RCWA

Residue-Class-Wise Affine Groups

Version 4.6.1

December 18, 2017

Stefan Kohl

Stefan Kohl Email: stefan@mcs.st-and.ac.uk
Homepage: https://stefan-kohl.github.io/

Abstract

RCWA is a package for GAP 4. It provides implementations of algorithms and methods for computing in certain infinite permutation groups acting on the set of integers. This package can be used to investigate the following types of groups and many more:

- Finite groups, and certain divisible torsion groups which they embed into.
- Free groups of finite rank.
- Free products of finitely many finite groups.
- Direct products of the above groups.
- Wreath products of the above groups with finite groups and with ($\mathbb{Z}$,+).
- Subgroups of any such groups.

With the help of this package, the author has found a countable simple group which is generated by involutions interchanging disjoint residue classes of $\mathbb{Z}$ and which all the above groups embed into – see [Koh10].

Acknowledgements

I am grateful to John P. McDermott for the discovery that the group discussed in Section 7.1 is isomorphic to Thompson's Group V in July 2008, and to Laurent Bartholdi for his hint on how to construct wreath products of residue-class-wise affine groups with ($\mathbb{Z}$,+) in April 2006. Further, I thank Bettina Eick for communicating this package and for her valuable suggestions on its manual in the time before its first public release in April 2005. Last but not least I thank the two anonymous referees for their constructive criticism and their helpful suggestions.

Contents

Chapter 1

About the RCWA Package

This package permits to compute in monoids, in particular groups, whose elements are *residue-class-wise affine* mappings. Probably the widest-known occurrence of such a mapping is in the statement of the $3n+1$ conjecture, which asserts that iterated application of the *Collatz mapping*

$$T: \ \mathbb{Z} \longrightarrow \mathbb{Z}, \quad n \longmapsto \begin{cases} \frac{n}{2} & \text{if } n \text{ is even,} \\ \frac{3n+1}{2} & \text{if } n \text{ is odd} \end{cases}$$

to any given positive integer eventually yields 1 (cf. [Lag03]). For definitions, see Section 2.1.

Presently, most research in computational group theory focuses on finite permutation groups, matrix groups, finitely presented groups, polycyclically presented groups and automata groups. For details, we refer to [HEO05]. The purpose of this package is twofold:

- On the one hand, it provides the means to deal with another large class of groups which are accessible to computational methods, and it therefore extends the range of groups which can be dealt with by means of computation.
- On the other – and perhaps more importantly – residue-class-wise affine groups appear to be interesting mathematical objects in their own right, and this package is intended to serve as a tool to obtain a better understanding of their rich and often complicated group theoretical and combinatorial structure.

In principle this package permits to construct and investigate all groups which have faithful representations as residue-class-wise affine groups. Among many others, the following groups and their subgroups belong to this class:

- Finite groups, and certain divisible torsion groups which they embed into.
- Free groups of finite rank.
- Free products of finitely many finite groups.
- Direct products of the above groups.
- Wreath products of the above groups with finite groups and with ($\mathbb{Z}$,+).

This list permits already to conclude that there are finitely generated residue-class-wise affine groups which do not have finite presentations, and such with algorithmically unsolvable membership problem. However the list is certainly by far not exhaustive, and using this package it is easy to construct groups of types which are not mentioned there.

The group $\mathrm{CT}(\mathbb{Z})$ which is generated by all *class transpositions* of $\mathbb{Z}$ – these are involutions which interchange two disjoint residue classes, see `ClassTransposition` (2.2.3) – is a simple group which has subgroups of all types listed above. It is countable, but it has an uncountable series of simple subgroups which is parametrized by the sets of odd primes.

Proofs of most of the results mentioned so far can be found in [Koh10]. Descriptions of a part of the algorithms and methods which are implemented in this package can be found in [Koh08].

The reader might want to know what type of results one can obtain with RCWA. However, the answer to this is that the package can be applied in various ways to various different problems, and it is simply not possible to say in general what can be found out with its help. So one really cannot give a better answer here than for the same question about GAP itself. The best way to get familiar with the package and its capabilities is likely to experiment with the examples discussed in this manual and the groups generated by 3 class transpositions from the corresponding data library.

Of course, sometimes this package does not provide an out-of-the-box solution for a given problem. But quite often it is still possible to find an answer by an interactive trial-and-error approach. With substantial help of this package, the author has found the results mentioned above. Interactive sessions with this package have also led to the development of most of the algorithms which are now implemented in it. Just to mention one example, developing the factorization method for residue-class-wise affine permutations (see `FactorizationIntoCSCRCT` (2.5.1)) solely by means of theory would likely have been very hard.

Chapter 2

Residue-Class-Wise Affine Mappings

This chapter contains the basic definitions, and it describes how to enter residue-class-wise affine mappings and how to compute with them.

How to compute with residue-class-wise affine groups is described in detail in the next chapter. The reader is encouraged to look there already after having read the first few pages of this chapter, and to look up definitions as he needs to.

2.1 Basic definitions

Residue-class-wise affine groups, or *rcwa* groups for short, are permutation groups whose elements are bijective residue-class-wise affine mappings.

A mapping $f : \mathbb{Z} \to \mathbb{Z}$ is called *residue-class-wise affine*, or for short an *rcwa* mapping, if there is a positive integer m such that the restrictions of f to the residue classes $r(m) \in \mathbb{Z}/m\mathbb{Z}$ are all affine, i.e. given by

$$f|_{r(m)} : \ r(m) \to \mathbb{Z}, \quad n \ \mapsto \ \frac{a_{r(m)} \cdot n + b_{r(m)}}{c_{r(m)}}$$

for certain coefficients $a_{r(m)}, b_{r(m)}, c_{r(m)} \in \mathbb{Z}$ depending on $r(m)$. The smallest possible m is called the *modulus* of f. It is understood that all fractions are reduced, i.e. that $\gcd(a_{r(m)}, b_{r(m)}, c_{r(m)}) = 1$, and that $c_{r(m)} > 0$. The lcm of the coefficients $a_{r(m)}$ is called the *multiplier* of f, and the lcm of the coefficients $c_{r(m)}$ is called the *divisor* of f.

It is easy to see that the residue-class-wise affine mappings of $\mathbb{Z}$ form a monoid under composition, and that the residue-class-wise affine permutations of $\mathbb{Z}$ form a countable subgroup of $\mathrm{Sym}(\mathbb{Z})$. We denote the former by $\mathrm{Rcwa}(\mathbb{Z})$, and the latter by $\mathrm{RCWA}(\mathbb{Z})$.

An rcwa mapping is called *tame* if the set of moduli of its powers is bounded, or equivalently if it permutes a partition of $\mathbb{Z}$ into finitely many residue classes on all of which it is affine. An rcwa group is called *tame* if there is a common such partition for all of its elements, or equivalently if the set of moduli of its elements is bounded. Rcwa mappings and -groups which are not tame are called *wild*. Tame rcwa mappings and -groups are something which one could call the “trivial cases” or “basic building blocks”, while wild rcwa groups are the objects of primary interest.

The definitions of residue-class-wise affine mappings and -groups can be generalized in the obvious way to suitable rings other than $\mathbb{Z}$. In fact, this package provides also some support for residue-class-wise affine groups over $\mathbb{Z}^2$, over semilocalizations of $\mathbb{Z}$ and over univariate polynomial rings over finite fields. The ring $\mathbb{Z}^2$ has been chosen as an example of a suitable ring which is not a principal ideal domain, the semilocalizations of $\mathbb{Z}$ have been chosen as examples of rings with only finitely many prime elements, and the univariate polynomial rings over finite fields have been chosen as examples of rings with nonzero characteristic.

2.2 Entering residue-class-wise affine mappings

Entering an rcwa mapping of $\mathbb{Z}$ requires giving the modulus m and the coefficients $a_{r(m)}$, $b_{r(m)}$ and $c_{r(m)}$ for $r(m)$ running over the residue classes (mod m).

This can be done easiest by `RcwaMapping( coeffs )`, where *coeffs* is a list of m coefficient triples `coeffs[`$r+1$`] = [`$a_{r(m)}, b_{r(m)}, c_{r(m)}$`]`, with r running from 0 to $m-1$.

If some coefficient $c_{r(m)}$ is zero or if images of some integers under the mapping to be defined would not be integers, an error message is printed and a break loop is entered. For example, the coefficient triple `[1,4,3]` is not allowed at the first position. The reason for this is that not all integers congruent to $1 \cdot 0 + 4 = 4 \bmod m$ are divisible by 3.

For the general constructor for rcwa mappings, see `RcwaMapping` (2.2.5).

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]); # The Collatz mapping.
<rcwa mapping of Z with modulus 2>
gap> [ IsSurjective(T), IsInjective(T) ];
[ true, false ]
gap> Display(T);

Surjective rcwa mapping of Z with modulus 2

        /
        | n/2      if n in 0(2)
 n |-> <  (3n+1)/2 if n in 1(2)
        |
        \

gap> a := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);
<rcwa mapping of Z with modulus 3>
gap> IsBijective(a);
true
gap> Display(a); # This is Collatz' permutation:

Rcwa permutation of Z with modulus 3

        /
        | 2n/3     if n in 0(3)
 n |-> <  (4n-1)/3 if n in 1(3)
        | (4n+1)/3 if n in 2(3)
        \

gap> Support(a);
Z \ [ -1, 0, 1 ]
gap> Cycle(a,44);
```

```
[ 44, 59, 79, 105, 70, 93, 62, 83, 111, 74, 99, 66 ]
```

There is computational evidence for the conjecture that any residue-class-wise affine permutation of $\mathbb{Z}$ can be factored into members of the following three series of permutations of particularly simple structure (cf. FactorizationIntoCSCRCT (2.5.1)):

2.2.1 ClassShift (r, m)

▷ ClassShift(*r*, *m*) (function)
▷ ClassShift(*cl*) (function)

Returns: the class shift $\nu_{r(m)}$.

The *class shift* $\nu_{r(m)}$ is the rcwa mapping of $\mathbb{Z}$ which maps $n \in r(m)$ to $n+m$ and which fixes $\mathbb{Z} \setminus r(m)$ pointwise.

In the one-argument form, the argument *cl* stands for the residue class $r(m)$. Enclosing the argument list in list brackets is permitted.

Example

```
gap> Display(ClassShift(5,12));

Tame rcwa permutation of Z with modulus 12, of order infinity

         /
         | n+12 if n in 5(12)
 n |-> < n     if n in Z \ 5(12)
         |
         \
```

2.2.2 ClassReflection (r, m)

▷ ClassReflection(*r*, *m*) (function)
▷ ClassReflection(*cl*) (function)

Returns: the class reflection $\varsigma_{r(m)}$.

The *class reflection* $\varsigma_{r(m)}$ is the rcwa mapping of $\mathbb{Z}$ which maps $n \in r(m)$ to $-n+2r$ and which fixes $\mathbb{Z} \setminus r(m)$ pointwise, where it is understood that $0 \leq r < m$.

In the one-argument form, the argument *cl* stands for the residue class $r(m)$. Enclosing the argument list in list brackets is permitted.

Example

```
gap> Display(ClassReflection(5,9));

Rcwa permutation of Z with modulus 9, of order 2

         /
         | -n+10 if n in 5(9)
 n |-> < n      if n in Z \ 5(9)
         |
         \
```

2.2.3 ClassTransposition (r1, m1, r2, m2)

▷ `ClassTransposition(r1, m1, r2, m2)` (function)
▷ `ClassTransposition(cl1, cl2)` (function)

Returns: the class transposition $\tau_{r_1(m_1),r_2(m_2)}$.

Given two disjoint residue classes $r_1(m_1)$ and $r_2(m_2)$ of the integers, the *class transposition* $\tau_{r_1(m_1),r_2(m_2)} \in \mathrm{RCWA}(\mathbb{Z})$ is defined as the involution which interchanges r_1+km_1 and r_2+km_2 for any integer k and which fixes all other points. It is understood that m_1 and m_2 are positive, that $0 \leq r_1 < m_1$ and that $0 \leq r_2 < m_2$. For a *generalized class transposition*, the latter assumptions are not made.

The class transposition $\tau_{r_1(m_1),r_2(m_2)}$ interchanges the residue classes $r_1(m_1)$ and $r_2(m_2)$ and fixes the complement of their union pointwise.

In the four-argument form, the arguments `r1`, `m1`, `r2` and `m2` stand for r_1, m_1, r_2 and m_2, respectively. In the two-argument form, the arguments `cl1` and `cl2` stand for the residue classes $r_1(m_1)$ and $r_2(m_2)$, respectively. Enclosing the argument list in list brackets is permitted. The residue classes $r_1(m_1)$ and $r_2(m_2)$ are stored as an attribute `TransposedClasses`.

A list of all class transpositions interchanging residue classes with moduli less than or equal to a given bound `m` can be obtained by `List(ClassPairs([P],m),ClassTransposition)`, where the function `ClassPairs` returns a list of all 4-tuples (r_1,m_1,r_2,m_2) of integers corresponding to the unordered pairs of disjoint residue classes $r_1(m_1)$ and $r_2(m_2)$ with m_1 and m_2 less than or equal to the specified bound. If a list of primes is given as optional argument `P`, then the returned list contains only those 4-tuples where all prime factors of m_1 and m_2 lie in `P`. If the option `divisors` is set, the returned list contains only the 4-tuples where m_1 and m_2 divide `m`.

The function `NrClassPairs(m)` returns the length of the list `ClassPairs(m)`, where the result is computed much faster and without actually generating the list of tuples. Given a class transposition `ct`, the corresponding 4-tuple can be obtained by `ExtRepOfObj(ct)`

A class transposition can be written as a product of any given number k of class transpositions. Such a decomposition can be obtained by `SplittedClassTransposition(ct,k)`.

Example

```
gap> Display(ClassTransposition(1,2,8,10):CycleNotation:=false);

Rcwa permutation of Z with modulus 10, of order 2

        /
        | 5n+3    if n in 1(2)
 n |-> <  (n-3)/5 if n in 8(10)
        | n       if n in 0(2) \ 8(10)
        \

gap> List(ClassPairs(4),ClassTransposition);
[ ( 0(2), 1(2) ), ( 0(2), 1(4) ), ( 0(2), 3(4) ), ( 0(3), 1(3) ),
  ( 0(3), 2(3) ), ( 0(4), 1(4) ), ( 0(4), 2(4) ), ( 0(4), 3(4) ),
  ( 1(2), 0(4) ), ( 1(2), 2(4) ), ( 1(3), 2(3) ), ( 1(4), 2(4) ),
  ( 1(4), 3(4) ), ( 2(4), 3(4) ) ]
gap> NrClassPairs(100);
3528138
gap> SplittedClassTransposition(ClassTransposition(0,2,1,4),3);
[ ( 0(6), 1(12) ), ( 2(6), 5(12) ), ( 4(6), 9(12) ) ]
```

The set of all class transpositions of the ring of integers generates the simple group CT($\mathbb{Z}$) mentioned in Chapter 1. This group has a representation as a GAP object – see CT (3.1.9). The set of all generalized class transpositions of $\mathbb{Z}$ generates a simple group as well, cf. [Koh10].

Class shifts, class reflections and class transpositions of rings R other than $\mathbb{Z}$ are defined in an entirely analogous way – all one needs to do is to replace $\mathbb{Z}$ by R and to read $<$ and $\leq$ in the sense of the ordering used by GAP. They can also be entered basically as described above – just prepend the desired ring R to the argument list. Often also a sensible "default ring" ($\rightarrow$ `DefaultRing` in the GAP Reference Manual) is chosen if that optional first argument is omitted.

On rings which have more than two units, there is another basic series of rcwa permutations which generalizes class reflections:

2.2.4 ClassRotation (r, m, u)

▷ `ClassRotation(r, m, u)` (function)
▷ `ClassRotation(cl, u)` (function)

Returns: the class rotation $\rho_{r(m),u}$.

Given a residue class $r(m)$ and a unit u of a suitable ring R, the *class rotation* $\rho_{r(m),u}$ is the rcwa mapping which maps $n \in r(m)$ to $un+(1-u)r$ and which fixes $R \setminus r(m)$ pointwise. Class rotations generalize class reflections, as we have $\rho_{r(m),-1} = \varsigma_{r(m)}$.

In the two-argument form, the argument *cl* stands for the residue class $r(m)$. Enclosing the argument list in list brackets is permitted. The argument `u` is stored as an attribute `RotationFactor`.

Example

```
gap> Display(ClassRotation(ResidueClass(Z_pi(2),2,1),1/3));

Tame rcwa permutation of Z_( 2 ) with modulus 2, of order infinity

        /
        | 1/3 n + 2/3 if n in 1(2)
 n |-> <  n           if n in 0(2)
        |
        \

gap> x := Indeterminate(GF(8),1);; SetName(x,"x");
gap> R := PolynomialRing(GF(8),1);;
gap> cr := ClassRotation(1,x,Z(8)*One(R)); Support(cr);
ClassRotation( 1(x), Z(2^3) )
1(x) \ [ 1 ]
gap> Display(cr);

Rcwa permutation of GF(2^3)[x] with modulus x, of order 7

        /
        | Z(2^3)*P + Z(2^3)^3 if P in 1(x)
 P |-> <  P                   otherwise
        |
        \
```

IsClassShift, IsClassReflection, IsClassRotation, IsClassTransposition and IsGeneralizedClassTransposition are properties which indicate whether a given rcwa mapping belongs to the corresponding series.

In the sequel we describe the general-purpose constructor for rcwa mappings. The constructor may look a bit technical on a first glance, but knowing all possible ways of entering an rcwa mapping is by no means necessary for understanding this manual or for using this package.

2.2.5 RcwaMapping (the general constructor)

▷ RcwaMapping(*R*, *m*, *coeffs*) (method)
▷ RcwaMapping(*R*, *coeffs*) (method)
▷ RcwaMapping(*coeffs*) (method)
▷ RcwaMapping(*perm*, *range*) (method)
▷ RcwaMapping(*m*, *values*) (method)
▷ RcwaMapping(*pi*, *coeffs*) (method)
▷ RcwaMapping(*q*, *m*, *coeffs*) (method)
▷ RcwaMapping(*P1*, *P2*) (method)
▷ RcwaMapping(*cycles*) (method)
▷ RcwaMapping(*expression*) (method)

Returns: an rcwa mapping.

In all cases the argument *R* is the underlying ring, *m* is the modulus and *coeffs* is the coefficient list. A *coefficient list* for an rcwa mapping with modulus m consists of $|R/mR|$ coefficient triples $[a_{r(m)}, b_{r(m)}, c_{r(m)}]$. Their ordering is determined by the ordering of the representatives of the residue classes (mod m) in the sorted list returned by AllResidues(*R*, *m*). In case $R = \mathbb{Z}$ this means that the coefficient triple for the residue class $0(m)$ comes first and is followed by the one for $1(m)$, the one for $2(m)$ and so on.

If one or several of the arguments *R*, *m* and *coeffs* are omitted or replaced by other arguments, the former are either derived from the latter or default values are chosen. The meaning of the other arguments is defined in the detailed description of the particular methods given in the sequel. The above methods return the rcwa mapping

(a) of *R* with modulus *m* and coefficients *coeffs*,

(b) of $R = \mathbb{Z}$ or $R = \mathbb{Z}_{(\pi)}$ with modulus Length(*coeffs*) and coefficients *coeffs*,

(c) of $R = \mathbb{Z}$ with modulus Length(*coeffs*) and coefficients *coeffs*,

(d) of $R = \mathbb{Z}$, permuting any set *range*+k*Length(*range*) like *perm* permutes *range*,

(e) of $R = \mathbb{Z}$ with modulus *m* and values given by a list *val* of 2 pairs [preimage, image] per residue class (mod *m*),

(f) of $R = \mathbb{Z}_{(\pi)}$ with modulus Length(*coeffs*) and coefficients *coeffs* (the set of primes π which denotes the underlying ring is passed as argument *pi*),

(g) of $R = \mathrm{GF}(q)[x]$ with modulus *m* and coefficients *coeffs*,

(h) an rcwa permutation which induces a bijection between the partitions *P1* and *P2* of *R* into residue classes and which is affine on the elements of *P1*,

(i) an rcwa permutation with "residue class cycles" given by a list *cycles* of lists of pairwise disjoint residue classes, each of which it permutes cyclically, or

(j) the rcwa permutation of $\mathbb{Z}$ given by the arithmetical expression *expression* – a string consisting of class transpositions (e.g. `"(0(2),1(4))"`) or cycles permuting residue classes (e.g. `"(0(2),1(8),3(4),5(8))"`), class shifts (e.g. `"cs(4(6))"`, class reflections (e.g. `"cr(3(4))"`), arithmetical operators (`"*"`, `"/"` and `"^"`) and brackets (`"("`, `")"`),

respectively. The methods for the operation `RcwaMapping` perform a number of argument checks, which can be skipped by using `RcwaMappingNC` instead.

Example

```
gap> R := PolynomialRing(GF(2),1);; x := X(GF(2),1);; SetName(x,"x");
gap> RcwaMapping(R,x+1,[[1,0,x+One(R)],[x+One(R),0,1]]*One(R));     # (a)
<rcwa mapping of GF(2)[x] with modulus x+1>
gap> RcwaMapping(Z_pi(2),[[1/3,0,1]]);                              # (b)
Rcwa mapping of Z_( 2 ): n -> 1/3 n
gap> a := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);                  # (c)
<rcwa mapping of Z with modulus 3>
gap> RcwaMapping((1,2,3),[1..4]);                                   # (d)
( 1(4), 2(4), 3(4) )
gap> T = RcwaMapping(2,[[1,2],[2,1],[3,5],[4,2]]);                  # (e)
true
gap> RcwaMapping([2],[[1/3,0,1]]);                                  # (f)
Rcwa mapping of Z_( 2 ): n -> 1/3 n
gap> RcwaMapping(2,x+1,[[1,0,x+One(R)],[x+One(R),0,1]]*One(R));     # (g)
<rcwa mapping of GF(2)[x] with modulus x+1>
gap> a = RcwaMapping(List([[0,3],[1,3],[2,3]],ResidueClass),
>                    List([[0,2],[1,4],[3,4]],ResidueClass));       # (h)
true
gap> RcwaMapping([List([[0,2],[1,4],[3,8],[7,16]],ResidueClass)]);  # (i)
( 0(2), 1(4), 3(8), 7(16) )
gap> Cycle(last,ResidueClass(0,2));
[ 0(2), 1(4), 3(8), 7(16) ]
gap> g := RcwaMapping("((0(4),1(6))*cr(0(6)))^2/cs(2(8))");         # (j)
<rcwa permutation of Z with modulus 72>
gap> g = (ClassTransposition(0,4,1,6) * ClassReflection(0,6))^2/
>          ClassShift(2,8);
true
```

Rcwa mappings of $\mathbb{Z}$ can be "translated" to rcwa mappings of some semilocalization $\mathbb{Z}_{(\pi)}$ of $\mathbb{Z}$:

2.2.6 LocalizedRcwaMapping (for an rcwa mapping of Z and a prime)

▷ `LocalizedRcwaMapping(f, p)` (function)
▷ `SemilocalizedRcwaMapping(f, pi)` (function)

Returns: the rcwa mapping of $\mathbb{Z}_{(p)}$ respectively $\mathbb{Z}_{(\pi)}$ with the same coefficients as the rcwa mapping *f* of $\mathbb{Z}$.

The argument *p* or *pi* must be a prime or a set of primes, respectively. The argument *f* must be an rcwa mapping of $\mathbb{Z}$ whose modulus is a power of *p*, or whose modulus has only prime divisors which lie in *pi*, respectively.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);; # The Collatz mapping.
gap> Cycle(LocalizedRcwaMapping(T,2),131/13);
[ 131/13, 203/13, 311/13, 473/13, 716/13, 358/13, 179/13, 275/13,
  419/13, 635/13, 959/13, 1445/13, 2174/13, 1087/13, 1637/13, 2462/13,
  1231/13, 1853/13, 2786/13, 1393/13, 2096/13, 1048/13, 524/13, 262/13 ]
```

Rcwa mappings can be `Viewed`, `Displayed`, `Printed` and written to a `String`. The output of the `View` method is kept reasonably short. In most cases it does not describe an rcwa mapping completely. In these cases the output is enclosed in brackets. There are options `CycleNotation`, `AsClassMapping`, `PrintNotation` and `AbridgedNotation` to take influence on how certain rcwa mappings are shown. These options can either be not set, set to `true` or set to `false`. If the option `CycleNotation` is set, it is tried harder to write down an rcwa permutation of $\mathbb{Z}$ of finite order as a product of disjoint residue class cycles, if this is possible. If the option `AsClassMapping` is set, `Display` shows which residue classes are mapped to which by the affine partial mappings, and marks any loops. The option `PrintNotation` influences the output in favour of GAP - readability, and the option `AbridgedNotation` can be used to abridge longer names like `ClassShift`, `ClassReflection` etc.. By default, the output of the methods for `Display` and `Print` describes an rcwa mapping in full. The `Printed` representation of an rcwa mapping is GAP - readable if and only if the `Printed` representation of the elements of the underlying ring is so.

There is also an operation `LaTeXStringRcwaMapping`, which takes as argument an rcwa mapping and returns a corresponding LaTeX string. The output makes use of the LaTeX macro package amsmath. If the option *`Factorization`* is set and the argument is bijective, a factorization into class shifts, class reflections, class transpositions and prime switches is printed (cf. `FactorizationIntoCSCRCT` (2.5.1)). For rcwa mappings with modulus greater than 1, an indentation by *`Indentation`* characters can be obtained by setting this option value accordingly.

Example

```
gap> Print(LaTeXStringRcwaMapping(T));
n \ \mapsto \
\begin{cases}
  n/2      & \text{if} \ n \in 0(2), \\
  (3n+1)/2 & \text{if} \ n \in 1(2).
\end{cases}
```

There is an operation `LaTeXAndXDVI` which displays an rcwa mapping in an xdvi window. This works as follows: The string returned by `LaTeXStringRcwaMapping` is inserted into a LaTeX template file. This file is LaTeX'ed, and the result is shown with xdvi. Calling `Display` with option *`xdvi`* has the same effect. The operation `LaTeXAndXDVI` is only available on UNIX systems, and requires suitable installations of LaTeX and xdvi.

