Cryst

Computing with Crystallographic Groups

A GAP4 Package

Version 4.1.23

Bettina Eick

Franz Gähler

Werner Nickel

December 2019

Contents

1 Introduction

The Cryst package, previously known as CrystGAP, provides functions for the computation with affine crystallographic groups, in particular space groups. For the definition of the standard crystallographic notions we refer to the International Tables [Hah95], in particular the chapter by Wondratschek [Won95], and to the introductory chapter in [BBNWZ78]. The principal algorithms used in this package are described in [EGN97].

The present version for GAP 4 has been considerably reworked from an earlier version for GAP 3.4.4. Most of the porting to GAP 4 has been done by Franz Gähler. Besides affine crystallographic groups acting from the right, also affine crystallographic groups acting from the left are now fully supported. Many algorithms have been added, extended, or improved in other ways.

Our warmest thanks go the Max Neunhöffer, whose extensive testing of the GAP 4 version of Cryst in connection with XGAP uncovered several bugs and led to many performance improvements.

Cryst is implemented in the GAP 4 language, and runs on any system supporting GAP 4. However, certain commands may require that other GAP packages such as CaratInterface or XGAP are installed. In particular, the routines in Section 2.8 are likely to require CaratInterface, and the function WyckoffGraph (see 2.7.9) requires XGAP. Both CaratInterface and XGAP may be available only under Unix.

The Cryst package is loaded with the command

```
gap> LoadPackage( "cryst" );
true
```

Cryst has been developed by

Bettina Eick
Fachbereich Mathematik und Informatik
Technische Universität Braunschweig
Pockelsstr. 14, D-38106 Braunschweig, Germany
e-mail: b.eick@tu-bs.de

Franz Gähler
Fakultät für Mathematik, Universität Bielefeld
Postfach 10 01 31, D-33501 Bielefeld, Germany
e-mail: gaehler@math.uni-bielefeld.de

Werner Nickel
Fachbereich Mathematik, AG2, Technische Universität Darmstadt,
Schlossgartenstraße 7, D-64289 Darmstadt, Germany
e-mail: nickel@mathematik.tu-darmstadt.de

For bug reports, suggestions and comments please please use the issue tracker on GitHub:

https://github.com/gap-packages/Cryst/issues/

2 Affine crystallographic groups

An affine crystallographic group G is a subgroup of the group of all Euclidean motions of d-dimensional space, with the property that its subgroup T of all pure translations is a discrete normal subgroup of finite index. If the rank of the translation subgroup T is d, G is called a space group. The quotient G/T is called the point group of G.

In this package, affine crystallographic groups are represented as groups of augmented matrices of dimension $d+1$. Most functions assume a group of rational matrices, but some may also work with cyclotomic matrix groups. In particular, it is possible to compute the translation basis of an affine crystallographic group given in a cyclotomic representation, and to pass to a rational representation by conjugating with that basis. Further functionality for cyclotomic crystallographic groups is currently not guaranteed.

Augmented matrices can take one of two forms. Matrices of the form

```
[ M 0 ]
[ t 1 ]
```

act from the right on row vectors $(x, 1)$. Such a matrix is said to be an affine matrix acting on the right. Since in GAP all groups act from the right, this is the preferred representation of an affine transformation.

The second representation of affine transformations is by augmented matrices of the form

```
[ M t ]
[ 0 1 ]
```

which act from the left on column vectors $(x, 1)$. Such matrices are said to be affine matrices acting on the left. This is the representation usually adopted by crystallographers.

Cryst supports affine crystallographic groups in both representations. Every affine crystallographic group is constructed in one of these two representations.

Affine crystallographic groups in different representations should never be mixed, however. It is recommended to adopt one of the two representations, and then to stick to that decision. In order to facilitate this, there is a global variable `CrystGroupDefaultAction`, whose value is either `RightAction` or `LeftAction`. The initial value is `RightAction`, but this can be changed with

1 ▶ `SetCrystGroupDefaultAction(` *action* `)` F

where *action* must be either `RightAction` or `LeftAction`. Constructor functions without explicit representation qualifier then will construct an affine crystallographic group in the representation specified by `CrystGroupDefaultAction`.

2.1 Construction

1 ▶ `AffineCrystGroupOnRight( gens )` F
▶ `AffineCrystGroupOnRight( genlist )` F
▶ `AffineCrystGroupOnRight( genlist, identity )` F

returns the matrix group generated by *gens* or *genlist*, which must be affine matrices acting on the right, as affine crystallographic group acting on the right. An already existing group S of affine matrices acting on the right can be converted to an affine crystallographic group acting on the right with

2 ▶ `AsAffineCrystGroupOnRight( S )` F

The property

3 ▶ `IsAffineCrystGroupOnRight( S )` P

is `true` exactly for those groups which have been constructed in the above two ways.

