HAPcryst

A HAP extension for crystallographic groups

Version 0.2.1

18 August 2026

Marc Roeder

Marc Roeder Email: roeder.marc@gmail.com

Copyright

© 2007 Marc Röder.

This package is distributed under the terms of the GNU General Public License version 2 or later (at your convenience). See the file `LICENSE` or `https://www.gnu.org/copyleft/gpl.html`

Acknowledgements

This work was supported by Marie Curie Grant No. MTKD-CT-2006-042685

Contents

1 Introduction **4**
- 1.1 Abstract and Notation . . . 4
- 1.2 Requirements . . . 5
- 1.3 Global Variables . . . 5

2 Bits and Pieces **6**
- 2.1 Matrices and Vectors . . . 6
- 2.2 Affine Matrices OnRight . . . 7
- 2.3 Geometry . . . 8
- 2.4 Space Groups . . . 9

3 Algorithms of Orbit-Stabilizer Type **10**
- 3.1 Orbit Stabilizer for Crystallographic Groups . . . 10

4 Resolutions of Crystallographic Groups **15**
- 4.1 Fundamental Domains . . . 15
- 4.2 Face Lattice and Resolution . . . 18

A Resolutions in Hap **21**
- A.1 The Standard Representation `HapResolutionRep` . . . 21
- A.2 The `HapLargeGroupResolutionRep` Representation . . . 22

B Accessing and Manipulating Resolutions **23**
- B.1 Representation-Independent Access Methods . . . 23
- B.2 Converting Between Representations . . . 25
- B.3 Special Methods for `HapResolutionRep` . . . 26
- B.4 The `HapLargeGroupResolutionRep` Representation . . . 27

C Contracting Homotopies **30**
- C.1 The `PartialContractingHomotopy` Data Type . . . 30

References **32**

Index **33**

Chapter 1

Introduction

1.1 Abstract and Notation

HAPcryst is an extension for "Homological Algebra Programming" (HAP, [Ell]) by Graham Ellis. It uses geometric methods to calculate resolutions for crystallographic groups. In this manual, we will use the terms "space group" and "crystallographic group" synonymous. As usual in GAP, group elements are supposed to act from the right. To emphasize this fact, some functions have names ending in "OnRight" (namely those, which rely on the action from the right). This is also meant to make work with HAPcryst and cryst [EGN] easier.

The functions called "somethingStandardSpaceGroup" are supposed to work for standard crystallographic groups on left and right some time in the future. Currently only the versions acting on right are implemented. As in cryst [EGN], space groups are represented as affine linear groups. For the computations in HAPcryst, crystallographic groups have to be in "standard form". That is, the translation basis has to be the standard basis of the space. This implies that the linear part of a group element need not be orthogonal with respect to the usual scalar product.

1.1.1 The natural action of crystallographic groups

There is some confusion about the way crystallographic groups are written. This concerns the question if we act on left or on right and if vectors are of the form `[1,...]` or `[...,1]`.

As mentioned, HAPcryst handles affine crystallographic groups on right (and maybe later also on left) acting on vectors of the form $[...,1]$.

BUT: The functions in HAPcryst do not take augmented vectors as input (no leading or ending ones). The handling of vectors is done internally. So in HAPcryst, a crystallographic group is a group of $n \times n$ matrices which acts on a vector space of dimension $n-1$ whose elements are vectors of length $n-1$ (not n). Example:

Example

```
gap> G:=SpaceGroup(3,4); #This group acts on 3-Space
SpaceGroupOnRightBBNWZ( 3, 2, 1, 1, 2 )
gap> Display(Representative(G));
[ [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  0,  0,  1 ] ]
gap> OrbitStabilizerInUnitCubeOnRight(G,[1/2,0,0]);
rec( orbit := [ [ 1/2, 0, 0 ], [ 1/2, 1/2, 0 ] ],
```

```
stabilizer := Group([ [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ],
        [ 0, 0, 0, 1 ] ] ]) )
```

1.2 Requirements

The following GAP packages are required

- polymaking which in turn depends on the computational geometry software polymake.
- HAP
- Cryst

The following GAP packages are not required but highly recommended:

- carat
- CrystCat
- GAPDoc is needed to display the online manual

1.2.1 Recommendation concerning polymake

Calculating resolutions of Bieberbach groups involves convex hull computations. polymake by default uses cdd to compute convex hulls. Experiments suggest that lrs is the more suitable algorithm for the computations done in HAPcryst than the default cdd. You can change the behaviour of by editing the file "yourhomedirectory/.polymake/prefer.pl". It should contain a section like this (just make sure lrs is before cdd, the position of beneath_beyond does not matter):

```
##########################################
application polytope;

prefer "*.convex_hull  lrs, beneath_beyond, cdd";
```

1.3 Global Variables

HAPcryst itself does only have one global variable, namely `InfoHAPcryst` (1.3.1). The location of files generated for interaction with polymake are determined by the polymaking.

1.3.1 InfoHAPcryst

▷ `InfoHAPcryst` (info class)

At a level of 1, only the most important messages are printed. At level 2, additional information is displayed, and level 3 is even more verbose. At level 0, HAPcryst remains silent.

Chapter 2

Bits and Pieces

This chapter contains a few very basic functions which are needed for space group calculations and were missing in standard GAP.

2.1 Matrices and Vectors

2.1.1 SignRat

▷ `SignRat(x)` (method)

Returns: sign of the rational number x (Standard GAP currently only has `SignInt`).

2.1.2 VectorModOne

▷ `VectorModOne(v)` (method)

Returns: Rational vector of the same length with enties in $[0, 1)$

For a rational vector v, this returns the vector with all entries taken "mod 1".

Example

```
gap> SignRat((-4)/(-2));
1
gap> SignRat(9/(-2));
-1
gap> VectorModOne([1/10,100/9,5/6,6/5]);
[ 1/10, 1/9, 5/6, 1/5 ]
```

2.1.3 IsSquareMat

▷ `IsSquareMat(matrix)` (method)

Returns: `true` if *matrix* is a square matrix and `false` otherwise.

2.1.4 DimensionSquareMat

▷ `DimensionSquareMat(matrix)` (method)

Returns: Number of lines in the matrix *matrix* if it is square and `fail` otherwise

Example

```
gap> m:=[[1,2,3],[4,5,6],[9,6,12]];
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 9, 6, 12 ] ]
```

```
gap> IsSquareMat(m);
true
gap> DimensionSquareMat(m);
3
gap> DimensionSquareMat([[1,2,3],[4,5,6]]);
fail
```

Affine mappings of n dimensional space are often written as a pair (A,v) where A is a linear mapping and v is a vector. GAP represents affine mappings by $n+1$ times $n+1$ matrices M which satisfy $M_{n+1,n+1}=1$ and $M_{i,n+1}=0$ for all $1\leq i\leq n$.

An affine matrix acts on an n dimensional space which is written as a space of $n+1$ tuples with $n+1$st entry 1. Here we give two functions to handle these affine matrices.

2.2 Affine Matrices OnRight

2.2.1 LinearPartOfAffineMatOnRight

▷ `LinearPartOfAffineMatOnRight(mat)` (method)

Returns: the linear part of the affine matrix `mat`. That is, everything except for the last row and column.

2.2.2 BasisChangeAffineMatOnRight

▷ `BasisChangeAffineMatOnRight(transform, mat)` (method)

Returns: affine matrix with same dimensions as `mat`

A basis change `transform` of an n dimensional space induces a transformation on affine mappings on this space. If `mat` is a affine matrix (in particular, it is $(n+1)\times(n+1)$), this method returns the image of `mat` under the basis transformation induced by `transform`.