2.3 Basic arithmetic for residue-class-wise affine mappings

Testing rcwa mappings for equality requires only comparing their coefficient lists, hence is cheap. Rcwa mappings can be multiplied, thus there is a method for `*`. Rcwa permutations can also be inverted, thus there is a method for `Inverse`. The latter method is usually accessed by raising a mapping to a power with negative exponent. Multiplying, inverting and computing powers of tame rcwa mappings is cheap. Computing powers of wild mappings is usually expensive – run time and memory requirements normally grow approximately exponentially with the exponent. How expensive multiplying a couple of wild mappings is, varies very much. In any case, the amount of memory required for storing an rcwa mapping is proportional to its modulus. Whether a given mapping is tame or wild can be determined by the operation `IsTame`. There is a method for `Order`, which can not only compute a finite order, but which can also detect infinite order.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);;          # The Collatz mapping.
gap> a := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);; # Collatz' permutation.
gap> List([-4..4],k->Modulus(a^k));
[ 256, 64, 16, 4, 1, 3, 9, 27, 81 ]
gap> IsTame(T) or IsTame(a);
false
gap> IsTame(ClassShift(0,1)) and IsTame(ClassTransposition(0,2,1,2));
true
gap> T^2*a*T*a^-3;
<rcwa mapping of Z with modulus 768>
gap> (ClassShift(1,3)*ClassReflection(2,7))^1000000;
<rcwa permutation of Z with modulus 21>
```

There are methods installed for `IsInjective`, `IsSurjective`, `IsBijective` and `Image`.

Example

```
gap> [ IsInjective(T), IsSurjective(T), IsBijective(a) ];
[ false, true, true ]
gap> Image(RcwaMapping([[2,0,1]]));
0(2)
```

Images of elements, of finite sets of elements and of unions of finitely many residue classes of the source of an rcwa mapping can be computed with `^`, the same symbol as used for exponentiation and conjugation. The same works for partitions of the source into a finite number of residue classes.

Example

```
gap> 15^T;
23
gap> ResidueClass(1,2)^T;
2(3)
gap> List([[0,3],[1,3],[2,3]],ResidueClass)^a;
[ 0(2), 1(4), 3(4) ]
```

For computing preimages of elements under rcwa mappings, there are methods for `PreImageElm` and `PreImagesElm`. The preimage of a finite set of ring elements or of a union of finitely many residue classes under an rcwa mapping can be computed by `PreImage`.

Example

```
gap> PreImagesElm(T,8);
[ 5, 16 ]
gap> PreImage(T,ResidueClass(Integers,3,2));
Z \ 0(6) U 2(6)
gap> M := [1];; l := [1];;
gap> while Length(M) < 5000 do M := PreImage(T,M); Add(l,Length(M)); od; l;
[ 1, 1, 2, 2, 4, 5, 8, 10, 14, 18, 26, 36, 50, 67, 89, 117, 157, 208,
  277, 367, 488, 649, 869, 1154, 1534, 2039, 2721, 3629, 4843, 6458 ]
```

There is a method for the operation `Support` for computing the support of an rcwa mapping. A synonym for `Support` is `MovedPoints`. The natural density of the support of an rcwa mapping of $\mathbb{Z}$ can be computed efficiently with the operation `DensityOfSupport`. Likewise, the natural density of the set of fixed points of an rcwa mapping of $\mathbb{Z}$ can be computed efficiently with the operation `DensityOfSetOfFixedPoints`. There is also a method for `RestrictedPerm` for computing the restriction of an rcwa permutation to a union of residue classes which it fixes setwise.

Example

```
gap> List([a,a^2],Support);
[ Z \ [ -1, 0, 1 ], Z \ [ -3, -2, -1, 0, 1, 2, 3 ] ]
gap> RestrictedPerm(ClassShift(0,2)*ClassReflection(1,2),
>                   ResidueClass(0,2));
<rcwa mapping of Z with modulus 2>
gap> last = ClassShift(0,2);
true
```

Rcwa mappings can be added and subtracted pointwise. However, please note that the set of rcwa mappings of a ring does not form a ring under `+` and `*`.

Example

```
gap> b := ClassShift(0,3) * a;;
gap> [ Image((a + b)), Image((a - b)) ];
[ 2(4), [ -2, 0 ] ]
```

There are operations `Modulus` (abbreviated `Mod`) and `Coefficients` for retrieving the modulus and the coefficient list of an rcwa mapping. The meaning of the return values is as described in Section 2.2.

General documentation for most operations mentioned in this section can be found in the GAP reference manual. For rcwa mappings of rings other than $\mathbb{Z}$, not for all operations applicable methods are available.

As in general a subring relation $R_1 < R_2$ does *not* give rise to a natural embedding of RCWA(R_1) into RCWA(R_2), there is no coercion between rcwa mappings or rcwa groups over different rings.

2.4 Attributes and properties of residue-class-wise affine mappings

A number of basic attributes and properties of an rcwa mapping are derived immediately from the coefficients of its affine partial mappings. This holds for example for the multiplier and the divisor. These two values are stored as attributes `Multiplier` and `Divisor`, or for short `Mult` and `Div`. The *prime set* of an rcwa mapping is the set of prime divisors of the product of its modulus and its multiplier. It is stored as an attribute `PrimeSet`. The *maximal shift* of an rcwa mapping of $\mathbb{Z}$ is the maximum of the absolute values of its coefficients $b_{r(m)}$ in the notation introduced in Section 2.1. It is stored as an attribute `MaximalShift`. An rcwa mapping is called *class-wise translating* if all of its affine partial mappings are translations, it is called *integral* if its divisor equals 1, and it is called *balanced* if its multiplier and its divisor have the same prime divisors. A class-wise translating mapping has the property `IsClassWiseTranslating`, an integral mapping has the property `IsIntegral` and a balanced mapping has the property `IsBalanced`. An rcwa mapping of the ring of integers or of one of its semilocalizations is called *class-wise order-preserving* if and only if all coefficients $a_{r(m)}$ (cf. Section 2.1) in the numerators of the affine partial mappings are positive. The corresponding property is `IsClassWiseOrderPreserving`. An rcwa mapping of $\mathbb{Z}$ is called *sign-preserving* if it does not map nonnegative integers to negative integers or vice versa. The corresponding property is `IsSignPreserving`. All elements of the simple group CT($\mathbb{Z}$) generated by the set of all class transpositions are sign-preserving.

Example

```
gap> u := RcwaMapping([[3,0,5],[9,1,5],[3,-1,5],[9,-2,5],[9,4,5]]);;
gap> IsBijective(u);; Display(u);

Rcwa permutation of Z with modulus 5

        /
        | 3n/5     if n in 0(5)
        | (9n+1)/5 if n in 1(5)
 n |-> <  (3n-1)/5 if n in 2(5)
        | (9n-2)/5 if n in 3(5)
        | (9n+4)/5 if n in 4(5)
        \

gap> Multiplier(u);
9
gap> Divisor(u);
5
gap> PrimeSet(u);
[ 3, 5 ]
gap> IsIntegral(u) or IsBalanced(u);
false
gap> IsClassWiseOrderPreserving(u) and IsSignPreserving(u);
true
```

There are a couple of further attributes and operations related to the affine partial mappings of an rcwa mapping:

2.4.1 LargestSourcesOfAffineMappings (for an rcwa mapping)

▷ `LargestSourcesOfAffineMappings(f)` (attribute)

Returns: the coarsest partition of `Source(f)` on whose elements the rcwa mapping `f` is affine.

Example

```
gap> LargestSourcesOfAffineMappings(ClassShift(3,7));
[ Z \ 3(7), 3(7) ]
gap> LargestSourcesOfAffineMappings(ClassReflection(0,1));
[ Integers ]
gap> u := RcwaMapping([[3,0,5],[9,1,5],[3,-1,5],[9,-2,5],[9,4,5]]);;
gap> List( [ u, u^-1 ], LargestSourcesOfAffineMappings );
[ [ 0(5), 1(5), 2(5), 3(5), 4(5) ], [ 0(3), 1(3), 2(9), 5(9), 8(9) ] ]
gap> kappa := ClassTransposition(2,4,3,4) * ClassTransposition(4,6,8,12)
>             * ClassTransposition(3,4,4,6);
<rcwa permutation of Z with modulus 12>
gap> LargestSourcesOfAffineMappings(kappa);
[ 2(4), 1(4) U 0(12), 3(12) U 7(12), 4(12), 8(12), 11(12) ]
```

2.4.2 FixedPointsOfAffinePartialMappings (for an rcwa mapping)

▷ `FixedPointsOfAffinePartialMappings(f)` (attribute)

Returns: a list of the sets of fixed points of the affine partial mappings of the rcwa mapping `f` in the quotient field of its source.

The returned list contains entries for the restrictions of `f` to all residue classes modulo `Mod(f)`. A list entry can either be an empty set, the source of `f` or a set of cardinality 1. The ordering of the entries corresponds to the ordering of the residues in `AllResidues(Source(f),m)`.

Example

```
gap> FixedPointsOfAffinePartialMappings(ClassShift(0,2));
[ [  ], Rationals ]
gap> List([1..3],k->FixedPointsOfAffinePartialMappings(T^k));
[ [ [ 0 ], [ -1 ] ], [ [ 0 ], [ 1 ], [ 2 ], [ -1 ] ],
  [ [ 0 ], [ -7 ], [ 2/5 ], [ -5 ], [ 4/5 ], [ 1/5 ], [ -10 ], [ -1 ] ] ]
```

2.4.3 Multpk (for an rcwa mapping, a prime and an exponent)

▷ `Multpk(f, p, k)` (operation)

Returns: the union of the residue classes $r(m)$ such that $p^k || a_{r(m)}$ if $k \geq 0$, and the union of the residue classes $r(m)$ such that $p^k || c_{r(m)}$ if $k \leq 0$. In this context, m denotes the modulus of `f`, and $a_{r(m)}$ and $c_{r(m)}$ denote the coefficients of `f` as introduced in Section 2.1.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);; # The Collatz mapping.
gap> [ Multpk(T,2,-1), Multpk(T,3,1) ];
[ Integers, 1(2) ]
gap> u := RcwaMapping([[3,0,5],[9,1,5],[3,-1,5],[9,-2,5],[9,4,5]]);;
gap> [ Multpk(u,3,0), Multpk(u,3,1), Multpk(u,3,2), Multpk(u,5,-1) ];
[ [  ], 0(5) U 2(5), Z \ 0(5) U 2(5), Integers ]
```

There are attributes `ClassWiseOrderPreservingOn`, `ClassWiseConstantOn` and `ClassWiseOrderReversingOn` which store the union of the residue classes (mod `Mod(f)`) on which an rcwa mapping `f` of $\mathbb{Z}$ or of a semilocalization thereof is class-wise order-preserving, class-wise constant or class-wise order-reversing, respectively.

Example

```
gap> List([ClassTransposition(1,2,0,4),ClassShift(2,3),
>          ClassReflection(2,5)],ClassWiseOrderPreservingOn);
[ Integers, Integers, Z \ 2(5) ]
```

Also there are attributes `ShiftsUpOn` and `ShiftsDownOn` which store the union of the residue classes (mod `Mod(f)`) on which an rcwa mapping `f` of $\mathbb{Z}$ induces affine mappings $n \mapsto n+c$ for $c > 0$, respectively, $c < 0$.

Finally, there are epimorphisms from the subgroup of RCWA($\mathbb{Z}$) formed by all class-wise order-preserving elements to ($\mathbb{Z}$,+) and from RCWA($\mathbb{Z}$) itself to the cyclic group of order 2, respectively:

2.4.4 Determinant (of an rcwa mapping of Z)

▷ `Determinant(f)` (method)

Returns: the determinant of the rcwa mapping `f` of $\mathbb{Z}$.

The *determinant* of an affine mapping $n \mapsto (an+b)/c$ whose source is a residue class $r(m)$ is defined by $b/|a|m$. This definition is extended additively to determinants of rcwa mappings.

Let f be an rcwa mapping of the integers, and let m denote its modulus. Using the notation $f|_{r(m)} : n \mapsto (a_{r(m)} \cdot n + b_{r(m)})/c_{r(m)}$ for the affine partial mappings, the *determinant* $\det(f)$ of f is given by

$$\sum_{r(m) \in \mathbb{Z}/m\mathbb{Z}} b_{r(m)}/(|a_{r(m)}| \cdot m).$$

The determinant mapping is an epimorphism from the group of all class-wise order-preserving rcwa permutations of $\mathbb{Z}$ to ($\mathbb{Z}$,+), see [Koh05], Theorem 2.11.9.

Example

```
gap> List([ClassTransposition(0,4,5,12),ClassShift(3,7)],Determinant);
[ 0, 1 ]
gap> Determinant(ClassTransposition(0,4,5,12)*ClassShift(3,7)^100);
100
```

2.4.5 Sign (of an rcwa permutation of Z)

▷ `Sign(g)` (attribute)

Returns: the sign of the rcwa permutation `g` of $\mathbb{Z}$.

Let σ be an rcwa permutation of the integers, and let m denote its modulus. Using the notation $\sigma|_{r(m)} : n \mapsto (a_{r(m)} \cdot n + b_{r(m)})/c_{r(m)}$ for the affine partial mappings, the *sign* of σ is defined by

$$(-1)^{\det(\sigma) + \sum_{r(m):\ a_{r(m)} < 0} \frac{m-2r}{m}}.$$

The sign mapping is an epimorphism from RCWA($\mathbb{Z}$) to the group $\mathbb{Z}^\times$ of units of $\mathbb{Z}$, see [Koh05], Theorem 2.12.8. Therefore the kernel of the sign mapping is a normal subgroup of RCWA($\mathbb{Z}$) of index 2. The simple group CT($\mathbb{Z}$) is a subgroup of this kernel.

Example

```
gap> List([ClassTransposition(3,4,2,6),
>          ClassShift(0,3),ClassReflection(2,5)],Sign);
[ 1, -1, -1 ]
```

2.5 Factoring residue-class-wise affine permutations

Factoring group elements into the members of some "nice" set of generators is often helpful. In this section we describe an operation which attempts to solve this problem for the group RCWA($\mathbb{Z}$). Elements of finitely generated rcwa groups can be factored into generators "as usual", see `PreImagesRepresentative` (3.2.3).

2.5.1 FactorizationIntoCSCRCT (for an rcwa permutation of Z)

▷ `FactorizationIntoCSCRCT(g)` (attribute)
▷ `Factorization(g)` (method)

Returns: a factorization of the rcwa permutation g of $\mathbb{Z}$ into class shifts, class reflections and class transpositions, provided that such a factorization exists and the method finds it.

The method may return `fail`, stop with an error message or run into an infinite loop. If it returns a result, this result is always correct.

The problem of obtaining a factorization as described is algorithmically difficult, and this factorization routine is currently perhaps the most sophisticated part of the RCWA package. Information about the progress of the factorization process can be obtained by setting the info level of the Info class `InfoRCWA` (9.5.1) to 2.

By default, prime switches (→ `PrimeSwitch` (2.5.2)) are taken as one factor. If the option *`ExpandPrimeSwitches`* is set, they are each decomposed into the 6 class transpositions given in the definition.

By default, the factoring process begins with splitting off factors from the right. This can be changed by setting the option *`Direction`* to `"from the left"`.

By default, a reasonably coarse respected partition of the integral mapping occurring in the final stage of the algorithm is computed. This can be suppressed by setting the option *`ShortenPartition`* equal to `false`.

By default, at the end it is checked whether the product of the determined factors indeed equals g. This check can be suppressed by setting the option *`NC`*.

Example

```
gap> Factorization(Comm(ClassShift(0,3)*ClassReflection(1,2),
>                       ClassShift(0,2)));
[ ClassReflection( 2(3) ), ClassShift( 2(6) )^-1, ( 0(6), 2(6) ),
  ( 0(6), 5(6) ) ]
```

For purposes of demonstrating the capabilities of the factorization routine, in Section 7.2 Collatz' permutation is factored. Lothar Collatz has investigated this permutation in 1932. Its cycle structure is unknown so far.

The permutations of the following kind play an important role in factoring rcwa permutations of $\mathbb{Z}$ into class shifts, class reflections and class transpositions:

2.5.2 PrimeSwitch (p)

▷ `PrimeSwitch(p)` (method)
▷ `PrimeSwitch(p, k)` (method)
▷ `PrimeSwitch(p, r, m)` (method)
▷ `PrimeSwitch(p, cl)` (method)

Returns: in the first form the *prime switch* $\sigma_p := \tau_{0(8),1(2p)} \cdot \tau_{4(8),-1(2p)} \cdot \tau_{0(4),1(2p)} \cdot \tau_{2(4),-1(2p)} \cdot \tau_{2(2p),1(4p)} \cdot \tau_{4(2p),2p+1(4p)}$, in the second form the restriction of σ_p by $n \mapsto kn$, and in the third and fourth form the *prime switch* $\sigma_{p,r(m)} := \tau_{r_1(m/2),r_2(m)} \cdot \tau_{r_2(m),r_1(pm/2)} \cdot \tau_{r(m/2),r_1(pm/2)}$. In the latter case, `cl` is the residue class $r(m)$, the residue r_1 is $1-(r \mod 2)$, and r_2 is defined by the equality $r(m) \cup r_2(m) = r(m/2)$.

For an odd prime p, the prime switch σ_p is an rcwa permutation of $\mathbb{Z}$ with modulus $4p$, multiplier p and divisor 2. The prime switch $\sigma_{p,r(m)}$ has multiplier p and divisor 2, and the class where the multiplication by p occurs is just $r(m)$. The key mathematical property of a prime switch is that it is a product of class transpositions whose multiplier and divisor are coprime.

Prime switches can be distinguished from other rcwa mappings by their GAP property `IsPrimeSwitch`.

Example

```
gap> Display(PrimeSwitch(3));

Wild rcwa permutation of Z with modulus 12

        /
        | (3n+4)/2 if n in 2(4)
        | n-1      if n in 5(6) U 8(12)
        | n+1      if n in 1(6)
 n |-> <  n/2      if n in 0(12)
        | n-3      if n in 4(12)
        | n        if n in 3(6)
        |
        \

gap> Display(PrimeSwitch(3):AsClassMapping);

Wild rcwa permutation of Z with modulus 12

  0(12) -> 0(6)  loop
   1(6) -> 2(6)
   2(4) -> 5(6)
   3(6) -> 3(6)  id
  4(12) -> 1(12)
   5(6) -> 4(6)
  8(12) -> 7(12)
```

```
gap> Factorization(PrimeSwitch(3));
[ ( 1(6), 0(8) ), ( 5(6), 4(8) ), ( 0(4), 1(6) ), ( 2(4), 5(6) ),
  ( 2(6), 1(12) ), ( 4(6), 7(12) ) ]
gap> Display(PrimeSwitch(5,3,4));

Wild rcwa permutation of Z with modulus 20

        /
        | n+1     if n in 0(2)
        | 5n-5    if n in 3(4)
 n |-> <  (n-1)/2 if n in 1(4) \ 1(20)
        | n-1     if n in 1(20)
        |
        \

gap> Multpk(PrimeSwitch(5,3,4),5,1);
3(4)
gap> PrimeSwitch(5,3,4) = PrimeSwitch(5,ResidueClass(3,4));
true
gap> Factorization(PrimeSwitch(5,3,4));
[ ( 0(2), 1(4) ), ( 1(4), 0(10) ), ( 1(2), 0(10) ) ]
```

Obtaining a factorization of an rcwa permutation into class shifts, class reflections and class transpositions is particularly difficult if multiplier and divisor are coprime. A prototype of permutations which have this property has been introduced in a different context in [Kel99]:

2.5.3 mKnot (for an odd integer)

▷ mKnot(*m*) (function)

Returns: the permutation g_m as defined in [Kel99].

The argument *m* must be an odd integer greater than 1.

Example

```
gap> Display(mKnot(5));

Wild rcwa permutation of Z with modulus 5

        /
        | 6n/5      if n in 0(5)
        | (4n+1)/5 if n in 1(5)
 n |-> <  (6n-2)/5 if n in 2(5)
        | (4n+3)/5 if n in 3(5)
        | (6n-4)/5 if n in 4(5)
        \
```

In his article, Timothy P. Keller shows that a permutation of this type cannot have infinitely many cycles of any given finite length.

2.6 Extracting roots of residue-class-wise affine mappings

2.6.1 Root (k-th root of an rcwa mapping)

▷ Root(*f*, *k*) (method)

Returns: an rcwa mapping g such that `g^k=f`, provided that such a mapping exists and that there is a method available which can determine it.

Currently, extracting roots is implemented for rcwa permutations of finite order.

Example

```
gap> Root(ClassTransposition(0,2,1,2),100);
( 0(8), 2(8), 4(8), 6(8), 1(8), 3(8), 5(8), 7(8) )
gap> Display(last:CycleNotation:=false);

Tame rcwa permutation of Z with modulus 8

        /
        | n+2 if n in Z \ 6(8) U 7(8)
 n |-> <  n-5 if n in 6(8)
        | n-7 if n in 7(8)
        \

gap> last^100 = ClassTransposition(0,2,1,2);
true
```

2.7 Special functions for non-bijective mappings

2.7.1 RightInverse (of an injective rcwa mapping)

▷ RightInverse(*f*) (attribute)

Returns: a right inverse of the injective rcwa mapping *f*, i.e. a mapping g such that $fg = 1$.

Example

```
gap> twice := 2*IdentityRcwaMappingOfZ;
Rcwa mapping of Z: n -> 2n
gap> twice * RightInverse(twice);
IdentityMapping( Integers )
```

2.7.2 CommonRightInverse (of two injective rcwa mappings)

▷ CommonRightInverse(*l*, *r*) (operation)

Returns: a mapping d such that $ld = rd = 1$.

The mappings *l* and *r* must be injective, and their images must form a partition of their source.

Example

```
gap> twice := 2*IdentityRcwaMappingOfZ; twiceplus1 := twice+1;
Rcwa mapping of Z: n -> 2n
Rcwa mapping of Z: n -> 2n + 1
```

```
gap> Display(CommonRightInverse(twice,twiceplus1));

Rcwa mapping of Z with modulus 2

        /
        | n/2     if n in 0(2)
 n |-> <  (n-1)/2 if n in 1(2)
        |
        \
```

2.7.3 ImageDensity (of an rcwa mapping)

▷ ImageDensity(*f*) (attribute)

Returns: the *image density* of the rcwa mapping *f*.

In the notation introduced in the definition of an rcwa mapping, the *image density* of an rcwa mapping f is defined by $\frac{1}{m}\sum_{r(m)\in R/mR}|R/c_{r(m)}R|/|R/a_{r(m)}R|$. The image density of an injective rcwa mapping is ≤ 1, and the image density of a surjective rcwa mapping is ≥ 1 (this can be seen easily). Thus in particular the image density of a bijective rcwa mapping is 1.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);; # The Collatz mapping.
gap> List( [ T, ClassShift(0,1), RcwaMapping([[2,0,1]]) ], ImageDensity );
[ 4/3, 1, 1/2 ]
```

Given an rcwa mapping f, the function InjectiveAsMappingFrom returns a set S such that the restriction of f to S is injective, and such that the image of S under f is the entire image of f.

Example

```
gap> InjectiveAsMappingFrom(T);
0(2)
```

2.8 On trajectories and cycles of residue-class-wise affine mappings

RCWA provides various methods to compute trajectories of rcwa mappings:

2.8.1 Trajectory (methods for rcwa mappings)

▷ Trajectory(*f*, *n*, *length*) (method)
▷ Trajectory(*f*, *n*, *length*, *m*) (method)
▷ Trajectory(*f*, *n*, *terminal*) (method)
▷ Trajectory(*f*, *n*, *terminal*, *m*) (method)

Returns: the first *length* iterates in the trajectory of the rcwa mapping *f* starting at *n*, respectively the initial part of the trajectory of the rcwa mapping *f* starting at *n* which ends at the first occurrence of an iterate in the set *terminal*. If the argument *m* is given, the iterates are reduced (mod *m*).

To save memory when computing long trajectories containing huge iterates, the reduction (mod *m*) is done each time before storing an iterate. In place of the ring element *n*, the methods also accept a finite set of ring elements or a union of residue classes.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);; # The Collatz mapping.
gap> Trajectory(T,27,15); Trajectory(T,27,20,5);
[ 27, 41, 62, 31, 47, 71, 107, 161, 242, 121, 182, 91, 137, 206, 103 ]
[ 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 3, 0, 3, 0, 0, 3 ]
gap> Trajectory(T,15,[1]); Trajectory(T,15,[1],2);
[ 15, 23, 35, 53, 80, 40, 20, 10, 5, 8, 4, 2, 1 ]
[ 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1 ]
gap> Trajectory(T,ResidueClass(Integers,3,0),Integers);
[ 0(3), 0(3) U 5(9), 0(3) U 5(9) U 7(9) U 8(27),
  <union of 20 residue classes (mod 27) (6 classes)>,
  <union of 73 residue classes (mod 81)>, Z \ 10(81) U 37(81), Integers ]
```

2.8.2 Trajectory (methods for rcwa mappings – “accumulated coefficients”)

▷ `Trajectory(f, n, length, whichcoeffs)` (method)
▷ `Trajectory(f, n, terminal, whichcoeffs)` (method)

Returns: either the list `c` of triples of coprime coefficients such that for any `k` it holds that `n^(f^(k-1)) = (c[k][1]*n + c[k][2])/c[k][3]` or the last entry of that list, depending on whether *whichcoeffs* is `"AllCoeffs"` or `"LastCoeffs"`.

The meanings of the arguments *length* and *terminal* are the same as in the methods for the operation `Trajectory` described above. In general, computing only the last coefficient triple (*whichcoeffs* = `"LastCoeffs"`) needs considerably less memory than computing the entire list.

Example

```
gap> Trajectory(T,27,[1],"LastCoeffs");
[ 36472996377170786403, 195820718533800070543, 1180591620717411303424 ]
gap> (last[1]*27+last[2])/last[3];
1
```

When dealing with problems like the $3n+1$-Conjecture or when determining the degree of transitivity of the natural action of an rcwa group on its underlying ring, an important task is to determine the residue classes whose elements get larger or smaller when applying a given rcwa mapping:

2.8.3 IncreasingOn & DecreasingOn (for an rcwa mapping)

▷ `IncreasingOn(f)` (attribute)
▷ `DecreasingOn(f)` (attribute)

Returns: the union of all residue classes $r(m)$ such that $|R/a_{r(m)}R| > |R/c_{r(m)}R|$ or $|R/a_{r(m)}R| < |R/c_{r(m)}R|$, respectively, where R denotes the source, m denotes the modulus and $a_{r(m)}$, $b_{r(m)}$ and $c_{r(m)}$ denote the coefficients of *f* as introduced in Section 2.1.

If the argument is an rcwa mapping of $\mathbb{Z}$ in sparse representation, an option `classes` is interpreted; if set, the step of forming the union of the residue classes in question is omitted, and the list of residue classes is returned instead of their union. This may save time and memory if the modulus is large.

Example

```
gap> List([1..3],k->IncreasingOn(T^k));
[ 1(2), 3(4), 3(4) U 1(8) U 6(8) ]
gap> List([1..3],k->DecreasingOn(T^k));
[ 0(2), Z \ 3(4), 0(4) U 2(8) U 5(8) ]
gap> a := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);; # Collatz' permutation
gap> List([-2..2],k->IncreasingOn(a^k));
[ Z \ 1(8) U 7(8), 0(2), [  ], Z \ 0(3), 1(9) U 4(9) U 5(9) U 8(9) ]
```

We assign certain directed graphs to rcwa mappings, which encode the order in which trajectories may traverse the residue classes modulo some modulus:

2.8.4 TransitionGraph (for an rcwa mapping and a modulus)

▷ `TransitionGraph(f, m)` (operation)

Returns: the transition graph of the rcwa mapping `f` for modulus `m`.

The *transition graph* $\Gamma_{f,m}$ of f for modulus m is defined as follows:

1. The vertices are the residue classes (mod m).
2. There is an edge from $r_1(m)$ to $r_2(m)$ if and only if there is some $n \in r_1(m)$ such that $n^f \in r_2(m)$.

The assignment of the residue classes (mod m) to the vertices of the graph corresponds to the ordering of the residues in `AllResidues(Source(f),m)`. The result is returned in the format used by the package GRAPE [Soi16].

There are a couple of operations and attributes which are based on these graphs:

2.8.5 OrbitsModulo (for an rcwa mapping and a modulus)

▷ `OrbitsModulo(f, m)` (operation)

Returns: the partition of `AllResidues(Source(f),m)` corresponding to the weakly connected components of the transition graph of the rcwa mapping `f` for modulus `m`.

Example

```
gap> OrbitsModulo(ClassTransposition(0,2,1,4),8);
[ [ 0, 1, 4 ], [ 2, 5, 6 ], [ 3 ], [ 7 ] ]
```

2.8.6 FactorizationOnConnectedComponents (for an rcwa mapping and a modulus)

▷ `FactorizationOnConnectedComponents(f, m)` (operation)

Returns: the set of restrictions of the rcwa mapping `f` to the weakly connected components of its transition graph $\Gamma_{f,m}$.

The product of the returned mappings is `f`. They have pairwise disjoint supports, hence any two of them commute.

Example

```
gap> sigma := ClassTransposition(1,4,2,4)  * ClassTransposition(1,4,3,4)
>          * ClassTransposition(3,9,6,18) * ClassTransposition(1,6,3,9);;
gap> List(FactorizationOnConnectedComponents(sigma,36),Support);
[ 33(36) U 34(36) U 35(36), 9(36) U 10(36) U 11(36),
  <union of 23 residue classes (mod 36)> \ [ -6, 3 ] ]
```

2.8.7 TransitionMatrix (for an rcwa mapping and a modulus)

▷ `TransitionMatrix(f, m)` (operation)

Returns: the transition matrix of the rcwa mapping `f` for modulus `m`.

Let M be this matrix. Then for any two residue classes $r_1(m), r_2(m) \in R/mR$, the entry $M_{r_1(m),r_2(m)}$ is defined by

$$M_{r_1(m),r_2(m)} \;:=\; \frac{|R/mR|}{|R/\hat{m}R|} \cdot \left|\left\{r(\hat{m}) \in R/\hat{m}R \middle| \, r \in r_1(m) \wedge r^f \in r_2(m)\right\}\right|,$$

where $\hat{m}$ is the product of `m` and the square of the modulus of `f`. The assignment of the residue classes (mod `m`) to the rows and columns of the matrix corresponds to the ordering of the residues in `AllResidues(Source(f),m)`.

The transition matrix is a weighted adjacency matrix of the corresponding transition graph `TransitionGraph(f,m)`. The sums of the rows of a transition matrix are always equal to 1.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);; # The Collatz mapping.
gap> Display(TransitionMatrix(T^3,3));
[ [  1/8,  1/4,  5/8 ],
  [    0,  1/4,  3/4 ],
  [    0,  3/8,  5/8 ] ]
```

2.8.8 Sources & Sinks (of an rcwa mapping)

▷ `Sources(f)` (attribute)

▷ `Sinks(f)` (attribute)

Returns: a list of unions of residue classes modulo the modulus m of the rcwa mapping `f`, as described below.

The returned list contains an entry for any strongly connected component of the transition graph of `f` for modulus `Mod(f)` which has only outgoing edges (“source”) or which has only ingoing edges (“sink”), respectively. The list entry corresponding to such a component is the union of the vertices belonging to it.

Example

```
gap> g := ClassTransposition(0,2,1,2)*ClassTransposition(0,2,1,4);;
gap> Sources(g); Sinks(g);
[ 0(4) ]
[ 1(4) ]
```

2.8.9 Loops (of an rcwa mapping)

▷ `Loops(f)` (attribute)

Returns: if `f` is bijective, the list of non-isolated vertices of the transition graph of `f` for modulus `Mod(f)` which carry a loop. In general, the list of vertices of that transition graph which carry a loop, but which `f` does not fix setwise.

The returned list may also include supersets of the named residue classes instead if `f` is affine even on these.

Example

```
gap> Loops(ClassTransposition(0,2,1,2)*ClassTransposition(0,2,1,4));
[ 0(4), 1(4) ]
```

There is a nice invariant of trajectories of the Collatz mapping:

2.8.10 GluckTaylorInvariant (of a trajectory)

▷ `GluckTaylorInvariant(a)` (function)

Returns: the invariant defined in [GT02]. This is $(\sum_{i=1}^{l} a_i \cdot a_{i \mod l+1})/(\sum_{i=1}^{l} a_i^2)$, where l denotes the length of `a`.

The argument `a` must be a list of integers. In [GT02] it is shown that if `a` is a trajectory of the 'original' Collatz mapping $n \mapsto (n/2$ if n even, $3n+1$ if n odd) starting at an odd integer ≥ 3 and ending at 1, then the invariant lies in the interval $]9/13, 5/7[$.

Example

```
gap> C := RcwaMapping([[1,0,2],[3,1,1]]);;
gap> List([3,5..49],n->Float(GluckTaylorInvariant(Trajectory(C,n,[1]))));
[ 0.701053, 0.696721, 0.708528, 0.707684, 0.706635, 0.695636, 0.711769,
  0.699714, 0.707409, 0.693833, 0.710432, 0.706294, 0.714242, 0.699935,
  0.714242, 0.705383, 0.706591, 0.698198, 0.712222, 0.714242, 0.709048,
  0.69612, 0.714241, 0.701076 ]
```

Quite often one can make certain "educated guesses" on the overall behaviour of the trajectories of a given rcwa mapping. For example it is reasonably straightforward to make the conjecture that all trajectories of the Collatz mapping eventually enter the finite set $\{-136,-91,-82,-68,-61,-55,-41,-37,-34,-25,-17,-10,-7,-5,-1,0,1,2\}$, or that "on average" the next number in a trajectory of the Collatz mapping is smaller than the preceding one by a factor of $\sqrt{3}/2$. However it is clear that such guesses can be wrong, and that they therefore cannot be used to prove anything. Nevertheless they can sometimes be useful:

2.8.11 LikelyContractionCentre (of an rcwa mapping)

▷ `LikelyContractionCentre(f, maxn, bound)` (operation)

Returns: a list of ring elements (see below).

This operation tries to compute the *contraction centre* of the rcwa mapping `f`. Assuming its existence this is the unique finite subset S_0 of the source of `f` on which `f` induces a permutation and which intersects non-trivially with any trajectory of `f`. The mapping `f` is assumed to be *contracting*, i.e. to have such a contraction centre. As in general contraction centres are likely not computable, the methods for this operation are probabilistic and may return wrong results. The argument *maxn* is a bound on the starting value and *bound* is a bound on the elements of the trajectories to be searched. If the limit *bound* is exceeded, an Info message on Info level 3 of `InfoRCWA` is given.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);; # The Collatz mapping.
gap> S0 := LikelyContractionCentre(T,100,1000);
#I  Warning: `LikelyContractionCentre' is highly probabilistic.
The returned result can only be regarded as a rough guess.
See ?LikelyContractionCentre for more information.
[ -136, -91, -82, -68, -61, -55, -41, -37, -34, -25, -17, -10, -7, -5,
  -1, 0, 1, 2 ]
```

2.8.12 GuessedDivergence (of an rcwa mapping)

▷ `GuessedDivergence(f)` (operation)

Returns: a floating point value which is intended to be a rough guess on how fast the trajectories of the rcwa mapping *f* diverge (return value greater than 1) or converge (return value smaller than 1).

