

images

Minimal and Canonical images

1.4.0

12 August 2026

Christopher Jefferson

Markus Pfeiffer

Rebecca Waldecker

Eliza Jonauskyte

Christopher Jefferson Email: caj21@st-andrews.ac.uk

Homepage: <https://heather.cafe/>

Markus Pfeiffer Email: markus.pfeiffer@morphism.de

Homepage: <http://www.morphism.de/~markusp/>

Rebecca Waldecker Email: rebecca.waldecker@mathematik.uni-halle.de

Homepage: <http://conway1.mathematik.uni-halle.de/~waldecker/>

Eliza Jonauskyte Email: ej31@st-andrews.ac.uk

Copyright

© 2013–2026

Contents

1	The Images Package	4
2	Minimal and Canonical Images	5
2.1	Supported objects and actions	6
2.2	Function documentation	7
3	Fundamental and Combinatorial Structures and Associated Graph Algorithms	12
3.1	Worked examples	12
3.2	Function documentation	13
4	Installing and Loading the Images Package	19
4.1	Loading the Images Package	19
	References	20
	Index	21

Chapter 1

The Images Package

The GAP package `Images` implements efficient algorithms to calculate minimal images and canonical images of several different types of objects, including sets and sets of sets, in permutation groups.

For the mathematical background, see [\[JWW23\]](#).

If you want help on applying `Images` to your problem, please contact the authors, who will be happy to look into if your problem can be solved with `Images`.

Chapter 2

Minimal and Canonical Images

Given a group G and action A , the minimal image of an object O is the smallest image of O under any element of G , under the action A .

As a more concrete example, let us consider the minimal image of the set $[2,3,5,7]$ under a group G .

We can calculate all the images of our set under G , then choose the smallest one.

Example

```
gap> G := Group((1,2,3)(4,5,6)(7,8,9), (1,4,7)(2,5,8)(3,6,9));;
gap> List(G, g -> OnSets([2,3,5,7], g) );
[ [ 2, 3, 5, 7 ], [ 1, 2, 4, 9 ], [ 1, 3, 6, 8 ], [ 2, 4, 8, 9 ],
  [ 1, 6, 7, 8 ], [ 3, 5, 7, 9 ], [ 1, 5, 6, 8 ], [ 3, 4, 5, 7 ],
  [ 2, 4, 6, 9 ] ]
gap> Minimum(List(G, g -> OnSets([2,3,5,7], g) ) );
[ 1, 2, 4, 9 ]
```

This is very inefficient, as it requires enumerating all members of G . The images package produces a function `MinimalImage` (2.2.1), which performs this same operation more efficiently.

Example

```
gap> LoadPackage("images", false);
true
gap> MinimalImage(G, [2,3,5,7], OnSets);
[ 1, 2, 4, 9 ]
```

The most common use of `MinimalImage` is to categorise objects into equivalence classes. This next example shows $[2,3,5,7]$ and $[1,6,7,8]$ are in the same orbit, while $[3,5,7,8]$ is in a different orbit.

Example

```
gap> MinimalImage(G, [2,3,5,7], OnSets);
[ 1, 2, 4, 9 ]
gap> MinimalImage(G, [1,6,7,8], OnSets);
[ 1, 2, 4, 9 ]
gap> MinimalImage(G, [3,5,7,8], OnSets);
[ 1, 2, 6, 8 ]
```

The result configuration option changes what is returned: the image itself (`GetImage`, the default), a permutation performing the mapping (`GetPerm`), or just whether the object already is its own image (`GetBool`, which is often much faster to compute).

Example

```
gap> p := MinimalImage(G, [2,3,5,7], OnSets, rec(result := GetPerm));
(1,3,2)(4,6,5)(7,9,8)
gap> OnSets([2,3,5,7], p);
[ 1, 2, 4, 9 ]
gap> MinimalImage(G, [2,3,5,7], OnSets, rec(result := GetBool));
false
```

In this situation, we do not really need the minimal image, just a method of telling if two sets are in the same equivalence class.

Motivated by this, this package provides `CanonicalImage` (2.2.5). `CanonicalImage(G,O,A)` returns some image of O by an element of G under the action A , guaranteeing that if two objects O_1 and O_2 are in the same orbit of G then `CanonicalImage(G,O1,A) = CanonicalImage(G,O2,A)`. However, the canonical image is not "minimal" under any sensible ordering. The advantage of `CanonicalImage` is that it is much faster than `MinimalImage`, often by orders of magnitude.

WARNING: The value of `MinimalImage` will remain identical between versions of GAP and the `Images` package, unless bugs are discovered. This is *not* true for `CanonicalImage`.