Example

```
gap> c:=[[0,1],[1,0]];
[ [ 0, 1 ], [ 1, 0 ] ]
gap> m:=[[1/2,0,0],[0,2/3,0],[1,0,1]];
[ [ 1/2, 0, 0 ], [ 0, 2/3, 0 ], [ 1, 0, 1 ] ]
gap> BasisChangeAffineMatOnRight(c,m);
[ [ 2/3, 0, 0 ], [ 0, 1/2, 0 ], [ 0, 1, 1 ] ]
```

2.2.3 TranslationOnRightFromVector

▷ `TranslationOnRightFromVector(v)` (method)

Returns: Affine matrix

Given a vector v with n entries, this method returns a $(n+1)\times(n+1)$ matrix which corresponds to the affine translation defined by v.

Example

```
gap> m:=TranslationOnRightFromVector([1,2,3]);;
gap> Display(m);
[ [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  1,  0 ],
```

```
    [  1,  2,  3,  1 ] ]
gap> LinearPartOfAffineMatOnRight(m);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> BasisChangeAffineMatOnRight([[3,2,1],[0,1,0],[0,0,1]],m);
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 3, 4, 4, 1 ] ]
```

2.3 Geometry

2.3.1 GramianOfAverageScalarProductFromFiniteMatrixGroup

▷ `GramianOfAverageScalarProductFromFiniteMatrixGroup(G)` (method)

Returns: Symmetric positive definite matrix

For a finite matrix group `G`, the gramian matrix of the average scalar product is returned. This is the sum over all gg^t with $g \in G$ (actually it is enough to take a generating set). The group `G` is orthogonal with respect to the scalar product induced by the returned matrix.

2.3.2 Inequalities

Inequalities are represented in the same way they are represented in polymaking. The vector $(v_0,\ldots,v_n)$ represents the inequality $0 \leq v_0 + v_1x_1 + \ldots + v_nx_n$.

2.3.3 BisectorInequalityFromPointPair

▷ `BisectorInequalityFromPointPair(v1, v2[, gram])` (method)

Returns: vector of length `Length(v1)+1`

Calculates the inequality defining the half-space containing `v1` such that `v1-v2` is perpendicular on the bounding hyperplane. And `(v1-v2)/2` is contained in the bounding hyperplane.
If the matrix `gram` is given, it is used as the gramian matrix. Otherwiese, the standard scalar product is used. It is not checked if `gram` is positive definite or symmetric.

2.3.4 WhichSideOfHyperplane

▷ `WhichSideOfHyperplane(v, ineq)` (method)
▷ `WhichSideOfHyperplaneNC(v, ineq)` (method)

Returns: -1 (below) 0 (in) or 1 (above).

Let `v` be a vector of length n and `ineq` an inequality represented by a vector of length $n+1$. Then `WhichSideOfHyperplane(v, ineq)` returns 1 if `v` is a solution of the inequality but not the equation given by `ineq`, it returns 0 if `v` is a solution to the equation and -1 if it is not a solution of the inequality `ineq`.

The NC version does not test the input for correctness.

Example

```
gap> BisectorInequalityFromPointPair([0,0],[1,0]);
[ 1, -2, 0 ]
gap> ineq:=BisectorInequalityFromPointPair([0,0],[1,0],[[5,4],[4,5]]);
[ 5, -10, -8 ]
gap> ineq{[2,3]}*[1/2,0];
-5
gap> WhichSideOfHyperplane([0,0],ineq);
```

```
1
gap> WhichSideOfHyperplane([1/2,0],ineq);
0
```

2.3.5 RelativePositionPointAndPolygon

▷ `RelativePositionPointAndPolygon(`*`point, poly`*`)` (operation)

Returns: one of `"VERTEX", "FACET", "OUTSIDE", "INSIDE"`

Let *`poly`* be a `PolymakeObject` and *`point`* a vector. If *`point`* is a vertex of *`poly`*, the string `"VERTEX"` is returned. If *`point`* lies inside *`poly`*, `"INSIDE"` is returned and if it lies in a facet, `"FACET"` is returned and if *`point`* does not lie inside *`poly`*, the function returns `"OUTSIDE"`.

2.4 Space Groups

2.4.1 PointGroupRepresentatives

▷ `PointGroupRepresentatives(`*`group`*`)` (attribute)

Returns: list of matrices

Given an `AffineCrystGroupOnLeftOrRight` *`group`*, this returns a list of representatives of the point group of *`group`*. That is, a system of representatives for the factor group modulo translations. This is an attribute of `AffineCrystGroupOnLeftOrRight`

Chapter 3

Algorithms of Orbit-Stabilizer Type

We introduce a way to calculate a sufficient part of an orbit and the stabilizer of a point.

3.1 Orbit Stabilizer for Crystallographic Groups

3.1.1 OrbitStabilizerInUnitCubeOnRight

▷ `OrbitStabilizerInUnitCubeOnRight(`*group*`, `*x*`)` (method)

Returns: A record containing

- `.stabilizer`: the stabilizer of *x*.
- `.orbit` set of vectors from $[0,1)^n$ which represents the orbit.

Let *x* be a rational vector from $[0,1)^n$ and *group* a space group in standard form. The function then calculates the part of the orbit which lies inside the cube $[0,1)^n$ and the stabilizer of *x*. Observe that every element of the full orbit differs from a point in the returned orbit only by a pure translation.

Note that the restriction to points from $[0,1)^n$ makes sense if orbits should be compared and the vector passed to `OrbitStabilizerInUnitCubeOnRight` should be an element of the returned orbit (part).

Example

```
gap> S:=SpaceGroup(3,5);;
gap> OrbitStabilizerInUnitCubeOnRight(S,[1/2,0,9/11]);
rec( orbit := [ [ 0, 1/2, 2/11 ], [ 1/2, 0, 9/11 ] ],
  stabilizer := Group([ [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ],
          [ 0, 0, 0, 1 ] ] ]) )
gap> OrbitStabilizerInUnitCubeOnRight(S,[0,0,0]);
rec( orbit := [ [ 0, 0, 0 ] ], stabilizer := <matrix group with 2 generators> )
```

If you are interested in other parts of the orbit, you can use `VectorModOne` (2.1.2) for the base point and the functions `ShiftedOrbitPart` (3.1.9), `TranslationsToOneCubeAroundCenter` (3.1.10) and `TranslationsToBox` (3.1.11) for the resulting orbit
Suppose we want to calculate the part of the orbit of `[4/3,5/3,7/3]` in the cube of sidelength `1` around this point:

Example

```
gap> S:=SpaceGroup(3,5);;
gap> p:=[4/3,5/3,7/3];;
```

```
gap> o:=OrbitStabilizerInUnitCubeOnRight(S,VectorModOne(p)).orbit;
[ [ 1/3, 2/3, 1/3 ], [ 1/3, 2/3, 2/3 ] ]
gap> box:=p+[[-1,1],[-1,1],[-1,1]];
[ [ 1/3, 8/3, 7/3 ], [ 1/3, 8/3, 7/3 ], [ 1/3, 8/3, 7/3 ] ]
gap> o2:=Concatenation(List(o,i->i+TranslationsToBox(i,box)));;
gap> # This is what we looked for. But it is somewhat large:
gap> Size(o2);
54
```

3.1.2 OrbitStabilizerInUnitCubeOnRightOnSets

▷ `OrbitStabilizerInUnitCubeOnRightOnSets(`*group, set*`)` (method)

Returns: A record containing

- `.stabilizer`: the stabilizer of *set*.
- `.orbit` set of sets of vectors from $[0,1)^n$ which represents the orbit.