Nothing particular is guaranteed.

Example

```
gap> GuessedDivergence(T);
#I  Warning: GuessedDivergence: no particular return value is guaranteed.
0.866025
```

2.9 Saving memory – the sparse representation of rcwa mappings

It is quite common that an rcwa mapping with large modulus has only few distinct affine partial mappings. In this case the "standard" representation which stores a coefficient triple for each residue class modulo the modulus is unsuitable. For this reason there is a second representation of rcwa mappings, the "sparse" representation. Depending on the rcwa mappings involved, using this representation may speed up computations and reduce memory requirements by orders of magnitude. For rcwa mappings with almost as many distinct affine partial mappings as there are residue classes modulo the modulus, using sparse representation makes computations slower and more memory-consuming. Presently, the sparse representation is only available for rcwa mappings of $\mathbb{Z}$.

The sparse representation of an rcwa mapping consists of the modulus and a list of 5-tuples $(r, m, a_{r(m)}, b_{r(m)}, c_{r(m)})$ of integers. Any such 5-tuple specifies the coefficients of the restriction $n \mapsto (a_{r(m)} \cdot n + b_{r(m)})/c_{r(m)}$ of the mapping to a residue class $r(m)$. The $r(m)$ are chosen to form the coarsest possible partition of $\mathbb{Z}$ into residue classes such that the restriction of the mapping to any of them is affine. Also the list of coefficient tuples is sorted, all $c_{r(m)}$ are positive and $\gcd(c_{r(m)}, \gcd(a_{r(m)}, b_{r(m)})) = 1$. This way the coefficient list of an rcwa mapping of $\mathbb{Z}$ is unique.

Changing the representation of rcwa mappings does not change their behaviour with respect to "=" and "<" The product of two rcwa mappings in sparse representation is in sparse representation again, just like the product of two rcwa mappings in standard representation is in standard representation. Also, inverses are in the same representation. The product of two rcwa mappings in different representation may be in any of the representations of the factors.

2.9.1 SparseRepresentation (of an rcwa mapping)

▷ SparseRepresentation(*f*) (operation)
▷ SparseRep(*f*) (operation)
▷ StandardRepresentation(*f*) (operation)
▷ StandardRep(*f*) (operation)

Returns: the rcwa mapping *f* in sparse, respectively, standard representation.

Appropriate attribute values and properties are copied over to the rcwa mapping in the "new" representation.

Example

```
gap> a := ClassTransposition(1,2,4,6);
( 1(2), 4(6) )
gap> b := ClassTransposition(1,3,2,6);
( 1(3), 2(6) )
gap> c := ClassTransposition(2,3,4,6);
( 2(3), 4(6) )
gap> g := (b*a*c)^2*a;
<rcwa permutation of Z with modulus 288>
gap> h := SparseRep(g);
<rcwa permutation of Z with modulus 288 and 21 affine parts>
gap> g = h;
true
gap> Coefficients(h);
[ [ 0, 6, 1, 0, 1 ], [ 1, 3, 16, -1, 3 ], [ 2, 96, 9, 14, 16 ],
  [ 3, 24, 9, 5, 4 ], [ 5, 24, 3, 1, 4 ], [ 8, 36, 2, -7, 9 ],
  [ 9, 48, 27, 29, 8 ], [ 11, 24, 9, 5, 4 ], [ 14, 48, 27, 38, 8 ],
  [ 15, 24, 27, 19, 4 ], [ 17, 48, 9, 7, 8 ], [ 20, 72, 3, 4, 4 ],
  [ 21, 24, 1, -3, 6 ], [ 23, 24, 27, 19, 4 ], [ 26, 48, 3, 2, 8 ],
  [ 32, 36, 4, -11, 9 ], [ 33, 48, 9, 7, 8 ], [ 38, 48, 9, 10, 8 ],
  [ 41, 48, 27, 29, 8 ], [ 50, 96, 27, 58, 16 ], [ 56, 72, 1, 0, 4 ] ]
gap> h^2;
<rcwa permutation of Z with modulus 13824 and 71 affine parts>
gap> h^3;
<rcwa permutation of Z with modulus 663552 and 201 affine parts>
```

Memory consumption may differ a lot between sparse- and standard representation:

Example

```
gap> MemoryUsage(h^3);               # on a 64-bit machine
18254
gap> MemoryUsage(StandardRep(h^3));  # on a 64-bit machine
42467894
```

2.10 The categories and families of rcwa mappings

2.10.1 IsRcwaMapping

▷ `IsRcwaMapping(f)` (filter)
▷ `IsRcwaMappingOfZ(f)` (filter)
▷ `IsRcwaMappingOfZ_pi(f)` (filter)
▷ `IsRcwaMappingOfGFqx(f)` (filter)

Returns: `true` if *f* is an rcwa mapping, an rcwa mapping of the ring of integers, an rcwa mapping of a semilocalization of the ring of integers or an rcwa mapping of a polynomial ring in one variable over a finite field, respectively, and `false` otherwise.

Often the same methods can be used for rcwa mappings of the ring of integers and of its semilocalizations. For this reason there is a category `IsRcwaMappingOfZOrZ_pi` which is the union of `IsRcwaMappingOfZ` and `IsRcwaMappingOfZ_pi`. The internal representation of rcwa mappings is called `IsRcwaMappingStandardRep`. There are methods available for `ExtRepOfObj` and `ObjByExtRep`.

2.10.2 RcwaMappingsFamily (of a ring)

▷ `RcwaMappingsFamily(R)` (function)

Returns: the family of rcwa mappings of the ring *R*.

Chapter 3

Residue-Class-Wise Affine Groups

In this chapter, we describe how to construct residue-class-wise affine groups and how to compute with them.

3.1 Constructing residue-class-wise affine groups

As any other groups in GAP, residue-class-wise affine (rcwa-) groups can be constructed by `Group`, `GroupByGenerators` or `GroupWithGenerators`.

Example

```
gap> G := Group(ClassTransposition(0,2,1,4),ClassShift(0,5));
<rcwa group over Z with 2 generators>
gap> IsTame(G); Size(G); IsSolvable(G); IsPerfect(G);
true
infinity
false
false
```

An rcwa group isomorphic to a given group can be obtained by taking the image of a faithful rcwa representation:

3.1.1 IsomorphismRcwaGroup (for a group, over a given ring)

▷ `IsomorphismRcwaGroup(G, R)` (attribute)
▷ `IsomorphismRcwaGroup(G)` (attribute)

Returns: a monomorphism from the group G to RCWA(R) or to RCWA($\mathbb{Z}$), respectively.

The best-supported case is $R = \mathbb{Z}$. Currently there are methods available for finite groups, for free products of finite groups and for free groups. The method for free products of finite groups uses the Table-Tennis Lemma (cf. e.g. Section II.B. in [dlH00]), and the method for free groups uses an adaptation of the construction given on page 27 in [dlH00] from PSL(2,$\mathbb{C}$) to RCWA($\mathbb{Z}$).

Example

```
gap> F := FreeProduct(Group((1,2)(3,4),(1,3)(2,4)),Group((1,2,3)),
>                     SymmetricGroup(3));
<fp group on the generators [ f1, f2, f3, f4, f5 ]>
```

```
gap> IsomorphismRcwaGroup(F);
[ f1, f2, f3, f4, f5 ] -> [ <rcwa permutation of Z with modulus 12>,
  <rcwa permutation of Z with modulus 24>,
  <rcwa permutation of Z with modulus 12>,
  <rcwa permutation of Z with modulus 72>,
  <rcwa permutation of Z with modulus 36> ]
gap> IsomorphismRcwaGroup(FreeGroup(2));
[ f1, f2 ] -> [ <wild rcwa permutation of Z with modulus 8>,
  <wild rcwa permutation of Z with modulus 8> ]
gap> F2 := Image(last);
<wild rcwa group over Z with 2 generators>
```

Further, new rcwa groups can be constructed from given ones by taking direct products and by taking wreath products with finite groups or with the infinite cyclic group:

3.1.2 DirectProduct (for rcwa groups over Z)

▷ `DirectProduct(G1, G2, ...)` (method)

Returns: an rcwa group isomorphic to the direct product of the rcwa groups over $\mathbb{Z}$ given as arguments.

There is certainly no unique or canonical way to embed a direct product of rcwa groups into RCWA($\mathbb{Z}$). This method chooses to embed the groups `G1`, `G2`, `G3` ... via restrictions by $n \mapsto mn$, $n \mapsto mn+1$, $n \mapsto mn+2$ ... (→ `Restriction` (3.1.6)), where m denotes the number of groups given as arguments.

Example

```
gap> F2 := Image(IsomorphismRcwaGroup(FreeGroup(2)));;
gap> F2xF2 := DirectProduct(F2,F2);
<wild rcwa group over Z with 4 generators>
gap> Image(Projection(F2xF2,1)) = F2;
true
```

3.1.3 WreathProduct (for an rcwa group over Z, with a permutation group or ($\mathbb{Z}$,+))

▷ `WreathProduct(G, P)` (method)
▷ `WreathProduct(G, Z)` (method)

Returns: an rcwa group isomorphic to the wreath product of the rcwa group `G` over $\mathbb{Z}$ with the finite permutation group `P` or with the infinite cyclic group `Z`, respectively.

The first-mentioned method embeds the `NrMovedPoints(P)`th direct power of `G` using the method for `DirectProduct`, and lets the permutation group `P` act naturally on the set of residue classes modulo `NrMovedPoints(P)`. The second-mentioned method restricts (→ `Restriction` (3.1.6)) the group `G` to the residue class 3(4), and maps the generator of the infinite cyclic group `Z` to `ClassTransposition(0,2,1,2) * ClassTransposition(0,2,1,4)`.

Example

```
gap> F2 := Image(IsomorphismRcwaGroup(FreeGroup(2)));;
gap> F2wrA5 := WreathProduct(F2,AlternatingGroup(5));;
```

```
gap> Embedding(F2wrA5,1);
[ <wild rcwa permutation of Z with modulus 8>,
  <wild rcwa permutation of Z with modulus 8> ] ->
[ <wild rcwa permutation of Z with modulus 40>,
  <wild rcwa permutation of Z with modulus 40> ]
gap> Embedding(F2wrA5,2);
[ (1,2,3,4,5), (3,4,5) ] -> [ ( 0(5), 1(5), 2(5), 3(5), 4(5) ),
  ( 2(5), 3(5), 4(5) ) ]
gap> ZwrZ := WreathProduct(Group(ClassShift(0,1)),Group(ClassShift(0,1)));
<wild rcwa group over Z with 2 generators>
gap> Embedding(ZwrZ,1);
[ ClassShift( Z ) ] ->
[ <tame rcwa permutation of Z with modulus 4, of order infinity> ]
gap> Embedding(ZwrZ,2);
[ ClassShift( Z ) ] -> [ <wild rcwa permutation of Z with modulus 4> ]
```

Also, rcwa groups can be obtained as particular extensions of finite permutation groups:

3.1.4 MergerExtension (for finite permutation groups)

▷ MergerExtension(*G, points, point*) (operation)

Returns: roughly spoken, an extension of *G* by an involution which “merges” *points* into *point*.

The arguments of this operation are a finite permutation group *G*, a set *points* of points moved by *G* and a single point *point* moved by *G* which is not in *points*.

Let n be the largest moved point of *G*, and let H be the tame subgroup of CT($\mathbb{Z}$) which respects the partition $\mathscr{P}$ of $\mathbb{Z}$ into the residue classes (mod n), and which acts on $\mathscr{P}$ as *G* acts on $\{1,\dots,n\}$. Further assume that *points* $= \{p_1,\dots,p_k\}$ and *point* $= p$, and put $r_i := p_i - 1$, $i = 1,\dots,k$ and $r := p-1$. Now let σ be the product of the class transpositions $\tau_{r_i(n),r+(i-1)n(kn)}$, $i = 1,\dots,k$. The group returned by this operation is the extension of H by the involution σ. – On first reading, this may look a little complicated, but really the code of the method is only about half as long as this description.

Example

```
gap> # First example -- a group isomorphic to PSL(2,Z):
gap> G := MergerExtension(Group((1,2,3)),[1,2],3);
<rcwa group over Z with 2 generators>
gap> Size(G);
infinity
gap> GeneratorsOfGroup(G);
[ ( 0(3), 1(3), 2(3) ), ( 0(3), 2(6) ) ( 1(3), 5(6) ) ]
gap> B := Ball(G,One(G),6:Spheres);;
gap> List(B,Length);
[ 1, 3, 4, 6, 8, 12, 16 ]
gap> #
gap> # Second example -- a group isomorphic to Thompson's group V:
gap> G := MergerExtension(Group((1,2,3,4),(1,2)),[1,2],3);
<rcwa group over Z with 3 generators>
gap> Size(G);
infinity
```

```
gap> GeneratorsOfGroup(G);
[ ( 0(4), 1(4), 2(4), 3(4) ), ( 0(4), 1(4) ),
  ( 0(4), 2(8) ) ( 1(4), 6(8) ) ]
gap> B := Ball(G,One(G),6:Spheres);;
gap> List(B,Length);
[ 1, 4, 11, 28, 69, 170, 413 ]
gap> G = Group(List([[0,2,1,2],[1,2,2,4],[0,2,1,4],[1,4,2,4]],
>                   ClassTransposition));
true
```

It is also possible to build an rcwa group from a list of residue classes:

3.1.5 GroupByResidueClasses (the group ‘permuting a given list of residue classes’)

▷ `GroupByResidueClasses(classes)` (function)

Returns: the group which is generated by all class transpositions which interchange disjoint residue classes in `classes`.

The argument `classes` must be a list of residue classes.

If the residue classes in `classes` are pairwise disjoint, then the returned group is the symmetric group on `classes`. If any two residue classes in `classes` intersect non-trivially, then the returned group is trivial. In many other cases, the returned group is infinite.

Example

```
gap> G := GroupByResidueClasses(List([[0,2],[0,4],[1,4],[2,4],[3,4]],
>                                    ResidueClass));
<rcwa group over Z with 8 generators>
gap> H := Group(List([[0,2,1,2],[1,2,2,4],[0,2,1,4],[1,4,2,4]],
>                    ClassTransposition)); # Thompson's group V
<(0(2),1(2)),(1(2),2(4)),(0(2),1(4)),(1(4),2(4))>
gap> G = H;
true
```

Various ways to construct rcwa groups are based on certain monomorphisms from the group RCWA(R) into itself. Examples are the constructions of direct products and wreath products described above. The support of the image of such a monomorphism is the image of a given injective rcwa mapping. For this reason, these monomorphisms are called *restriction monomorphisms*. The following operation computes images of rcwa mappings and -groups under these embeddings of RCWA(R) into itself:

3.1.6 Restriction (of an rcwa mapping or -group, by an injective rcwa mapping)

▷ `Restriction(g, f)` (operation)

▷ `Restriction(G, f)` (operation)

Returns: the restriction of the rcwa mapping `g` (respectively the rcwa group `G`) by the injective rcwa mapping `f`.

By definition, the *restriction* g_f of an rcwa mapping g by an injective rcwa mapping `f` is the unique rcwa mapping which satisfies the equation $f \cdot g_f = g \cdot f$ and which fixes the complement of the image of `f` pointwise. If `f` is bijective, the restriction of `g` by `f` is just the conjugate of `g` under `f`.

The *restriction* of an rcwa group `G` by an injective rcwa mapping `f` is defined as the group whose elements are the restrictions of the elements of `G` by `f`. The restriction of `G` by `f` acts on the image of `f` and fixes its complement pointwise.

Example

```
gap> F2tilde := Restriction(F2,RcwaMapping([[5,3,1]]));
<wild rcwa group over Z with 2 generators>
gap> Support(F2tilde);
3(5)
```

3.1.7 Induction (of an rcwa mapping or -group, by an injective rcwa mapping)

▷ `Induction(g, f)` (operation)
▷ `Induction(G, f)` (operation)

Returns: the induction of the rcwa mapping g (respectively the rcwa group `G`) by the injective rcwa mapping `f`.

Induction is the right inverse of restriction, i.e. it is `Induction(Restriction(g,f),f) = g` and `Induction(Restriction(G,f),f) = G`. The mapping g respectively the group `G` must not move points outside the image of `f`.

Example

```
gap> Induction(F2tilde,RcwaMapping([[5,3,1]])) = F2;
true
```

Once having constructed an rcwa group, it is sometimes possible to obtain a smaller generating set by the operation `SmallGeneratingSet`.

There are methods for the operations `View`, `Display`, `Print` and `String` which are applicable to rcwa groups.

Basic attributes of an rcwa group which are derived from the coefficients of its elements are `Modulus`, `Multiplier`, `Divisor` and `PrimeSet`. The *modulus* of an rcwa group is the lcm of the moduli of its elements if such an lcm exists, i.e. if the group is tame, and 0 otherwise. The *multiplier* respectively *divisor* of an rcwa group is the lcm of the multipliers respectively divisors of its elements in case such an lcm exists and ∞ otherwise. The *prime set* of an rcwa group is the union of the prime sets of its elements. There are shorthands `Mod`, `Mult` and `Div` defined for `Modulus`, `Multiplier` and `Divisor`, respectively. An rcwa group is called *class-wise translating*, *integral* or *class-wise order-preserving* if all of its elements are so. There are corresponding methods available for `IsClassWiseTranslating`, `IsIntegral` and `IsClassWiseOrderPreserving`. There is a property `IsSignPreserving`, which indicates whether a given rcwa group over $\mathbb{Z}$ acts on the set of nonnegative integers. The latter holds for any subgroup of CT($\mathbb{Z}$) (cf. below).

Example

```
gap> G := Group(ClassTransposition(0,2,1,2),ClassTransposition(1,3,2,6),
>               ClassReflection(2,4));
<rcwa group over Z with 3 generators>
gap> List([Modulus,Multiplier,Divisor,PrimeSet,IsClassWiseTranslating,
>          IsIntegral,IsClassWiseOrderPreserving,IsSignPreserving],f->f(G));
[ 24, 2, 2, [ 2, 3 ], false, false, false, false ]
```

All rcwa groups over a ring R are subgroups of RCWA(R). The group RCWA(R) itself is not finitely generated, thus cannot be constructed as described above. It is handled as a special case:

3.1.8 RCWA (the group formed by all rcwa permutations of a ring)

▷ RCWA(*R*) (function)

Returns: the group RCWA(*R*) of all residue-class-wise affine permutations of the ring *R*.

Example
```
gap> RCWA_Z := RCWA(Integers);
RCWA(Z)
gap> IsSubgroup(RCWA_Z,G);
true
```

Examples of rcwa permutations can be obtained via `Random(RCWA(R))`, see Section 3.5. The number of conjugacy classes of RCWA($\mathbb{Z}$) of elements of given order is known, cf. Corollary 2.7.1 (b) in [Koh05]. It can be determined by the function `NrConjugacyClassesOfRCWAZOfOrder`:

Example
```
gap> List([2,105],NrConjugacyClassesOfRCWAZOfOrder);
[ infinity, 218 ]
```

We denote the group which is generated by all class transpositions of the ring R by CT(R). This group is handled as a special case as well:

3.1.9 CT (the group generated by all class transpositions of a ring)

▷ CT(*R*) (function)

▷ CT(*P*, *Integers*) (function)

Returns: the group CT(*R*) which is generated by all class transpositions of the ring *R*, respectively, the group CT(*P*,$\mathbb{Z}$) which is generated by all class transpositions of $\mathbb{Z}$ which interchange residue classes whose moduli have only prime factors in the finite set *P*.

Example
```
gap> CT_Z := CT(Integers);
CT(Z)
gap> IsSimple(CT_Z); # One of a number of stored attributes/properties.
true
gap> V := CT([2],Integers);
CT_[ 2 ](Z)
gap> GeneratorsOfGroup(V);
[ ( 0(2), 1(2) ), ( 1(2), 2(4) ), ( 0(2), 1(4) ), ( 1(4), 2(4) ) ]
gap> G := CT([2,3],Integers);
CT_[ 2, 3 ](Z)
gap> GeneratorsOfGroup(G);
[ ( 0(2), 1(2) ), ( 0(3), 1(3) ), ( 1(3), 2(3) ), ( 0(2), 1(4) ),
  ( 0(2), 5(6) ), ( 0(3), 1(6) ) ]
```

The group CT($\mathbb{Z}$) has an outer automorphism which is given by conjugation with $n \mapsto -n-1$. This automorphism can be applied to an rcwa mapping of $\mathbb{Z}$ or to an rcwa group over $\mathbb{Z}$ by the operation `Mirrored`. The group `Mirrored(G)` acts on the nonnegative integers as `G` acts on the negative integers, and vice versa.

Example

```
gap> ct := ClassTransposition(0,2,1,6);
( 0(2), 1(6) )
gap> Mirrored(ct);
( 1(2), 4(6) )
gap> G := Group(List([[0,2,1,2],[0,3,2,3],[2,4,1,6]],ClassTransposition));;
gap> ShortOrbits(G,[-100..100],100);
[ [ 0, 1, 2, 3, 4, 5 ] ]
gap> ShortOrbits(Mirrored(G),[-100..100],100);
[ [ -6, -5, -4, -3, -2, -1 ] ]
```

Under the hypothesis that CT($\mathbb{Z}$) is the setwise stabilizer of $\mathbb{N}_0$ in RCWA($\mathbb{Z}$), the elements of CT($\mathbb{Z}$) with modulus dividing a given positive integer m are parametrized by the ordered partitions of $\mathbb{Z}$ into m residue classes. The list of these elements for given m can be obtained by the function `AllElementsOfCTZWithGivenModulus`, and the numbers of such elements for $m \leq 24$ are stored in the list `NrElementsOfCTZWithGivenModulus`.

Example

```
gap> NrElementsOfCTZWithGivenModulus{[1..8]};
[ 1, 1, 17, 238, 4679, 115181, 3482639, 124225680 ]
```

The number of conjugacy classes of CT($\mathbb{Z}$) of elements of given order is also known under the hypothesis that CT($\mathbb{Z}$) is the setwise stabilizer of $\mathbb{N}_0$ in RCWA($\mathbb{Z}$). It can be determined by the function `NrConjugacyClassesOfCTZOfOrder`.

3.2 Basic routines for investigating residue-class-wise affine groups

In the previous section we have seen how to construct rcwa groups. The purpose of this section is to describe how to obtain information on the structure of an rcwa group and on its action on the underlying ring. The easiest way to get a little (but really only *a very little*!) information on the group structure is a dedicated method for the operation `StructureDescription`:

3.2.1 StructureDescription (for an rcwa group)

▷ `StructureDescription(G)` (method)

Returns: a string which sometimes gives a little glimpse of the structure of the rcwa group `G`.

The attribute `StructureDescription` for finite groups is documented in the GAP Reference Manual. Therefore we describe here only issues which are specific to infinite groups, and in particular to rcwa groups.

Wreath products are denoted by `wr`, and free products are denoted by `*`. The infinite cyclic group ($\mathbb{Z}$,+) is denoted by `Z`, the infinite dihedral group is denoted by `D0` and free groups of rank $2,3,4,\ldots$ are denoted by `F2`, `F3`, `F4`, ... While for finite groups the symbol `.` is used to denote a non-split extension, for rcwa groups in general it stands for an extension which may be split or not. For wild groups in most cases it happens that there is a large section on which no structural information can be obtained. Such sections of the group with unknown structure are denoted by `<unknown>`. In general, the structure of a section denoted by `<unknown>` can be very complicated and very difficult to exhibit.

Example

```
gap> G := Group(ClassTransposition(0,2,1,4),ClassShift(0,5));;
gap> StructureDescription(G);
"(Z x Z x Z x Z x Z x Z x Z) . (C2 x S7)"
gap> G := Group(ClassTransposition(0,2,1,4),
>               ClassShift(2,4),ClassReflection(1,2));;
gap> StructureDescription(G:short);
"Z^2.((S3xS3):2)"
gap> F2 := Image(IsomorphismRcwaGroup(FreeGroup(2)));;
gap> PSL2Z := Image(IsomorphismRcwaGroup(FreeProduct(CyclicGroup(3),
>                                                     CyclicGroup(2))));;
gap> G := DirectProduct(PSL2Z,F2);
<wild rcwa group over Z with 4 generators>
gap> StructureDescription(G);
"(C3 * C2) x F2"
gap> G := WreathProduct(G,CyclicGroup(IsRcwaGroupOverZ,infinity));
<wild rcwa group over Z with 5 generators>
gap> StructureDescription(G);
"((C3 * C2) x F2) wr Z"
gap> Collatz := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);;
gap> G := Group(Collatz,ClassShift(0,1));;
gap> StructureDescription(G:short);
"<unknown>.Z"
```

The extent to which the structure of an rcwa group can be exhibited automatically is severely limited. In general, one can find out much more about the structure of a given rcwa group in an interactive session using the functionality described in the rest of this section and elsewhere in this manual.

The order of an rcwa group can be computed by the operation `Size`. An rcwa group is finite if and only if it is tame and its action on a suitably chosen respected partition (see `RespectedPartition` (3.4.1)) is faithful. Hence the problem of computing the order of an rcwa group reduces to the problem of deciding whether it is tame, the problem of deciding whether it acts faithfully on a respected partition and the problem of computing the order of the finite permutation group induced on the respected partition.

Example

```
gap> G := Group(ClassTransposition(0,2,1,2),ClassTransposition(1,3,2,3),
>               ClassReflection(0,5));
<rcwa group over Z with 3 generators>
gap> Size(G);
46080
```

For a finite rcwa group, an isomorphism to a permutation group can be computed by `IsomorphismPermGroup`:

Example

```
gap> G := Group(ClassTransposition(0,2,1,2),ClassTransposition(0,3,1,3));;
gap> IsomorphismPermGroup(G);
[ ( 0(2), 1(2) ), ( 0(3), 1(3) ) ] -> [ (1,2)(3,4)(5,6), (1,2)(4,5) ]
```

In general the membership problem for rcwa groups is algorithmically unsolvable, see Corollary 4.5 in [Koh10]. A consequence of this is that a membership test "`g in G`" may run into an infinite loop if the rcwa permutation `g` is not an element of the rcwa group `G`. For tame rcwa groups however membership can always be decided. For wild rcwa groups, membership can very often be decided quite quick as well, but – as said – not always. Anyway, if `g` is contained in `G`, the membership test will eventually always return `true`, provided that there are sufficient computing resources available (memory etc.).

On Info level 2 of `InfoRCWA` the membership test provides information on reasons why the given rcwa permutation is an element of the given rcwa group or not.

The membership test "`g in G`" recognizes an option `OrbitLengthBound`. If this option is set, it returns `false` once it has computed balls of size exceeding `OrbitLengthBound` about 1 and `g` in `G`, and these balls are still disjoint. Note however that due to the algorithmic unsolvability of the membership problem, RCWA has no means to check the correctness of such bound in a given case. So the correct use of this option has to remain within the full responsibility of the user.

Example

```
gap> G := Group(ClassShift(0,3),ClassTransposition(0,3,2,6));;
gap>  ClassShift(2,6)^7 * ClassTransposition(0,3,2,6)
>   * ClassShift(0,3)^-3 in G;
true
gap> ClassShift(0,1) in G;
false
```

The conjugacy problem for rcwa groups is difficult, and RCWA provides only methods to solve it in some reasonably easy cases.