4 ▶ `AffineCrystGroupOnLeft( gens )` F
▶ `AffineCrystGroupOnLeft( genlist )` F
▶ `AffineCrystGroupOnLeft( genlist, identity )` F

returns the matrix group generated by *gens* or *genlist*, which must be affine matrices acting on the left, as affine crystallographic group acting on the left. An already existing group S of affine matrices acting on the left can be converted to an affine crystallographic group acting on the left with

5 ▶ `AsAffineCrystGroupOnLeft( S )` F

The property

6 ▶ `IsAffineCrystGroupOnLeft( S )` P

is `true` exactly for those groups which have been constructed in the above two ways.

It is recommended to adopt one representation for affine crystallographic groups, and then to stick to it. To facilitate this, routines are provided which assume a default representation.

7 ▶ `AffineCrystGroup( gens )` F
▶ `AffineCrystGroup( genlist )` F
▶ `AffineCrystGroup( genlist, identity )` F

calls `AffineCrystGroupOnRight` or `AffineCrystGroupOnLeft` with the same arguments, depending on the value of `CrystGroupDefaultAction`.

8 ▶ `AsAffineCrystGroup( S )` F

calls `AsAffineCrystGroupOnRight` or `AsAffineCrystGroupOnLeft` with the same argument, depending on the value of `CrystGroupDefaultAction`.

9 ▶ `IsAffineCrystGroup( S )` F

calls `IsAffineCrystGroupOnRight` or `IsAffineCrystGroupOnLeft` with the same argument, depending on the value of `CrystGroupDefaultAction`.

10 ▶ `TransposedMatrixGroup( S )` A

returns the transpose of the affine crystallographic group S. If S is acting on the right, its transpose is acting on the left, and vice versa.

2.2 Point group

The point group P of an affine crystallographic group S is the quotient S/T, where T is the normal subgroup of all pure translations. P is isomorphic to the group generated by the linear parts of all affine matrices contained in S. In Cryst this latter group is identified with the point group of S.

1 ► `PointGroup( S )` A

returns the point group of S.

2 ► `IsPointGroup( P )` P

returns `true` if and only if P has been constructed as the point group of an affine crystallographic group S.

3 ► `AffineCrystGroupOfPointGroup( P )` A

returns the affine crystallographic group S, from which P has been constructed.

4 ► `PointHomomorphism( S )` A

returns a homomorphism from the affine crystallographic group to its point group.

5 ► `IsPointHomomorphism( H )` P

returns `true` if and only if H has been constructed as the `PointHomomorphism` of an affine crystallographic group.

2.3 Translation lattice

The vectors by which the pure translations in an affine crystallographic group translate form a discrete lattice, L, called the translation lattice of S.

1 ► `TranslationBasis( S )` A

returns a basis of the translation lattice of S. The basis returned is unique for the translation lattice.

2 ► `InternalBasis( S )` A

returns a basis used internally for many computations. It consists of the translation basis B of S, extended by further standard basis vectors if B has not full rank.

If a generating set B of the translation lattice of S is known from somewhere, this knowledge can be added to S with

3 ► `AddTranslationBasis( S, B )` F

This function must do further work, so that `SetTranslationBasis` cannot be used for this purpose. If doubts arise about the correctness of the translation basis that has been added by hand, one can check the correctness of the stored value with

4 ► `CheckTranslationBasis( S )` F

An affine crystallographic group S acting on d-dimensional Euclidean space is called a **space group** if its translation lattice has rank d.

5 ► `IsSpaceGroup( S )` P

tests if the affine crystallographic group S is a space group.

Since many computations are done internally in the `InternalBasis` of S, we say that S is in standard form if the `InternalBasis` is the standard basis of Euclidean row space or column space, respectively. This means that the translation lattice is generated by the first k standard basis vectors, where k is the rank of the translation lattice.

6 ► `IsStandardAffineCrystGroup( S )` P

checks if S is in standard form.

7 ► `IsStandardSpaceGroup( S )` P

checks if *S* is a space group in standard form.

8 ► `StandardAffineCrystGroup( S )` F

returns a conjugate of *S* which is in standard form.

If an space group is a semi-direct product of its point group with its translation subgroup, *S* is said to be symmorphic.

9 ► `IsSymmorphicSpaceGroup( S )` P

checks if the space group *S* is symmorphic.

2.4 Special methods

In the representation by augmented matrices, affine crystallographic groups are infinite matrix groups. Their infinity is relatively trivial in the sense that they have an abelian normal subgroup of finite index. Nevertheless, for many operations special methods have to be installed that avoid to attempt algorithms that never finish. These methods all make essential use of the exactness of the sequence of homomorphism `0 -> T -> S -> P -> 1`, where *T* is the translation subgroup of *S*, and *P* its point group.