Calculates orbit and stabilizer of a set of vectors. Just as `OrbitStabilizerInUnitCubeOnRight` (3.1.1), it needs input from $[0,1)^n$. The returned orbit part `.orbit` is a set of sets such that every element of `.orbit` has a non-trivial intersection with the cube $[0,1)^n$. In general, these sets will not lie inside $[0,1)^n$ completely.

Example

```
gap> S:=SpaceGroup(3,5);;
gap> OrbitStabilizerInUnitCubeOnRightOnSets(S,[[0,0,0],[0,1/2,0]]);
rec(
  orbit := [ [ [ -1/2, 0, 0 ], [ 0, 0, 0 ] ], [ [ 0, 0, 0 ], [ 0, 1/2, 0 ] ],
      [ [ 1/2, 0, 0 ], [ 1, 0, 0 ] ] ], stabilizer := <matrix group of size
    1 with 3 generators> )
```

3.1.3 OrbitPartInVertexSetsStandardSpaceGroup

▷ `OrbitPartInVertexSetsStandardSpaceGroup(`*group, vertexset, allvertices*`)` (method)

Returns: Set of subsets of *allvertices*.

If *allvertices* is a set of vectors and *vertexset* is a subset thereof, then `OrbitPartInVertexSetsStandardSpaceGroup` returns that part of the orbit of *vertexset* which consists entirely of subsets of *allvertices*. Note that,unlike the other `OrbitStabilizer` algorithms, this does not require the input to lie in some particular part of the space.

Example

```
gap> S:=SpaceGroup(3,5);;
gap> OrbitPartInVertexSetsStandardSpaceGroup(S,[[0,1,5],[1,2,0]],
> Set([[1,2,0],[2,3,1],[1,2,6],[1,1,0],[0,1,5],[3/5,7,12],[1/17,6,1/2]]));
[ [ [ 0, 1, 5 ], [ 1, 2, 0 ] ], [ [ 1, 2, 6 ], [ 2, 3, 1 ] ] ]
gap> OrbitPartInVertexSetsStandardSpaceGroup(S, [[1,2,0]],
> Set([[1,2,0],[2,3,1],[1,2,6],[1,1,0],[0,1,5],[3/5,7,12],[1/17,6,1/2]]));
[ [ [ 0, 1, 5 ] ], [ [ 1, 1, 0 ] ], [ [ 1, 2, 0 ] ], [ [ 1, 2, 6 ] ], [ [ 2, 3, 1 ] ] ]
```

3.1.4 OrbitPartInFacesStandardSpaceGroup

▷ OrbitPartInFacesStandardSpaceGroup(*group, vertexset, faceset*) (method)

Returns: Set of subsets of *faceset*.

This calculates the orbit of a space group on sets restricted to a set of faces.

If *faceset* is a set of sets of vectors and *vertexset* is an element of *faceset*, then OrbitPartInFacesStandardSpaceGroup returns that part of the orbit of *vertexset* which consists entirely of elements of *faceset*.

Note that,unlike the other OrbitStabilizer algorithms, this does not require the input to lie in some particular part of the space.

3.1.5 OrbitPartAndRepresentativesInFacesStandardSpaceGroup

▷ OrbitPartAndRepresentativesInFacesStandardSpaceGroup(*group, vertexset, faceset*) (method)

Returns: A set of face-matrix pairs .

This is a slight variation of OrbitPartInFacesStandardSpaceGroup (3.1.4) that also returns a representative for every orbit element.

Example

```
gap> S:=SpaceGroup(3,5);;
gap> OrbitPartInVertexSetsStandardSpaceGroup(S,[[0,1,5],[1,2,0]],
> Set([[1,2,0],[2,3,1],[1,2,6],[1,1,0],[0,1,5],[3/5,7,12],[1/17,6,1/2]]));
[ [ [ 0, 1, 5 ], [ 1, 2, 0 ] ], [ [ 1, 2, 6 ], [ 2, 3, 1 ] ] ]
gap> OrbitPartInFacesStandardSpaceGroup(S,[[0,1,5],[1,2,0]],
> Set( [ [ [ 0, 1, 5 ], [ 1, 2, 0 ] ], [[1/17,6,1/2],[1,2,7]]]));
[ [ [ 0, 1, 5 ], [ 1, 2, 0 ] ] ]
gap> OrbitPartAndRepresentativesInFacesStandardSpaceGroup(S,[[0,1,5],[1,2,0]],
> Set( [ [ [ 0, 1, 5 ], [ 1, 2, 0 ] ], [[1/17,6,1/2],[1,2,7]]]));
[ [ [ [ 0, 1, 5 ], [ 1, 2, 0 ] ],
      [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ] ] ]
```

3.1.6 StabilizerOnSetsStandardSpaceGroup

▷ StabilizerOnSetsStandardSpaceGroup(*group, set*) (method)

Returns: finite group of affine matrices (OnRight)

Given a set *set* of vectors and a space group *group* in standard form, this method calculates the stabilizer of that set in the full crystallographic group.

Example

```
gap> G:=SpaceGroup(3,12);;
gap> v:=[ 0, 0,0 ];;
gap> s:=StabilizerOnSetsStandardSpaceGroup(G,[v]);
<matrix group with 2 generators>
gap> s2:=OrbitStabilizerInUnitCubeOnRight(G,v).stabilizer;
<matrix group with 2 generators>
gap> s2=s;
true
```

3.1.7 RepresentativeActionOnRightOnSets

▷ RepresentativeActionOnRightOnSets(*group, set, imageset*) (method)

Returns: Affine matrix.

Returns an element of the space group S which takes the set *set* to the set *imageset*. The group must be in standard form and act on the right.

Example

```
gap> S:=SpaceGroup(3,5);;
gap> RepresentativeActionOnRightOnSets(G, [[0,0,0],[0,1/2,0]],
>        [ [ 0, 1/2, 0 ], [ 0, 1, 0 ] ]);
[ [ -1, 0, 0, 0 ], [ 0, -1, 0, 0 ], [ 0, 0, -1, 0 ], [ 0, 1, 0, 1 ] ]
```

3.1.8 Getting other orbit parts

HAPcryst does not calculate the full orbit but only the part of it having coefficients between $-1/2$ and $1/2$. The other parts of the orbit can be calculated using the following functions.

3.1.9 ShiftedOrbitPart

▷ ShiftedOrbitPart(*point, orbitpart*) (method)

Returns: Set of vectors

Takes each vector in *orbitpart* to the cube unit cube centered in *point*.

Example

```
gap> ShiftedOrbitPart([0,0,0],[[1/2,1/2,1/3],-[1/2,1/2,1/2],[19,3,1]]);
[ [ 0, 0, 0 ], [ 1/2, 1/2, 1/3 ], [ 1/2, 1/2, 1/2 ] ]
gap> ShiftedOrbitPart([1,1,1],[[1/2,1/2,1/2],-[1/2,1/2,1/2]]);
[ [ 3/2, 3/2, 3/2 ] ]
```

3.1.10 TranslationsToOneCubeAroundCenter

▷ TranslationsToOneCubeAroundCenter(*point, center*) (method)

Returns: List of integer vectors

This method returns the list of all integer vectors which translate *point* into the box $center+[-1/2,1/2]^n$

Example

```
gap> TranslationsToOneCubeAroundCenter([1/2,1/2,1/3],[0,0,0]);
[ [ 0, 0, 0 ], [ 0, -1, 0 ], [ -1, 0, 0 ], [ -1, -1, 0 ] ]
gap> TranslationsToOneCubeAroundCenter([1,0,1],[0,0,0]);
[ [ -1, 0, -1 ] ]
```

3.1.11 TranslationsToBox

▷ TranslationsToBox(*point, box*) (method)

Returns: List of integer vectors or the empty list

Given a vector v and a list of pairs, this function returns the translation vectors (integer vectors) which take v into the box *box*. The box *box* has to be given as a list of pairs.