Example

```
gap> IsConjugate(RCWA(Integers),
>                ClassTransposition(0,2,1,4),ClassShift(0,1));
false
gap> IsConjugate(CT(Integers),ClassTransposition(0,2,1,6),
>                             ClassTransposition(1,4,0,8));
true
gap> g := RepresentativeAction(CT(Integers),ClassTransposition(0,2,1,6),
>                                           ClassTransposition(1,4,0,8));
<rcwa permutation of Z with modulus 48>
gap> ClassTransposition(0,2,1,6)^g = ClassTransposition(1,4,0,8);
true
```

There is a property `IsTame` which indicates whether an rcwa group is tame or not:

Example

```
gap> G := Group(ClassTransposition(0,2,1,4),ClassShift(1,3));;
gap> H := Group(ClassTransposition(0,2,1,6),ClassShift(1,3));;
gap> IsTame(G);
true
gap> IsTame(H);
false
```

For tame rcwa groups, there are methods for `IsSolvable` and `IsPerfect` available, and usually derived subgroups and subgroup indices can be computed as well. Linear representations of tame groups over the rationals can be determined by the operation `IsomorphismMatrixGroup`. Testing a wild group for solvability or perfectness is currently not always feasible, and wild groups have in general no faithful finite-dimensional linear representations. There is a method for `Exponent` available, which works basically for any rcwa group.

Example

```
gap> G := Group(ClassTransposition(0,2,1,4),ClassShift(1,2));;
gap> IsPerfect(G);
false
gap> IsSolvable(G);
true
gap> D1 := DerivedSubgroup(G);; D2 := DerivedSubgroup(D1);;
gap> IsAbelian(D2);
true
gap> Index(G,D1); Index(D1,D2);
infinity
9
gap> StructureDescription(G); StructureDescription(D1);
"(Z x Z x Z) . S3"
"(Z x Z) . C3"
gap> Q := D1/D2;
Group([ (), (1,2,4)(3,5,7)(6,8,9), (1,3,6)(2,5,8)(4,7,9) ])
gap> StructureDescription(Q);
"C3 x C3"
gap> Exponent(G);
infinity
gap> phi := IsomorphismMatrixGroup(G);;
gap> Display(Image(phi,ClassTransposition(0,2,1,4)));
[ [     0,     0,   1/2,  -1/2,     0,     0 ],
  [     0,     0,     0,     1,     0,     0 ],
  [     2,     1,     0,     0,     0,     0 ],
  [     0,     1,     0,     0,     0,     0 ],
  [     0,     0,     0,     0,     1,     0 ],
  [     0,     0,     0,     0,     0,     1 ] ]
```

When investigating a group, a basic task is to find relations among the generators:

3.2.2 EpimorphismFromFpGroup (for an rcwa group and a search radius)

▷ `EpimorphismFromFpGroup(G, r)` (method)
▷ `EpimorphismFromFpGroup(G, r, maxparts)` (method)

Returns: an epimorphism from a finitely presented group to the rcwa group *G*.

The argument *r* is the "search radius", i.e. the radius of the ball around 1 which is scanned for relations. In general, the larger *r* is chosen the smaller the kernel of the returned epimorphism is. If the group *G* has finite presentations, the kernel will in principle get trivial provided that *r* is chosen large enough. If the optional argument *maxparts* is given, it limits the search space to elements with at most *maxparts* affine parts.

Example

```
gap> a := ClassTransposition(2,4,3,4);;
gap> b := ClassTransposition(4,6,8,12);;
gap> c := ClassTransposition(3,4,4,6);;
gap> G := SparseRep(Group(a,b,c));
<(2(4),3(4)),(4(6),8(12)),(3(4),4(6))>
gap> phi := EpimorphismFromFpGroup(G,6);
#I  there are 3 generators and 12 relators of total length 330
#I  there are 3 generators and 11 relators of total length 312
[ a, b, c ] -> [ ( 2(4), 3(4) ), ( 4(6), 8(12) ), ( 3(4), 4(6) ) ]
gap> RelatorsOfFpGroup(Source(phi));
[ a^2, b^2, c^2, (b*c)^3, (a*b)^6, (a*b*c*b)^3, (a*c*b*c)^3,
  (a*b*a*c)^12, ((a*b)^2*a*c)^12, (a*b*(a*c)^2)^12, (a*b*c*a*c*b)^12 ]
```

A related very common task is to factor group elements into generators:

3.2.3 PreImagesRepresentative (for an epi. from a free group to an rcwa group)

▷ `PreImagesRepresentative(phi, g)` (method)

Returns: a representative of the set of preimages of *g* under the epimorphism *phi* from a free group to an rcwa group.

The epimorphism *phi* must map the generators of the free group to the generators of the rcwa group one-by-one.

This method can be used for factoring elements of rcwa groups into generators. The implementation is based on `RepresentativeActionPreImage`, see `RepresentativeAction` (3.3.10).

Quite frequently, computing several preimages is not harder than computing just one, i.e. often several preimages are found simultaneously. The operation `PreImagesRepresentatives` takes care of this. It takes the same arguments as `PreImagesRepresentative` and returns a list of preimages. If multiple preimages are found, their quotients give rise to nontrivial relations among the generators of the image of *phi*.

Example

```
gap> a := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);; SetName(a,"a");
gap> b := ClassShift(0,1);; SetName(b,"b");
gap> G := Group(a,b);; # G = <<Collatz permutation>, n -> n + 1>
gap> phi := EpimorphismFromFreeGroup(G);;
gap> g := Comm(a^2*b^4,a*b^3); # a sample element to be factored
<rcwa permutation of Z with modulus 8>
```

```
gap> PreImagesRepresentative(phi,g); # -> a factorization of g
b^-3*(b^-1*a^-1)^2*b^3*a*b^-1*a*b^3
gap> g = b^-4*a^-1*b^-1*a^-1*b^3*a*b^-1*a*b^3; # check
true
gap> g := Comm(a*b,Comm(a,b^3));
<rcwa permutation of Z with modulus 8>
gap> pre := PreImagesRepresentatives(phi,g);
[ (b^-1*a^-1)^2*b^2*(b*a)^2*b^-2, b^-1*(a^-1*b)^2*b^2*(a*b^-1)^2*b^-1 ]
gap> rel := pre[1]/pre[2]; # -> a nontrivial relation
(b^-1*a^-1)^2*b^3*a*b^2*a^-1*b^-2*(b^-1*a)^2*b
gap> rel^phi;
IdentityMapping( Integers )
```

3.3 The natural action of an rcwa group on the underlying ring

Knowing a natural permutation representation of a group usually helps significantly in computing in it and in obtaining results on its structure. This holds particularly for the natural action of an rcwa group on its underlying ring. In this section we describe RCWA's functionality related to this action.

The support, i.e. the set of moved points, of an rcwa group can be determined by `Support` or `MovedPoints` (these are synonyms). Testing for transitivity on the underlying ring or on a union of residue classes thereof is often feasible:

Example

```
gap> G := Group(ClassTransposition(1,2,0,4),ClassShift(0,2));;
gap> IsTransitive(G,Integers);
true
```

Groups generated by class transpositions of the integers act on the set of nonnegative integers. There is a property `IsTransitiveOnNonnegativeIntegersInSupport(G)` which indicates whether such group acts transitively on the set of nonnegative integers in its support. Since such transitivity test is a computationally hard problem, methods may fail. If `IsTransitiveOnNonnegativeIntegersInSupport` returns `true`, an attribute `TransitivityCertificate` is set; this is a record containing components `phi`, `words`, `classes`, `smallpointbound`, `status` and `complete` as follows:

`phi` is an epimorphism from a free group to `G` which maps generators to generators.

`words`, `classes`
: two lists. – `words[i]` is a preimage under `phi` of an element of `G` which maps all sufficiently large positive integers in the residue classes `classes[i]` to smaller nonnegative integers.

`smallpointbound`
: in addition to finding a list of group elements g_i such that for any large enough integer n in the support of `G` there is some g_i such that $n^{g_i} < n$, for verifying transitivity it was necessary to check that all integers less than or equal to `smallpointbound` in the support of `G` lie in the same orbit.

`status`
: the string `"transitive"` in case all checks have been completed successfully.

`complete`
: `true` in case all checks have been completed successfully.

Parts of this information for possibly intransitive groups can be obtained by the operation `TryToComputeTransitivityCertificate(`*G*`,`*searchlimit*`)`, where *searchlimit* is the maximum radius about a point within which smaller points are searched and taken into consideration. This operation interprets an option `abortdensity` – if set, the operation returns the data computed so far once the density of the set of positive integers in the support of *G* for which no group element is found which maps them to smaller integers reaches or drops below `abortdensity`. A simplified certificate can be obtained via `SimplifiedCertificate(`*cert*`)`.

Example

```
gap> G := Group(List([[0,2,1,2],[0,3,2,3],[1,2,2,4]],
>                    ClassTransposition));
<(0(2),1(2)),(0(3),2(3)),(1(2),2(4))>
gap> IsTransitiveOnNonnegativeIntegersInSupport(G);
true
gap> TransitivityCertificate(G);
rec(
  classes := [ [ 1(2) ], [ 2(6) ], [ 6(12), 10(12) ], [ 0(12) ],
      [ 4(12) ] ], complete := true,
  phi := [ a, b, c ] -> [ ( 0(2), 1(2) ), ( 0(3), 2(3) ), ( 1(2), 2(4) )
     ], smallpointbound := 4, status := "transitive",
  words := [ a, b, c, b*c, a*b ] )
gap> SimplifiedCertificate(last);
rec( classes := [ [ 1(2) ], [ 2(4) ], [ 4(12) ], [ 0(12), 8(12) ] ],
  complete := true,
  phi := [ a, b, c ] -> [ ( 0(2), 1(2) ), ( 0(3), 2(3) ), ( 1(2), 2(4) )
     ], smallpointbound := 4, status := "transitive",
  words := [ a, c, a*b, b*c ] )
gap> G := Group(List([[0,2,1,2],[1,2,2,4],[1,4,2,6]],
>                    ClassTransposition));             # '3n+1 group'
<(0(2),1(2)),(1(2),2(4)),(1(4),2(6))>
gap> cert := TryToComputeTransitivityCertificate(G,10);
rec(
  classes := [ [ 1(2) ], [ 2(4) ], [ 4(32) ], [ 8(24), 44(48), 20(96) ],
      [ 0(24), 16(24) ] ], complete := false,
  phi := [ a, b, c ] -> [ ( 0(2), 1(2) ), ( 1(2), 2(4) ), ( 1(4), 2(6) )
     ], remaining := [ 12(48), 28(48), 52(96), 84(96) ],
  smallpointbound := 42, status := "unclear",
  words := [ a, b, (a*c)^2*b*a*b, c, a*c*b ] )
gap> Union(Flat(cert.classes));
<union of 90 residue classes (mod 96) (6 classes)>
gap> Difference(Integers,Union(Flat(cert.classes)));
12(48) U 28(48) U 52(96) U 84(96)
gap> cert := TryToComputeTransitivityCertificate(G,20); # try larger bound
rec(
  classes := [ [ 1(2) ], [ 2(4) ], [ 4(32) ], [ 8(24), 44(48), 20(96) ],
      [ 0(24), 16(24) ], [ 12(768), 268(768) ], [ 28(768), 540(768) ] ],
```

```
  complete := false,
  phi := [ a, b, c ] -> [ ( 0(2), 1(2) ), ( 1(2), 2(4) ), ( 1(4), 2(6) )
     ],
  remaining := [ 52(96), 84(96), 60(192), 108(192), 124(192), 172(192),
      76(384), 204(384), 220(384), 348(384), 156(768), 396(768),
      412(768), 652(768) ], smallpointbound := 1074, status := "unclear",
  words := [ a, b, (a*c)^2*b*a*b, c, a*c*b, (a*c)^3*b*c*b*a*b,
      (a*c)^4*b*a*b*a*b ] )
gap> Difference(Integers,Union(Flat(cert.classes)));
<union of 44 residue classes (mod 768) (14 classes)>
gap> Intersection([0..100],last);
[ 52, 60, 76, 84 ]
```

Further, there are methods to compute orbits under the action of an rcwa group:

3.3.1 Orbit (for an rcwa group and either a point or a set)

▷ `Orbit(G, point)` (method)
▷ `Orbit(G, set)` (method)

Returns: the orbit of the point *point* respectively the set *set* under the natural action of the rcwa group *G* on its underlying ring.

The second argument can either be an element or a subset of the underlying ring of the rcwa group *G*. Since orbits under the action of rcwa groups can be finite or infinite, and since infinite orbits are not necessarily residue class unions, the orbit may either be returned in the form of a list, in the form of a residue class union or in the form of an orbit object. It is possible to loop over orbits returned as orbit objects, they can be compared and there is a membership test for them. However note that equality and membership for such orbits cannot always be decided.

Example

```
gap> G := Group(ClassShift(0,2),ClassTransposition(0,3,1,3));
<rcwa group over Z with 2 generators>
gap> Orbit(G,0);
Z \ 5(6)
gap> Orbit(G,5);
[ 5 ]
gap> Orbit(G,ResidueClass(0,2));
[ 0(2), 1(6) U 2(6) U 3(6), 1(3) U 3(6), 0(3) U 1(6), 0(3) U 4(6),
  1(3) U 0(6), 0(3) U 2(6), 0(6) U 1(6) U 2(6), 2(6) U 3(6) U 4(6),
  1(3) U 2(6) ]
gap> Length(Orbit(G,ResidueClass(0,4)));
80
gap> G := Group(ClassTransposition(0,2,1,2),ClassTransposition(0,2,1,4),
>               ClassReflection(0,3));
<rcwa group over Z with 3 generators>
gap> orb := Orbit(G,2);
<orbit of 2 under <wild rcwa group over Z with 3 generators>>
gap> 1015808 in orb;
true
gap> First(orb,n->ForAll([n,n+2,n+6,n+8,n+30,n+32,n+36,n+38],IsPrime));
```

```
-19
```

3.3.2 GrowthFunctionOfOrbit (for an rcwa group, a point and bounds on radius and sphere size)

▷ `GrowthFunctionOfOrbit(G, n, r_max, size_max)` (operation)
▷ `GrowthFunctionOfOrbit(orb, r_max, size_max)` (method)

Returns: a list whose $(r+1)$-th entry is the size of the sphere of radius r about `n` under the action of the group `G`, where the argument `r_max` is the largest possible radius to be considered, and the computation stops once the sphere size exceeds `size_max`.

An option `"small"` is interpreted – see example below. In place of the group `G` and the point `n`, one can pass as first argument also an rcwa group orbit object `orb`.

Example

```
gap> G := Group(List([[0,4,1,4],[0,3,5,6],[0,4,5,6]],ClassTransposition));
<(0(4),1(4)),(0(3),5(6)),(0(4),5(6))>
gap> GrowthFunctionOfOrbit(G,18,100,20);
[ 1, 1, 2, 3, 4, 3, 4, 4, 4, 4, 3, 3, 3, 4, 3, 4, 4, 5, 5, 6, 8, 6, 5,
  5, 4, 3, 3, 4, 4, 4, 3, 3, 5, 4, 5, 6, 5, 2, 3, 3, 2, 3, 3, 4, 5, 4,
  4, 4, 6, 5, 5, 3, 4, 2, 3, 4, 4, 2, 3, 4, 4, 2, 3, 3, 4, 3, 5, 3, 5,
  4, 5, 6, 5, 3, 4, 5, 6, 5, 4, 3, 5, 4, 5, 5, 4, 4, 5, 5, 3, 4, 5, 3,
  3, 4, 5, 4, 2, 3, 4, 4, 4 ]
gap> last = GrowthFunctionOfOrbit(Orbit(G,18),100,20);
true
gap> GrowthFunctionOfOrbit(G,18,20,20:small:=[0..100]);
rec( smallpoints := [ 18, 24, 25, 27, 30, 32, 33, 36, 37, 39, 40, 41,
      42, 44, 45, 48, 49, 51, 52, 53, 56, 57, 59, 60, 61, 64, 65, 66,
      68, 69, 71, 75, 76, 77, 80, 81, 83, 88, 89, 92, 93, 95, 100 ],
  spheresizes := [ 1, 1, 2, 3, 4, 3, 4, 4, 4, 4, 3, 3, 3, 4, 3, 4, 4, 5,
      5, 6, 8 ] )
gap> G := Group(List([[0,2,1,2],[1,2,2,4],[1,4,2,6]],ClassTransposition));
<(0(2),1(2)),(1(2),2(4)),(1(4),2(6))>
gap> GrowthFunctionOfOrbit(G,0,100,10000);
[ 1, 1, 1, 1, 1, 1, 1, 2, 3, 3, 4, 5, 7, 6, 7, 9, 12, 14, 19, 21, 28,
  29, 37, 42, 55, 57, 72, 78, 99, 113, 148, 164, 215, 226, 288, 344,
  462, 478, 612, 686, 894, 985, 1284, 1416, 1847, 2018, 2620, 2902,
  3786, 4167, 5432, 5958, 7749, 8568, 11178 ]
```

Given an rcwa group `G` over $\mathbb{Z}$ and an integer `n`, `DistanceToNextSmallerPointInOrbit(G,n)` computes the smallest number d such that there is a product g of d generators or inverses of generators of `G` which maps `n` to an integer with absolute value less than |`n`|, provided that the orbit of `n` contains such integer. RCWA provides a function to draw pictures of orbits of rcwa groups on $\mathbb{Z}^2$. The pictures are written to files in bitmap- (bmp-) format. The author has successfully tested this feature both under Linux and under Windows, and the generated pictures can be processed further with many common graphics programs:

3.3.3 DrawOrbitPicture (G, p0, bound, h, w, colored, palette, filename)

▷ `DrawOrbitPicture(G, p0, bound, h, w, colored, palette, filename)` (function)

Returns: nothing.

Draws a picture of the orbit(s) of the point(s) *p0* under the action of the group *G* on $\mathbb{Z}^2$. The argument *p0* is either one point or a list of points. The argument *bound* denotes the bound to which the ball about *p0* is to be computed, in terms of absolute values of coordinates. The size of the generated picture is *h* x *w* pixels. The argument *colored* is a boolean which indicates whether a 24-bit true color picture or a monochrome picture should be generated. In the former case, *palette* must be a list of triples of integers in the range $0,\dots,255$, denoting the RGB values of the colors to be used. In the latter case, *palette* is not used, and any value can be passed. The picture is written in bitmap- (bmp-) format to a file named *filename*. This is done using the utility function `SaveAsBitmapPicture` from ResClasses.

Example

```
gap> PSL2Z := Image(IsomorphismRcwaGroup(FreeProduct(CyclicGroup(2),
>                                                    CyclicGroup(3))));;
gap> DrawOrbitPicture(PSL2Z,[0,1],2000,512,512,false,fail,"example1.bmp");
gap> DrawOrbitPicture(PSL2Z,Combinations([1..4],2),2000,512,512,true,
>                     [[255,0,0],[0,255,0],[0,0,255]],"example2.bmp");
```

The pictures drawn in the examples are shown on RCWA's webpage.

Finite orbits give rise to finite quotients of a group, and finite cycles can help to check for conjugacy. Therefore it is important to be able to determine them:

3.3.4 ShortOrbits (for rcwa groups) & ShortCycles (for rcwa permutations)

▷ `ShortOrbits(G, S, maxlng)` (operation)
▷ `ShortOrbits(G, S, maxlng, maxn)` (operation)
▷ `ShortCycles(g, S, maxlng)` (operation)
▷ `ShortCycles(g, S, maxlng, maxn)` (operation)
▷ `ShortCycles(g, maxlng)` (operation)

Returns: in the first form a list of all orbits of the rcwa group *G* of length at most *maxlng* which intersect non-trivially with the set *S*. In the second form a list of all orbits of the rcwa group *G* of length at most *maxlng* which intersect non-trivially with the set *S* and which, in terms of euclidean norm, do not exceed *maxn*. In the third form a list of all cycles of the rcwa permutation *g* of length at most *maxlng* which intersect non-trivially with the set *S*. In the fourth form a list of all cycles of the rcwa permutation *g* of length at most *maxlng* which intersect non-trivially with the set *S* and which, in terms of euclidean norm, do not exceed *maxn*. In the fifth form a list of all cycles of the rcwa permutation *g* of length at most *maxlng* which do not correspond to cycles consisting of residue classes.

The operation `ShortOrbits` recognizes an option *finite*. If this option is set, it is assumed that all orbits are finite, in order to speed up the computation. If furthermore *maxlng* is negative, a list of *all* orbits which intersect non-trivially with *S* is returned.

There is an operation `CyclesOnFiniteOrbit(G,g,n)` which returns a list of all cycles of the rcwa permutation *g* on the orbit of the point *n* under the action of the rcwa group *G*. Here *g* is assumed to be an element of *G*, and the orbit of *n* is assumed to be finite.

Example

```
gap> G := Group(ClassTransposition(1,4,2,4)*ClassTransposition(1,4,3,4),
>               ClassTransposition(3,9,6,18)*ClassTransposition(1,6,3,9));;
gap> List(ShortOrbits(G,[-15..15],100),
>         orb->StructureDescription(Action(G,orb)));
[ "A15", "A4", "1", "1", "C3", "1", "((C2 x C2 x C2) : C7) : C3", "1",
  "1", "C3", "A19" ]
gap> ShortCycles(mKnot(7),[1..100],20);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 7, 8 ], [ 9, 10 ],
  [ 11, 12 ], [ 13, 14, 16, 18, 20, 22, 19, 17, 15 ], [ 21, 24 ],
  [ 23, 26 ], [ 25, 28, 32, 36, 31, 27, 30, 34, 38, 33, 29 ],
  [ 35, 40 ], [ 37, 42, 48, 54, 47, 41, 46, 52, 45, 39, 44, 50, 43 ],
  [ 77, 88, 100, 114, 130, 148, 127, 109, 124, 107, 122, 105, 120, 103,
      89 ] ]
gap> G := Group(List([[0,2,1,2],[0,5,4,5],[1,4,0,6]],ClassTransposition));;
gap> CyclesOnFiniteOrbit(G,G.1*G.2,0);
[ [ 0, 1, 4, 9, 8, 5 ], [ 6, 7 ], [ 10, 11, 14, 19, 18, 15 ], [ 12, 13 ] ]
gap> List(CyclesOnFiniteOrbit(G,G.1*G.2*G.3*G.1*G.3*G.2,32),Length);
[ 3148, 3148 ]
```

3.3.5 ShortResidueClassOrbits & ShortResidueClassCycles

▷ `ShortResidueClassOrbits(G, modulusbound, maxlng)` (operation)

▷ `ShortResidueClassCycles(g, modulusbound, maxlng)` (operation)

Returns: in the first form a list of all orbits of residue classes under the action of the rcwa group `G` which contain a residue class $r(m)$ such that m divides `modulusbound` and which are not longer than `maxlng`. In the second form a list of all cycles of residue classes of the rcwa permutation g which contain a residue class $r(m)$ such that m divides `modulusbound` and which are not longer than `maxlng`.

We are only talking about a *cycle* of residue classes of an rcwa permutation g if the restrictions of g to all contained residue classes are affine. Similarly we are only talking about an *orbit* of residue classes under the action of an rcwa group G if the restrictions of all elements of G to all residue classes in the orbit are affine.

The returned lists may contain additional cycles, resp., orbits, which do not contain a residue class $r(m)$ such that m divides `modulusbound`, but which happen to be found without additional efforts.

Example

```
gap> g := ClassTransposition(0,2,1,2)*ClassTransposition(0,4,1,6);
<rcwa permutation of Z with modulus 12>
gap> ShortResidueClassCycles(g,Mod(g)^2,20);
[ [ 2(12), 3(12) ], [ 10(12), 11(12) ], [ 4(24), 5(24), 7(36), 6(36) ],
  [ 20(24), 21(24), 31(36), 30(36) ],
  [ 8(48), 9(48), 13(72), 19(108), 18(108), 12(72) ],
  [ 40(48), 41(48), 61(72), 91(108), 90(108), 60(72) ],
  [ 16(96), 17(96), 25(144), 37(216), 55(324), 54(324), 36(216), 24(144)
     ],
  [ 80(96), 81(96), 121(144), 181(216), 271(324), 270(324), 180(216),
      120(144) ] ]
gap> G := Group(List([[0,6,5,6],[1,4,4,6],[2,4,3,6]],ClassTransposition));
```

```
<(0(6),5(6)),(1(4),4(6)),(2(4),3(6))>
gap> ShortResidueClassOrbits(G,48,10);
[ [ 7(12) ], [ 8(12) ], [ 1(24), 4(36) ], [ 2(24), 3(36) ],
  [ 12(24), 17(24), 28(36) ], [ 18(24), 23(24), 27(36) ],
  [ 37(48), 58(72), 87(108) ], [ 38(48), 57(72), 88(108) ],
  [ 0(48), 5(48), 10(72), 15(108) ], [ 6(48), 11(48), 9(72), 16(108) ] ]
```

3.3.6 ComputeCycleLength (for an rcwa permutation and a point)

▷ `ComputeCycleLength(g, n)` (function)

Returns: a record containing the length, the largest point and the position of the largest point of the cycle of the rcwa permutation *g* which contains the point *n*, provided that this cycle is finite.

If the cycle is infinite, the function will run into an infinite loop unless the option `"abortat"` is set to the maximum number of iterates to be tried before aborting. Iterates are not stored, to save memory. The function interprets an option `"notify"`, which defaults to 10000; every "notify" iterations, the number of binary digits of the latest iterate is printed. This output can be suppressed by the option `quiet`. The function also interprets an option `"small"`, which may be set to a range within which small points are recorded and returned in a component `smallpoints`.

Example

```
gap> g := Product(List([[0,5,3,5],[1,2,0,6],[2,4,3,6]],
>                       ClassTransposition));
<rcwa permutation of Z with modulus 180>
gap> ComputeCycleLength(g,20:small:=[0..1000]);
n = 20: after 10000 steps, the iterate has 1919 binary digits.
n = 20: after 20000 steps, the iterate has 2908 binary digits.
n = 20: after 30000 steps, the iterate has 1531 binary digits.
n = 20: after 40000 steps, the iterate has 708 binary digits.
rec( aborted := false, g := <rcwa permutation of Z with modulus 180>,
  length := 45961,
  maximum := 1804799284115095270913147901449294800414733098629573943847831\
052593543743102144234616642220125093526855394515808576992444838872467975 3\
5271669245363980744610119632280105994423399614803956244808653465492205657\
8650363041608376587943180444494842094693691286183613056599672737336761093\
3101035841077322874883200384115281051837032147150147712534199292886436789\
7520389780289517825203780151058517520194926468391308525704499649905091899\
9667529835495635671154681958992898010506577172313321500572646883756736685\
0158653917532084531267455434808219032998691038943070902228427549279555530\
6429870190316109419051531138721361826083376315737131067799731181096142797\
4868525347003646887454985757711743327946232372385342293662007684758208408\
8635715976464060647431260835037213863991037813998261883899050447111540742\
5857187943077255493709629738212709349458790098815926920248565399938335540\
8092502449690267365120996852, maxpos := 19825, n := 20,
  smallpoints := [ 20, 23, 66, 99, 294, 295, 298, 441, 447, 882, 890,
      893 ] )
```

3.3.7 CycleRepresentativesAndLengths (for rcwa permutation and set of seed points)

▷ `CycleRepresentativesAndLengths(g, S)` (operation)

Returns: a list of pairs (cycle representative, length of cycle) for all cycles of the rcwa permutation g which have a nontrivial intersection with the set S, where fixed points are omitted.

The rcwa permutation g is assumed to have only finite cycles. If g has an infinite cycle which intersects non-trivially with S, this may cause an infinite loop unless a cycle length limit is set via the option `abortat`. The output can be suppressed by the option `quiet`.

Example

```
gap> g := ClassTransposition(0,2,1,2)*ClassTransposition(0,4,1,6);;
gap> CycleRepresentativesAndLengths(g,[0..50]);
[ [ 2, 2 ], [ 4, 4 ], [ 8, 6 ], [ 10, 2 ], [ 14, 2 ], [ 16, 8 ],
  [ 20, 4 ], [ 22, 2 ], [ 26, 2 ], [ 28, 4 ], [ 32, 10 ], [ 34, 2 ],
  [ 38, 2 ], [ 40, 6 ], [ 44, 4 ], [ 46, 2 ], [ 50, 2 ] ]
gap> g := Product(List([[0,5,3,5],[1,2,0,6],[2,4,3,6]],
>                      ClassTransposition));
<rcwa permutation of Z with modulus 180>
gap> CycleRepresentativesAndLengths(g,[0..100]:abortat:=100000);
n = 20: after 10000 steps, the iterate has 1919 binary digits.
n = 20: after 20000 steps, the iterate has 2908 binary digits.
n = 20: after 30000 steps, the iterate has 1531 binary digits.
n = 20: after 40000 steps, the iterate has 708 binary digits.
n = 79: after 10000 steps, the iterate has 1679 binary digits.
n = 100: after 10000 steps, the iterate has 712 binary digits.
n = 100: after 20000 steps, the iterate has 2507 binary digits.
n = 100: after 30000 steps, the iterate has 3311 binary digits.
n = 100: after 40000 steps, the iterate has 3168 binary digits.
n = 100: after 50000 steps, the iterate has 3947 binary digits.
n = 100: after 60000 steps, the iterate has 4793 binary digits.
n = 100: after 70000 steps, the iterate has 5325 binary digits.
n = 100: after 80000 steps, the iterate has 6408 binary digits.
n = 100: after 90000 steps, the iterate has 7265 binary digits.
n = 100: after 100000 steps, the iterate has 7918 binary digits.
[ [ 0, 7 ], [ 5, 3 ], [ 7, 7159 ], [ 11, 9 ], [ 19, 342 ],
  [ 20, 45961 ], [ 25, 3 ], [ 26, 21 ], [ 29, 2 ], [ 31, 3941 ],
  [ 34, 19 ], [ 37, 7 ], [ 40, 5 ], [ 41, 7 ], [ 46, 3 ], [ 49, 2 ],
  [ 59, 564 ], [ 61, 577 ], [ 65, 3 ], [ 67, 23 ], [ 71, 41 ],
  [ 79, 16984 ], [ 80, 5 ], [ 85, 3 ], [ 86, 3 ], [ 89, 2 ], [ 91, 9 ],
  [ 94, 1355 ], [ 97, 7 ], [ 100, fail ] ]
```

Often one also wants to know which residue classes an rcwa mapping or an rcwa group fixes setwise:

3.3.8 FixedResidueClasses (for rcwa mapping and bound on modulus)

▷ `FixedResidueClasses(g, maxmod)` (operation)
▷ `FixedResidueClasses(G, maxmod)` (operation)

Returns: the set of residue classes with modulus greater than 1 and less than or equal to *maxmod* which the rcwa mapping g, respectively the rcwa group G, fixes setwise.