2.1 Supported objects and actions

`MinimalImage` (2.2.1), `CanonicalImage` (2.2.5) and their variants accept the following combinations of object and action. The action argument defaults to `OnPoints` in every case, which for lists is not a supported action, so for lists the action should always be given.

- a set of positive integers, with `OnSets`;
- a list of positive integers, with `OnTuples`;
- a set of sets of positive integers, with `OnSetsSets`;
- a list of sets of positive integers, with `OnTuplesSets`;
- a positive integer, with `OnPoints`;
- a permutation, transformation or partial permutation, with `OnPoints` (that is, up to conjugacy);
- a digraph (from the `Digraphs` package), with `OnDigraphs` (the group must move only vertices of the digraph, and multidigraphs are not supported);
- a multiplication (Cayley) table, with `OnMultiplicationTables` (2.2.2);
- a fundamental structure (see Chapter 3), with `OnFundamental`.

For digraphs the order minimised is the sorted list of arcs, compared lexicographically, so the minimal image is the relabelling whose arc list `Set(DigraphEdges(D))` is smallest. This is neither the ordering of $<$ on digraph objects nor the lexicographic order on adjacency matrices read row by row as 0/1 strings. An undirected graph is represented by a symmetric digraph and needs no separate treatment.

Not every configuration option applies to every combination: the `stabilizer` and `getStab` options only affect sets, transformations, permutations, partial permutations, digraphs and multiplication tables, and the `order` option is ignored for `OnSetsSets`. See `ImagesAdvancedConfig` (2.2.6) for the details.

2.2 Function documentation

2.2.1 MinimalImage

- ▷ `MinimalImage(G, pnt[, act][, Config])` (function)
- ▷ `IsMinimalImage(G, pnt[, act][, Config])` (function)
- ▷ `MinimalImagePerm(G, pnt[, act][, Config])` (function)

`MinimalImage` returns the minimal image of *pnt* under the group *G*. `IsMinimalImage` returns a boolean which is true if `MinimalImage` would return *pnt* (so the value is its own minimal image).

`MinimalImagePerm` returns a permutation in *G* which maps *pnt* to its minimal image.

The supported combinations of *pnt* and *act* are listed in Section 2.1. The option *Config* defines a number of advanced configuration options, which are described in `ImagesAdvancedConfig` (2.2.6). Note that passing an order option changes which image these functions compute: with any order other than `CanonicalConfig_Minimum` they behave like `CanonicalImage` (2.2.5) and the result need not be minimal.

2.2.2 OnMultiplicationTables

- ▷ `OnMultiplicationTables(table, g)` (function)

The action of a permutation *g* on a multiplication (Cayley) table: the table of the isomorphic structure on the relabelled elements, so `OnMultiplicationTables(T, g)[i^g][j^g] = T[i][j]^g`. A table is a list of *n* rows of length *n* with entries in $[1..n]$, as produced by `MultiplicationTable`; two magmas are isomorphic precisely when their tables lie in the same orbit under `SymmetricGroup(n)`.

`MinimalImage` (2.2.1) with this action returns the lexicographically least table in the orbit (comparing tables row by row, which is GAP's ordering of the tables as lists), so it is a distinguished representative of the isomorphism class of the magma.

Example

```
gap> T := MultiplicationTable(CyclicGroup(4));;
gap> MinimalImage(SymmetricGroup(4), T, OnMultiplicationTables);
[ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 2, 1 ], [ 4, 3, 1, 2 ] ]
```

2.2.3 IsMinimalImageLessThan

- ▷ `IsMinimalImageLessThan(G, A, B, act)` (function)

`IsMinimalImageLessThan` checks if the minimal image of *A* under the group *G* is smaller than *B*.

It returns `MinImage.Smaller`, `MinImage.Equal` or `MinImage.Larger`, if the minimal image of *A* is smaller, equal or larger than *B*.

A and *B* must be sets of the same size, and *act* must be `OnSets`; no other actions are currently supported, and this function accepts no configuration record.

2.2.4 MinimalImageOrderedPair

- ▷ `MinimalImageOrderedPair(G, pair[], act)` (operation)
- ▷ `MinimalImageUnorderedPair(G, pair[], act)` (operation)

`MinimalImageOrderedPair` returns the lexicographically smallest pair `[A,B]` such that some single `g` in `G` maps `pair[1]` to `A` and `pair[2]` to `B` under `act`. `MinimalImageUnorderedPair` instead minimises over both orderings of the pair, so exchanging the two entries of `pair` does not change the result. The default action is `OnPoints`.