Example

```
gap> TranslationsToBox([0,0],[[1/2,2/3],[1/2,2/3]]);
[  ]
gap> TranslationsToBox([0,0],[[-3/2,1/2],[1,4/3]]);
[ [ -1, 1 ], [ 0, 1 ] ]
gap> TranslationsToBox([0,0],[[-3/2,1/2],[2,1]]);
Error, Box must not be empty
```

Chapter 4

Resolutions of Crystallographic Groups

4.1 Fundamental Domains

Let S be a crystallographic group. A Fundamental domain is a closed convex set containing a system of representatives for the Orbits of S in its natural action on euclidian space.
There are two algorithms for calculating fundamental domains in HAPcryst. One uses the geometry and relies on having the standard rule for evaluating the scalar product (i.e. the gramian matrix is the identity). The other one is independent of the gramian matrix but does only work for Bieberbach groups, while the first ("geometric") algorithm works for arbitrary crystallographic groups given a point with trivial stabilizer.

4.1.1 FundamentalDomainStandardSpaceGroup

▷ `FundamentalDomainStandardSpaceGroup(`*`[v, ]G`*`)` (operation)

Returns: a `PolymakeObject`

Let *G* be an `AffineCrystGroupOnRight` and *v* a vector. A fundamental domain containing *v* is calculated and returned as a `PolymakeObject`. The vector *v* is used as the starting point for a Dirichlet-Voronoi construction. If no *v* is given, the origin is used as starting point if it has trivial stabiliser. Otherwise an error is cast.

Example

```
gap> fd:=FundamentalDomainStandardSpaceGroup([1/2,0,1/5],SpaceGroup(3,9));
<polymake object>
gap> Polymake(fd,"N_VERTICES");
24
gap> fd:=FundamentalDomainStandardSpaceGroup(SpaceGroup(3,9));
<polymake object>
gap> Polymake(fd,"N_VERTICES");
8
```

4.1.2 FundamentalDomainBieberbachGroup

▷ `FundamentalDomainBieberbachGroup(`*`[v, ]G[, gram]`*`)` (operation)

Returns: a `PolymakeObject`

Given a starting vector *v* and a Bieberbach group *G* in standard form, this method calculates the Dirichlet domain with respect to *v*. If *gram* is not supplied, the average gramian matrix is used (see

`GramianOfAverageScalarProductFromFiniteMatrixGroup` (2.3.1)). It is not tested if *gram* is symmetric and positive definite. It is also not tested, if the product defined by *gram* is invariant under the point group of *G*.

The behaviour of this function is influenced by the option `ineqThreshold`. The algorithm calculates approximations to a fundamental domain by iteratively adding inequalities. For an approximating polyhedron, every vertex is tested to find new inequalities. When all vertices have been considered or the number of new inequalities already found exceeds the value of `ineqThreshold`, a new approximating polyhedron in calculated. The default for `ineqThreshold` is 200. Roughly speaking, a large threshold means shifting work from `polymake` to GAP, a small one means more calls of (and work for) `polymake`.

If the value of `InfoHAPcryst` (1.3.1) is 2 or more, for each approximation the number of vertices of the approximation, the number of vertices that have to be considered during the calculation, the number of facets, and new inequalities is shown.

Note that the algorithm chooses vertices in random order and also writes inequalities for `polymake` in random order.

Example

```
gap> a0:=[[ 1, 0, 0, 0, 0, 0, 0 ], [ 0, -1, 0, 0, 0, 0, 0 ],
>     [ 0, 0, 1, 0, 0, 0, 0 ], [ 0, 0, 0, 1, 0, 0, 0 ],
>     [ 0, 0, 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, -1, -1, 0 ],
>     [ -1/2, 0, 0, 1/6, 0, 0, 1 ]
>     ];;
gap> a1:=[[ 0, -1, 0, 0, 0, 0, 0 ],[ 0, 0, -1, 0, 0, 0, 0 ],
>         [ 1, 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 1, 0, 0, 0 ],
>         [ 0, 0, 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 0, 1, 0 ],
>         [ 0, 0, 0, 0, 1/3, -1/3, 1 ]
>        ];;
gap> trans:=List(Group((1,2,3,4,5,6)),g->
>           TranslationOnRightFromVector(Permuted([1,0,0,0,0,0],g)));;
gap> S:=AffineCrystGroupOnRight(Concatenation(trans,[a0,a1]));
<matrix group with 8 generators>
gap> SetInfoLevel(InfoHAPcryst,2);
gap> FundamentalDomainBieberbachGroup(S:ineqThreshold:=10);
#I  v: 104/104 f:15
#I  new: 201
#I  v: 961/961 f:58
#I  new: 20
#I  v: 1143/805 f:69
#I  new: 12
#I  v: 1059/555 f:64
#I  new: 15
#I  v: 328/109 f:33
#I  new: 12
#I  v: 336/58 f:32
#I  new: 0
<polymake object>
gap> FundamentalDomainBieberbachGroup(S:ineqThreshold:=1000);
#I  v: 104/104 f:15
#I  new: 149
#I  v: 635/635 f:41
#I  new: 115
#I  v: 336/183 f:32
```

```
#I  new: 0
#I  out of inequalities
<polymake object>
```

4.1.3 FundamentalDomainFromGeneralPointAndOrbitPartGeometric

▷ FundamentalDomainFromGeneralPointAndOrbitPartGeometric(*v, orbit*) (method)

Returns: a PolymakeObject

This uses an alternative algorithm based on geometric considerations. It is not used in any of the high-level methods. Let *v* be a vector and *orbit* a sufficiently large part of the orbit of *v* under a crystallographic group with standard- orthogonal point group (satisfying $A^t = A^{-}1$). A geometric algorithm is then used to calculate the Dirichlet domain with respect to *v*. This also works for crystallographic groups which are not Bieberbach. The point *v* has to have trivial stabilizer.
The intersection of the full orbit with the unit cube around *v* is sufficiently large.

Example

```
gap> G:=SpaceGroup(3,9);;
gap> v:=[0,0,0];
[ 0, 0, 0 ]
gap> orbit:=OrbitStabilizerInUnitCubeOnRight(G,v).orbit;
[ [ 0, 0, 0 ], [ 0, 0, 1/2 ] ]
gap> fd:=FundamentalDomainFromGeneralPointAndOrbitPartGeometric(v,orbit);
<polymake object>
gap> Polymake(fd,"N_VERTICES");
8
```

4.1.4 IsFundamentalDomainStandardSpaceGroup

▷ IsFundamentalDomainStandardSpaceGroup(*poly, G*) (method)

Returns: true or false

This tests if a PolymakeObject *poly* is a fundamental domain for the affine crystallographic group *G* in standard form.
The function tests the following: First, does the orbit of any vertex of *poly* have a point inside *poly* (if this is the case, false is returned). Second: Is every facet of *poly* the image of a different facet under a group element which does not fix *poly*. If this is satisfied, true is returned.