Example

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

Frequently one needs to compute balls of certain radius around points or group elements, be it to estimate the growth of a group, be it to see how an orbit looks like, be it to search for a group element with certain properties or be it for other purposes:

3.3.9 Ball (for group, element and radius or group, point, radius and action)

▷ `Ball(G, g, r)` (method)
▷ `Ball(G, p, r, action)` (method)
▷ `Ball(G, p, r)` (method)

Returns: the ball of radius *r* around the element *g* in the group *G*, respectively the ball of radius *r* around the point *p* under the action *action* of the group *G*, respectively the ball of radius *r* around the point *p* under the action `OnPoints` of the group *G*,

All balls are understood with respect to `GeneratorsOfGroup(G)`. As membership tests can be expensive, the former method does not check whether *g* is indeed an element of *G*. The methods require that element- / point comparisons are cheap. They are not only applicable to rcwa groups. If the option *Spheres* is set, the ball is split up and returned as a list of spheres. There is a related operation `RestrictedBall(G,g,r,modulusbound)` specifically for rcwa groups which computes only those elements of the ball whose moduli do not exceed *modulusbound*, and which can be reached from *g* without computing intermediate elements whose moduli do exceed *modulusbound*. The latter operation interprets an option `"boundaffineparts"`. – If this option is set and the group *G* and the element *g* are in sparse representation, then *modulusbound* is actually taken to be a bound on the number of affine parts rather than a bound on the modulus.

Example

```
gap> PSL2Z := Image(IsomorphismRcwaGroup(FreeProduct(CyclicGroup(2),
>                                                    CyclicGroup(3))));;
gap> List([1..10],k->Length(Ball(PSL2Z,[0,1],k,OnTuples)));
[ 4, 8, 14, 22, 34, 50, 74, 106, 154, 218 ]
gap> Ball(Group((1,2),(2,3),(3,4)),(),2:Spheres);
[ [ () ], [ (3,4), (2,3), (1,2) ],
  [ (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,3,2) ] ]
gap> G := Group(List([[1,2,4,6],[1,3,2,6],[2,3,4,6]],ClassTransposition));;
gap> B := RestrictedBall(G,One(G),20,36:Spheres);; # try replacing 36 by 72
gap> List(B,Length);
[ 1, 3, 6, 12, 4, 6, 6, 4, 4, 4, 6, 6, 3, 3, 2, 0, 0, 0, 0, 0, 0 ]
```

It is possible to determine group elements which map a given tuple of elements of the underlying ring to a given other tuple, if such elements exist:

3.3.10 RepresentativeAction (G, source, destination, action)

▷ `RepresentativeAction(G, source, destination, action)` (method)

Returns: an element of *G* which maps *source* to *destination* under the action given by *action*.

If an element satisfying this condition does not exist, this method either returns `fail` or runs into an infinite loop. The problem whether *source* and *destination* lie in the same orbit under the action *action* of *G* is hard, and in its general form most likely computationally undecidable.

In cases where rather a word in the generators of *G* than the actual group element is needed, one should use the operation `RepresentativeActionPreImage` instead. This operation takes five arguments. The first four are the same as those of `RepresentativeAction`, and the fifth is a free group whose generators are to be used as letters of the returned word. Note that `RepresentativeAction` calls `RepresentativeActionPreImage` and evaluates the returned word. The evaluation of the word can very well take most of the time if *G* is wild and coefficient explosion occurs.

The algorithm is based on computing balls of increasing radius around *source* and *destination* until they intersect non-trivially.

Example

```
gap> a := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);; SetName(a,"a");
gap> b := ClassShift(1,4:Name:="b");; G := Group(a,b);;
gap> elm := RepresentativeAction(G,[7,4,9],[4,5,13],OnTuples);;
gap> Display(elm);

Rcwa permutation of Z with modulus 12

        /
        | n-3 if n in 1(6) U 10(12)
        | n+4 if n in 5(12) U 9(12)
 n |-> <  n+1 if n in 4(12)
        | n   if n in 0(6) U 2(6) U 3(12) U 11(12)
        |
        \

gap> List([7,4,9],n->n^elm);
[ 4, 5, 13 ]
gap> elm := RepresentativeAction(G,[6,-3,8],[-9,4,11],OnPoints);;
gap> Display(elm);

Rcwa permutation of Z with modulus 12

        /
        | 2n/3      if n in 0(6) U 3(12)
        | (4n+1)/3  if n in 2(6) U 11(12)
        | (4n-1)/3  if n in 4(6) U 7(12)
 n |-> <  (2n-8)/3  if n in 1(12)
        | (4n-17)/3 if n in 5(12)
        | (4n-15)/3 if n in 9(12)
        |
        \

gap> [6,-3,8]^elm; List([6,-3,8],n->n^elm); # 'OnPoints' allows reordering
[ -9, 4, 11 ]
```

```
[ 4, -9, 11 ]
gap> F := FreeGroup("a","b");; phi := EpimorphismByGenerators(F,G);;
gap> w := RepresentativeActionPreImage(G,[10,-4,9,5],[4,5,13,-8],
>                                      OnTuples,F);
a*b^-1*a^-1*(b^-1*a)^2*b*a*b^-2*a*b*a^-1*b
gap> elm := w^phi;
<rcwa permutation of Z with modulus 324>
gap> List([10,-4,9,5],n->n^elm);
[ 4, 5, 13, -8 ]
```

Sometimes an rcwa group fixes a certain partition of the underlying ring into unions of residue classes. If this happens, then any orbit is clearly a subset of exactly one of these parts. Further, such a partition often gives rise to proper quotients of the group:

3.3.11 ProjectionsToInvariantUnionsOfResidueClasses (for rcwa group and modulus)

▷ `ProjectionsToInvariantUnionsOfResidueClasses(G, m)` (operation)

Returns: the projections of the rcwa group `G` to the unions of residue classes (mod `m`) which it fixes setwise.

The corresponding partition of a set of representatives for the residue classes (mod `m`) can be obtained by the operation `OrbitsModulo(G,m)`.

Example
```
gap> G := Group(ClassTransposition(0,2,1,2),ClassShift(3,4));;
gap> ProjectionsToInvariantUnionsOfResidueClasses(G,4);
[ [ ( 0(2), 1(2) ), ClassShift( 3(4) ) ] ->
    [ ( 0(4), 1(4) ), IdentityMapping( Integers ) ],
  [ ( 0(2), 1(2) ), ClassShift( 3(4) ) ] ->
    [ ( 2(4), 3(4) ), <rcwa permutation of Z with modulus 4> ] ]
gap> List(last,phi->Support(Image(phi)));
[ 0(4) U 1(4), 2(4) U 3(4) ]
```

Given two partitions of the underlying ring into the same number of unions of residue classes, there is always an rcwa permutation which maps the one to the other:

3.3.12 RepresentativeAction (for RCWA(R) and 2 partitions of R into residue classes)

▷ `RepresentativeAction(RCWA(R), P1, P2)` (method)

Returns: an element of RCWA(R) which maps the partition `P1` to `P2`.

The arguments `P1` and `P2` must be partitions of the underlying ring R into the same number of unions of residue classes. The method for $R = \mathbb{Z}$ recognizes the option `IsTame`, which can be used to demand a tame result. If this option is set and there is no tame rcwa permutation which maps `P1` to `P2`, the method runs into an infinite loop. This happens if the condition in Theorem 2.8.9 in [Koh05] is not satisfied. If the option `IsTame` is not set and the partitions `P1` and `P2` both consist entirely of single residue classes, then the returned mapping is affine on any residue class in `P1`.

Example
```
gap> P1 := AllResidueClassesModulo(3);
```

```
[ 0(3), 1(3), 2(3) ]
gap> P2 := List([[0,2],[1,4],[3,4]],ResidueClass);
[ 0(2), 1(4), 3(4) ]
gap> elm := RepresentativeAction(RCWA(Integers),P1,P2);
<rcwa permutation of Z with modulus 3>
gap> P1^elm = P2;
true
gap> IsTame(elm);
false
gap> elm := RepresentativeAction(RCWA(Integers),P1,P2:IsTame);
<tame rcwa permutation of Z with modulus 24>
gap> P1^elm = P2;
true
gap> elm := RepresentativeAction(RCWA(Integers),
>              [ResidueClass(1,3),
>               ResidueClassUnion(Integers,3,[0,2])],
>              [ResidueClassUnion(Integers,5,[2,4]),
>               ResidueClassUnion(Integers,5,[0,1,3])]);
<rcwa permutation of Z with modulus 6>
gap> [ResidueClass(1,3),ResidueClassUnion(Integers,3,[0,2])]^elm;
[ 2(5) U 4(5), Z \ 2(5) U 4(5) ]
```

3.3.13 CollatzLikeMappingByOrbitTree (for rcwa group, root point and range of radii)

▷ `CollatzLikeMappingByOrbitTree(G, n, min_r, max_r)` (operation)

Returns: either an rcwa mapping f which maps the sphere of radius r about `n` under the action of `G` into the sphere of radius $r-1$ about `n` for every r ranging from `min_r` to `max_r`, or `fail`.

Obviously not for every rcwa group and every root point a mapping f with these properties exists, and if it exists, it usually depends on the choice of generators of the group.

Example

```
gap> G := Group(ClassTransposition(0,2,1,2),ClassTransposition(1,2,2,4),
>                ClassTransposition(1,4,2,6));;
gap> G := SparseRep(G);;
gap> f := CollatzLikeMappingByOrbitTree(G,0,4,10);
<rcwa mapping of Z with modulus 4 and 4 affine parts>
gap> Display(f);

Rcwa mapping of Z with modulus 4 and 4 affine parts

         /
         | n+1        if n in 0(4)
         | (3n+1)/2 if n in 1(4)
 n |-> <  n/2        if n in 2(4)
         | n-1        if n in 3(4)
         |
         \

gap> B := Ball(G,0,15:Spheres);
```

```
[ [ 0 ], [ 1 ], [ 2 ], [ 3 ], [ 6 ], [ 7 ], [ 14 ], [ 9, 15 ], [ 8, 18, 30 ],
  [ 5, 19, 31 ], [ 4, 10, 38, 62 ], [ 11, 25, 39, 41, 63 ],
  [ 22, 24, 40, 50, 78, 82, 126 ], [ 23, 33, 51, 79, 83, 127 ],
  [ 32, 46, 66, 102, 158, 166, 254 ],
  [ 21, 47, 67, 103, 105, 159, 167, 169, 255 ] ]
gap> List([3..15],i->IsSubset(B[i-1],B[i]^f));
[ true, true, true, true, true, true, true, true, true, true, true, true,
  true ]
gap> Trajectory(f,52,[0,1]);
[ 52, 53, 80, 81, 122, 61, 92, 93, 140, 141, 212, 213, 320, 321, 482, 241,
  362, 181, 272, 273, 410, 205, 308, 309, 464, 465, 698, 349, 524, 525, 788,
  789, 1184, 1185, 1778, 889, 1334, 667, 666, 333, 500, 501, 752, 753, 1130,
  565, 848, 849, 1274, 637, 956, 957, 1436, 1437, 2156, 2157, 3236, 3237,
  4856, 4857, 7286, 3643, 3642, 1821, 2732, 2733, 4100, 4101, 6152, 6153,
  9230, 4615, 4614, 2307, 2306, 1153, 1730, 865, 1298, 649, 974, 487, 486,
  243, 242, 121, 182, 91, 90, 45, 68, 69, 104, 105, 158, 79, 78, 39, 38, 19,
  18, 9, 14, 7, 6, 3, 2, 1 ]
```

3.4 Special attributes of tame residue-class-wise affine groups

There are a couple of attributes which a priori make only sense for tame rcwa groups. With their help, various structural information about a given such group can be obtained. We have already seen above that there are for example methods for `IsSolvable`, `IsPerfect` and `DerivedSubgroup` available for tame rcwa groups, while testing wild groups for solvability or perfectness is currently not always feasible. The purpose of this section is to describe the specific attributes of tame groups which are needed for these computations.

3.4.1 RespectedPartition (of a tame rcwa group or -permutation)

▷ `RespectedPartition(G)` (attribute)
▷ `RespectedPartition(g)` (attribute)

Returns: a shortest and coarsest possible respected partition of the rcwa group `G` / of the rcwa permutation `g`.

A tame element $g \in \mathrm{RCWA}(R)$ permutes a partition of R into finitely many residue classes on all of which it is affine. Given a tame group $G < \mathrm{RCWA}(R)$, there is a common such partition for all elements of G. We call the mentioned partitions *respected partitions* of g or G, respectively.

An rcwa group or an rcwa permutation has a respected partition if and only if it is tame. This holds either by definition or by Theorem 2.5.8 in [Koh05], depending on how one introduces the notion of tameness.

There is an operation `RespectsPartition(G,P)` / `RespectsPartition(g,P)`, which tests whether `G` or `g` respects a given partition `P`. The permutation induced by `g` on `P` can be computed efficiently by `PermutationOpNC(g,P,OnPoints)`.

Example

```
gap> G := Group(ClassTransposition(0,4,1,6),ClassShift(0,2));
<rcwa group over Z with 2 generators>
gap> IsTame(G);
true
```

```
gap> Size(G);
infinity
gap> P := RespectedPartition(G);
[ 0(4), 2(4), 1(6), 3(6), 5(6) ]
```

3.4.2 ActionOnRespectedPartition & KernelOfActionOnRespectedPartition

▷ `ActionOnRespectedPartition(G)` (attribute)
▷ `KernelOfActionOnRespectedPartition(G)` (attribute)
▷ `RankOfKernelOfActionOnRespectedPartition(G)` (attribute)

Returns: the action of the tame rcwa group `G` on `RespectedPartition(G)`, the kernel of this action or the rank of the latter, respectively.

The method for `KernelOfActionOnRespectedPartition` uses the package Polycyclic [EHN13]. The rank of the largest free abelian subgroup of the kernel of the action of `G` on its stored respected partition is `RankOfKernelOfActionOnRespectedPartition(G)`.

Example

```
gap> G := Group(ClassTransposition(0,4,1,6),ClassShift(0,2));;
gap> H := ActionOnRespectedPartition(G);
Group([ (1,3), (1,2) ])
gap> H = Action(G,P);
true
gap> Size(H);
6
gap> K := KernelOfActionOnRespectedPartition(G);
<rcwa group over Z with 3 generators>
gap> RankOfKernelOfActionOnRespectedPartition(G);
3
gap> Index(G,K);
6
gap> List(GeneratorsOfGroup(K),Factorization);
[ [ ClassShift( 0(4) ) ], [ ClassShift( 2(4) ) ], [ ClassShift( 1(6) ) ] ]
gap> Image(IsomorphismPcpGroup(K));
Pcp-group with orders [ 0, 0, 0 ]
```

Let G be a tame rcwa group over $\mathbb{Z}$, let $\mathscr{P}$ be a respected partition of G and put $m := |\mathscr{P}|$. Then there is an rcwa permutation g which maps $\mathscr{P}$ to the partition of $\mathbb{Z}$ into the residue classes (mod m), and the conjugate G^g of G under such a permutation is integral (cf. [Koh05], Theorem 2.5.14).

The conjugate G^g can be determined by the operation `IntegralConjugate`, and the conjugating permutation g can be determined by the operation `IntegralizingConjugator`. Both operations are applicable to rcwa permutations as well. Note that a tame rcwa group does not determine its integral conjugate uniquely.

Example

```
gap> G := Group(ClassTransposition(0,4,1,6),ClassShift(0,2));;
gap> G^IntegralizingConjugator(G) = IntegralConjugate(G);
true
gap> RespectedPartition(G);
```

```
[ 0(4), 2(4), 1(6), 3(6), 5(6) ]
gap> RespectedPartition(G)^IntegralizingConjugator(G);
[ 0(5), 1(5), 2(5), 3(5), 4(5) ]
gap> last = RespectedPartition(IntegralConjugate(G));
true
```

3.5 Generating pseudo-random elements of RCWA(R) and CT(R)

There are methods for the operation `Random` for RCWA(R) and CT(R). These methods are designed to be suitable for generating interesting examples. No particular distribution is guaranteed.

Example

```
gap> elm := Random(RCWA(Integers));;
gap> Display(elm);

Rcwa permutation of Z with modulus 180

        /
        | 6n+12     if n in 2(10) U 4(10) U 6(10) U 8(10)
        | 3n+3      if n in 1(20) U 5(20) U 9(20) U 17(20)
        | 6n+10     if n in 0(10)
        | (n+1)/2   if n in 15(60) U 27(60) U 39(60) U 51(60)
        | (n+7)/2   if n in 19(60) U 31(60) U 43(60) U 55(60)
        | 3n+1      if n in 13(20)
        | (-n+17)/6 if n in 23(180) U 35(180) U 59(180) U 71(180) U
 n |-> <                    95(180) U 131(180) U 143(180) U 179(180)
        | (-n-1)/6  if n in 11(180) U 47(180) U 83(180) U 155(180)
        | (-n+7)/2  if n in 3(60)
        | (n+3)/2   if n in 7(60)
        | (n-17)/6  if n in 107(180)
        | (-n+11)/6 if n in 119(180)
        | (-n+29)/6 if n in 167(180)
        |
        \
```

The elements which are returned by this method are obtained by multiplying class shifts (see `ClassShift` (2.2.1)), class reflections (see `ClassReflection` (2.2.2)) and class transpositions (see `ClassTransposition` (2.2.3)). These factors can be retrieved by factoring:

Example

```
gap> Factorization(elm);
[ ClassTransposition(0,2,3,4), ClassTransposition(1,2,4,6), ClassShift(0,2),
  ClassShift(1,3), ClassReflection(2,5), ClassReflection(1,3),
  ClassReflection(1,2) ]
```

3.6 The categories of residue-class-wise affine groups

3.6.1 IsRcwaGroup

▷ `IsRcwaGroup(G)` (filter)
▷ `IsRcwaGroupOverZ(G)` (filter)
▷ `IsRcwaGroupOverZ_pi(G)` (filter)
▷ `IsRcwaGroupOverGFqx(G)` (filter)

Returns: `true` if *G* is an rcwa group, an rcwa group over the ring of integers, an rcwa group over a semilocalization of the ring of integers or an rcwa group over a polynomial ring in one variable over a finite field, respectively, and `false` otherwise.

Often the same methods can be used for rcwa groups over the ring of integers and over its semilocalizations. For this reason there is a category `IsRcwaGroupOverZOrZ_pi` which is the union of `IsRcwaGroupOverZ` and `IsRcwaGroupOverZ_pi`.

To allow distinguishing the groups RCWA(R) and CT(R) from others, they have the characteristic property `IsNaturalRCWA` or `IsNaturalCT`, respectively.

Chapter 4

Residue-Class-Wise Affine Monoids

In this short chapter, we describe how to compute with residue-class-wise affine monoids. *Residue-class-wise affine* monoids, or *rcwa* monoids for short, are monoids whose elements are residue-class-wise affine mappings.

4.1 Constructing residue-class-wise affine monoids

As any other monoids in GAP, residue-class-wise affine monoids can be constructed by `Monoid` or `MonoidByGenerators`.

Example

```
gap> M := Monoid(RcwaMapping([[ 0,1,1],[1,1,1]]),
>                RcwaMapping([[-1,3,1],[0,2,1]]));
<rcwa monoid over Z with 2 generators>
gap> Size(M);
11
gap> Display(MultiplicationTable(M));
[ [   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11 ],
  [   2,   8,   5,  11,   8,   3,  10,   5,   2,   8,   5 ],
  [   3,  10,  11,   5,   5,   5,   8,   8,   8,   2,   3 ],
  [   4,   9,   6,   8,   8,   8,   5,   5,   5,   7,   4 ],
  [   5,   8,   5,   8,   8,   8,   5,   5,   5,   8,   5 ],
  [   6,   7,   4,   8,   8,   8,   5,   5,   5,   9,   6 ],
  [   7,   5,   8,   6,   5,   4,   9,   8,   7,   5,   8 ],
  [   8,   5,   8,   5,   5,   5,   8,   8,   8,   5,   8 ],
  [   9,   5,   8,   4,   5,   6,   7,   8,   9,   5,   8 ],
  [  10,   8,   5,   3,   8,  11,   2,   5,  10,   8,   5 ],
  [  11,   2,   3,   5,   5,   5,   8,   8,   8,  10,  11 ] ]
```

There are methods for the operations `View`, `Display`, `Print` and `String` which are applicable to rcwa monoids. All rcwa monoids over a ring R are submonoids of Rcwa(R). The monoid Rcwa(R) itself is not finitely generated, thus cannot be constructed as described above. It is handled as a special case:

4.1.1 Rcwa (the monoid formed by all rcwa mappings of a ring)

▷ Rcwa(*R*) (function)

Returns: the monoid Rcwa(*R*) of all residue-class-wise affine mappings of the ring *R*.

Example

```
gap> RcwaZ := Rcwa(Integers);
Rcwa(Z)
gap> IsSubset(RcwaZ,M);
true
```

In our methods to construct rcwa groups, two kinds of mappings played a crucial role, namely the restriction monomorphisms (cf. `Restriction` (3.1.6)) and the induction epimorphisms (cf. `Induction` (3.1.7)). The restriction monomorphisms extend in a natural way to the monoids Rcwa(R), and the induction epimorphisms have corresponding generalizations, also. Therefore the operations `Restriction` and `Induction` can be applied to rcwa monoids as well:

Example

```
gap> M2 := Restriction(M,2*One(Rcwa(Integers)));
<rcwa monoid over Z with 2 generators, of size 11>
gap> Support(M2);
0(2)
gap> Action(M2,ResidueClass(1,2));
Trivial rcwa group over Z
gap> Induction(M2,2*One(Rcwa(Integers))) = M;
true
```

4.2 Computing with residue-class-wise affine monoids

There is a method for `Size` which computes the order of an rcwa monoid. Further there is a method for `in` which checks whether a given rcwa mapping lies in a given rcwa monoid (membership test), and there is a method for `IsSubset` which checks for a submonoid relation.

There are also methods for `Support`, `Modulus`, `IsTame`, `PrimeSet`, `IsIntegral`, `IsClassWiseOrderPreserving` and `IsSignPreserving` available for rcwa monoids.

The *support* of an rcwa monoid is the union of the supports of its elements. The *modulus* of an rcwa monoid is the lcm of the moduli of its elements in case such an lcm exists and 0 otherwise. An rcwa monoid is called *tame* if its modulus is nonzero, and *wild* otherwise. The *prime set* of an rcwa monoid is the union of the prime sets of its elements. An rcwa monoid is called *integral*, *class-wise order-preserving* or *sign-preserving* if all of its elements are so.

Example

```
gap> f1 := RcwaMapping([[-1, 1, 1],[ 0,-1, 1]]);;
gap> f2 := RcwaMapping([[ 1,-1, 1],[-1,-2, 1],[-1, 2, 1]]);;
gap> f3 := RcwaMapping([[ 1, 0, 1],[-1, 0, 1]]);;
gap> N := Monoid(f1,f2,f3);;
gap> Size(N);
366
```

```
gap> List([Monoid(f1,f2),Monoid(f1,f3),Monoid(f2,f3)],Size);
[ 96, 6, 66 ]
gap> f1*f2*f3 in N;
true
gap> IsSubset(N,M);
false
gap> IsSubset(N,Monoid(f1*f2,f3*f2));
true
gap> Support(N);
Integers
gap> Modulus(N);
6
gap> IsTame(N) and IsIntegral(N);
true
gap> IsClassWiseOrderPreserving(N) or IsSignPreserving(N);
false
gap> Collected(List(AsList(N),Image)); # The images of the elements of N.
[ [ Integers, 2 ], [ 1(2), 2 ], [ Z \ 1(3), 32 ], [ 0(6), 44 ],
  [ 0(6) U 1(6), 4 ], [ Z \ 4(6) U 5(6), 32 ], [ 0(6) U 2(6), 4 ],
  [ 0(6) U 5(6), 4 ], [ 1(6), 44 ], [ 1(6) U [ -1 ], 2 ],
  [ 1(6) U 3(6), 4 ], [ 1(6) U 5(6), 40 ], [ 2(6), 44 ],
  [ 2(6) U 3(6), 4 ], [ 3(6), 44 ], [ 3(6) U 5(6), 4 ], [ 5(6), 44 ],
  [ 5(6) U [ 1 ], 2 ], [ [ -5 ], 1 ], [ [ -4 ], 1 ], [ [ -3 ], 1 ],
  [ [ -1 ], 1 ], [ [ 0 ], 1 ], [ [ 1 ], 1 ], [ [ 2 ], 1 ], [ [ 3 ], 1 ],
  [ [ 5 ], 1 ], [ [ 6 ], 1 ] ]
```

Finite forward orbits under the action of an rcwa monoid can be found by the operation `ShortOrbits`:

4.2.1 ShortOrbits (for rcwa monoid, set of points and bound on length)

▷ `ShortOrbits(M, S, maxlng)` (method)

Returns: a list of finite forward orbits of the rcwa monoid *M* of length at most *maxlng* which start at points in the set *S*.

Example

```
gap> ShortOrbits(M,[-5..5],20);
[ [ -5, -4, 1, 2, 7, 8 ], [ -3, -2, 1, 2, 5, 6 ], [ -1, 0, 1, 2, 3, 4 ] ]
gap> Print(Action(M,last[1]),"\n");
Monoid( [ Transformation( [ 2, 3, 4, 3, 6, 3 ] ),
  Transformation( [ 4, 5, 4, 3, 4, 1 ] ) ] )
gap> orbs := ShortOrbits(N,[0..10],100);
[ [ -5, -4, -3, -1, 0, 1, 2, 3, 5, 6 ],
  [ -11, -10, -9, -7, -6, -5, -4, -3, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
      11, 12 ],
  [ -17, -16, -15, -13, -12, -11, -10, -9, -7, -6, -5, -4, -3, -1, 0, 1,
      2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18 ] ]
gap> quots := List(orbs,orb->Action(N,orb));;
gap> List(quots,Size);
[ 268, 332, 366 ]
```

Balls of given radius around an element of an rcwa monoid can be computed by the operation `Ball`. This operation can also be used for computing forward orbits or subsets of such under the action of an rcwa monoid:

4.2.2 Ball (for monoid, element and radius or monoid, point, radius and action)

▷ `Ball(M, f, r)` (method)
▷ `Ball(M, p, r, action)` (method)

Returns: the ball of radius *r* around the element *f* in the monoid *M*, respectively the ball of radius *r* around the point *p* under the action *action* of the monoid *M*.

All balls are understood with respect to `GeneratorsOfMonoid(M)`. As membership tests can be expensive, the first-mentioned method does not check whether *f* is indeed an element of *M*. The methods require that point- / element comparisons are cheap. They are not only applicable to rcwa monoids. If the option *Spheres* is set, the ball is split up and returned as a list of spheres.

Example

```
gap> List([0..12],k->Length(Ball(N,One(N),k)));
[ 1, 4, 11, 26, 53, 99, 163, 228, 285, 329, 354, 364, 366 ]
gap> Ball(N,[0..3],2,OnTuples);
[ [ -3, 3, 3, 3 ], [ -1, -3, 0, 2 ], [ -1, -1, -1, -1 ],
  [ -1, -1, 1, -1 ], [ -1, 1, 1, 1 ], [ -1, 3, 0, -4 ], [ 0, -1, 2, -3 ],
  [ 0, 1, 2, 3 ], [ 1, -1, -1, -1 ], [ 1, 3, 0, 2 ], [ 3, -4, -1, 0 ] ]
gap> l := 2*IdentityRcwaMappingOfZ; r := l+1;
Rcwa mapping of Z: n -> 2n
Rcwa mapping of Z: n -> 2n + 1
gap> Ball(Monoid(l,r),1,4,OnPoints:Spheres);
[ [ 1 ], [ 2, 3 ], [ 4, 5, 6, 7 ], [ 8, 9, 10, 11, 12, 13, 14, 15 ],
  [ 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31 ] ]
```

Chapter 5

Residue-Class-Wise Affine Mappings, Groups and Monoids over $\mathbb{Z}^2$

This chapter describes how to compute with residue-class-wise affine mappings of $\mathbb{Z}^2$ and with groups and monoids formed by them.