2.2.5 CanonicalImage

- ▷ `CanonicalImage(G, pnt[], act] [, Config])` (function)
- ▷ `IsCanonicalImage(G, pnt[], act] [, Config])` (function)
- ▷ `CanonicalImagePerm(G, pnt[], act] [, Config])` (function)

`CanonicalImage` returns a canonical image of `pnt` under the group `G`. `IsCanonicalImage` returns a boolean which is true if `CanonicalImage` would return `pnt` (so the value is its own canonical image).

`CanonicalImagePerm` returns a permutation in `G` which maps `pnt` to its canonical image.

By default, these functions use `CanonicalConfig_Fast`, an alias for the fastest known ordering (currently `CanonicalConfig_RareRatioOrbitFixPlusMin`), which may change in new versions of the package. The supported combinations of `pnt` and `act` are listed in Section 2.1. The option `Config` defines a number of advanced configuration options, which are described in `ImagesAdvancedConfig` (2.2.6). These include the ability to choose the canonicalising algorithm used.

2.2.6 ImagesAdvancedConfig

- ▷ `ImagesAdvancedConfig` (global variable)

This section describes the advanced configuration options for both `MinimalImage` (2.2.1) and `CanonicalImage` (2.2.5). Assume we have called `MinimalImage` (2.2.1) or `CanonicalImage` (2.2.5) with arguments `(G, O, A)`.

`order`

The search ordering used while building the image. The most useful values are:

`CanonicalConfig_Minimum`

Lexicographically smallest image -- same as using `MinimalImage`.

`CanonicalConfig_FixedMinOrbit`

Lexicographically smallest set under the ordering of the integers given by the `MinOrbitPerm` function. This ordering (and `CanonicalConfig_FixedMaxOrbit`) is not supported when canonicalising transformations, permutations or partial permutations, and will raise an error there.

`CanonicalConfig_Fast`

The current best algorithm, and the default for `CanonicalImage` (2.2.5). It is an alias, currently for `CanonicalConfig_RareRatioOrbitFixPlusMin`, and may change between versions of the package.

The full list of orderings, whose behaviour is described in the paper [JJPW19], is:

CanonicalConfig_Minimum,	CanonicalConfig_MinOrbit,
CanonicalConfig_MaxOrbit,	CanonicalConfig_SingleMaxOrbit,
CanonicalConfig_RareOrbit,	CanonicalConfig_CommonOrbit,
CanonicalConfig_RareRatioOrbit,	CanonicalConfig_CommonRatioOrbit,
CanonicalConfig_RareRatioOrbitFix,	CanonicalConfig_CommonRatioOrbitFix,
CanonicalConfig_RareRatioOrbitFixPlusMin,	CanonicalConfig_RareRatioOrbitFixPlusRare,
CanonicalConfig_RareRatioOrbitFixPlusCommon,	CanonicalConfig_RareOrbitPlusMin,
CanonicalConfig_RareOrbitPlusRare,	CanonicalConfig_RareOrbitPlusCommon,
CanonicalConfig_FixedMinOrbit,	CanonicalConfig_FixedMaxOrbit

and CanonicalConfig_Fast.

Note that these values are the value of the order component of the configuration record, as in `rec(order := CanonicalConfig_Fast)` -- passing one directly as the whole configuration record is an error. For the action `OnSetsSets` the ordering is currently ignored, and the minimal image is computed whatever order is given.

result

What to return: `GetImage` (the image, the default), `GetPerm` (a permutation in G mapping O to its image, as `MinimalImagePerm` (2.2.1) returns), or `GetBool` (true if O is its own image, as `IsMinimalImage` (2.2.1) returns, which is often much faster than computing the image).

stabilizer

The group `Stabilizer( $G, O, A$ )`, or a subgroup of this group; see `Stabilizer` (**Reference: Stabilizer**). If this group is large, it is more efficient to pre-calculate it. Default behaviour is to calculate the group, pass `Group()` to disable this behaviour. The generators of the given group are checked to stabilize O (which characterises being a subgroup of the stabilizer), and a group failing the check is rejected with an error, so accidentally reusing a stabilizer computed for a different object cannot silently produce wrong answers. The "vole" engine computes its own stabilizer and ignores this option.

When canonicalising transformations, permutations, partial permutations or digraphs, the default stabilizer is computed with the `ferret` package when it is loaded, which is much faster for groups of large degree; without `ferret` a slower fallback is used. For permutations the default is the centralizer, and for digraphs under the full symmetric group on their vertices it is the automorphism group of the digraph. For the minimum orderings (which `MinimalImage` (2.2.1) and its variants use), an object whose orbit under G is small is answered by enumerating the orbit directly, and then no stabilizer is needed at all.