All operations for general groups that make sense for affine crystallographic groups should work also in that case. In particular, there should be no restrictions for finite `AffineCrystGroups`. For infinite groups, some restrictions apply, however. For instance, algorithms from the orbit-stabilizer family can work only if the orbits generated are finite. Note, however, that `Normalizer`, `Centralizer` and `RepresentativeAction` in an `AffineCrystGroup` work even if the corresponding orbit is infinite.

Some methods installed for affine crystallographic groups have a special behavior.

1 ► `\^( S, conj )`

If *S* is an `AffineCrystGroupOnRight`, the group *conj * S * conj^-1* is returned. *conj* must be an affine matrix acting on the right. If *S* is an `AffineCrystGroupOnLeft`, the group *conj^-1 * S * conj* is returned. *conj* must be an affine matrix acting on the left.

2 ► `IsomorphismFpGroup( P )` A

returns an isomorphism from the `PointGroup` *P* to an isomorphic `FpGroup` *F*. If *P* is solvable, *F* is given in a power-commutator presentation.

3 ► `IsomorphismFpGroup( S )` A

returns an isomorphism from the `AffineCrystGroup` *S* to an isomorphic `FpGroup` *F*. If *S* is solvable, *F* is given in a power-commutator presentation. The presentation of *F* is an extension of the presentation of the point group *P* of *S* used in `IsomorphismFpGroup( P )`.

If the package polycyclic is installed, Cryst automatically loads it, and then provides special methods for `IsomorphismPcpGroup`.

4 ► `IsomorphismPcpGroup( P )` A

with *P* a solvable `PointGroup`, returns an isomorphism from *P* to an isomorphic `PcpGroup` *pcp*. For details about `PcpGroups`, we refer to the documentation of the package polycyclic.

5 ► `IsomorphismPcpGroup( S )` A

with *S* a solvable `AffineCrystGroup` (i.e., one with a solvable `PointGroup`), returns an isomorphism from *S* to an isomorphic `PcpGroup` *pcp*. The presentation of *pcp* is an extension of the presentation of the point group *P* of *S* used in `IsomorphismPcpGroup( P )`.

2.5 Maximal subgroups

Since an `AffineCrystGroup` has infinitely many maximal subgroups in general, in the computation of maximal subgroups it must be further specified which maximal subgroups are desired. Recall that a maximal subgroup of an `AffineCrystGroup` is either latticeequal or classequal. A latticeequal subgroup has the same translation lattice as the parent, while a classequal subgroup has the same point group as the parent. In the classequal case a maximal subgroup always has prime-power index, whereas in the latticeequal case this is so only in dimensions up to 3.

1 ▶ `MaximalSubgroupClassReps( S, flags )` O

returns a list of conjugacy class representatives of maximal subgroups of the `AffineCrystGroup` *S*.

2 ▶ `ConjugacyClassesMaximalSubgroups( S, flags )` O

returns a list of conjugacy classes of maximal subgroups of the `AffineCrystGroup` *S*.

In these two functions, the argument *flags* specifies which maximal subgroups are computed. *flags* is a record which may have the following components:

`flags.primes := [p1 .. pr]`
: only maximal subgroups of p-power index for the given primes p are computed

`flags.latticeequal := true`
: only latticeequal maximal subgroups are computed

`flags.classequal := true`
: only classequal maximal subgroups are computed

`flags.latticeequal` and `flags.classequal` must not both be bound and `true`. `flags.primes` may be omitted only if `flags.latticeequal` is bound and `true`.

```
gap> S := SpaceGroupIT(3,222);
SpaceGroupOnRightIT(3,222,'2')
gap> L := MaximalSubgroupClassReps( S, rec( primes := [3,5] ) );
[ <matrix group with 7 generators>, <matrix group with 8 generators>,
  <matrix group with 8 generators> ]
gap> List( L, IndexInParent );
[ 3, 27, 125 ]
gap> L := MaximalSubgroupClassReps( S,
>                  rec( classequal := true, primes := [3,5] ) );
[ <matrix group with 8 generators>, <matrix group with 8 generators> ]
gap> List( L, IndexInParent );
[ 27, 125 ]
gap> L := MaximalSubgroupClassReps( S,
>                  rec( latticeequal := true, primes := [3,5] ) );
[ <matrix group with 7 generators> ]
gap> List( L, IndexInParent );
[ 3 ]
gap> L := MaximalSubgroupClassReps( S, rec( latticeequal := true ) );
[ <matrix group with 7 generators>, <matrix group with 7 generators>,
  <matrix group with 7 generators>, <matrix group with 7 generators>,
  <matrix group with 6 generators> ]
gap> List( L, IndexInParent );
[ 2, 2, 2, 3, 4 ]
```