4.1.5 IsFundamentalDomainBieberbachGroup

▷ IsFundamentalDomainBieberbachGroup(*poly, G*) (method)

Returns: true, false or fail

This tests if a PolymakeObject *poly* is a fundamental domain for the affine crystallographic group *G* in standard form and if this group is torsion free (ie a Bieberbach group)
It returns true if *G* is torsion free and *poly* is a fundamental domain for *G*. If *poly* is not a fundamental domain, false is returned regardless of the structure of *G*. And if *G* is not torsion free, the method returns fail. If *G* is polycyclic, torsion freeness is tested using a representation as pcp group. Otherwise the stabilisers of the faces of the fundamental domain *poly* are calculated (*G* is torsion free if and only if it all these stabilisers are trivial).

4.2 Face Lattice and Resolution

For Bieberbach groups (torsion free crystallographic groups), the following functions calculate free resolutions. This calculation is done by finding a fundamental domain for the group. For a description of the HapResolution datatype, see the Hap data types documentation or the experimental datatypes documentation A

4.2.1 ResolutionBieberbachGroup

▷ ResolutionBieberbachGroup(*G[, v]*) (method)

Returns: a HAPresolution

Let *G* be a Bieberbach group given as an AffineCrystGroupOnRight and *v* a vector. Then a Dirichlet domain with respect to *v* is calculated using FundamentalDomainBieberbachGroup (4.1.2). From this domain, a resolution is calculated using FaceLatticeAndBoundaryBieberbachGroup (4.2.2) and ResolutionFromFLandBoundary (4.2.3). If *v* is not given, the origin is used.

Example

```
gap> R:=ResolutionBieberbachGroup(SpaceGroup(3,9));
Resolution of length 3 in characteristic
0 for SpaceGroupOnRightBBNWZ( 3, 2, 2, 2, 2 ) .
No contracting homotopy available.

gap> List([0..3],Dimension(R));
[ 1, 3, 3, 1 ]
gap> R:=ResolutionBieberbachGroup(SpaceGroup(3,9),[1/2,0,0]);
Resolution of length 3 in characteristic
0 for SpaceGroupOnRightBBNWZ( 3, 2, 2, 2, 2 ) .
No contracting homotopy available.

gap> List([0..3],Dimension(R));
[ 6, 12, 7, 1 ]
```

4.2.2 FaceLatticeAndBoundaryBieberbachGroup

▷ FaceLatticeAndBoundaryBieberbachGroup(*poly, group*) (method)

Returns: Record with entries .hasse and .elts representing a part of the hasse diagram and a lookup table of group elements

Let *group* be a torsion free AffineCrystGroupOnRight (that is, a Bieberbach group). Given a PolymakeObject *poly* representing a fundamental domain for *group*, this method uses polymaking to calculate the face lattice of *poly*. From the set of faces, a system of representatives for *group*-orbits is chosen. For each representative, the boundary is then calculated. The list .elts contains elements of *group* (in fact, it is even a set). The structure of the returned list .hasse is as follows:

- The i-th entry contains a system of representatives for the $i-1$ dimensional faces of *poly*.
- Each face is represented by a pair of lists [vertices,boundary]. The list of integers vertices represents the vertices of *poly* which are contained in this face. The enumeration is chosen such that an i in the list represents the i-th entry of the list Polymake(poly,"VERTICES");

- The list `boundary` represents the boundary of the respective face. It is a list of pairs of integers `[j,g]`. The first entry lies between $-n$ and n, where n is the number of faces of dimension $i-1$. This entry represents a face of dimension $i-1$ (or its additive inverse as a module generator). The second entry `g` is the position of the matrix in `.elts`.

This representation is compatible with the representation of free $\mathbb{Z}G$ modules in Hap and this method essentially calculates a free resolution of *group*. If the value of `InfoHAPcryst` (1.3.1) is 2 or more, additional information about the number of faces in every codimension, the number of orbits of the group on the free module generated by those faces, and the time it took to calculate the orbit decomposition is output.

Example

```
gap> SetInfoLevel(InfoHAPcryst,2);
gap> G:=SpaceGroup(3,165);
SpaceGroupOnRightBBNWZ( 3, 6, 1, 1, 4 )
gap> fd:=FundamentalDomainBieberbachGroup(G);
<polymake object>
gap> fl:=FaceLatticeAndBoundaryBieberbachGroup(fd,G);;
#I  1(4/8): 0:00:00.004
#I  2(5/18): 0:00:00.000
#I  3(2/12): 0:00:00.000
#I  Face lattice done ( 0:00:00.004). Calculating boundary
#I  done ( 0:00:00.004) Reformating...
gap> Set(RecNames(fl));
[ "elts", "groupring", "hasse" ]
gap> fl.groupring;
<free left module over Integers, and ring-with-one, with 10 generators>
```

4.2.3 ResolutionFromFLandBoundary

▷ `ResolutionFromFLandBoundary(`*fl, group*`)` (method)

Returns: Free resolution

If *fl* is the record output by `FaceLatticeAndBoundaryBieberbachGroup` (4.2.2) and *group* is the corresponding group, this function returns a `HapResolution`. Of course, *fl* has to be generated from a fundamental domain for *group*

Example

```
gap> G:=SpaceGroup(3,165);
SpaceGroupOnRightBBNWZ( 3, 6, 1, 1, 4 )
gap> fd:=FundamentalDomainBieberbachGroup(G);
<polymake object>
gap> fl:=FaceLatticeAndBoundaryBieberbachGroup(fd,G);;
gap> ResolutionFromFLandBoundary(fl,G);
Resolution of length 3 in characteristic
0 for SpaceGroupOnRightBBNWZ( 3, 6, 1, 1, 4 ) .
No contracting homotopy available.

gap> ResolutionFromFLandBoundary(fl,G);
Resolution of length 3 in characteristic
0 for SpaceGroupOnRightBBNWZ( 3, 6, 1, 1, 4 ) .
No contracting homotopy available.
```

```
gap> List([0..4],Dimension(last));
[ 2, 5, 4, 1, 0 ]
```

Appendix A

Resolutions in Hap

This document is only concerned with the representation of resolutions in Hap. Note that it is not a part of Hap. The framework provided here is just an extension of Hap data types used in HAPcryst and HAPprime.

From now on, let G be a group and $\ldots \to M_n \to M_{n-1} \to \ldots \to M_1 \to M_0 \to Z$ be a resolution with free ZG modules M_i.

The elements of the modules M_i can be represented in different ways. This is what makes different representations for resolutions desirable. First, we will look at the standard representation (`HapResolutionRep`) as it is defined in Hap. After that, we will present another representation for infinite groups. Note that all non-standard representations must be sub-representations of the standard representation to ensure compatibility with Hap.

A.1 The Standard Representation `HapResolutionRep`

For every M_i we fix a basis and number its elements. Furthermore, it is assumed that we have a (partial) enumeration of the group of a resolution. In practice this is done by generating a lookup table on the fly.

In standard representation, the elements of the modules M_k are represented by lists `-"words"-` of pairs of integers. A letter `[i,g]` of such a word consists of the number of a basis element `i` or `-i` for its additive inverse and a number g representing a group element.