This option is honoured for sets, transformations, permutations, partial permutations and digraphs. It is silently ignored for `OnTuples`, `OnTuplesSets`, points and fundamental structures, and for `OnSetsSets` only the trivial group is accepted.

Beware of passing `Group()` together with one of the dynamic orderings when the true stabilizer of O is very large: the search then rediscovers the stabilizer piecemeal, and can take many seconds on instances the default settings solve instantly. The same applies to `disableStabilizerCheck`.

Passing a stabilizer can also change *which* representative a non-minimum ordering selects. `MinimalImage` (2.2.1) and its variants are unaffected -- the minimum of an orbit is the minimum of that orbit whatever the search was told -- but the dynamic orderings prune and rank

using the stabilizer, so `CanonicalImage` (2.2.5) may return a different (equally valid) representative when a stabilizer is supplied than when it is computed. The same holds for the other options which change how the stabilizer is obtained or used, namely `disableStabilizerCheck` and `bruteForce`. (`getStab` is not among them: it only reports the stabilizer the computation arrived at anyway, and never changes the computation.) Each fixed choice of settings still gives a canonical form -- the answer is constant on the orbit -- so what matters is to use one setting throughout a computation, and not to compare canonical images produced under different ones. No attempt is made here to say which orderings are sensitive to this and which are not.

`disableStabilizerCheck` (default `false`)

By default, during search we perform cheap checks to try to find extra elements of the stabilizer. Pass true to disable this check, this will make the algorithm MUCH slower if the stabilizer argument is a subgroup.

`getStab` (default `false`)

Store the stabilizer calculated during the search in the `stab` component of the configuration record that was passed in. With the "vole" engine this is `Stabilizer( $G, O, A$ )`; with the native engine it is a subgroup stabilizing the returned image, and may be a proper subgroup. It is honoured on the same paths as `stabilizer`.

This reports a stabilizer only when a search was run to obtain one. A transformation, permutation, partial permutation or digraph whose orbit is short is answered by the enumeration pre-pass described under `bruteForce` below, which walks the orbit and computes no stabilizer at all; such a call deposits `fail` in `stab`. Pass `bruteForce := false` alongside `getStab` to insist on the search, and so on a stabilizer, at the price of the search's cost on an object the enumeration answers in microseconds. Test the component rather than assuming a group.

`bruteForce` (default `"auto"`)

Whether to try the orbit-enumeration pre-pass described under `stabilizer` above, which answers an object whose orbit under G is small without any stabilizer chain at all. The pre-pass applies to transformations, permutations, partial permutations and digraphs; on every other path this option has no effect.

Under the default `"auto"` the pre-pass enumerates up to a work budget balancing the measured cost of enumeration against the measured cost of the search, computed from the degree, the number of generators and the size of the encoded object, and gives up and runs the search when the orbit does not close within it. The budget consults nothing but those constants, so it does not depend on session state (in particular, whether a stabilizer chain for G has already been computed changes nothing). Passing `false` always runs the search. Passing `true` removes the budget: the orbit is enumerated however large it turns out to be, so pass it only when you know the orbit is small. The budget is a heuristic and is sometimes wrong in both directions, which is what these two overrides are for.

For the minimum orderings all three settings compute the same answer, because the pre-pass minimises exactly the order the search minimises; they differ only in how long they take. Note that `IsMinimalImage` (2.2.1) stops the enumeration at the first image smaller than O , so it can answer `false` even for orbits which would run past the budget.

Under a non-minimum ordering the pre-pass returns the minimum of the orbit, which is constant on the orbit and so is a canonical form, but is not the representative the search would have

selected. Both are valid; they are different. This is the same settings-dependence described under `stabilizer` above, and the same rule applies: use one setting throughout. Whether the pre-pass runs at all is decided from the orbit alone, so it cannot split a single orbit between the two.

search (default "bfs")

Which search strategy the native engine uses. The default "bfs" is the frontier search: it stores every partial image achieving the minimal prefix, which can exhaust memory on highly symmetric inputs (a cyclic group's multiplication table of order 12 exceeds 8GB). Passing "iterative" stores none of them, re-enumerating the realisations of the fixed prefix at every level: bounded memory, at the price of re-enumeration time. Passing "hybrid" runs the frontier search while each stored level fits under `frontierLimit` nodes, and switches to re-enumeration from the last stored frontier only when a level would exceed the cap, so it matches the default search's speed when memory suffices and degrades gracefully instead of running out of memory. All three produce identical results.