2.6 Space groups with a given point group

1 ▶ `SpaceGroupsByPointGroupOnRight( P )` O
▶ `SpaceGroupsByPointGroupOnRight( P, norm )` O
▶ `SpaceGroupsByPointGroupOnRight( P, norm, orbsflag )` O

where P is any finite subgroup of $GL(d,\mathbb{Z})$, returns a list of all space groups (acting on the right) with point group P, up to conjugacy in the full translation group of Euclidean space. All these space groups are returned as `AffineCrystGroupOnRight` in standard representation. If a second argument is present, which must be a list of elements of the normalizer of P in $GL(d,\mathbb{Z})$, only space groups inequivalent under conjugation with these elements are returned. If these normalizer elements, together with P, generate the full normalizer of P in $GL(d,\mathbb{Z})$, then exactly one representative of each space group type is obtained. If the third argument *orbsflag*, which must be `false` or `true`, is also present and `true`, all space groups up to conjugacy in the full translation group are returned, but these space groups are collected into orbits under the conjugation action with elements from *norm*.

```
gap> P := Group([ [ [ -1, 0 ], [ 0, -1 ] ], [ [ -1, 0 ], [ 0, 1 ] ] ]);
Group([ [ [ -1, 0 ], [ 0, -1 ] ], [ [ -1, 0 ], [ 0, 1 ] ] ])
gap> norm := GeneratorsOfGroup( NormalizerInGLnZ( P ) );
[ [ [ -1, 0 ], [ 0, -1 ] ], [ [ -1, 0 ], [ 0, 1 ] ], [ [ -1, 0 ], [ 0, -1 ] ],
  [ [ 1, 0 ], [ 0, -1 ] ], [ [ 0, 1 ], [ 1, 0 ] ] ]
gap> SpaceGroupsByPointGroupOnRight( P );
[ <matrix group with 4 generators>, <matrix group with 4 generators>,
  <matrix group with 4 generators>, <matrix group with 4 generators> ]
gap> SpaceGroupsByPointGroupOnRight( P, norm );
[ <matrix group with 4 generators>, <matrix group with 4 generators>,
  <matrix group with 4 generators> ]
gap> SpaceGroupsByPointGroupOnRight( P, norm, true );
[ [ <matrix group with 4 generators> ],
  [ <matrix group with 4 generators>, <matrix group with 4 generators> ],
  [ <matrix group with 4 generators> ] ]
```

2 ▶ `SpaceGroupTypesByPointGroupOnRight( P )` O
▶ `SpaceGroupTypesByPointGroupOnRight( P, orbsflag )` O

returns a list of space group type representatives (acting on the right) of the point group P. As in the case of `SpaceGroupsByPointGroupOnRight`, if the boolean argument *orbsflag* is present and `true`, not only space group type representatives, but all space groups up to conjugacy in the full translation group are returned. These are then collected into lists of space groups of the same space group type.

```
gap> SpaceGroupTypesByPointGroupOnRight( P );
[ <matrix group with 4 generators>, <matrix group with 4 generators>,
  <matrix group with 4 generators> ]
gap> SpaceGroupTypesByPointGroupOnRight( P, true );
[ [ <matrix group with 4 generators> ],
  [ <matrix group with 4 generators>, <matrix group with 4 generators> ],
  [ <matrix group with 4 generators> ] ]
```

3 ▶ `SpaceGroupsByPointGroupOnLeft( P )` O
▶ `SpaceGroupsByPointGroupOnLeft( P, norm )` O
▶ `SpaceGroupsByPointGroupOnLeft( P, norm, orbsflag )` O

works the same way as `SpaceGroupsByPointGroupOnRight`, except that the space groups acting from the left are returned.

4 ▶ `SpaceGroupTypesByPointGroupOnLeft( P )` O
▶ `SpaceGroupTypesByPointGroupOnLeft( P, orbsflag )` O

works the same way as `SpaceGroupTypesByPointGroupOnRight`, except that the space groups acting from the left are returned.

5 ▶ `SpaceGroupsByPointGroup( P )` O
▶ `SpaceGroupsByPointGroup( P, norm )` O
▶ `SpaceGroupsByPointGroup( P, norm, orbsflag )` O

calls `SpaceGroupByPointGroupOnRight` or `SpaceGroupByPointGroupOnLeft` with the same arguments, depending on the value of `CrystGroupDefaultAction`.

6 ▶ `SpaceGroupTypesByPointGroupOnLeft( P )` O
▶ `SpaceGroupTypesByPointGroupOnLeft( P, orbsflag )` O

calls either `SpaceGroupTypesByPointGroupOnRight` or `SpaceGroupTypesByPointGroupOnLeft` with the same arguments, depending on the variable `CrystGroupDefaultAction`.