A `HapResolution` in `HapResolutionRep` representation is a component object with the components

- `group`, a group of arbitrary type.
- `elts`, a (partial) list of (possibly duplicate) elements in G. This list provides the "enumeration" of the group. Note that there are functions in Hap which assume that `elts[1]` is the identity element of G.
- `appendToElts(g)` a function that appends the group element `g` to `.elts`. This is not documented in Hap 1.8.6 but seems to be required for infinite groups. This requirement might vanish in some later version of Hap [G. Ellis, private communication].
- `dimension(k)`, a function which returns the ZG-rank of the Module M_k

- `boundary(k,j)`, a function which returns the image in M_{k-1} of the jth free generator of M_k. Note that negative j are valid as input as well. In this case the additive inverse of the boundary of the jth generator is returned

- `homotopy(k,[i,g])` a function which returns the image in M_{k+1}, under a contracting homotopy $M_k \to M_{k+1}$, of the element `[[i,g]]` in M_k. The value of this might be `fail`. However, currently (version 1.8.4) some Hap functions assume that `homotopy` is a function without testing.

- `properties`, a list of pairs `["name","value"]` "name" is a string and value is anything (boolean, number, string...). Every `HapResolution` (regardless of representation) has to have `["type","resolution"]`, `["length",length]` where `length` is the length of the resolution and `["characteristic",char]`. Currently (Hap 1.8.6), `length` must not be `infinity`. The values of these properties can be tested using the Hap function `EvaluateProperty(resolution,propertyname)`.

Note that making `HapResolutions` immutable will make the `.elts` component immutable. As this lookup table might change during calculations, we do not recommend using immutable resolutions (in any representation).

A.2 The `HapLargeGroupResolutionRep` Representation

In this sub-representation of the standard representation, the module elements in this resolution are lists of groupring elements. So the lookup table `.elts` is not used as long as no conversion to standard representation takes place. In addition to the components of a `HapResolution`, a resolution in large group representation has the following components:

- `boundary2(resolution,term,gen)`, a function that returns the boundary of the *`gen`*th generator of the *`term`*th module.

- `groupring` the group ring of the resolution *`resolution`*.

- `dimension2(resolution,term)` a function that returns the dimension of the *`term`*th module of the resolution *`resolution`*.

The effort of having two versions of `boundary` and `dimension` is necessary to keep the structure compatible with the usual Hap resolution.

Appendix B

Accessing and Manipulating Resolutions

B.1 Representation-Independent Access Methods

All methods listed below take a HapResolution in any representation. If the other arguments are compatible with the representation of the resolution, the returned value will be in the form defined by this representation. If the other arguments are in a different representation, GAPs method selection is used via TryNextMethod() to find an applicable method (a suitable representation).

The idea behind this is that the results of computations have the same form as the input. And as all representations are sub-representations of the HapResolutionRep representation, input which is compatible with the HapResolutionRep representation is always valid.

Every new representation must support the functions of this section.

B.1.1 StrongestValidRepresentationForLetter

▷ StrongestValidRepresentationForLetter(*resolution, term, letter*) (method)

Returns: filter

Finds the sub-representation of HapResolutionRep for which *letter* is a valid letter of the *term*th module of *resolution*. Note that *resolution* automatically is in some sub-representation of HapResolutionRep.This is mainly meant for debugging.

B.1.2 StrongestValidRepresentationForWord

▷ StrongestValidRepresentationForWord(*resolution, term, word*) (method)

Returns: filter

Finds the sub-representation of HapResolutionRep for which *word* is a valid word of the *term*th module of *resolution*. Note that *resolution* automatically is in some sub-representation of HapResolutionRep. This is mainly meant for debugging.

B.1.3 PositionInGroupOfResolution

▷ PositionInGroupOfResolution(*resolution, g*) (method)

▷ PositionInGroupOfResolutionNC(*resolution, g*) (method)

Returns: positive integer

This returns the position of the group element *g* in the enumeration of the group of *resolution*. The NC version does not check if *g* really is an element of the group of *resolution*.

B.1.4 IsValidGroupInt

▷ `IsValidGroupInt(`*`resolution, n`*`)` (method)

Returns: boolean

Returns true if the *n*th element of the group of *resolution* is known.

B.1.5 GroupElementFromPosition

▷ `GroupElementFromPosition(`*`resolution, n`*`)` (method)

Returns: group element or `fail`

Returns *n*th element of the group of *resolution*. If the *n*th element is not known, `fail` is returned.

B.1.6 MultiplyGroupElts

▷ `MultiplyGroupElts(`*`resolution, x, y`*`)` (method)

Returns: positive integer or group element, depending on the type of *x* and *y*

If `x` and `y` are given in standard representation (i.e. as integers), this returns the position of the product of the group elements represented by the positive integers *x* and *x*.

If `x` and `y` are given in any other representation, the returned group element will also be represented in this way.

B.1.7 MultiplyFreeZGLetterWithGroupElt

▷ `MultiplyFreeZGLetterWithGroupElt(`*`resolution, letter, g`*`)` (method)

Returns: A letter

Multiplies the letter *letter* with the group element *g* and returns the result. If *resolution* is in standard representation, *g* has to be an integer and *letter* has to be a pair of integer. If *resolution* is in any other representation, *letter* and *g* can be in a form compatible with that representation or in the standard form (in the latter case, the returned value will also have standard form).

B.1.8 MultiplyFreeZGWordWithGroupElt

▷ `MultiplyFreeZGWordWithGroupElt(`*`resolution, word, g`*`)` (method)

Returns: A word

Multiplies the word *word* with the group element *g* and returns the result. If *resolution* is in standard representation, *g* has to be an integer and *word* has to be a list of pairs of integers. If *resolution* is in any other representation, *word* and *g* can be in a form compatible with that representation or in the standard form (in the latter case, the returned value will also have standard form).

B.1.9 BoundaryOfFreeZGLetter

▷ `BoundaryOfFreeZGLetter(`*`resolution, term, letter`*`)` (method)

Returns: free ZG word (in the same representation as *letter*)

Calculates the boundary of the letter (word of length 1) *letter* of the *term*th module of *resolution*.

The returned value is a word of the *term*-1st module and comes in the same representation as *letter*.

B.1.10 BoundaryOfFreeZGWord

▷ `BoundaryOfFreeZGWord(`*`resolution, term, word`*`)` (method)

Returns: free ZG word (in the same representation as *`letter`*)

Calculates the boundary of the word *`word`* of the *`term`*th module of *`resolution`*.

The returned value is a word of the *`term`*-1st module and comes in the same representation as *`word`*.

B.2 Converting Between Representations

Four methods are provided to convert letters and words from standard representation to any other representation and back again.

B.2.1 ConvertStandardLetter

▷ `ConvertStandardLetter(`*`resolution, term, letter`*`)` (method)

▷ `ConvertStandardLetterNC(`*`resolution, term, letter`*`)` (method)

Returns: letter in the representation of *`resolution`*

Converts the letter *`letter`* in standard representation to the representation of *`resolution`*. The NC version does not check whether *`letter`* really is a letter in standard representation.

B.2.2 ConvertStandardWord

▷ `ConvertStandardWord(`*`resolution, term, word`*`)` (method)

▷ `ConvertStandardWordNC(`*`resolution, term, word`*`)` (method)

Returns: word in the representation of *`resolution`*

Converts the word *`word`* in standard representation to the representation of *`resolution`*. The NC version does not check whether *`word`* is a valid word in standard representation.

B.2.3 ConvertLetterToStandardRep

▷ `ConvertLetterToStandardRep(`*`resolution, term, letter`*`)` (method)

▷ `ConvertLetterToStandardRepNC(`*`resolution, term, letter`*`)` (method)

Returns: letter in standard representation

Converts the letter *`letter`* in the representation of *`resolution`* to the standard representation. The NC version does not check whether *`letter`* is a valid letter of *`resolution`*.