The rings on which we have defined residue-class-wise affine mappings so far have all been principal ideal domains, and it has been crucial that all nontrivial principal ideals had finite index. However, the rings $\mathbb{Z}^d$, $d > 1$ are not principal ideal domains. Furthermore, their principal ideals have infinite index. Therefore as moduli of residue-class-wise affine mappings we can only use lattices of full rank, for these are precisely the ideals of $\mathbb{Z}^d$ of finite index. However, on the other hand we can also be more permissive and look at $\mathbb{Z}^d$ not as a ring, but rather as a free $\mathbb{Z}$-module. The consequence of this is that then an affine mapping of $\mathbb{Z}^d$ is not just given by $v \mapsto (av+b)/c$ for some $a,b,c \in \mathbb{Z}^d$, but rather by $v \mapsto (vA+b)/c$, where $A \in \mathbb{Z}^{d \times d}$. Also for technical reasons concerning the implementation in GAP, looking at $\mathbb{Z}^d$ as a free $\mathbb{Z}$-module is preferable – in GAP, `Integers^d` is not a ring, and multiplying lists of integers means forming their scalar product.

5.1 The definition of residue-class-wise affine mappings of $\mathbb{Z}^d$

Let $d \in \mathbb{N}$. We call a mapping $f : \mathbb{Z}^d \to \mathbb{Z}^d$ *residue-class-wise affine* if there is a lattice $L = \mathbb{Z}^d M$ where $M \in \mathbb{Z}^{d \times d}$ is a matrix of full rank, such that the restrictions of f to the residue classes $r + L \in \mathbb{Z}^d/L$ are all affine. This means that for any residue class $r + L \in \mathbb{Z}^d/L$, there is a matrix $A_{r+L} \in \mathbb{Z}^{d \times d}$, a vector $b_{r+L} \in \mathbb{Z}^d$ and a positive integer c_{r+L} such that the restriction of f to $r+L$ is given by $f|_{r+L} :\ r+L \ \to\ \mathbb{Z}^d,\ \ v \ \mapsto\ (v \cdot A_{r+L} + b_{r+L})/c_{r+L}$. For reasons of uniqueness, we assume that L is chosen maximal with respect to inclusion, and that no prime factor of c_{r+L} divides all coefficients of A_{r+L} and b_{r+L}.

We call the lattice L the *modulus* of f, written $\mathrm{Mod}(f)$. Further we define the *prime set* of f as the set of all primes which divide the determinant of at least one of the coefficients A_{r+L} or which divide the determinant of M, and we call the mapping f *class-wise translating* if all coefficients A_{r+L} are identity matrices and all coefficients c_{r+L} are equal to 1.

For the sake of simplicity, we identify a lattice with the Hermite normal form of the matrix by whose rows it is spanned.

5.2 Entering residue-class-wise affine mappings of $\mathbb{Z}^2$

Residue-class-wise affine mappings of $\mathbb{Z}^2$ can be entered using the general constructor `RcwaMapping` (2.2.5) or the more specialized functions `ClassTransposition` (2.2.3), `ClassRotation` (2.2.4) and `ClassShift` (2.2.1). The arguments differ only slightly.

5.2.1 RcwaMapping (the general constructor; methods for $\mathbb{Z}^2$)

▷ `RcwaMapping(R, L, coeffs)` (method)
▷ `RcwaMapping(P1, P2)` (method)
▷ `RcwaMapping(cycles)` (method)
▷ `RcwaMapping(f, g)` (method)

Returns: an rcwa mapping of $\mathbb{Z}^2$.

The above methods return

(a) the rcwa mapping of `R = Integers^2` with modulus `L` and coefficients `coeffs`,

(b) an rcwa permutation which induces a bijection between the partitions `P1` and `P2` of $\mathbb{Z}^2$ into residue classes and which is affine on the elements of `P1`,

(c) an rcwa permutation with "residue class cycles" given by a list `cycles` of lists of pairwise disjoint residue classes of $\mathbb{Z}^2$ each of which it permutes cyclically, and

(d) the rcwa mapping of $\mathbb{Z}^2$ whose projections to the coordinates are given by `f` and `g`,

respectively.

The modulus of an rcwa mapping of $\mathbb{Z}^2$ is a lattice of full rank. It is represented by a matrix L in Hermite normal form, whose rows are the spanning vectors.

A coefficient list for an rcwa mapping of $\mathbb{Z}^2$ with modulus L consists of $|\det(L)|$ coefficient triples $[A_{r+\mathbb{Z}^2L}, b_{r+\mathbb{Z}^2L}, c_{r+\mathbb{Z}^2L}]$. The entries $A_{r+\mathbb{Z}^2L}$ are 2×2 integer matrices, the $b_{r+\mathbb{Z}^2L}$ are elements of $\mathbb{Z}^2$, i.e. lists of two integers, and the $c_{r+\mathbb{Z}^2L}$ are integers. The ordering of the coefficient triples is determined by the ordering of the representatives of the residue classes $r+\mathbb{Z}^2L$ in the sorted list returned by `AllResidues(Integers^2,L)`.

The methods for the operation `RcwaMapping` perform a number of argument checks, which can be skipped by using `RcwaMappingNC` instead.

Last but not least, regarding Method (d) it should be mentioned that only very special rcwa mappings of $\mathbb{Z}^2$ have projections to coordinates.

Example

```
gap> R := Integers^2;;
gap> twice := RcwaMapping(R,[[1,0],[0,1]],
>                        [[[[2,0],[0,2]],[0,0],1]]);       # method (a)
Rcwa mapping of Z^2: (m,n) -> (2m,2n)
gap> [4,5]^twice;
[ 8, 10 ]
gap> twice1 := RcwaMapping(R,[[1,0],[0,1]],
>                         [[[[2,0],[0,1]],[0,0],1]]);      # method (a)
Rcwa mapping of Z^2: (m,n) -> (2m,n)
gap> [4,5]^twice1;
[ 8, 5 ]
gap> Image(twice1);
```

```
(0,0)+(2,0)Z+(0,1)Z
gap> hyperbolic := RcwaMapping(R,[[1,0],[0,2]],
>                                [[[[4,0],[0,1]],[0, 0],2],
>                                 [[[4,0],[0,1]],[2,-1],2]]); # method (a)
<rcwa mapping of Z^2 with modulus (1,0)Z+(0,2)Z>
gap> IsBijective(hyperbolic);
true
gap> Display(hyperbolic);

Rcwa permutation of Z^2 with modulus (1,0)Z+(0,2)Z

              /
              | (2m,n/2)       if (m,n) in (0,0)+(1,0)Z+(0,2)Z
 (m,n) |-> <  (2m+1,(n-1)/2) if (m,n) in (0,1)+(1,0)Z+(0,2)Z
              |
              \

gap> Trajectory(hyperbolic,[0,10000],20);
[ [ 0, 10000 ], [ 0, 5000 ], [ 0, 2500 ], [ 0, 1250 ], [ 0, 625 ],
  [ 1, 312 ], [ 2, 156 ], [ 4, 78 ], [ 8, 39 ], [ 17, 19 ], [ 35, 9 ],
  [ 71, 4 ], [ 142, 2 ], [ 284, 1 ], [ 569, 0 ], [ 1138, 0 ],
  [ 2276, 0 ], [ 4552, 0 ], [ 9104, 0 ], [ 18208, 0 ] ]
gap> P1 := AllResidueClassesModulo(R,[[2,1],[0,2]]);
[ (0,0)+(2,1)Z+(0,2)Z, (0,1)+(2,1)Z+(0,2)Z, (1,0)+(2,1)Z+(0,2)Z,
  (1,1)+(2,1)Z+(0,2)Z ]
gap> P2 := AllResidueClassesModulo(R,[[1,0],[0,4]]);
[ (0,0)+(1,0)Z+(0,4)Z, (0,1)+(1,0)Z+(0,4)Z, (0,2)+(1,0)Z+(0,4)Z,
  (0,3)+(1,0)Z+(0,4)Z ]
gap> g := RcwaMapping(P1,P2);                              # method (b)
<rcwa permutation of Z^2 with modulus (2,1)Z+(0,2)Z>
gap> P1^g = P2;
true
gap> Display(g:AsTable);

Rcwa permutation of Z^2 with modulus (2,1)Z+(0,2)Z

   [m,n] mod (2,1)Z+(0,2)Z   |              Image of [m,n]
-----------------------------+-----------------------------------------
 [0,0]                       | [m/2,-m+2n]
 [0,1]                       | [m/2,-m+2n-1]
 [1,0]                       | [(m-1)/2,-m+2n+3]
 [1,1]                       | [(m-1)/2,-m+2n+2]

gap> classes := List([[[0,0],[[2,1],[0,2]]],[[1,0],[[2,1],[0,4]]],
>                     [[1,1],[[4,2],[0,4]]]],ResidueClass);
[ (0,0)+(2,1)Z+(0,2)Z, (1,0)+(2,1)Z+(0,4)Z, (1,1)+(4,2)Z+(0,4)Z ]
gap> g := RcwaMapping([classes]);                          # method (c)
<rcwa permutation of Z^2 with modulus (4,2)Z+(0,4)Z, of order 3>
gap> Permutation(g,classes);
(1,2,3)
gap> Support(g);
(0,0)+(2,1)Z+(0,2)Z U (1,0)+(2,1)Z+(0,4)Z U (1,1)+(4,2)Z+(0,4)Z
gap> Display(g);
```

```
Rcwa permutation of Z^2 with modulus (4,2)Z+(0,4)Z, of order 3

             /
             | (m+1,(-m+4n)/2)   if (m,n) in (0,0)+(2,1)Z+(0,2)Z
             | (2m-1,(m+2n+1)/2) if (m,n) in (1,0)+(2,1)Z+(0,4)Z
 (m,n) |-> <  ((m-1)/2,(n-1)/2) if (m,n) in (1,1)+(4,2)Z+(0,4)Z
             | (m,n)             otherwise
             |
             \

gap> g := RcwaMapping(ClassTransposition(0,2,1,2),
>                     ClassReflection(0,2));                # method (d)
<rcwa mapping of Z^2 with modulus (2,0)Z+(0,2)Z>
gap> Display(g);

Rcwa mapping of Z^2 with modulus (2,0)Z+(0,2)Z

             /
             | (m+1,-n) if (m,n) in (0,0)+(2,0)Z+(0,2)Z
             | (m+1,n)  if (m,n) in (0,1)+(2,0)Z+(0,2)Z
 (m,n) |-> <  (m-1,-n) if (m,n) in (1,0)+(2,0)Z+(0,2)Z
             | (m-1,n)  if (m,n) in (1,1)+(2,0)Z+(0,2)Z
             |
             \

gap> g^2;
IdentityMapping( ( Integers^2 ) )
gap> List(ProjectionsToCoordinates(g),Factorization);
[ [ ( 0(2), 1(2) ) ], [ ClassReflection( 0(2) ) ] ]
```

5.2.2 ClassTransposition (for $\mathbb{Z}^2$)

▷ `ClassTransposition(r1, L1, r2, L2)` (function)

▷ `ClassTransposition(cl1, cl2)` (function)

Returns: the class transposition $\tau_{r_1+\mathbb{Z}^2L_1,r_2+\mathbb{Z}^2L_2}$.

Let $d \in \mathbb{N}$, and let $L_1, L_2 \in \mathbb{Z}^{d\times d}$ be matrices of full rank which are in Hermite normal form. Further let $r_1 + \mathbb{Z}^d L_1$ and $r_2 + \mathbb{Z}^d L_2$ be disjoint residue classes, and assume that the representatives r_1 and r_2 are reduced modulo $\mathbb{Z}^d L_1$ and $\mathbb{Z}^d L_2$, respectively. Then we define the *class transposition* $\tau_{r_1+\mathbb{Z}^dL_1,r_2+\mathbb{Z}^dL_2} \in \mathrm{Sym}(\mathbb{Z}^d)$ as the involution which interchanges $r_1 + kL_1$ and $r_2 + kL_2$ for all $k \in \mathbb{Z}^d$.

The class transposition $\tau_{r_1+\mathbb{Z}^dL_1,r_2+\mathbb{Z}^dL_2}$ interchanges the residue classes $r_1 + \mathbb{Z}^d L_1$ and $r_2 + \mathbb{Z}^d L_2$, and fixes the complement of their union pointwise. The set of all class transpositions of $\mathbb{Z}^d$ generates the simple group $\mathrm{CT}(\mathbb{Z}^d)$ (cf. [Koh13]).

In the four-argument form, the arguments `r1`, `L1`, `r2` and `L2` stand for r_1, L_1, r_2 and L_2, respectively. In the two-argument form, the arguments `cl1` and `cl2` stand for the residue classes $r_1 + \mathbb{Z}^2 L_1$ and $r_2 + \mathbb{Z}^2 L_2$, respectively. Enclosing the argument list in list brackets is permitted. The residue classes $r_1 + \mathbb{Z}^2 L_1$ and $r_2 + \mathbb{Z}^2 L_2$ are stored as an attribute `TransposedClasses`.

There is also a method for `SplittedClassTransposition` available for class transpositions of $\mathbb{Z}^2$. This method takes as first argument the class transposition, and as second argument a list of two integers. These integers are the numbers of parts into which the class transposition is to be sliced in each dimension. Note that the product of the returned class transpositions is not always equal to the class transposition passed as first argument. However this equality holds if the first entry of the second argument is 1.

Example

```
gap> ct := ClassTransposition([0,0],[[2,1],[0,2]],[1,0],[[2,1],[0,4]]);
( (0,0)+(2,1)Z+(0,2)Z, (1,0)+(2,1)Z+(0,4)Z )
gap> Display(ct);

Rcwa permutation of Z^2 with modulus (2,1)Z+(0,4)Z, of order 2

             /
             | (m+1,(-m+4n)/2)  if (m,n) in (0,0)+(2,1)Z+(0,2)Z
 (m,n) |-> <  (m-1,(m+2n-1)/4) if (m,n) in (1,0)+(2,1)Z+(0,4)Z
             | (m,n)            otherwise
             \

gap> TransposedClasses(ct);
[ (0,0)+(2,1)Z+(0,2)Z, (1,0)+(2,1)Z+(0,4)Z ]
gap> ct = ClassTransposition(last);
true
gap> SplittedClassTransposition(ct,[1,2]);
[ ( (0,0)+(2,1)Z+(0,4)Z, (1,0)+(2,1)Z+(0,8)Z ),
  ( (0,2)+(2,1)Z+(0,4)Z, (1,4)+(2,1)Z+(0,8)Z ) ]
gap> Product(last) = ct;
true
gap> SplittedClassTransposition(ct,[2,1]);
[ ( (0,0)+(4,0)Z+(0,2)Z, (1,0)+(4,2)Z+(0,4)Z ),
  ( (2,1)+(4,0)Z+(0,2)Z, (3,1)+(4,2)Z+(0,4)Z ) ]
gap> Product(last) = ct;
false
```

5.2.3 ClassRotation (for $\mathbb{Z}^2$)

▷ `ClassRotation(r, L, u)` (function)

▷ `ClassRotation(cl, u)` (function)

Returns: the class rotation $\rho_{r(m),u}$.

Let $d \in \mathbb{N}$. Given a residue class $r+\mathbb{Z}^d L$ and a matrix $u \in \mathrm{GL}(d,\mathbb{Z})$, the *class rotation* $\rho_{r+\mathbb{Z}^d L,u}$ is the rcwa mapping which maps $v \in r+\mathbb{Z}^d L$ to $vu+r(1-u)$ and which fixes $\mathbb{Z}^d \setminus r+\mathbb{Z}^d L$ pointwise. In the two-argument form, the argument `cl` stands for the residue class $r+\mathbb{Z}^d L$. Enclosing the argument list in list brackets is permitted. The argument `u` is stored as an attribute `RotationFactor`.

Example

```
gap> interchange := ClassRotation([0,0],[[1,0],[0,1]],[[0,1],[1,0]]);
ClassRotation( Z^2, [ [ 0, 1 ], [ 1, 0 ] ] )
gap> Display(interchange);
Rcwa permutation of Z^2: (m,n) -> (n,m)
gap> classes := AllResidueClassesModulo(Integers^2,[[2,1],[0,3]]);
[ (0,0)+(2,1)Z+(0,3)Z, (0,1)+(2,1)Z+(0,3)Z, (0,2)+(2,1)Z+(0,3)Z,
```

```
  (1,0)+(2,1)Z+(0,3)Z, (1,1)+(2,1)Z+(0,3)Z, (1,2)+(2,1)Z+(0,3)Z ]
gap> transvection := ClassRotation(classes[5],[[1,1],[0,1]]);
ClassRotation((1,1)+(2,1)Z+(0,3)Z,[[1,1],[0,1]])
gap> Display(transvection);

Tame rcwa permutation of Z^2 with modulus (2,1)Z+(0,3)Z, of order infinity

              /
              | (m,(3m+2n-3)/2) if (m,n) in (1,1)+(2,1)Z+(0,3)Z
 (m,n) |-> <  (m,n)           otherwise
              |
              \
```

5.2.4 ClassShift (for $\mathbb{Z}^2$)

▷ `ClassShift(r, L, k)` (function)

▷ `ClassShift(cl, k)` (function)

Returns: the class shift $\nu_{r+\mathbb{Z}^d L,k}$.

Let $d \in \mathbb{N}$. Given a residue class $r+\mathbb{Z}^d L$ and an integer $k \in \{1,\dots,d\}$, the *class shift* $\nu_{r+\mathbb{Z}^d L,k}$ is the rcwa mapping which maps $v \in r+\mathbb{Z}^d L$ to $v+L_k$ and which fixes $\mathbb{Z}^d \setminus r+\mathbb{Z}^d L$ pointwise. Here L_k denotes the kth row of L.

In the two-argument form, the argument `cl` stands for the residue class $r+\mathbb{Z}^d L$. Enclosing the argument list in list brackets is permitted.

Example

```
gap> shift1 := ClassShift([0,0],[[1,0],[0,1]],1);
ClassShift( Z^2, 1 )
gap> Display(shift1);
Tame rcwa permutation of Z^2: (m,n) -> (m+1,n)
gap> s := ClassShift(ResidueClass([1,1],[[2,1],[0,2]]),2);
ClassShift((1,1)+(2,1)Z+(0,2)Z,2)
gap> Display(s);

Tame rcwa permutation of Z^2 with modulus (2,1)Z+(0,2)Z, of order infinity

              /
              | (m,n+2) if (m,n) in (1,1)+(2,1)Z+(0,2)Z
 (m,n) |-> <  (m,n)   if (m,n) in (0,0)+(2,0)Z+(0,1)Z U
              |                   (1,0)+(2,1)Z+(0,2)Z
              \
```

As for other rings, class transpositions, class rotations and class shifts of $\mathbb{Z}^2$ have the distinguishing properties `IsClassTransposition`, `IsClassRotation` and `IsClassShift`.

5.3 Methods for residue-class-wise affine mappings of $\mathbb{Z}^2$

There are methods available for rcwa mappings of $\mathbb{Z}^2$ for the following general operations:

Output
: `View`, `Display`, `Print`, `String`, `LaTeXStringRcwaMapping`, `LaTeXAndXDVI`.

Access to components
: `Modulus`, `Coefficients`.

Attributes
: `Support` / `MovedPoints`, `Order`, `Multiplier`, `Divisor`, `PrimeSet`, `One`, `Zero`.

Properties
: `IsInjective`, `IsSurjective`, `IsBijective`, `IsTame`, `IsIntegral`, `IsBalanced`, `IsClassWiseOrderPreserving`, `IsOne`, `IsZero`.

Action on $\mathbb{Z}^d$
: `^` (for points / finite sets / residue class unions), `Trajectory`, `ShortCycles`, `Multpk`, `ClassWiseOrderPreservingOn`, `ClassWiseOrderReversingOn`, `ClassWiseConstantOn`.

Arithmetical operations
: `=`, `*` (multiplication / composition and multiplication by a 2×2 matrix or an integer), `^` (exponentiation and conjugation), `Inverse`, `+` (addition of a constant).

The above operations are documented either in the GAP Reference Manual or earlier in this manual. The operations which are special for rcwa mappings of $\mathbb{Z}^2$ are described in the sequel.

5.3.1 ProjectionsToCoordinates (for an rcwa mapping of Z x Z)

▷ `ProjectionsToCoordinates(`*f*`)` (attribute)

Returns: the projections of the rcwa mapping *f* of $\mathbb{Z}^2$ to the coordinates if such projections exist, and `fail` otherwise.

An rcwa mapping can be projected to the first / second coordinate if and only if the first / second coordinate of the image of a point depends only on the first / second coordinate of the preimage. Note that this is a very strong and restrictive condition.

Example

```
gap> f := RcwaMapping(ClassTransposition(0,2,1,2),ClassReflection(0,2));;
gap> Display(f);

Rcwa mapping of Z^2 with modulus (2,0)Z+(0,2)Z

            /
            | (m+1,-n) if (m,n) in (0,0)+(2,0)Z+(0,2)Z
            | (m+1,n)  if (m,n) in (0,1)+(2,0)Z+(0,2)Z
 (m,n) |-> <  (m-1,-n) if (m,n) in (1,0)+(2,0)Z+(0,2)Z
            | (m-1,n)  if (m,n) in (1,1)+(2,0)Z+(0,2)Z
            |
            \

gap> List(ProjectionsToCoordinates(f),Factorization);
[ [ ( 0(2), 1(2) ) ], [ ClassReflection( 0(2) ) ] ]
```

5.4 Methods for residue-class-wise affine groups and -monoids over $\mathbb{Z}^2$

Residue-class-wise affine groups over $\mathbb{Z}^2$ can be entered by `Group`, `GroupByGenerators` and `GroupWithGenerators`, like any groups in GAP. Likewise, residue-class-wise affine monoids over $\mathbb{Z}^2$ can be entered by `Monoid` and `MonoidByGenerators`. The groups RCWA($\mathbb{Z}^2$) and CT($\mathbb{Z}^2$) are entered as `RCWA(Integers^2)` and `CT(Integers^2)`, respectively. The monoid Rcwa($\mathbb{Z}^2$) is entered as `Rcwa(Integers^2)`.

There are methods provided for the operations `Size`, `IsIntegral`, `IsClassWiseTranslating`, `IsTame`, `Modulus`, `Multiplier` and `Divisor`.

There are methods for `IsomorphismRcwaGroup` (3.1.1) which embed the groups SL(2,$\mathbb{Z}$) and GL(2,$\mathbb{Z}$) into RCWA($\mathbb{Z}^2$) in such a way that the support of the image is a specified residue class:

5.4.1 IsomorphismRcwaGroup (Embeddings of SL(2,$\mathbb{Z}$) and GL(2,$\mathbb{Z}$))

▷ `IsomorphismRcwaGroup(`*sl2z, cl*`)` (attribute)
▷ `IsomorphismRcwaGroup(`*gl2z, cl*`)` (attribute)

Returns: a monomorphism from *sl2z* respectively *gl2z* to RCWA($\mathbb{Z}^2$), such that the support of the image is the residue class *cl* and the generators are affine on *cl*.

Example

```
gap> sl := SL(2,Integers);
SL(2,Integers)
gap> phi := IsomorphismRcwaGroup(sl,ResidueClass([1,0],[[2,2],[0,3]]));
[ [ [ 0, 1 ], [ -1, 0 ] ], [ [ 1, 1 ], [ 0, 1 ] ] ] ->
[ ClassRotation((1,0)+(2,2)Z+(0,3)Z,[[0,1],[-1,0]]),
  ClassRotation((1,0)+(2,2)Z+(0,3)Z,[[1,1],[0,1]]) ]
gap> Support(Image(phi));
(1,0)+(2,2)Z+(0,3)Z
gap> gl := GL(2,Integers);
GL(2,Integers)
gap> phi := IsomorphismRcwaGroup(gl,ResidueClass([1,0],[[2,2],[0,3]]));
[ [ [ 0, 1 ], [ 1, 0 ] ], [ [ -1, 0 ], [ 0, 1 ] ],
  [ [ 1, 1 ], [ 0, 1 ] ] ] ->
[ ClassRotation((1,0)+(2,2)Z+(0,3)Z,[[0,1],[1,0]]),
  ClassRotation((1,0)+(2,2)Z+(0,3)Z,[[-1,0],[0,1]]),
  ClassRotation((1,0)+(2,2)Z+(0,3)Z,[[1,1],[0,1]]) ]
gap> [[-47,-37],[61,48]]^phi;
ClassRotation((1,0)+(2,2)Z+(0,3)Z,[[-47,-37],[61,48]])
gap> Display(last:AsTable);

Rcwa permutation of Z^2 with modulus (2,2)Z+(0,3)Z, of order 6

   [m,n] mod (2,2)Z+(0,3)Z    |               Image of [m,n]
------------------------------+------------------------------------------
 [0,0] [0,1] [0,2] [1,1]      |
 [1,2]                        | [m,n]
 [1,0]                        | [(-263m+122n+266)/3,(-1147m+532n+1147)/6]
```

The function `DrawOrbitPicture` (3.3.3) can also be used to depict orbits under the action of rcwa groups over $\mathbb{Z}^2$. Further there is a function which depicts residue class unions of $\mathbb{Z}^2$ and partitions of $\mathbb{Z}^2$ into such:

5.4.2 DrawGrid

▷ `DrawGrid(U, yrange, xrange, filename)` (function)

▷ `DrawGrid(P, yrange, xrange, filename)` (function)

Returns: nothing.

This function depicts the residue class union *U* of $\mathbb{Z}^2$ or the partition *P* of $\mathbb{Z}^2$ into residue class unions, respectively. The arguments *yrange* and *xrange* are the coordinate ranges of the rectangular snippet to be drawn, and the argument *filename* is the name, i.e. the full path name, of the output file. If the first argument is a residue class union, the output picture is black-and-white, where black pixels represent members of *U* and white pixels represent non-members. If the first argument is a partition of $\mathbb{Z}^2$ into residue class unions, the produced picture is colored, and different colors are used to denote membership in different parts.

Chapter 6

Databases of Residue-Class-Wise Affine Groups and -Mappings

The RCWA package contains a number of databases of rcwa groups and rcwa mappings. They can be loaded into a GAP session by the functions described in this chapter.

6.1 The collection of examples

6.1.1 LoadRCWAExamples

▷ `LoadRCWAExamples()` (function)

Returns: the name of the variable to which the record containing the collection of examples of rcwa groups and -mappings loaded from the file `pkg/rcwa/examples/examples.g` got bound.

The components of the examples record are records which contain the individual groups and mappings. A detailed description of some of the examples can be found in Chapter 7.

Example

```
gap> LoadRCWAExamples();
"RCWAExamples"
gap> Set(RecNames(RCWAExamples));
[ "AbelianGroupOverPolynomialRing", "Basics", "CT3Z", "CTPZ",
  "CheckingForSolvability", "ClassSwitches",
  "ClassTranspositionProducts", "ClassTranspositionsAsCommutators",
  "CollatzFactorizationOld", "CollatzMapping", "CollatzlikePerms",
  "CoprimeMultDiv", "F2_PSL2Z", "Farkas", "FiniteQuotients",
  "FiniteVsDenseCycles", "GF2xFiniteCycles", "GrigorchukQuotients",
  "Hexagon", "HicksMullenYucasZavislak", "HigmanThompson",
  "LongCyclesOfPrimeLength", "MatthewsLeigh",
  "MaybeInfinitelyPresentedGroup", "ModuliOfPowers",
  "OddNumberOfGens_FiniteOrder", "Semilocals",
  "SlowlyContractingMappings", "Syl3_S9", "SymmetrizingCollatzTree",
  "TameGroupByCommsOfWildPerms", "Venturini", "ZxZ" ]
gap> AssignGlobals(RCWAExamples.CollatzMapping);
The following global variables have been assigned:
[ "T", "T5", "T5m", "T5p", "Tm", "Tp" ]
```

6.2 Databases of rcwa groups

6.2.1 LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions (small database)

▷ `LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions()` (function)

Returns: the name of the variable to which the record containing the database of all groups generated by 3 class transpositions which interchange residue classes with moduli ≤ 6 got bound.

The database record has at least the following components (the index `i` is always an integer in the range `[1..52394]`, and the term "indices" always refers to list indices in that range):

`cts` The list of all 69 class transpositions which interchange residue classes with moduli ≤ 6.

`grps`
The list of the 52394 groups – 21948 finite and 30446 infinite ones.

`sizes`
The list of group orders – it is `Size(grps[i]) = sizes[i]`.

`mods`
The list of moduli of the groups – it is `Mod(grps[i]) = mods[i]`.

`equalityclasses`
A list of lists of indices `i` of groups which are known to be equal, i.e. if `i` and `j` lie in the same list, then `grps[i] = grps[j]`.

`samegroups`
A list of lists, where `samegroups[i]` is a list of indices of groups which are known to be equal to `grps[i]`.

`conjugacyclasses`
A list of lists of indices of groups which are known to be conjugate in RCWA($\mathbb{Z}$).

`subgroups`
A list of lists, where `subgroups[i]` is a list of indices of groups which are known to be proper subgroups of `grps[i]`.

`supergroups`
A list of lists, where `supergroups[i]` is a list of indices of groups which are known to be proper supergroups of `grps[i]`.

`chains`
A list of lists, where each list contains the indices of the groups in a descending chain of subgroups.

`respectedpartitions`
The list of shortest respected partitions. If `grps[i]` is finite, then `respectedpartitions[i]` is a list of pairs (residue, modulus) for the residue classes in the shortest respected partition `grps[i]`. If `grps[i]` is infinite, then `respectedpartitions[i] = fail`.

`partitionlengths`
The list of lengths of shortest respected partitions. If the group `grps[i]` is finite, then `partitionlengths[i]` is the length of the shortest respected partition of `grps[i]`. If `grps[i]` is infinite, then `partitionlengths[i] = 0`.