2.7 Wyckoff positions

A Wyckoff position of a space group S is an equivalence class of points in Euclidean space, having stabilizers which are conjugate subgroups of S. Apart from a subset of lower dimension, which contains points with even bigger stabilizers, a Wyckoff position consists of an S-orbit of some affine subspace A. In Cryst, a Wyckoff position W is specified by such a representative affine subspace.

1 ▶ `WyckoffPositions( S )` A

returns the list of Wyckoff positions of the space group S.

```
gap> S := SpaceGroupIT(2,14);
SpaceGroupOnRightIT(2,14,'1')
gap> W := WyckoffPositions(S);
[ < Wyckoff position, point group 3, translation := [ 0, 0 ],
    basis := [  ] >
    , < Wyckoff position, point group 3, translation := [ 2/3, 1/3 ],
    basis := [  ] >
    , < Wyckoff position, point group 3, translation := [ 1/3, 2/3 ],
    basis := [  ] >
    , < Wyckoff position, point group 2, translation := [ 0, 0 ],
    basis := [ [ 1, -1 ] ] >
    , < Wyckoff position, point group 1, translation := [ 0, 0 ],
    basis := [ [ 1, 0 ], [ 0, 1 ] ] >
     ]
```

In the previous example, S has three kinds of special points (the basis is empty), whose representatives all have a stabilizer with the same point group (with label 1), one kind of special line (the basis has length 1), and the general position.

2 ▶ `WyckoffPositionsByStabilizer( S, sub )` O

where S is a space group and *sub* a subgroup of the point group or a list of such subgroups, determines only the Wyckoff positions whose representatives have a stabilizer with a point group equal to the subgroup *sub* or contained in the list *sub*, respectively.

```
gap> sub := Group([ [ [ 0, -1 ], [ -1, 0 ] ] ]);
Group([ [ [ 0, -1 ], [ -1, 0 ] ] ])
gap> IsSubgroup( PointGroup( S ), sub );
true
gap> WyckoffPositionsByStabilizer( S, sub );
[ < Wyckoff position, point group 1, translation := [ 0, 0 ],
    basis := [ [ 1, -1 ] ] >
 ]
```

3 ► `IsWyckoffPosition( obj )` R

checks whether *obj* is a Wyckoff position.

```
gap> ForAll( W, IsWyckoffPosition );
true
```

4 ► `WyckoffBasis( W )` O

returns a basis of the representative affine subspace of the Wyckoff position *W*.

```
gap> WyckoffBasis( W[4] );
[ [ 1, -1 ] ]
```

5 ► `WyckoffTranslation( W )` O

returns a point of the representative affine subspace of the Wyckoff position *W*.

```
gap> WyckoffTranslation( W[3] );
[ 1/3, 2/3 ]
```

6 ► `WyckoffSpaceGroup( W )` O

returns the space group of which *W* is a Wyckoff position.

```
gap> WyckoffSpaceGroup( W[1] );
SpaceGroupOnRightIT(2,14,'1')
```

7 ► `WyckoffStabilizer( W )` O

returns the stabilizer of the (generic) points in the representative affine subspace of the Wyckoff position *W*. This stabilizer is a subgroup of the space group of *W*, and thus an `AffineCrystGroup`.

```
gap> stab := WyckoffStabilizer( W[4] );
Group([ [ [ 0, -1, 0 ], [ -1, 0, 0 ], [ 0, 0, 1 ] ] ])
gap> IsAffineCrystGroupOnRight( stab );
true
```

8 ► `WyckoffOrbit( W )` O

determines the orbit of the representative affine subspace *A* of the Wyckoff position *W* under the space group *S* of *W* (modulo lattice translations). The affine subspaces in this orbit are then converted into a list of Wyckoff positions, which is returned. The Wyckoff positions in this list are just different representations of *W*. Their `WyckoffBasis` and `WyckoffTranslation` are chosen such that the induced parametrizations of their representative subspaces are mapped onto each other under the space group operation.

```
gap> orb := WyckoffOrbit( W[4] );
[ < Wyckoff position, point group 2, translation := [ 0, 0 ],
    basis := [ [ 1, -1 ] ] >
    , < Wyckoff position, point group 2, translation := [ 0, 0 ],
    basis := [ [ 1, 2 ] ] >
    , < Wyckoff position, point group 2, translation := [ 0, 0 ],
    basis := [ [ -2, -1 ] ] >
     ]
gap> Set(orb);
[ < Wyckoff position, point group 2, translation := [ 0, 0 ],
    basis := [ [ 1, -1 ] ] >
     ]
```

9 ▶ `WyckoffGraph( W [, def ] )` O
▶ `WyckoffGraph( S [, def ] )` O

displays the incidence relations of a set of Wyckoff positions graphically. This function is available only under XGAP. In the first form, *W* is a list of Wyckoff positions, which must belong to the same space group. In the second form, *S* is a space group; in this case, the function is applied to the complete list of Wyckoff positions of *S*. In both forms, a second argument, *def*, is possible, which is a record with optional components `title`, `width` and `height`, specifying the title, width and height of the graphic sheet on which the graph will be displayed.