B.2.4 ConvertWordToStandardRep

▷ `ConvertWordToStandardRep(`*`resolution, term, word`*`)` (method)

▷ `ConvertWordToStandardRepNC(`*`resolution, term, word`*`)` (method)

Returns: word in standard representation

Converts the word *`word`* in the representation of *`resolution`* to the standard representation. The NC version does not check whether *`word`* is a valid word of *`resolution`*.

B.3 Special Methods for HapResolutionRep

Some methods explicitely require the input to be in the standard representation (*HapResolutionRep*). Two of these test if a word/letter is really in standard representation, the other ones are non-check versions of the universal methods.

B.3.1 IsFreeZGLetter

▷ IsFreeZGLetter(*resolution, term, letter*) (method)

Returns: boolean

Checks if *letter* is an valid letter (word of length 1) in standard representation of the *term*th module of *resolution*.

B.3.2 IsFreeZGWord

▷ IsFreeZGWord(*resolution, term, word*) (method)

Returns: boolean

Check if *word* is a valid word in large standard representation of the *term*th module in *resolution*.

B.3.3 MultiplyGroupEltsNC

▷ MultiplyGroupEltsNC(*resolution, x, y*) (method)

Returns: positive integer

Given positive integers x and y, this returns the position of the product of the group elements represented by the positive integers *x* and *x*. This assumes that all input is in standard representation and does not check the input.

B.3.4 MultiplyFreeZGLetterWithGroupEltNC

▷ MultiplyFreeZGLetterWithGroupEltNC(*resolution, letter, g*) (method)

Returns: A letter in standard representation

Multiplies the letter *letter* with the group element represented by the positive integer *g* and returns the result. The input is assumed to be in *HapResolutionRep* and is not checked.

B.3.5 MultiplyFreeZGWordWithGroupEltNC

▷ MultiplyFreeZGWordWithGroupEltNC(*resolution, word, g*) (method)

Returns: A letter in standard representation

Multiplies the word *word* with the group element represented by the positive integer *g* and returns the result. The input is assumed to be in *HapResolutionRep* and is not checked.

B.3.6 BoundaryOfFreeZGLetterNC

▷ BoundaryOfFreeZGLetterNC(*resolution, term, letter*) (method)

Returns: free ZG word in standard representation

Calculates the boundary of the letter (word of length 1) *letter* of the *term*th module of *resolution*. The input is assumed to be in standard representation and not checked.

B.3.7 BoundaryOfFreeZGWordNC

▷ `BoundaryOfFreeZGWordNC(`*`resolution, term, word`*`)` (method)

Returns: free ZG word in standard representation

Calculates the boundary of the word *word* of the *term*th module of *resolution*. The input is assumed to be in standard representation and not checked.

B.4 The `HapLargeGroupResolutionRep` Representation

The large group representation has one additional component called `groupring`. Elements of the modules in a resolution are represented by lists of group ring elements. The length of the list corresponds to the dimension of the free module.

All methods for the generic representation do also work for the large group representation. Some of them are wrappers for special methods which do only work for this representation. The following list only contains the methods which are not already present in the generic representation.

Note that the input or the output of these functions does not comply with the standard representation.

B.4.1 GroupRingOfResolution

▷ `GroupRingOfResolution(`*`resolution`*`)` (method)

Returns: group ring

This returns the group ring of *resolution*. Note that by the way that group rings are handled in GAP, this is not equal to `GroupRing(R,GroupOfResolution(`*`resolution`*`))` where `R` is the ring of the resolution.

B.4.2 MultiplyGroupElts_LargeGroupRep

▷ `MultiplyGroupElts_LargeGroupRep(`*`resolution, x, y`*`)` (method)

▷ `MultiplyGroupEltsNC_LargeGroupRep(`*`resolution, x, y`*`)` (method)

Returns: group element

Returns the product of *x* and *y*. The NC version does not check whether *x* and *y* are actually elements of the group of *resolution*.

B.4.3 IsFreeZGLetterNoTermCheck_LargeGroupRep

▷ `IsFreeZGLetterNoTermCheck_LargeGroupRep(`*`resolution, letter`*`)` (method)

Returns: boolean

Returns `true` if *letter* has the form of a letter (a module element with exactly one non-zero entry which has exactly one non-zero coefficient) a module of *resolution* in the `HapLargeGroupResolution` representation. Note that it is not tested if *letter* actually is a letter in any term of *resolution*

B.4.4 IsFreeZGWordNoTermCheck_LargeGroupRep

▷ `IsFreeZGWordNoTermCheck_LargeGroupRep(`*`resolution, word`*`)` (method)

Returns: boolean

Returns `true` if *word* has the form of a word of a module of *resolution* in the `HapLargeGroupResolution` representation. Note that it is not tested if *word* actually is a word in any term of *resolution*.

B.4.5 IsFreeZGLetter_LargeGroupRep

▷ `IsFreeZGLetter_LargeGroupRep(`*resolution, term, letter*`)` (method)

Returns: boolean

Returns `true` if and only if *letter* is a letter (a word of length 1) of the *term*th module of *resolution* in the `hapLargeGroupResolution` representation. I.e. it tests if *letter* is a module element with exactly one non-zero entry which has exactly one non-zero coefficient.

B.4.6 IsFreeZGWord_LargeGroupRep

▷ `IsFreeZGWord_LargeGroupRep(`*resolution, term, word*`)` (method)

Returns: boolean

Tests if *word* is an element of the *term*th module of *resoultion*.

B.4.7 MultiplyFreeZGLetterWithGroupElt_LargeGroupRep

▷ `MultiplyFreeZGLetterWithGroupElt_LargeGroupRep(`*resolution, letter, g*`)` (method)
▷ `MultiplyFreeZGLetterWithGroupEltNC_LargeGroupRep(`*resolution, letter, g*`)` (method)

Returns: free ZG letter in large group representation

Multiplies the letter *letter* with the group element *g* and returns the result. The NC version does not check whether *g* is an element of the group of *resolution* and *letter* can be a letter.

B.4.8 MultiplyFreeZGWordWithGroupElt_LargeGroupRep

▷ `MultiplyFreeZGWordWithGroupElt_LargeGroupRep(`*resolution, word, g*`)` (method)
▷ `MultiplyFreeZGWordWithGroupEltNC_LargeGroupRep(`*resolution, word, g*`)` (method)

Returns: free ZG word in large group representation

Multiplies the word *word* with the group element *g* and returns the result. The NC version does not check whether *g* is an element of the group of *resolution* and *word* can be a word.

B.4.9 GeneratorsOfModuleOfResolution_LargeGroupRep

▷ `GeneratorsOfModuleOfResolution_LargeGroupRep(`*resolution, term*`)` (method)

Returns: list of letters/words in large group representation

Returns a set of generators for the *term*th module of *resolution*. The returned value is a list of vectors of group ring elements.

B.4.10 BoundaryOfGenerator_LargeGroupRep

▷ `BoundaryOfGenerator_LargeGroupRep(`*resolution, term, n*`)` (method)
▷ `BoundaryOfGeneratorNC_LargeGroupRep(`*resolution, term, n*`)` (method)

Returns: free ZG word in the large group representation

Returns the boundary of the *n*th generator of the *term*th module of *resolution* as a word in the *n-1*st module (in large group representation). The NC version does not check whether there is a *term*th module and if it has at least *n* generators.