`degrees`
The list of permutation degrees, i.e. numbers of moved points, in the action of the finite groups on their shortest respected partitions. If there is no respected partition, i.e. if `grps[i]` is infinite, then `degrees[i] = 0`.

`orbitlengths`
The list of lists of orbit lengths in the action of the finite groups on their shortest respected partitions. If `grps[i]` is infinite, then `orbitlengths[i] = fail`.

`permgroupgens`
The list of lists of generators of the isomorphic permutation groups induced by the finite groups on their shortest respected partitions. If `grps[i]` is infinite, then `permgroupgens[i] = fail`.

`stabilize_digitsum_base2_mod2`
The list of indices of groups which stabilize the digit sum in base 2 modulo 2.

`stabilize_digitsum_base2_mod3`
The list of indices of groups which stabilize the digit sum in base 2 modulo 3.

`stabilize_digitsum_base3_mod2`
The list of indices of groups which stabilize the digit sum in base 3 modulo 2.

`freeproductcandidates`
A list of indices of groups which may be isomorphic to the free product of 3 copies of the cyclic group of order 2.

`freeproductlikes`
A list of indices of groups which are not isomorphic to the free product of 3 copies of the cyclic group of order 2, but where the shortest relation indicating this is relatively long.

`abc_torsion`
A list of pairs (index, order of product of generators) for all infinite groups for which the product of the generators has finite order.

`cyclist`
A list described in the comments in `rcwa/data/3ctsgroups6/spheresizecycles.g`.

`finiteorbits`
A record described in the comments in `rcwa/data/3ctsgroups6/finite-orbits.g`.

`intransitivemodulo`
For every modulus `m` from 1 to 60, `intransitivemodulo[m]` is the list of indices of groups none of whose orbits on $\mathbb{Z}$ has nontrivial intersection with all residue classes modulo `m`.

`trsstatus`
A list of strings which describe what is known about whether the groups `grps[i]` act transitively on the nonnegative integers in their support, or how the computation has failed.

`orbitgrowthtype`
A list of integers and lists of integers which encode what has been observed heuristically on the growth of the orbits of the groups `grps[i]` on $\mathbb{Z}$.

Note that the database contains an entry for every unordered 3-tuple of distinct class transpositions in `cts`, which means that it contains multiple copies of equal groups – cf. the components `equalityclasses` and `samegroups` described above.

To mention an example, the group `grps[44132]` might be called the “Collatz group” – its action on the set of positive integers which are not multiples of 6 is transitive if and only if the Collatz conjecture holds.

Example

```
gap> LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions();
"3CTsGroups6"
gap> AssignGlobals(3CTsGroups6); # for convenience
The following global variables have been assigned:
[ "3CTsGroupsWithGivenOrbit", "Id3CTsGroup",
  "ProbablyFixesDigitSumsModulo", "ProbablyStabilizesDigitSumsModulo",
  "TriangleTypes", "abc_torsion", "chains", "conjugacyclasses", "cts",
  "cyclist", "degrees", "epifromfpgroupto_ct23z",
  "epifromfpgrouptocollatzgroup_c", "epifromfpgrouptocollatzgroup_t",
  "equalityclasses", "finiteorbits", "freeproductcandidates",
  "freeproductlikes", "groups", "grps", "intransitivemodulo",
  "minwordlengthcoprimemultdiv", "minwordlengthnonbalanced", "mods",
  "orbitgrowthtype", "orbitlengths", "partitionlengths", "permgroupgens",
  "redundant_generator", "refinementseqlngs", "respectedpartitions",
  "samegroups", "shortresidueclassorbitlengths", "sizes", "sizespos",
  "sizesset", "spheresizebound_12", "spheresizebound_24",
  "spheresizebound_4", "spheresizebound_6",
  "stabilize_digitsum_base2_mod2", "stabilize_digitsum_base2_mod3",
  "stabilize_digitsum_base3_mod2", "subgroups", "supergroups",
  "trsstatus", "trsstatuspos", "trsstatusset" ]
gap> grps[44132]; # the "3n+1 group"
<(2(3),4(6)),(1(3),2(6)),(1(2),4(6))>
gap> trsstatus[44132]; # deciding this would solve the 3n+1 problem
"exceeded memory bound"
gap> Length(Set(sizes));
1066
gap> Maximum(Filtered(sizes,IsInt)); # order of largest finite group stored
71650335897938526975314569807067325484356096450918222967779764651168249559\
21354991746178379117549210141381841552049349610040738533234583155394615430\
44805152608184099138461614735360000000000000000000000000000000000000000000\
000000
gap> PrintFactorsInt(last);
2^200*3^103*5^48*7^28*11^16*13^13*17^8*19^6*23^6*29
gap> Positions(sizes,last);
[ 33814, 36548 ]
gap> grps{last};
[ <(1(5),4(5)),(0(3),1(6)),(3(4),0(6))>,
  <(0(5),3(5)),(2(3),4(6)),(0(4),5(6))> ]
gap> samegroups[1];
[ 1, 2, 68 ]
gap> grps[1] = grps[68];
true
gap> Maximum(mods);
77760
```

```
gap> Positions(mods,last);
[ 26311, 26313, 26452, 26453, 26455, 26456, 26457, 26459, 26461, 26462,
  27781, 27784, 27785, 27786, 27788, 27789, 27790, 27791, 27829, 27832,
  30523, 30524, 30525, 30526, 30529, 30530, 30532, 30534, 32924, 32927,
  32931, 32933 ]
gap> Set(sizes{last});
[ 45509262704640000 ]
gap> Collected(mods);
[ [ 0, 30446 ], [ 3, 1 ], [ 4, 37 ], [ 5, 120 ], [ 6, 1450 ], [ 8, 18 ],
  [ 10, 45 ], [ 12, 3143 ], [ 15, 165 ], [ 18, 484 ], [ 20, 528 ],
  [ 24, 1339 ], [ 30, 2751 ], [ 36, 2064 ], [ 40, 26 ], [ 48, 515 ],
  [ 60, 2322 ], [ 72, 2054 ], [ 80, 44 ], [ 90, 108 ], [ 96, 108 ],
  [ 108, 114 ], [ 120, 782 ], [ 144, 310 ], [ 160, 26 ], [ 180, 206 ],
  [ 192, 6 ], [ 216, 72 ], [ 240, 304 ], [ 270, 228 ], [ 288, 14 ],
  [ 360, 84 ], [ 432, 36 ], [ 480, 218 ], [ 540, 18 ], [ 720, 120 ],
  [ 810, 112 ], [ 864, 8 ], [ 960, 94 ], [ 1080, 488 ], [ 1620, 44 ],
  [ 1920, 38 ], [ 2160, 506 ], [ 3240, 34 ], [ 3840, 12 ],
  [ 4320, 218 ], [ 4860, 16 ], [ 6480, 282 ], [ 7680, 10 ],
  [ 8640, 16 ], [ 12960, 120 ], [ 14580, 2 ], [ 25920, 34 ],
  [ 30720, 2 ], [ 38880, 12 ], [ 51840, 8 ], [ 77760, 32 ] ]
gap> Collected(trsstatus);
[ [ "> 1 orbit (mod m)", 593 ],
  [ "Mod(U DecreasingOn) exceeded <maxmod>", 23 ],
  [ "U DecreasingOn stable and exceeded memory bound", 11 ],
  [ "U DecreasingOn stable for <maxeq> steps", 5753 ],
  [ "exceeded memory bound", 497 ], [ "finite", 21948 ],
  [ "intransitive, but finitely many orbits", 8 ],
  [ "seemingly only finite orbits (long)", 1227 ],
  [ "seemingly only finite orbits (medium)", 2501 ],
  [ "seemingly only finite orbits (short)", 4816 ],
  [ "seemingly only finite orbits (very long)", 230 ],
  [ "seemingly only finite orbits (very long, very unclear)", 76 ],
  [ "seemingly only finite orbits (very short)", 208 ],
  [ "there are infinite orbits which have exponential sphere size growth"
      , 2934 ],
  [ "there are infinite orbits which have linear sphere size growth",
    10881 ],
  [ "there are infinite orbits which have unclear sphere size growth",
    86 ], [ "transitive", 562 ],
  [ "transitive up to one finite orbit", 40 ] ]
```

6.2.2 LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions (both databases)

▷ `LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions(max_m)` (function)

Returns: the name of the variable to which the record containing the database of all groups generated by 3 class transpositions which interchange residue classes with moduli less than or equal to *max_m* got bound, where *max_m* is either 6 or 9.

If *max_m* is 6, this is equivalent to the call of the function without argument described above. If *max_m* is 9, the function returns a record with at least the following components (in the sequel, the indices `i > j > k` are always integers in the range `[1..264]`):

`cts` The list of all 264 class transpositions which interchange residue classes with moduli ≤ 9.

`mods`
The list of moduli of the groups, i.e. `Mod(Group(cts{[i,j,k]})) = mods[i][j][k]`.

`partlengths`
The list of lengths of shortest respected partitions of the groups in the database, i.e. `Length(RespectedPartition(Group(cts{[i,j,k]}))) = partlengths[i][j][k]`.

`sizes`
The list of orders of the groups, i.e. `Size(Group(cts{[i,j,k]})) = sizes[i][j][k]`.

`All3CTs9Indices`
A selector function which takes as argument a function *func* of three arguments *i*, *j* and *k*. It returns a list of all triples of indices `[`*i*`,`*j*`,`*k*`]` where $264 \geq i > j > k \geq 1$ for which *func* returns `true`.

`All3CTs9Groups`
A selector function which takes as argument a function *func* of three arguments *i*, *j* and *k*. It returns a list of all groups `Group(cts{[`*i*`,`*j*`,`*k*`]})` from the database for which *func*`(`*i*`,`*j*`,`*k*`)` returns `true`.

Example

```
gap> LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions(9);
"3CTsGroups9"
gap> AssignGlobals(3CTsGroups9);
The following global variables have been assigned:
[ "All3CTs9Groups", "All3CTs9Indices", "cts", "mods", "partlengths",
  "sizes" ]
gap> PrintFactorsInt(Maximum(Filtered(Flat(sizes),n->n<>infinity)));
2^1283*3^673*5^305*7^193*11^98*13^84*17^50*19^41*23^25*29^13*31^4
```

6.2.3 LoadDatabaseOfGroupsGeneratedBy4ClassTranspositions

▷ `LoadDatabaseOfGroupsGeneratedBy4ClassTranspositions()` (function)

Returns: the name of the variable to which the record containing the database of all groups generated by 4 class transpositions which interchange residue classes with moduli ≤ 6 for which all subgroups generated by 3 out of the 4 generators are finite got bound.

The record has at least the following components (the index `i` is always an integer in the range `[1..140947]`, and the term “indices” always refers to list indices in that range):

`cts` The list of all 69 class transpositions which interchange residue classes with moduli ≤ 6.

`grps4_3finite`
The list of all 140947 groups in the database.

`grps4_3finitepos`
The list obtained from `grps4_3finite` by replacing every group by the list of positions of its generators in the list `cts`.

`sizes4`
: The list of group orders – it is `Size(grps4_3finite[i]) = sizes4[i]`.

`mods4`
: The list of moduli of the groups – it is `Mod(grps4_3finite[i]) = mods4[i]`.

`conjugacyclasses4cts`
: A list of lists of indices of groups which are known to be conjugate in RCWA($\mathbb{Z}$).

`grps4_3finite_reps`
: Tentative conjugacy class representatives from the list `grps4_3finite` – *tentative* in the sense that likely some of the groups in the list are still conjugate.

Note that the database contains an entry for every suitable unordered 4-tuple of distinct class transpositions in `cts`, which means that it contains multiple copies of equal groups.

Example

```
gap> LoadDatabaseOfGroupsGeneratedBy4ClassTranspositions();
"4CTsGroups6"
gap> AssignGlobals(4CTsGroups6);
The following global variables have been assigned:
[ "conjugacyclasses4cts", "cts", "grps4_3finite", "grps4_3finite_reps",
  "grps4_3finitepos", "mods4", "sizes4", "sizes4pos", "sizes4set" ]
gap> Length(grps4_3finite);
140947
gap> Length(sizes4);
140947
gap> Size(grps4_3finite[1]);
518400
gap> sizes4[1];
518400
gap> Maximum(Filtered(sizes4,IsInt));
<integer 420...000 (3852 digits)>
gap> Modulus(grps4_3finite[1]);
12
gap> mods4[1];
12
gap> Length(Set(sizes4));
7339
gap> Length(Set(mods4));
91
gap> conjugacyclasses4cts{[1..4]};
[ [ 1, 23, 563, 867 ], [ 2, 859 ], [ 3, 622 ], [ 4, 16, 868, 873 ] ]
gap> grps4_3finite[1] = grps4_3finite[23];
true
gap> grps4_3finite[4] = grps4_3finite[16];
false
```

6.3 Databases of rcwa mappings

6.3.1 LoadDatabaseOfProductsOf2ClassTranspositions

▷ `LoadDatabaseOfProductsOf2ClassTranspositions()` (function)

Returns: the name of the variable to which the record containing the database of products of 2 class transpositions got bound.

There are 69 class transpositions which interchange residue classes with moduli ≤ 6, thus there is a total of $(69 \cdot 68)/2 = 2346$ unordered pairs of distinct such class transpositions. Looking at intersection- and subset relations between the 4 involved residue classes, we can distinguish 17 different "intersection types" (or 18, together with the trivial case of equal class transpositions). The intersection type does not fully determine the cycle structure of the product. – In total, we can distinguish 88 different cycle types of products of 2 class transpositions which interchange residue classes with moduli ≤ 6.

The components of the database record are a list `CTPairs` of all 2346 pairs of distinct class transpositions which interchange residue classes with moduli ≤ 6, functions `CTPairsIntersectionTypes`, `CTPairIntersectionType` and `CTPairProductType`, as well as data lists `OrdersMatrix`, `CTPairsProductClassification`, `CTPairsProductType`, `CTProds12` and `CTProds32`. – For the description of these components, see the file `pkg/rcwa/data/ctproducts/ctprodclass.g`.

Example

```
gap> LoadDatabaseOfProductsOf2ClassTranspositions();
"CTProducts"
gap> Set(RecNames(CTProducts));
[ "CTPairIntersectionType", "CTPairProductType", "CTPairs",
  "CTPairsIntersectionTypes", "CTPairsProductClassification",
  "CTPairsProductType", "CTProds12", "CTProds32", "OrdersMatrix" ]
gap> Length(CTProducts.CTPairs);
2346
gap> Collected(List(CTProducts.CTPairsProductType,l->l[2])); # order stats
[ [ 2, 165 ], [ 3, 255 ], [ 4, 173 ], [ 6, 693 ], [ 10, 2 ],
  [ 12, 345 ], [ 15, 4 ], [ 20, 10 ], [ 30, 120 ], [ 60, 44 ],
  [ infinity, 535 ] ]
```

6.3.2 LoadDatabaseOfNonbalancedProductsOfClassTranspositions

▷ `LoadDatabaseOfNonbalancedProductsOfClassTranspositions()` (function)

Returns: the name of the variable to which the record containing the database of non-balanced products of class transpositions got bound.

This database contains a list of the 24 pairs of class transpositions which interchange residue classes with moduli ≤ 6 and whose product is not balanced, as well as a list of all 36 essentially distinct triples of such class transpositions whose product has coprime multiplier and divisor.

Example

```
gap> LoadDatabaseOfNonbalancedProductsOfClassTranspositions();
"CTProductsNB"
gap> Set(RecNames(CTProductsNB));
[ "PairsOfCTsWhoseProductIsNotBalanced",
```

```
  "TriplesOfCTsWhoseProductHasCoprimeMultiplierAndDivisor" ]
gap> CTProductsNB.PairsOfCTsWhoseProductIsNotBalanced;
[ [ ( 1(2), 2(4) ), ( 2(4), 3(6) ) ], [ ( 1(2), 2(4) ), ( 2(4), 5(6) ) ],
  [ ( 1(2), 2(4) ), ( 2(4), 1(6) ) ], [ ( 1(2), 0(4) ), ( 0(4), 1(6) ) ],
  [ ( 1(2), 0(4) ), ( 0(4), 3(6) ) ], [ ( 1(2), 0(4) ), ( 0(4), 5(6) ) ],
  [ ( 0(2), 1(4) ), ( 1(4), 2(6) ) ], [ ( 0(2), 1(4) ), ( 1(4), 4(6) ) ],
  [ ( 0(2), 1(4) ), ( 1(4), 0(6) ) ], [ ( 0(2), 3(4) ), ( 3(4), 4(6) ) ],
  [ ( 0(2), 3(4) ), ( 3(4), 2(6) ) ], [ ( 0(2), 3(4) ), ( 3(4), 0(6) ) ],
  [ ( 1(2), 2(6) ), ( 3(4), 2(6) ) ], [ ( 1(2), 2(6) ), ( 1(4), 2(6) ) ],
  [ ( 1(2), 4(6) ), ( 3(4), 4(6) ) ], [ ( 1(2), 4(6) ), ( 1(4), 4(6) ) ],
  [ ( 1(2), 0(6) ), ( 1(4), 0(6) ) ], [ ( 1(2), 0(6) ), ( 3(4), 0(6) ) ],
  [ ( 0(2), 1(6) ), ( 2(4), 1(6) ) ], [ ( 0(2), 1(6) ), ( 0(4), 1(6) ) ],
  [ ( 0(2), 3(6) ), ( 2(4), 3(6) ) ], [ ( 0(2), 3(6) ), ( 0(4), 3(6) ) ],
  [ ( 0(2), 5(6) ), ( 2(4), 5(6) ) ], [ ( 0(2), 5(6) ), ( 0(4), 5(6) ) ]
 ]
```

Chapter 7

Examples

This chapter discusses a number of examples of rcwa mappings and -groups in detail. All of them show different aspects of the package, and the order in which they appear is entirely arbitrary. In particular they are not ordered by degree of difficulty or interest.

The rcwa mappings, rcwa groups and other objects defined in this chapter can be found in the file `pkg/rcwa/examples/examples.g`. This file can be read into the current GAP session by the function `LoadRCWAExamples` (6.1.1) which takes no arguments and returns the name of a variable which the record containing the examples got assigned to. The global variable assignments made in a section of this chapter can be made by applying the function `AssignGlobals` to the respective component of the examples record. The component names are given at the end of the corresponding sections.

The discussions of the examples are typically far from being exhaustive. It is quite likely that in many instances by just a few little modifications or additional easy commands you can find out interesting things yourself – have fun!

7.1 Thompson's group V

Thompson's group V, also known as Higman-Thompson group, is a finitely presented infinite simple group. This group has been found by Graham Higman, cf. [Hig74]. We show that the group

Example
```
gap> G := Group(List([[0,2,1,4],[0,4,1,4],[1,4,2,4],[2,4,3,4]],
>                    ClassTransposition));
<(0(2),1(4)),(0(4),1(4)),(1(4),2(4)),(2(4),3(4))>
```

is isomorphic to Thompson's group V. This isomorphism has been pointed out by John P. McDermott. We take a slightly different set of generators:

Example
```
gap> k := ClassTransposition(0,2,1,2);;
gap> l := ClassTransposition(1,2,2,4);;
gap> m := ClassTransposition(0,2,1,4);;
gap> n := ClassTransposition(1,4,2,4);;
gap> H := Group(k,l,m,n);
<(0(2),1(2)),(1(2),2(4)),(0(2),1(4)),(1(4),2(4))>
```

```
gap> G = H; # k, l, m and n generate G as well
true
```

Now we verify that our four generators satisfy the relations given on page 50 in [Hig74], when we read k as κ, l as λ, m as μ and n as ν:

Example

```
gap> HigmanThompsonRels :=
> [ k^2, l^2, m^2, n^2,                              # (1) in Higman's book
>   l*k*m*k*l*n*k*n*m*k*l*k*m,                       # (2)        "
>   k*n*l*k*m*n*k*l*n*m*n*l*n*m,                     # (3)        "
>   (l*k*m*k*l*n)^3, (m*k*l*k*m*n)^3,                # (4)        "
>   (l*n*m)^2*k*(m*n*l)^2*k,                         # (5)        "
>   (l*n*m*n)^5,                                     # (6)        "
>   (l*k*n*k*l*n)^3*k*n*k*(m*k*n*k*m*n)^3*k*n*k*n,# (7)        "
>   ((l*k*m*n)^2*(m*k*l*n)^2)^3,                     # (8)        "
>   (l*n*l*k*m*k*m*n*l*n*m*k*m*k)^4,                 # (9)        "
>   (m*n*m*k*l*k*l*n*m*n*l*k*l*k)^4,                 #(10)        "
>   (l*m*k*l*k*m*l*k*n*k)^2,                         #(11)        "
>   (m*l*k*m*k*l*m*k*n*k)^2 ];                       #(12)        "
[ IdentityMapping( Integers ), IdentityMapping( Integers ),
  IdentityMapping( Integers ), IdentityMapping( Integers ),
  IdentityMapping( Integers ), IdentityMapping( Integers ),
  IdentityMapping( Integers ), IdentityMapping( Integers ),
  IdentityMapping( Integers ), IdentityMapping( Integers ),
  IdentityMapping( Integers ), IdentityMapping( Integers ),
  IdentityMapping( Integers ), IdentityMapping( Integers ),
  IdentityMapping( Integers ), IdentityMapping( Integers ) ]
```

We conclude that our group is an homomorphic image of Thompson's group V. But since Thompson's group V is simple and our group is not trivial, this means indeed that the two groups are isomorphic.

In fact it is straightforward to show that G is the group CT([2],Integers) which is generated by the set of all class transpositions which interchange residue classes modulo powers of 2. First we check that G contains all 11 class transpositions which interchange residue classes modulo 2 or 4:

Example

```
gap> S := Filtered(List(ClassPairs(4),ClassTransposition),
>                  ct->Mod(ct) in [2,4]);
[ ( 0(2), 1(2) ), ( 0(2), 1(4) ), ( 0(2), 3(4) ), ( 0(4), 1(4) ),
  ( 0(4), 2(4) ), ( 0(4), 3(4) ), ( 1(2), 0(4) ), ( 1(2), 2(4) ),
  ( 1(4), 2(4) ), ( 1(4), 3(4) ), ( 2(4), 3(4) ) ]
gap> IsSubset(G,S);
true
```

Then we give a function which takes a class transposition $\tau \in \mathrm{CT}_{\emptyset}(\mathbb{Z})$, and which returns a factorization of an element γ satisfying $\tau^\gamma \in S$ into $g_1 := \tau_{0(2),1(4)} \in S$, $g_2 := \tau_{0(2),3(4)} \in S$, $g_3 := \tau_{1(2),0(4)} \in S$, $g_4 := \tau_{1(2),2(4)} \in S$, $h_1 := \tau_{0(4),1(4)} \in S$ and $h_2 := \tau_{1(4),2(4)} \in S$:

GAP code

```
ReducingConjugator := function ( tau )

  local  w, F, g1, g2, g3, g4, h1, h2, h, cls, cl, r;

  g1 := ClassTransposition(0,2,1,4); h1 := ClassTransposition(0,4,1,4);
  g2 := ClassTransposition(0,2,3,4); h2 := ClassTransposition(1,4,2,4);
  g3 := ClassTransposition(1,2,0,4);
  g4 := ClassTransposition(1,2,2,4);

  F := FreeGroup("g1","g2","g3","g4","h1","h2");

  w := One(F); if Mod(tau) <= 4 then return w; fi;

  # Before we can reduce the moduli of the interchanged residue classes,
  # we must make sure that both of them have at least modulus 4.
  cls := TransposedClasses(tau);
  if Mod(cls[1]) = 2 then
    if Residue(cls[1]) = 0 then
      if Residue(cls[2]) mod 4 = 1 then tau := tau^g2; w := w * F.2;
                                   else tau := tau^g1; w := w * F.1; fi;
    else
      if Residue(cls[2]) mod 4 = 0 then tau := tau^g4; w := w * F.4;
                                   else tau := tau^g3; w := w * F.3; fi;
    fi;
  fi;

  while Mod(tau) > 4 do # Now we can successively reduce the moduli.
    if not ForAny(AllResidueClassesModulo(2),
                  cl -> IsEmpty(Intersection(cl,Support(tau))))
    then
      cls := TransposedClasses(tau);
      h := Filtered([h1,h2],
             hi->Length(Filtered(cls,cl->IsSubset(Support(hi),cl)))=1);
      h := h[1]; tau := tau^h;
      if h = h1 then w := w * F.5; else w := w * F.6; fi;
    fi;
    cl := TransposedClasses(tau)[2]; # class with larger modulus
    r  := Residue(cl);
    if   r mod 4 = 1 then tau := tau^g1; w := w * F.1;
    elif r mod 4 = 3 then tau := tau^g2; w := w * F.2;
    elif r mod 4 = 0 then tau := tau^g3; w := w * F.3;
    elif r mod 4 = 2 then tau := tau^g4; w := w * F.4; fi;
  od;

  return w;
end;
```

After assigning g1, g2, g3, g4, h1 and h2 appropriately, we obtain for example:

Example

```
gap> ReducingConjugator(ClassTransposition(3,16,34,256));
h2*g1*h1*g1*h1*g1*h1*g1*h2*g2*h2*g4*h2*g4*h2*g3
gap> gamma := h2*g1*h1*g1*h1*g1*h1*g1*h2*g2*h2*g4*h2*g4*h2*g3;
<rcwa permutation of Z with modulus 256>
gap> ct := ClassTransposition(3,16,34,256)^gamma;;
gap> IsClassTransposition(ct);;
gap> ct;
ClassTransposition(1,4,2,4)
```

Thompson's group V can also be embedded in a natural way into CT(GF(2)[x]):

Example

```
gap> x := Indeterminate(GF(2));; SetName(x,"x");
gap> R := PolynomialRing(GF(2),1);;
gap> k := ClassTransposition(0,x,1,x);;
gap> l := ClassTransposition(1,x,x,x^2);;
gap> m := ClassTransposition(0,x,1,x^2);;
gap> n := ClassTransposition(1,x^2,x,x^2);;
gap> G := Group(k,l,m,n);
<rcwa group over GF(2)[x] with 4 generators>
```

The correctness of this representation can likewise be verified by simply checking the defining relations given above.

Enter `AssignGlobals(LoadRCWAExamples().HigmanThompson);` in order to assign the global variables defined in this section.

7.2 Factoring Collatz' permutation of the integers

In 1932, Lothar Collatz mentioned in his notebook the following permutation of the integers:

Example

```
gap> Collatz := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);;
gap> Display(Collatz);

Rcwa mapping of Z with modulus 3

        /
        | 2n/3     if n in 0(3)
 n |-> <  (4n-1)/3 if n in 1(3)
        | (4n+1)/3 if n in 2(3)
        \

gap> ShortCycles(Collatz,[-50..50],50); # There are some finite cycles:
[ [ 0 ], [ -1 ], [ 1 ], [ 2, 3 ], [ -2, -3 ], [ 4, 5, 7, 9, 6 ],
  [ -4, -5, -7, -9, -6 ],
  [ 44, 59, 79, 105, 70, 93, 62, 83, 111, 74, 99, 66 ],
```

```
  [ -44, -59, -79, -105, -70, -93, -62, -83, -111, -74, -99, -66 ] ]
```

The cycle structure of Collatz' permutation has not been completely determined yet. In particular it is not known whether the cycle containing 8 is finite or infinite. Nevertheless, the factorization routine included in this package can determine a factorization of this permutation into class transpositions, i.e. involutions interchanging two disjoint residue classes:

Example

```
gap> Collatz in CT(Integers);  # 'Collatz' lies in the simple group CT(Z).
true
gap> Length(Factorization(Collatz));
212
```

Setting the Info level of `InfoRCWA` equal to 2 (simply issue `RCWAInfo(2);`) causes the factorization routine to display detailed information on the progress of the factoring process. For reasons of saving space, this is not done in this manual.

We would like to get a factorization into fewer factors. Firstly, we try to factor the inverse – just like the various options interpreted by the factorization routine, this has influence on decisions taken during the factoring process:

Example

```
gap> Length(Factorization(Collatz^-1));
129
```

This is already a shorter product, but can still be improved. We remember the `mKnot`'s, of which the permutation `mKnot(3)` looks very similar to Collatz' permutation. Therefore it is straightforward to try to factor both `mKnot(3)` and `Collatz/mKnot(3)`, and to look whether the sum of the numbers of factors is less than 129:

Example

```
gap> KnotFacts := Factorization(mKnot(3));;
gap> QuotFacts := Factorization(Collatz/mKnot(3));;
gap> List([KnotFacts,QuotFacts],Length);
[ 59, 9 ]
gap> CollatzFacts := Concatenation(QuotFacts,KnotFacts);
[ ( 0(6), 4(6) ), ( 0(6), 5(6) ), ( 0(6), 3(6) ), ( 0(6), 1(6) ),
  ( 0(6), 2(6) ), ( 2(3), 4(6) ), ( 0(3), 4(6) ), ( 2(3), 1(6) ),
  ( 0(3), 1(6) ), ( 0(36), 35(36) ), ( 0(36), 22(36) ),
  ( 0(36), 18(36) ), ( 0(36), 17(36) ), ( 0(36), 14(36) ),
  ( 0(36), 20(36) ), ( 0(36), 4(36) ), ( 2(36), 8(36) ),
  ( 2(36), 16(36) ), ( 2(36), 13(36) ), ( 2(36), 9(36) ),
  ( 2(36), 7(36) ), ( 2(36), 6(36) ), ( 2(36), 3(36) ),
  ( 2(36), 10(36) ), ( 2(36), 15(36) ), ( 2(36), 12(36) ),
  ( 2(36), 5(36) ), ( 21(36), 28(36) ), ( 21(36), 33(36) ),
  ( 21(36), 30(36) ), ( 21(36), 23(36) ), ( 21(36), 34(36) ),
  ( 21(36), 31(36) ), ( 21(36), 27(36) ), ( 21(36), 25(36) ),
  ( 21(36), 24(36) ), ( 26(36), 32(36) ), ( 26(36), 29(36) ),
  ( 10(18), 35(36) ), ( 5(18), 35(36) ), ( 10(18), 17(36) ),
```

```
  ( 5(18), 17(36) ), ( 8(12), 14(24) ), ( 6(9), 17(18) ),
  ( 3(9), 17(18) ), ( 0(9), 17(18) ), ( 6(9), 16(18) ), ( 3(9), 16(18) ),
  ( 0(9), 16(18) ), ( 6(9), 11(18) ), ( 3(9), 11(18) ), ( 0(9), 11(18) ),
  ( 6(9), 4(18) ), ( 3(9), 4(18) ), ( 0(9), 4(18) ), ( 0(6), 14(24) ),
  ( 0(6), 2(24) ), ( 8(12), 17(18) ), ( 7(12), 17(18) ),
  ( 8(12), 11(18) ), ( 7(12), 11(18) ), PrimeSwitch(3)^-1,
  ( 7(12), 17(18) ), ( 2(6), 17(18) ), ( 0(3), 17(18) ),
  PrimeSwitch(3)^-1, PrimeSwitch(3)^-1, PrimeSwitch(3)^-1 ]
gap> Product(CollatzFacts) = Collatz; # Check.
true
```

The factors `PrimeSwitch(3)` are products of 6 class transpositions (cf. `PrimeSwitch` (2.5.2)).