Each vertex of the graph represents a Wyckoff position. Vertices are arranged in horizontal layers, determined by the dimension *s* of the Wyckoff position and the size *s* of its stabilizer. For each layer, the list *[d, s]* is displayed at the right border of the graphic sheet. The vertical positions of the layers are ordered according to the dimension of the Wyckoff position (primary criterion, smaller dimension above) and the size of the stabilizer (secondary criterion, bigger stabilizer above). Two Wyckoff positions are connected if the closure of the lower one contains the upper one. Two Wyckoff positions are connected by a line only if there is no Wyckoff position in between. The connection line is labelled with the number of affine subspaces contained in the lower Wyckoff position that contain a fixed representative affine subspace of the upper Wyckoff position. For instance, if the lower Wyckoff position consists of a space group orbit of lines (and thus the upper one of an orbit of points), the label of the connection line is the number of lines in the orbit which cross a fixed representative point of the upper Wyckoff position.

The initial layout of the graph is not always optimal. In particular, several connection lines can be drawn on top of each other, so that it is not easy to see who is connected with whom. With the left mouse button, the graph can be rearranged, however. Just drag each vertex to a more suitable place. Note, however, that a vertex can not leave its layer. For more details, please consult the XGAP manual.

By right-clicking on a vertex, a popup menu with information on the Wyckoff position of that vertex appears. It informs on the size of the `WyckoffStabilizer`, the dimension of the Wyckoff position, the length of the `WyckoffOrbit` (modulo lattice translations), the translation and basis of a representative affine subspace, the isomorphims type of the `WyckoffStabilizer`, and the ConjugacyClassInfo of the point group *P* of the `WyckoffStabilizer`. The ConjugacyClassInfo lists for each conjugacy class of elements of *P* the number of that class, the order, trace and determinant of its elements, and the size of the class. This information is useful to identify the geometric operation of the stabilizer. The isomorphism type and ConjugacyClassInfo may not be displayed initially. It this case, they can be obtained by left-clicking on them, or by left-clicking on the button labelled *all*. Unfortunately, the popup window cannot be resized automatically, and since the ConjugacyClassInfo needs several lines for the display, the information may be hidden behind the border of the window. You will have to use the slider of the popup window to make it visible, or resize the window with the help of your window manager. Alternatively, you can right-click again on the same vertex, in which case a new popup window of sufficient size appears.

2.8 Normalizers

At present, most of the functions in this section require that the GAP package CaratInterface is installed (and compiled). Otherwise, they are available only for space groups from the crystallographic groups catalogue or the International Tables (section 2.11).

1 ► `NormalizerPointGroupInGLnZ( P )` A

returns the normalizer of the `PointGroup` *P* in the group of all unimodular transformations of the lattice spanned by the `InternalBasis` *B* of the `AffineCrystGroup` *S* of *P*. If *S* is in standard representation, this is the same as `Normalizer( GL(dim,Integers), P )`, otherwise it is `Normalizer( GL(dim,Integers), P^(B^-1) )^B`. This notion probably makes sense only if *S* is a space group. Note that *P* must have elements with integer entries (which is the case if *S* is a space group).

2 ► `CentralizerPointGroupInGLnZ( P )` A

returns the centralizer of the `PointGroup` *P* in the group of all unimodular transformations of the lattice spanned by the `InternalBasis` *B* of the `AffineCrystGroup` *S* of *P*. If *S* is in standard representation, this is the same as `Centralizer( GL(dim,Integers), P )`, otherwise it is `Centralizer( GL(dim,Integers), P^(B^-1) )^B`. This notion probably makes sense only if *S* is a space group. Note that *P* must have elements with integer entries (which is the case if *S* is a space group).

3 ► `TranslationNormalizer( S )` F

returns the normalizer of the space group *S* in the full translation group. At present, this function is implemented only for space groups, not for general `AffineCrystGroups`. The translation normalizer *TN* of *S* may contain a continuous subgroup *C*. A basis of the space of such continuous translations is bound in `TN!.continuousTranslations`. Since this subgroup is not finitely generated, it is **not** contained in the group generated by `GeneratorsOfGroup( TN )`. Properly speaking, the translation normalizer is the span of *TN* and *C* together.

4 ► `AffineNormalizer( S )` F

returns the affine normalizer of the space group *S*. The affine normalizer *AF* contains the translation normalizer as a subgroup. Similarly as with `TranslationNormalizer`, the subgroup *C* of continuous translations, which is not finitely generated, is not part of the group that is returned. However, a basis of the space of continuous translations is bound in the component `AF!.continuousTranslations`.