"iterative" and "hybrid" are experimental. They support only the minimum ordering on unblocked domains (which excludes sets of sets); any other ordering is rejected with an error.

frontierLimit (default chosen from the input)

The cap on the nodes stored per level by `search := "hybrid"`; the other searches ignore it. When it is not given, the cap is `Maximum(1000, QuoInt(50000000, m))` nodes for an object encoded on `m` points, which roughly bounds the stored list entries rather than the node count. After a hybrid run the global `_IMAGES_HYBRID_STATS` holds the widest frontier actually stored and the level at which the search switched to re-enumeration (fail if it never did), which is the information needed to tune the cap.

engine (default "native")

Which algorithm to use to compute the canonical image. The default "native" uses this package's own algorithm. Passing "vole" instead computes the canonical image using the `vole` package (via `VoleFind.Canonical`), which supports the same actions except for fundamental structures (Chapter 3). `vole` must already be loaded; an error is raised if it is requested but not available. Note that `vole` produces a different (but equally valid) canonical representative, so the two engines must not be mixed for a given computation. The "vole" engine only applies to `CanonicalImage` (2.2.5); it cannot compute minimal images and will raise an error if requested to.

Chapter 3

Fundamental and Combinatorial Structures and Associated Graph Algorithms

This chapter details the `Fundamental` and `Combinatorial` records, which are used for creating and manipulating structured data representations within GAP. It also covers functions for converting these structures into graphs and for computing their stabilizers and canonical forms.

Fundamental structures can be passed directly to `CanonicalImage` (2.2.5), `CanonicalImagePerm` (2.2.5) and `IsCanonicalImage` (2.2.5), with the action `OnFundamental` (which is also the default action for them). NOTE: unless the group is a direct product of natural symmetric groups (such as the full symmetric group on the atoms), canonicalising a fundamental structure requires the optional `vole` package to be loaded. The graph computations in this chapter use `bliss`, through the required `Digraphs` package.

3.1 Worked examples

As a first example we canonicalise Latin squares, considering two squares equivalent (isotopic) if one can be reached from the other by permuting rows, permuting columns, and renaming symbols. We represent a square as a 2D matrix whose row and column indices are drawn from points disjoint from the symbols, and attach a colouring so rows, columns and symbols cannot be exchanged with each other.

Example

```
gap> LoadPackage("images", false);
true
gap> n := 4;;
gap> row := [n+1..2*n];; col := [2*n+1..3*n];;
gap> enc := m -> Combinatorial.WithColoring(
>   Combinatorial.Matrix2D(m, row, col), [[1..n], row, col]);;
gap> klein := [[1,2,3,4],[2,1,4,3],[3,4,1,2],[4,3,2,1]];
```

Now any group acting on the points `[1..3*n]` can be used to canonicalise; permutations which mix the colour classes are excluded by the colouring. The canonical image is the same whichever equivalent square we start from: here we scramble the Klein table by swapping two symbols, two rows and

two columns. The order 4 Latin squares fall into exactly two isotopy classes, represented by the Klein and cyclic group tables, and the canonical images separate them.

Example

```
gap> klein2 := [[3,1,2,4],[1,3,4,2],[2,4,3,1],[4,2,1,3]];;
gap> z4 := [[1,2,3,4],[2,3,4,1],[3,4,1,2],[4,1,2,3]];;
gap> can1 := CanonicalImage(SymmetricGroup(3*n), enc(klein), OnFundamental);;
gap> can2 := CanonicalImage(SymmetricGroup(3*n), enc(klein2), OnFundamental);;
gap> can3 := CanonicalImage(SymmetricGroup(3*n), enc(z4), OnFundamental);;
gap> can1 = can2;
true
gap> can1 = can3;
false
```

As a second example, the multiplication table of a semigroup of order n can be canonicalised up to isomorphism as a 2D matrix on which `SymmetricGroup(n)` acts simultaneously on values, rows and columns.

Example

```
gap> mat := [[1,2,3,4],[1,4,3,2],[1,2,4,3],[4,2,1,3]];;
gap> fullm := Combinatorial.Matrix2D(mat, [1..4], [1..4]);;
gap> CanonicalImagePerm(SymmetricGroup(4), fullm);
(1,4,3,2)
```

A pair of such tables can be treated as an ordered pair with `Combinatorial.Tuple`, or as an unordered pair with `Combinatorial.Multiset`.

3.2 Function documentation

3.2.1 Fundamental

▷ `Fundamental`

(global variable)

The `Fundamental` record provides the basic building blocks for representing structured data. These structures are records typically containing `kind`, `contents`, and `type` fields. It also defines constants for different kinds of structures.

The following components are available in the `Fundamental` record:

`AtomOf( a )`

Returns a new fundamental structure representing an `Atom` with value `a`.