B.4.11 BoundaryOfFreeZGLetterNC_LargeGroupRep

▷ `BoundaryOfFreeZGLetterNC_LargeGroupRep(`*resolution, term, letter*`)` (method)
▷ `BoundaryOfFreeZGLetter_LargeGroupRep(`*resolution, term, letter*`)` (method)

Returns: free ZG word in large group representation

Calculates the boundary of the letter *letter* of the *term*th module of *resolution* in large group representation. The NC version does not check whether *letter* actually is a letter in the *term*th module.

B.4.12 BoundaryOfFreeZGWord_LargeGroupRep

▷ `BoundaryOfFreeZGWord_LargeGroupRep(`*resolution, term, word*`)` (method)

Returns: free ZG word in large group representation

Calculates the boundary of the element *word* of the *term*th module of *resolution* in large group representation. The NC version does not check whether *word* actually is a word in the *term*th module.

Appendix C

Contracting Homotopies

C.1 The `PartialContractingHomotopy` Data Type

A partial contracting homotopy is a component object that knows the values of a contracting homotopy on some subspace of a resolution. It has two mandatory components:

- `.resolution` a `HapResolution` on which the contraction is defined.
- `.knownPartOfHomotopy` a list of `Records` with components `.space` and `.map`.

Let `h` be a contracting homotopy. The lookup table `.knownPartOfHomotopy` has one entry for each term of the resolution `h.resolution` (that is, one more than `Length(h.resolution)`).

The i th element of `.knownPartOfHomotopy` contains a record with components `.space` and `.map` where `.space` is a `FreeZGWord` of the $i-1$ st term of the resolution. The component `.map` is a list of length `Dimension(h.resolution)(i-1)`. The entries of this list are pairs `[g,im]` where `g` represents a group element and `im` represents the image of the contraction. So the entry `[g,im]` in the `k`th component of the list `.map` means that the `k`th free generator of the corresponding module multiplied with the group element represented by `g` is mapped to `im` under the partial contracting homotopy. Note that the data type of `g` or `im` are not fixed at this level. They must be specified by the sub representations. Also, `im` need not represent the actual image under a contracting homotopy. It is possible to just store a bit of information that is then used to generate the actual image.

As this is a very general data type, it has very few methods.

C.1.1 ResolutionOfContractingHomotopy

▷ `ResolutionOfContractingHomotopy(`*`homotopy`*`)` (method)

Returns: A `HapResolution`

This returns the resolution of the homotopy *`homotopy`* (the component *`homotopy!.resolution`*).

C.1.2 PartialContractingHomotopyLookup

▷ `PartialContractingHomotopyLookup(`*`homotopy, term, generator, groupel`*`)` (method)
▷ `PartialContractingHomotopyLookupNC(`*`homotopy, term, generator, groupel`*`)` (method)

Returns: The entry `im` of the corresponding lookup table

Looks up the known part of the contracting homotopy *homotopy* and returns the corresponding image. More precisely, it returns the image of the *generator*th generator times the group element represented by *groupel* in term *term* under the partial homotopy. The data type of this image depends on the representation of *homotopy*.

term has to be an integer and *generator* a positive integer. *groupel* only has to be an Object.

The NC version does not do any checks on the input. The other version checks if *term* and *generator* are sensible. It does not check *groupel*.

References

[EGN] Bettina Eick, Franz Gahler, and Werner Nickel. cryst. https://www.gap-system.org/Packages/cryst.html. 4

[Ell] Graham Ellis. Hap. http://hamilton.nuigalway.ie/Hap/www/. 4

Index

action of crystallographic groups, 4

BasisChangeAffineMatOnRight, 7
BisectorInequalityFromPointPair, 8
BoundaryOfFreeZGLetter, 24
BoundaryOfFreeZGLetterNC, 26
BoundaryOfFreeZGLetterNC_LargeGroupRep, 29
BoundaryOfFreeZGLetter_LargeGroupRep, 29
BoundaryOfFreeZGWord, 25
BoundaryOfFreeZGWordNC, 27
BoundaryOfFreeZGWord_LargeGroupRep, 29
BoundaryOfGeneratorNC_LargeGroupRep, 28
BoundaryOfGenerator_LargeGroupRep, 28

ConvertLetterToStandardRep, 25
ConvertLetterToStandardRepNC, 25
ConvertStandardLetter, 25
ConvertStandardLetterNC, 25
ConvertStandardWord, 25
ConvertStandardWordNC, 25
ConvertWordToStandardRep, 25
ConvertWordToStandardRepNC, 25

DimensionSquareMat, 6

FaceLatticeAndBoundaryBieberbachGroup, 18
FundamentalDomainBieberbachGroup, 15
FundamentalDomainFromGeneralPointAndOrbitPartGeometric, 17
FundamentalDomainStandardSpaceGroup, 15

GeneratorsOfModuleOfResolution_LargeGroupRep, 28
GramianOfAverageScalarProductFromFiniteMatrixGroup, 8
GroupElementFromPosition, 24
GroupRingOfResolution, 27

ineqThreshold, 16
InfoHAPcryst, 5
installation, 5
IsFreeZGLetter, 26
IsFreeZGLetterNoTermCheck_LargeGroupRep, 27
IsFreeZGLetter_LargeGroupRep, 28
IsFreeZGWord, 26
IsFreeZGWordNoTermCheck_LargeGroupRep, 27
IsFreeZGWord_LargeGroupRep, 28
IsFundamentalDomainBieberbachGroup, 17
IsFundamentalDomainStandardSpaceGroup, 17
IsSquareMat, 6
IsValidGroupInt, 24

LinearPartOfAffineMatOnRight, 7

MultiplyFreeZGLetterWithGroupElt, 24
MultiplyFreeZGLetterWithGroupEltNC, 26
MultiplyFreeZGLetterWithGroupEltNC_LargeGroupRep, 28
MultiplyFreeZGLetterWithGroupElt_LargeGroupRep, 28
MultiplyFreeZGWordWithGroupElt, 24
MultiplyFreeZGWordWithGroupEltNC, 26
MultiplyFreeZGWordWithGroupEltNC_LargeGroupRep, 28
MultiplyFreeZGWordWithGroupElt_LargeGroupRep, 28
MultiplyGroupElts, 24
MultiplyGroupEltsNC, 26
MultiplyGroupEltsNC_LargeGroupRep, 27
MultiplyGroupElts_LargeGroupRep, 27

OrbitPartAndRepresentativesInFacesStandardSpaceGroup, 12
OrbitPartInFacesStandardSpaceGroup, 12

OrbitPartInVertexSetsStandardSpace-
Group, 11
OrbitStabilizerInUnitCubeOnRight, 10
OrbitStabilizerInUnitCubeOnRightOn-
Sets, 11

PartialContractingHomotopyLookup, 30
PartialContractingHomotopyLookupNC, 30
PointGroupRepresentatives, 9
polymake, 5
PositionInGroupOfResolution, 23
PositionInGroupOfResolutionNC, 23

RelativePositionPointAndPolygon, 9
RepresentativeActionOnRightOnSets, 13
ResolutionBieberbachGroup, 18
ResolutionFromFLandBoundary, 19
ResolutionOfContractingHomotopy, 30

ShiftedOrbitPart, 13
SignRat, 6
StabilizerOnSetsStandardSpaceGroup, 12
StrongestValidRepresentationForLetter,
23
StrongestValidRepresentationForWord, 23

TranslationOnRightFromVector, 7
TranslationsToBox, 13
TranslationsToOneCubeAroundCenter, 13

VectorModOne, 6

WhichSideOfHyperplane, 8
WhichSideOfHyperplaneNC, 8