5 ► `AffineInequivalentSubgroups( S, sub )` F

takes as input a space group *S* and list of subgroups of *S*, and returns a sublist of affine inequivalent subgroups. Note that the affine normalizer of *S* must be discrete in the current implementation. If it is not, `fail` is returned.

For two space groups *S1* and *S2* of the same dimension (and acting from the same side),

6 ► `ConjugatorSpaceGroups( S1, S2 )` F

returns an affine matrix *m* such that `S1^m = S2`, of `fail` if no such matrix exists, i.e., if the two space groups are not equivalent. This function requires that the GAP package CaratInterface is installed (and compiled).

2.9 Color groups

A color group *C* is a group whose elements are colored in the following way. The elements having the same color as the identity element `One(C)` form a subgroup *H* of finite index *n*. *H* is called the `ColorSubgroup` of *C*. Elements of *C* have the same color if and only if they are in the same right coset of *H* in *C*. The labelling of the colors, which runs from 1 to *n*, is determined by a fixed labelling of the right cosets of *H*. The list of right cosets of *H* is stored in the attribute `ColorCosetList`. The color of the elements of a coset corresponds to the position of the coset in that list. Elements of *H* by definition have color 1, i.e., the coset with representative `One(C)` is always the first element of

the ColorCosetList of C. Color groups which have a parent inherit their coloring from that parent, including the labelling of the colors. As with other groups, color groups having no parent are their own parent.

Right multiplication by a fixed element g of C induces a permutation $p(g)$ of the colors of the parent of C. This defines a natural homomorphism of C into the symmetric group of degree n. The image of this homomorphism is called the ColorPermGroup of C, and the homomorphism to it is called the ColorHomomorphism of C.

1 ▶ ColorGroup(G, H) F

constructs a colored copy of G, with color subgroup H (which should have finite index in G). Color groups constructed in this way are always their own parent. It is not possible to set their parent attribute to a different value.

Groups which may be colored include, in particular, AffineCrystGroups, but coloring of any finite group should work as well.

2 ▶ IsColorGroup(G) P

checks whether G is a color group.

3 ▶ ColorSubgroup(G) A

returns the color subgroup of G.

4 ▶ ColorCosetList(G) A

returns the color labelling cosets of G.

5 ▶ ColorOfElement(G, *elem*) F

returns the color of an element of G.

6 ▶ ColorPermGroup(G) A

returns the ColorPermGroup of G, which is the permutation group induced by G acting on the colors of the parent of G.

7 ▶ ColorHomomorphism(G) A

returns the homomomorphism from G to its ColorPermGroup.

8 ▶ Subgroup(C, *elems*) O

where C is a color group, returns the colored subgroup U of C generated by *elems*. The parent of U is set to the parent of C, from which the coloring of U is inherited.

```
gap> G := Group(  (1,2,3), (2,3,4) );
Group([ (1,2,3), (2,3,4) ])
gap> H := Group( (1,2,3) );
Group([ (1,2,3) ])
gap> C := ColorGroup( G, H );
Group([ (1,2,3), (2,3,4) ])
gap> ColorSubgroup( C ) = H;
true
gap> ColorCosetList( C );
[ RightCoset(Group( [ (1,2,3) ] ),()), RightCoset(Group( [ (1,2,3) ] ),(1,2)
    (3,4)), RightCoset(Group( [ (1,2,3) ] ),(1,3)(2,4)),
  RightCoset(Group( [ (1,2,3) ] ),(1,4)(2,3)) ]
gap> List( last, x -> ColorOfElement( C, Representative(x) ) );
[ 1, 2, 3, 4 ]
gap> U := Subgroup( C, [(1,3)(2,4)] );
Group([ (1,3)(2,4) ])
```

```
gap> IsColorGroup( U );
true
gap> ColorSubgroup( U );
Group(())
gap> ColorCosetList( U );
[ RightCoset(Group( () ),()), RightCoset(Group( () ),(1,3)(2,4)) ]
gap> List( last, x -> ColorOfElement( U, Representative(x) ) );
[ 1, 3 ]
```

2.10 Colored AffineCrystGroups

If C is a colored `AffineCrystGroup` whose `ColorSubgroup` is lattice-equal (translationengleich) with C, then the `PointGroup` of C can consistently be colored. In that case,