`AtomOfWithType( a, t )`

Returns a new fundamental structure representing an `Atom` with value `a` and type `t`.

`CollectionOf( l )`

Returns a new fundamental structure representing the collection containing the list `l`.

`CollectionOfWithType( l, t )`

Returns a new fundamental structure representing the collection containing the list `l`, of type `t`.

`TupleOf( l )`

Returns a new fundamental structure representing the tuple containing the list `l`.

`TupleOfWithType( l, t )`

Returns a new fundamental structure representing the tuple containing the list *l*, of type *t*.

The record also contains the constants `AtomType`, `CollectionType` and `TupleType`, the possible values of the kind field of a fundamental structure.

The `Fundamental` record provides the basic building blocks such as atoms, collections, and tuples. Building upon these, the `Combinatorial` record offers convenient constructors for common mathematical objects.

3.2.2 Combinatorial

▷ `Combinatorial`

(global variable)

The `Combinatorial` record provides a collection of functions to construct various standard combinatorial objects. These objects are built using the underlying structures defined in `Fundamental` (3.2.1).

The following components are available in the `Combinatorial` record:

`Atom( a )`

Returns a fundamental atom structure representing *a*. This is equivalent to calling `Fundamental.AtomOfWithType( a, "atom" )`.

`Set( l )`

Returns a fundamental collection structure representing a set from the list *l*. The elements in *l* are typically fundamental structures themselves. This is equivalent to calling `Fundamental.CollectionOfWithType( l, "set" )`.

`Multiset( l )`

Returns a fundamental collection structure representing a multiset from the list *l*. This is equivalent to calling `Fundamental.CollectionOfWithType( l, "multiset" )`.

`Tuple( l )`

Returns a fundamental tuple structure from the list *l*. This is equivalent to calling `Fundamental.TupleOfWithType( l, "tuple" )`.

`Matrix( vals, index )`

Constructs a 1D matrix representation. *vals* is a list of values and *index* is a list of corresponding indices. The function asserts that *vals* and *index* have the same length. The matrix is represented as a fundamental collection of type "matrix1dtop", where each element is a fundamental tuple of type "matrix1d" containing a pair [*val*, *idx*].

`Matrix2D( vals, index1, index2 )`

Constructs a 2D matrix representation. *vals* is a 2D list (list of lists) of values, *index1* is a list of row indices, and *index2* is a list of column indices. The function asserts that the dimensions match. The matrix is represented as a fundamental collection of type "matrix2dtop", where each element is a fundamental tuple of type "matrix2d" containing a triplet [*val*, *idx1*, *idx2*].

`Permutation( p )`

Converts a GAP permutation *p* into a fundamental structure. It lists the moved points of *p*

as pairs $[\text{point}, \text{point}^p]$. Each such pair is converted into a fundamental tuple of atoms. These tuples are then collected into a fundamental collection of type "permutation".

`Transformation( p )`

Converts a GAP transformation p into a fundamental structure. Similar to `Permutation`, it uses moved points $[\text{point}, \text{point}^p]$ and forms a fundamental collection of type "transformation".

`PartialPermutation( p )`

Converts a GAP partial permutation p into a fundamental structure. Similar to `Permutation`, it uses moved points $[\text{point}, \text{point}^p]$ and forms a fundamental collection of type "partialpermutation".

`OrderedPartition( p )`

Converts a list of lists p into a fundamental tuple of collections of type "orderedpartition": the order of the parts matters, the order within each part does not.

`WithColoring( f, cols )`

Attaches a colouring to the fundamental structure f : $cols$ is a list of lists of points, and two points may only be mapped to each other if they lie in the same list. This is the way to restrict which relabellings of f are considered.

The following functions operate on these fundamental structures:

3.2.3 OnFundamental

▷ `OnFundamental( f, p )`

(function)

Applies a permutation p to a fundamental structure f or to an integer.

If f is an integer, it returns the image of f under p (i.e., f^p).

If f is a fundamental structure (a record with `kind`, `contents`, and `type` fields):

- If $f.\text{kind}$ is `Fundamental.AtomType`, it applies p to $f.\text{contents}$.
- If $f.\text{kind}$ is `Fundamental.CollectionType`, it recursively calls `OnFundamental` on each element of $f.\text{contents}$ with p , and the resulting list of contents is sorted.
- If $f.\text{kind}$ is `Fundamental.TupleType`, it recursively calls `OnFundamental` on each element of $f.\text{contents}$ with p . The order of elements in the tuple is preserved.

A new fundamental structure is returned with the modified contents, while the `kind` and `type` fields are preserved from the original structure f . An error is raised if f has an invalid `kind`.

The action is also installed as the power operation, so f^p may be used instead.

This function describes how permutations act on fundamental structures.

3.2.4 AtomsOfFundamentalStructure

▷ `AtomsOfFundamentalStructure( f )`

(function)

Returns the set of all atoms (the underlying points) occurring anywhere in the fundamental structure f .

3.2.5 GraphOfFundamentalStructure

▷ `GraphOfFundamentalStructure(s, omega, parts)` (function)

Constructs a graph representation of a fundamental structure *s*.

s is the fundamental structure to be converted into a graph. *omega* is a list of atomic elements (often integers) that form the base points of the graph. These are typically the objects upon which permutations will act. *parts* is a list of lists, representing a partition of *omega*. This partition is used to assign initial colors to the vertices corresponding to elements of *omega*. For an element *j* in *parts* [*i*], its corresponding vertex is colored with `Fundamental.PAtom, i`. Elements of *omega* not in any list in *parts* are colored with `Fundamental.PAtom`.

The function returns a record, let's call it graph, with the following components:

vertices

A list of records, where each record represents a vertex in the graph. Each vertex record has at least `name`, `colour`, `height`, and `id` fields.

edges

A list of pairs [*u*, *v*], where *u* and *v* are IDs of vertices, representing directed edges from *u* to *v*.

omega

The length of the input list *omega*.

atoms

A hash map where keys are the elements from *omega* and values are the IDs of their corresponding vertices in `graph.vertices`.

The graph construction recursively traverses the fundamental structure *s*, creating vertices for atoms, collections, and tuples, and connecting them appropriately. Optimizations (`_IMAGES_DO_ATOM_OPT`, `_IMAGES_DO_TUPLE_OPT`) might affect the exact structure for performance.

This function explains the conversion of a fundamental structure into a graph representation, which is essential for the subsequent algorithms. The stabilizer of these structures can be computed, either over the full symmetric group or within a user-specified group:

3.2.6 StabilizerOfFundamentalStructure

▷ `StabilizerOfFundamentalStructure(fs, omega[, parts])` (function)

Computes the stabilizer group of the fundamental structure *fs* with respect to a set of base points *omega*.

fs is the fundamental structure. *omega* is a list of atomic elements, representing the set of points on which the resulting group will act. *parts* (optional) is a partition of *omega*, used for coloring the graph derived from *fs*. If not provided, an empty partition [] is used.

The function first converts the fundamental structure *fs* into a digraph using `_convertToDigraph` (which internally calls `GraphOfFundamentalStructure` (3.2.5)). This digraph also has an associated vertex coloring based on *parts* and the types of internal nodes. Then, it computes the automorphism group of this colored digraph using `BlissAutomorphismGroup` from the `Digraphs` package. Finally, the resulting automorphism group is returned as a permutation group acting on the points in *omega*.

3.2.7 StabilizerOfFundamentalStructureWithGroup

▷ `StabilizerOfFundamentalStructureWithGroup(fs, omega, grp)` (function)

Computes the stabilizer of the fundamental structure *fs* within a given permutation group *grp*. This function requires the `vole` package to be loaded.

fs is the fundamental structure. *omega* is a list of atomic elements, which must contain every atom of *fs* and every point moved by *grp*. *grp* is a permutation group acting on the points in *omega*. The search for stabilizing permutations is restricted to *grp*.

The function converts *fs* into a colored digraph (using `_convertToDigraph` with *omega* as a single part for coloring). It then constructs a candidate group for `VoleFind.Group` by combining *grp* (acting on *omega* vertices) with the symmetric group on the remaining non-*omega* vertices of the graph. `VoleFind.Group` is used to find the subgroup of this candidate group that stabilizes the digraph and its coloring. The resulting group is then restricted to act only on the vertices corresponding to *omega* and is returned.

Similarly, canonical permutations can be found:

3.2.8 CanonicalPermOfFundamentalStructure

▷ `CanonicalPermOfFundamentalStructure(fs, omega)` (function)

Computes a canonicalizing permutation for the fundamental structure *fs* with respect to *omega*. This function assumes the full symmetric group is acting on *omega*.

fs is the fundamental structure. *omega* is a list of atomic elements.

This function is a convenience wrapper that calls `CanonicalPermOfFundamentalStructureWithGroup` (3.2.9) with *fs*, *omega*, and `SymmetricGroup(omega)`. It requires the `vole` package to be loaded. It returns a permutation acting on *omega* that maps *fs* to its canonical form.