1 ▶ `PointGroup( C )` A

returns a colored point group. Otherwise, the `PointGroup` of C is an ordinary, uncolored group.

```
gap> S := SpaceGroupIT( 2, 10 );
SpaceGroupOnRightIT(2,10,'1')
gap> m := MaximalSubgroupClassReps( S, rec( primes := [2] ) );
[ <matrix group with 4 generators>, <matrix group with 3 generators>,
  <matrix group with 4 generators> ]
gap> List( last, x -> TranslationBasis(x) = TranslationBasis(S) );
[ false, true, false ]
gap> C := ColorGroup( S, m[1] );; IsColorGroup( PointGroup( C ) );
false
gap> C := ColorGroup( S, m[2] );; IsColorGroup( PointGroup( C ) );
true
```

Two colorings of a **space group** S are **equivalent** if the two `ColorSubgroups` are conjugate in the affine normalizer of S. For instance, a list of inequivalent index-2 `ColorSubgroups` of S can be obtained with the following code:

```
gap> sub := MaximalSubgroupClassReps( S, rec( primes := [2] ) );
[ <matrix group with 4 generators>, <matrix group with 3 generators>,
  <matrix group with 4 generators> ]
gap> List( sub, Size );
[ infinity, infinity, infinity ]
gap> sub := Filtered( sub, s -> IndexInParent( s ) = 2 );
[ <matrix group of size infinity with 4 generators>,
  <matrix group of size infinity with 3 generators>,
  <matrix group of size infinity with 4 generators> ]
gap> Length( AffineInequivalentSubgroups( S, sub ) );
2
```

Note that `AffineInequivalentSubgroups` requires the GAP package CaratInterface to be installed. Otherwise, this function is supported only for `AffineCrystGroups` constructed from the crystallographic groups catalog.

2.11 International Tables

For the user's convenience, a table with the 17 plane groups and the 230 space groups is included in Cryst. These groups are given in exactly the same settings (i.e., choices of basis and origin) as in the International Tables. Space groups with a centered lattice are therefore given in the non-primitive basis crystallographers are used to. This is in contrast to the crystallographic groups catalogue, where always a primitive basis is used.

For some of the 3D space groups, two different settings are available. The possible settings are labelled with the characters `'1'`, `'2'`,`'b'`,`'c'`,`'h'` and `'r'`. If only one setting is available, it is labelled `'1'`. For some space groups there exists a point with higher symmetry than the origin of the `'1'` setting. In such cases, a second setting `'2'` is available, which has this high symmetry point as origin. This second setting `'2'` then is the default setting. Space groups which have a unique axis can have this axis in *b* direction (setting`'b'`) or *c* direction (setting `'c'`). `'b'` is the default setting. Rhombohedral space groups are given in a hexagonal basis (setting `'h'`) and in a rhombohedral basis (setting `'r'`). `'h'` is the default setting.

1 ▶ `SpaceGroupSettingsIT( dim, nr )` F

returns a string, whose characters label the available settings of the space group with number *nr* and dimension *dim*.

2 ▶ `SpaceGroupOnRightIT( dim, nr )` F
▶ `SpaceGroupOnRightIT( dim, nr, setting )` F

returns space group number *nr* in dimension *dim* in the representation acting on the right. In the third argument, the desired setting can be specified. Otherwise, the space group is returned in the default setting for that space group.

3 ▶ `SpaceGroupOnLeftIT( dim, nr )` F
▶ `SpaceGroupOnLeftIT( dim, nr, setting )` F

returns space group number *nr* in dimension *dim* in the representation acting on the left. In the third argument, the desired setting can be specified. Otherwise, the space group is returned in the default setting for that space group.

4 ▶ `SpaceGroupIT( dim, nr )` F
▶ `SpaceGroupIT( dim, nr, setting )` F

returns either `SpaceGroupOnRightIT` or `SpaceGroupOnLeftIT` with the same arguments, depending on the value of `CrystGroupDefaultAction`.

```
gap> SpaceGroupSettingsIT( 3, 146 );
"hr"
gap> SpaceGroupOnRightIT( 3, 146 );
SpaceGroupOnRightIT(3,146,'h')
gap> SpaceGroupOnRightIT( 3, 146, 'r' );
SpaceGroupOnRightIT(3,146,'r')
```

Bibliography

[BBNWZ78] H. Brown, R. Bülow, J. Neubüser, H. Wondratschek, and H. Zassenhaus. *Crystallographic Groups of Four-Dimensional Space*. John Wiley, New York, 1978.

[EGN97] F. Gähler, B. Eick and W. Nickel. Computing maximal subgroups and wyckoff positions of space groups. *Acta Cryst A* 53, 467–474 (1997).

[Hah95] T. Hahn, editor. *International Tables for Crystallography, Volume A, Space-group Symmetry, 4th Edition*. Kluwer, Dordrecht, 1995.

[Won95] H. Wondratschek. Introduction to space-group symmetry. In *Internation Tables for Crystallography, Vol. A* [Hah95], pages 711–735.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "PermutationCharacter" comes before "permutation group".

T

W