3.2.9 CanonicalPermOfFundamentalStructureWithGroup

▷ `CanonicalPermOfFundamentalStructureWithGroup(fs, omega, grp)` (function)

Computes a canonicalizing permutation for the fundamental structure *fs* with respect to *omega*, restricting the search to the group *grp*. This function requires the `vole` package to be loaded.

fs is the fundamental structure. *omega* is a list of atomic elements, which must contain every atom of *fs* and every point moved by *grp*. *grp* is a permutation group acting on the points in *omega*.

Similar to `StabilizerOfFundamentalStructureWithGroup` (3.2.7), this function converts *fs* to a colored digraph. It forms a candidate group by combining *grp* (acting on *omega* vertices) with the symmetric group on non-*omega* vertices. `VoleFind.CanonicalPerm` is then used to find a permutation from this candidate group that maps the digraph (and its coloring) to a canonical form. The resulting permutation is restricted to act on *omega* and is returned. This permutation, when applied to *omega* and used to relabel *fs*, would yield a canonical representation of *fs* under the action of *grp*.

Finally, a utility function for refining canonical labellings with respect to colorings is provided:

3.2.10 MakeCanonicalLabellingRespectColors

▷ `MakeCanonicalLabellingRespectColors(n, p, colours)` (function)

Adjusts a permutation p (acting on $[1..n]$) to create a new permutation that respects a given coloring. The intent is to refine a canonical labeling p such that elements within the same color class are ordered canonically based on their preimages under p .

n is the number of points being permuted; it is raised internally if p or *colours* mention larger points, and points of $[1..n]$ not in any colour class are treated as one extra class. p is the input permutation, typically a canonical labeling permutation obtained from a graph algorithm. *colours* is a list of lists, where each inner list *colours*[i] contains points belonging to the i -th color class. These inner lists are treated as sets.

The function works as follows:

- It first determines the color class for each point j in $[1..n]$.
- It computes the inverse of p , let this be p_inv . The list `ListPerm( $p\_inv$ ,  $n$ )` gives the order in which points val appear if we iterate i from 1 to n and take $val = i^{(p_inv)}$.
- It then constructs a new permutation. For each i from 1 to n , it considers the point $val = i^{(p_inv)}$. This point val belongs to some color class, say col_val .
- The point val is mapped by the new permutation to the next available (i.e., smallest unassigned) point within its own color class *colours*[col_val], according to the ordering defined by iterating through i .

The effect is that the output permutation, when applied, will order the points such that all points of the first color class come first (ordered among themselves by their p_inv values), then all points of the second color class (similarly ordered), and so on.

Chapter 4

Installing and Loading the Images Package

To install the `Images` package simply place `Images` into the `pkg` directory of your GAP installation.

The package requires GAP 4.13 or later, together with the `Digraphs` and `Datastructures` packages (both part of the standard GAP package distribution). Two further packages are optional but recommended, and are used automatically when available: `ferret` greatly speeds up the stabilizer computations used when canonicalising transformations, permutations, partial permutations and sets of sets, and `vole` is required for the "vole" engine and for canonicalising fundamental structures under groups which are not direct products of symmetric groups (see Chapter 3).

4.1 Loading the Images Package

To use the `Images` Package you have to request it explicitly. This is done by calling `LoadPackage` (**Reference:** `LoadPackage`):

Example

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

References

- [JJPW19] Christopher Jefferson, Eliza Jonauskyte, Markus Pfeiffer, and Rebecca Waldecker. Minimal and canonical images. *Journal of Algebra*, 521:481–506, 2019. [9](#)
- [JWW23] Christopher Jefferson, Rebecca Waldecker, and Wilf A. Wilson. Computing canonical images in permutation groups with graph backtracking. *Journal of Computational Algebra*, 8:100010, December 2023. [4](#)

Index

Images package, [4](#)

AtomsOfFundamentalStructure, [15](#)

CanonicalImage, [8](#)

CanonicalImagePerm, [8](#)

CanonicalPermOfFundamentalStructure, [17](#)

CanonicalPermOfFundamentalStructure-
WithGroup, [17](#)

Combinatorial, [14](#)

Fundamental, [13](#)

GraphOfFundamentalStructure, [16](#)

ImagesAdvancedConfig, [8](#)

IsCanonicalImage, [8](#)

IsMinimalImage, [7](#)

IsMinimalImageLessThan, [7](#)

MakeCanonicalLabellingRespectColors, [17](#)

MinimalImage, [7](#)

MinimalImageOrderedPair, [8](#)

MinimalImagePerm, [7](#)

MinimalImageUnorderedPair, [8](#)

OnFundamental, [15](#)

OnMultiplicationTables, [7](#)

StabilizerOfFundamentalStructure, [16](#)

StabilizerOfFundamentalStructureWith-
Group, [17](#)