Enter `AssignGlobals(LoadRCWAExamples().CollatzlikePerms);` in order to assign the global variables defined in this section.

7.3 The $3n+1$ group

The following group acts transitively on the set of positive integers for which the $3n+1$ conjecture holds and which are not divisible by 6:

Example

```
gap> a := ClassTransposition(1,2,4,6);;
gap> b := ClassTransposition(1,3,2,6);;
gap> c := ClassTransposition(2,3,4,6);;
gap> G := Group(a,b,c);
<(1(2),4(6)),(1(3),2(6)),(2(3),4(6))>
gap> LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions();
"3CTsGroups6"
gap> 3CTsGroups6.Id3CTsGroup(G,3CTsGroups6.grps); # 'catalogue number' of G
44132
```

To see this, consider the action of G on the "$3n+1$ tree". The vertices of this tree are the positive integers for which the $3n+1$ conjecture holds, and for every vertex n there is an edge from n to $T(n)$, where T denotes the Collatz mapping

$$T: \mathbb{Z} \longrightarrow \mathbb{Z}, \quad n \longmapsto \begin{cases} \frac{n}{2} & \text{if } n \text{ is even,} \\ \frac{3n+1}{2} & \text{if } n \text{ is odd} \end{cases}$$

(cf. Chapter 1). It is easy to check that for every vertex n, either a, b or c maps n to $T(n)$, and that the other two generators either fix n or map it to one of its preimages under T. So the $3n+1$ conjecture is equivalent to the assertion that the group G acts transitively on $\mathbb{N} \setminus 0(6)$. First let's have a look at balls of small radius about 1 under the action of G – these consist of those numbers whose trajectory under T reaches 1 quickly:

Example

```
gap> Ball(G,1,5,OnPoints);
[ 1, 2, 4, 5, 8, 10, 16, 32, 64 ]
gap> Ball(G,1,10,OnPoints);
```

```
[ 1, 2, 3, 4, 5, 8, 10, 13, 16, 20, 21, 26, 32, 40, 52, 53, 64, 80, 85,
  128, 160, 170, 256, 320, 340, 341, 512, 1024, 2048 ]
gap> Ball(G,1,15,OnPoints);
[ 1, 2, 3, 4, 5, 7, 8, 10, 11, 13, 16, 17, 20, 21, 22, 23, 26, 32, 34,
  35, 40, 44, 45, 46, 52, 53, 64, 68, 69, 70, 75, 80, 85, 104, 106, 113,
  128, 136, 140, 141, 151, 160, 170, 208, 212, 213, 226, 227, 256, 272,
  277, 280, 301, 302, 320, 340, 341, 416, 424, 452, 453, 454, 512, 640,
  680, 682, 832, 848, 853, 904, 908, 909, 1024, 1280, 1360, 1364, 1365,
  1664, 1696, 1706, 1808, 1813, 1816, 2048, 2560, 2720, 2728, 4096,
  5120, 5440, 5456, 5461, 8192, 10240, 10880, 10912, 10922, 16384,
  32768, 65536 ]
gap> Ball(G,1,15,OnPoints:Spheres);
[ [ 1 ], [ 2, 4 ], [ 8 ], [ 16 ], [ 5, 32 ], [ 10, 64 ],
  [ 3, 20, 21, 128 ], [ 40, 256 ], [ 13, 80, 85, 512 ],
  [ 26, 160, 170, 1024 ], [ 52, 53, 320, 340, 341, 2048 ],
  [ 17, 104, 106, 113, 640, 680, 682, 4096 ],
  [ 34, 35, 208, 212, 213, 226, 227, 1280, 1360, 1364, 1365, 8192 ],
  [ 11, 68, 69, 70, 75, 416, 424, 452, 453, 454, 2560, 2720, 2728, 16384
     ],
  [ 22, 23, 136, 140, 141, 151, 832, 848, 853, 904, 908, 909, 5120,
      5440, 5456, 5461, 32768 ],
  [ 7, 44, 45, 46, 272, 277, 280, 301, 302, 1664, 1696, 1706, 1808,
      1813, 1816, 10240, 10880, 10912, 10922, 65536 ] ]
gap> List(Ball(G,1,50,OnPoints:Spheres),Length);
[ 1, 2, 1, 1, 2, 2, 4, 2, 4, 4, 6, 8, 12, 14, 17, 20, 26, 32, 43, 52,
  66, 81, 104, 133, 170, 211, 271, 335, 424, 542, 686, 873, 1096, 1376,
  1730, 2205, 2794, 3522, 4429, 5611, 7100, 8978, 11343, 14296, 18058,
  22828, 28924, 36532, 46146, 58399, 73713 ]
gap> FloatQuotientsList(last);
[ 2., 0.5, 1., 2., 1., 2., 0.5, 2., 1., 1.5, 1.33333, 1.5, 1.16667,
  1.21429, 1.17647, 1.3, 1.23077, 1.34375, 1.2093, 1.26923, 1.22727,
  1.28395, 1.27885, 1.2782, 1.24118, 1.28436, 1.23616, 1.26567, 1.2783,
  1.26568, 1.27259, 1.25544, 1.25547, 1.25727, 1.27457, 1.26712,
  1.26056, 1.25752, 1.26688, 1.26537, 1.26451, 1.26342, 1.26034,
  1.26315, 1.26415, 1.26704, 1.26303, 1.26317, 1.26553, 1.26223 ]
gap> Difference(Filtered([1..100],n->n mod 6 <> 0),Ball(G,1,40,OnPoints));
[ 27, 31, 41, 47, 55, 62, 63, 71, 73, 82, 83, 91, 94, 95, 97 ]
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);;
gap> List(last2,n->Length(Trajectory(T,n,[1])));
[ 71, 68, 70, 67, 72, 69, 69, 66, 74, 71, 71, 60, 68, 68, 76 ]
```

It is convenient to define an epimorphism from the free group of rank 3 to G:

Example

```
gap> F := FreeGroup("a","b","c");
<free group on the generators [ a, b, c ]>
gap> phi := EpimorphismByGenerators(F,G);
[ a, b, c ] -> [ ( 1(2), 4(6) ), ( 1(3), 2(6) ), ( 2(3), 4(6) ) ]
```

We can compute balls about 1 in G:

Example

```
gap> B := Ball(G,One(G),7:Spheres);;
gap> List(B,Length);
[ 1, 3, 6, 12, 24, 48, 96, 192 ]
gap> List(B[3],Order);
[ 12, infinity, infinity, infinity, infinity, 12 ]
gap> List(B[3],g->PreImagesRepresentative(phi,g));
[ b*a, c*b, c*a, b*c, a*c, a*b ]
gap> g := a*b;; Order(g);;
gap> Display(g);

Rcwa permutation of Z with modulus 18, of order 12

( 1(6), 8(36), 4(18), 2(12) ) ( 3(6), 20(36), 10(18) )
( 5(6), 32(36), 16(18) )
```

Spending some more time to compute `B := Ball(G,One(G),12:Spheres);;`, one can check that $(ab)^{12}$ is the shortest word in the generators of G which does not represent the identity in the free product of 3 cyclic groups of order 2, but which represents the identity in G. However, the group G has elements of other finite orders as well – for example:

Example

```
gap> g := (b*a)^3*b*c;; Order(g);;
gap> Display(g);

Rcwa permutation of Z with modulus 36, of order 105

( 8(9), 16(18), 64(72), 256(288), 85(96), 128(144), 32(36) )
( 7(12), 11(18), 22(36) ) ( 5(18), 10(36), 40(144), 13(48),
  20(72) ) ( 1(24), 2(36), 4(72) ) ( 14(36), 28(72), 112(288),
  37(96), 56(144) )

gap> Order(a*c*b*a*b*c*a*c);
60
```

With some more efforts, one finds that e.g. $(abc)^2c^b$ has order 616, that $(abc)^2b$ has order 2310, that $(ab)^2a^ca^bc$ has order 27720, and that $a(c(ab)^2)^2$ has order 65520. Of course G has many elements of infinite order as well. Some of them have infinite cycles, like e.g.

Example

```
gap> g := b*c;;
gap> Display(g);

Rcwa permutation of Z with modulus 12

        /
        | 4n  if n in 1(3)
        | 2n  if n in 5(6)
```

```
 n |-> <  n/2 if n in 2(12)
       | n/4 if n in 8(12)
       | n   if n in 0(3)
       \

gap> Sinks(g);
[ 4(12) ]
gap> Trajectory(g,last[1],10);
[ 4(12), 16(48), 64(192), 256(768), 1024(3072), 4096(12288),
  16384(49152), 65536(196608), 262144(786432), 1048576(3145728) ]
gap> Trajectory(g,4,20);
[ 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304,
  16777216, 67108864, 268435456, 1073741824, 4294967296, 17179869184,
  68719476736, 274877906944, 1099511627776 ]
```

Others seem to have only finite cycles. Some of these appear to have "on average" comparatively "short" cycles, like e.g.

Example

```
gap> g := a*b*a*c*b*c;
<rcwa permutation of Z with modulus 144>
gap> cycs := ShortCycles(g,[0..10000],100,10^20);;
gap> Difference([0..10000],Union(cycs));
[  ]
gap> Collected(List(cycs,Length));
[ [ 1, 2222 ], [ 3, 1945 ], [ 4, 1111 ], [ 5, 93 ], [ 6, 926 ],
  [ 7, 31 ], [ 8, 864 ], [ 9, 10 ], [ 10, 289 ], [ 11, 4 ], [ 12, 95 ],
  [ 13, 1 ], [ 14, 31 ], [ 16, 12 ], [ 18, 4 ], [ 20, 1 ] ]
```

If the cycle of g containing some $n \in \mathbb{Z}$ is finite and has a certain length l, then there is some $m \in \mathbb{Z}$ such that for every $k \in \mathbb{Z}$ the cycle of g containing $n+km$ has length l as well. Thus, in other words, every finite cycle of g "belongs to" a cycle of residue classes. (This is a special property of g which is not shared by every rcwa permutation – cf. e.g. Collatz' permutation from Section 7.2.) We can find some of these infinitely many "residue class cycles":

Example

```
gap> cycsrc := ShortResidueClassCycles(g,Mod(g),20);
[ [ 0(6) ], [ 3(6), 160(288), 20(36) ],
  [ 7(18), 352(864), 44(108), 28(72) ],
  [ 11(18), 544(864), 2896(4608), 362(576), 68(108), 88(144) ],
  [ 13(18), 640(864), 80(108), 52(72) ], [ 10(36) ], [ 34(36) ],
  [ 1(54), 64(2592), 8(324), 4(216), 16(1152), 2(144) ],
  [ 5(54), 256(2592), 1360(13824), 170(1728), 32(324), 40(432),
      208(2304), 26(288) ],
  [ 17(54), 832(2592), 4432(13824), 23632(73728), 2954(9216), 554(1728),
      104(324), 136(432) ],
  [ 37(54), 1792(2592), 224(324), 148(216), 784(1152), 98(144) ],
  [ 41(54), 1984(2592), 10576(13824), 1322(1728), 248(324), 328(432),
      1744(2304), 218(288) ],
  [ 53(54), 2560(2592), 13648(13824), 72784(73728), 9098(9216),
```

```
    1706(1728), 320(324), 424(432) ], [ 38(72), 58(108), 304(576) ],
  [ 62(72), 94(108), 496(576) ] ]
gap> List(cycsrc,Length);
[ 1, 3, 4, 6, 4, 1, 1, 6, 8, 8, 6, 8, 8, 3, 3 ]
gap> Sum(List(Flat(cycsrc),cl->1/Mod(cl)));
97459/110592
gap> Float(last); # about 88% 'coverage'
0.881248
gap> cycsrc := ShortResidueClassCycles(g,3*Mod(g),20);
[ [ 0(6) ], [ 3(6), 160(288), 20(36) ],
  [ 7(18), 352(864), 44(108), 28(72) ],
  [ 11(18), 544(864), 2896(4608), 362(576), 68(108), 88(144) ],
  [ 13(18), 640(864), 80(108), 52(72) ], [ 10(36) ], [ 34(36) ],
  [ 1(54), 64(2592), 8(324), 4(216), 16(1152), 2(144) ],
  [ 5(54), 256(2592), 1360(13824), 170(1728), 32(324), 40(432),
      208(2304), 26(288) ],
  [ 17(54), 832(2592), 4432(13824), 23632(73728), 2954(9216), 554(1728),
      104(324), 136(432) ],
  [ 37(54), 1792(2592), 224(324), 148(216), 784(1152), 98(144) ],
  [ 41(54), 1984(2592), 10576(13824), 1322(1728), 248(324), 328(432),
      1744(2304), 218(288) ],
  [ 53(54), 2560(2592), 13648(13824), 72784(73728), 9098(9216),
      1706(1728), 320(324), 424(432) ], [ 38(72), 58(108), 304(576) ],
  [ 62(72), 94(108), 496(576) ],
  [ 23(162), 1120(7776), 5968(41472), 746(5184), 140(972), 184(1296),
      976(6912), 5200(36864), 650(4608), 122(864) ],
  [ 35(162), 1696(7776), 9040(41472), 48208(221184), 257104(1179648),
      32138(147456), 6026(27648), 1130(5184), 212(972), 280(1296) ],
  [ 73(162), 3520(7776), 440(972), 292(648), 1552(3456), 8272(18432),
      1034(2304), 194(432) ],
  [ 77(162), 3712(7776), 19792(41472), 2474(5184), 464(972), 616(1296),
      3280(6912), 17488(36864), 2186(4608), 410(864) ],
  [ 89(162), 4288(7776), 22864(41472), 121936(221184), 650320(1179648),
      81290(147456), 15242(27648), 2858(5184), 536(972), 712(1296) ],
  [ 127(162), 6112(7776), 764(972), 508(648), 2704(3456), 14416(18432),
      1802(2304), 338(432) ],
  [ 14(216), 22(324), 112(1728), 592(9216), 74(1152) ],
  [ 86(216), 130(324), 688(1728), 3664(9216), 458(1152) ] ]
gap> List(cycsrc,Length);
[ 1, 3, 4, 6, 4, 1, 1, 6, 8, 8, 6, 8, 8, 3, 3, 10, 10, 8, 10, 10, 8, 5,
  5 ]
gap> Sum(List(Flat(cycsrc),Density));
5097073/5308416
gap> Float(last); # already about 96% 'coverage'
0.960187
```

There are also some elements of infinite order whose cycles seem to be all finite, but "on average" pretty "long" – e.g.

Example

```
gap> g := (b*a*c)^2*a;;
gap> Display(g);
```

```
Rcwa permutation of Z with modulus 288

        /
        | (16n-1)/3   if n in 1(3)
        | (9n+5)/4    if n in 3(24) U 11(24)
        | (27n+19)/4  if n in 15(24) U 23(24)
        | (3n+1)/4    if n in 5(24)
        | (n-3)/6     if n in 21(24)
        | (27n+29)/8  if n in 9(48) U 41(48)
        | (9n+7)/8    if n in 17(48) U 33(48)
        | (2n-7)/9    if n in 8(36)
 n |-> <  (4n-11)/9   if n in 32(36)
        | (27n+38)/8  if n in 14(48)
        | (3n+2)/8    if n in 26(48)
        | (9n+10)/8   if n in 38(48)
        | (3n+4)/4    if n in 20(72)
        | n/4         if n in 56(72)
        | (9n+14)/16  if n in 2(96)
        | (27n+58)/16 if n in 50(96)
        | n           if n in 0(6)
        \

gap> List([1..100],n->Length(Cycle(g,n)));
[ 6, 1, 6, 6, 6, 1, 194, 6, 216, 26, 26, 1, 26, 194, 65, 26, 26, 1, 216,
  26, 6, 216, 46, 1, 640, 26, 70, 194, 216, 1, 70, 26, 216, 216, 26, 1,
  194, 216, 73, 26, 110, 1, 194, 216, 194, 111, 39, 1, 194, 640, 640,
  194, 26, 1, 171, 194, 204, 640, 216, 1, 111, 70, 91, 26, 194, 1, 216,
  216, 26, 111, 65, 1, 50, 194, 26, 216, 640, 1, 502, 26, 111, 40, 110,
  1, 26, 194, 385, 640, 88, 1, 100, 111, 65, 110, 416, 1, 171, 194, 194,
  640 ]
gap> Length(Cycle(g,25));
640
gap> Maximum(Cycle(g,25));
323270249684063829
gap> Length(Cycle(g,25855));
4751
gap> Maximum(Cycle(g,25855));
507359605810239426786254778159924369135184044618585904603866210104085
gap> cycs := ShortCycles(g,[0..50000],10000,10^100);;
gap> S := [0..50000];;
gap> for cyc in cycs do S := Difference(S,cyc); od;
gap> S; # no cycle containing some n in [0..50000] has length > 10000
[  ]
```

Taking a look at the lengths of the trajectories of the Collatz mapping T starting at the points in a cycle, we can see how a cycle of g goes "up and down" in the $3n+1$ tree:

Example

```
gap> List(Cycle(g,25),n->Length(Trajectory(T,n,[1])));
[ 17, 21, 25, 29, 33, 31, 35, 34, 32, 33, 37, 41, 45, 44, 42, 39, 43,
  41, 45, 44, 42, 43, 40, 38, 35, 39, 37, 41, 40, 44, 48, 46, 50, 49,
```

```
  47, 48, 45, 42, 46, 44, 48, 47, 45, 46, 50, 49, 47, 43, 41, 38, 39,
  36, 34, 30, 27, 31, 29, 33, 32, 30, 31, 35, 33, 37, 36, 40, 39, 43,
  41, 45, 44, 42, 43, 47, 51, 55, 53, 57, 56, 54, 55, 59, 58, 62, 66,
  64, 68, 67, 65, 66, 63, 60, 64, 62, 66, 65, 63, 64, 68, 67, 65, 61,
  59, 56, 52, 49, 53, 51, 55, 54, 52, 53, 57, 55, 59, 58, 56, 57, 54,
  50, 48, 45, 49, 47, 51, 50, 54, 52, 56, 55, 53, 54, 58, 62, 66, 70,
  74, 72, 76, 75, 79, 83, 87, 91, 90, 94, 93, 97, 95, 99, 98, 96, 97,
  94, 91, 88, 85, 89, 87, 91, 90, 88, 89, 86, 84, 81, 85, 83, 87, 86,
  90, 94, 98, 97, 101, 105, 109, 107, 111, 110, 108, 109, 113, 117, 115,
  119, 118, 122, 126, 125, 123, 120, 124, 122, 126, 125, 123, 124, 121,
  119, 116, 117, 114, 111, 115, 113, 117, 116, 114, 115, 119, 123, 122,
  120, 117, 121, 119, 123, 122, 120, 121, 118, 116, 112, 110, 106, 103,
  107, 105, 109, 108, 106, 107, 111, 109, 113, 112, 116, 114, 118, 117,
  115, 116, 113, 110, 111, 108, 104, 102, 99, 103, 101, 105, 104, 108,
  106, 110, 109, 107, 108, 112, 111, 109, 105, 102, 103, 100, 98, 95,
  92, 96, 94, 98, 97, 95, 96, 93, 91, 88, 92, 90, 94, 93, 97, 101, 105,
  109, 108, 106, 103, 107, 105, 109, 108, 106, 107, 104, 102, 99, 103,
  101, 105, 104, 108, 112, 110, 114, 113, 111, 112, 116, 115, 113, 109,
  106, 110, 108, 112, 111, 109, 110, 114, 112, 116, 115, 113, 114, 111,
  107, 105, 102, 103, 100, 98, 95, 99, 97, 101, 100, 104, 103, 107, 105,
  109, 108, 106, 107, 104, 101, 98, 99, 96, 94, 91, 92, 89, 87, 84, 85,
  82, 80, 77, 81, 79, 83, 82, 86, 85, 89, 88, 86, 83, 80, 81, 78, 76,
  73, 74, 71, 68, 72, 70, 74, 73, 71, 72, 76, 80, 79, 83, 87, 91, 90,
  88, 85, 89, 87, 91, 90, 88, 89, 86, 84, 81, 85, 83, 87, 86, 90, 94,
  92, 96, 95, 93, 94, 98, 96, 100, 99, 97, 98, 102, 106, 110, 114, 113,
  111, 108, 112, 110, 114, 113, 111, 112, 109, 107, 104, 108, 106, 110,
  109, 113, 117, 115, 119, 118, 116, 117, 114, 111, 115, 113, 117, 116,
  114, 115, 119, 118, 116, 112, 110, 107, 108, 105, 103, 100, 104, 102,
  106, 105, 109, 108, 112, 110, 114, 113, 111, 112, 116, 115, 113, 109,
  106, 103, 104, 101, 99, 95, 91, 88, 92, 90, 94, 93, 91, 92, 96, 94,
  98, 97, 95, 96, 100, 98, 102, 101, 105, 104, 102, 99, 100, 97, 93, 89,
  87, 84, 85, 82, 80, 77, 74, 78, 76, 80, 79, 77, 78, 75, 73, 69, 67,
  64, 68, 66, 70, 69, 73, 71, 75, 74, 72, 73, 70, 67, 68, 65, 63, 60,
  64, 62, 66, 65, 69, 68, 66, 63, 64, 61, 59, 56, 60, 58, 62, 61, 65,
  64, 62, 59, 60, 57, 55, 51, 48, 49, 46, 44, 40, 37, 34, 35, 32, 28,
  26, 23, 27, 25, 29, 28, 32, 30, 34, 33, 31, 32, 36, 35, 33, 29, 26,
  27, 24, 22, 19, 23, 21, 25, 24, 28, 27, 25, 22, 23, 20, 18, 14, 18,
  22, 20, 24, 23, 21, 22, 19, 16, 20, 18, 22, 21, 19, 20, 24, 23, 21,
  17, 15, 17, 15, 19, 18, 16 ]
gap> lngs := List(Cycle(g,25855),n->Length(Trajectory(T,n,[1])));;
gap> Minimum(lngs);
55
gap> Maximum(lngs);
521
gap> Position(lngs,55);
15
gap> Position(lngs,521);
2807
```

Finally let's have a look at elements of G with small modulus:

Example

```
gap> B := RestrictedBall(G,One(G),20,36:Spheres);;
gap> List(B,Length);
[ 1, 3, 6, 12, 4, 6, 6, 4, 4, 4, 6, 6, 3, 3, 2, 0, 0, 0, 0, 0, 0 ]
gap> Sum(last);
70
gap> Position(last2,0)-2;
14
```

So we have 70 elements of modulus 36 or less in G which can be reached from the identity by successive multiplication with generators without passing elements with mudulus exceeding 36. Further we see that the longest word in the generators yielding an element with modulus at most 36 has length 14. Now we double our bound on the modulus:

Example

```
gap> B := RestrictedBall(G,One(G),100,72:Spheres);;
gap> List(B,Length);
[ 1, 3, 6, 12, 22, 14, 18, 22, 24, 26, 26, 34, 35, 32, 37, 38, 46, 58,
  65, 73, 82, 91, 93, 96, 110, 121, 114, 117, 146, 138, 148, 168, 174,
  196, 215, 214, 232, 255, 280, 305, 315, 359, 377, 371, 363, 366, 397,
  419, 401, 405, 405, 401, 407, 415, 435, 424, 401, 359, 338, 330, 332,
  281, 278, 271, 269, 254, 255, 257, 258, 258, 233, 215, 202, 185, 154,
  121, 88, 55, 35, 20, 10, 5, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0 ]
gap> Sum(last);
15614
gap> Position(last2,0)-2;
83
gap> Collected(List(Flat(B),Modulus));
[ [ 1, 1 ], [ 6, 3 ], [ 12, 4 ], [ 18, 2 ], [ 24, 4 ], [ 36, 56 ],
  [ 48, 4 ], [ 72, 15540 ] ]
```

We observe that there are 15540 elements in G with modulus 72 which are "reachable" from the identity by successive multiplication with generators without passing elements with mudulus exceeding 72. Further we see that the longest word in the generators yielding an element with modulus at most 72 has length 83.

It is obvious that many questions regarding the group G remain open.

7.4 A group with huge finite orbits

In this section we investigate a group which has huge finite orbits on $\mathbb{Z}$.

Example

```
gap> a := ClassTransposition(0,2,1,2);;
gap> b := ClassTransposition(0,5,4,5);;
gap> c := ClassTransposition(1,4,0,6);;
gap> G := Group(a,b,c);
<(0(2),1(2)),(0(5),4(5)),(1(4),0(6))>
```

```
gap> LoadDatabaseOfGroupsGeneratedBy3ClassTranspositions();
"3CTsGroups6"
gap> 3CTsGroups6.Id3CTsGroup(G,3CTsGroups6.grps); # 'catalogue number' of G
1284
```

We look for orbits of length at most 100 containing an integer in the range `[0..1000]`:

Example

```
gap> orbs := ShortOrbits(G,[0..1000],100);;
gap> List(orbs,Length);
[ 16, 2, 24, 2, 2, 2, 8, 2, 8, 2, 2, 8, 2, 8, 2, 2, 2, 40, 2, 8, 24, 2,
  8, 2, 2, 8, 2, 24, 8, 2, 56, 2, 2, 2, 8, 2, 8, 2, 2, 8, 2, 8, 2, 2, 2,
  24, 2, 8, 2, 8, 2, 2, 8, 2, 8, 2, 24, 2, 2, 2, 8, 2, 8, 2, 2, 8, 2, 8,
  2, 2, 2, 2, 8, 24, 2, 8, 2, 2, 8, 2, 24, 8, 2, 2, 2, 2, 8, 2, 8, 2, 2,
  8, 2, 8, 2, 2, 2, 24, 2, 8, 2, 8, 2, 2, 8, 2, 8, 2, 24, 2, 2 ]
gap> Collected(last);
[ [ 2, 67 ], [ 8, 32 ], [ 16, 1 ], [ 24, 9 ], [ 40, 1 ], [ 56, 1 ] ]
gap> Length(Difference([0..1000],Union(orbs)));
491
```

So almost half of the integers in the range `[0..1000]` lie in orbits of length larger than 100. In fact there are much larger orbits. For example:

Example

```
gap> B := Ball(G,32,500,OnPoints:Spheres);; # compute ball about 32
gap> Position(B,[]); # <> fail -> we have exhausted the orbit
354
gap> Sum(List(B,Length)); # the orbit length
6296
gap> Maximum(Flat(B)); # the largest integer in the orbit
330163638160950979743767 9
gap> B := Ball(G,736,5000,OnPoints:Spheres);; # the same for 736 ...
gap> Position(B,[]);
2997
gap> Sum(List(B,Length)); # the orbit length for this time
495448
gap> Maximum(Flat(B));
24613742765227139490361518119031497856901514673563546528602769571523014 65\
05463606966271871948494398819734424516866850247086526345938611467097523 78\
84707849340628785457338192055371315596774155049883 9
```

It seems that the cycles of *abc* completely traverse all orbits of G, with the only exception of the orbit of 0. Let's check this in the above examples:

Example

```
gap> g := a*b*c;;
gap> Display(g);

Rcwa permutation of Z with modulus 60
```

```
        /
        | n-1        if n in 3(30) U 9(30) U 17(30) U 23(30) U 27(30) U
        |                    29(30)
        | 3n/2       if n in 0(20) U 12(20) U 16(20)
        | n+1        if n in 2(20) U 6(20) U 10(20)
        | (2n+1)/3   if n in 7(30) U 13(30) U 19(30)
        | n+3        if n in 1(30) U 11(30)
 n |-> <  n-5        if n in 15(30) U 25(30)
        | (3n+12)/2 if n in 4(20)
        | (3n-12)/2 if n in 8(20)
        | n+5        if n in 14(20)
        | n-3        if n in 18(20)
        | (2n-7)/3   if n in 5(30)
        | (2n+9)/3   if n in 21(30)
        \

gap> Length(Cycle(g,32));
6296
gap> Length(Cycle(g,736));
495448
```

Representatives and lengths of the cycles of g which intersect nontrivially with the range `[0..1000]` are as follows:

Example

```
gap> CycleRepresentativesAndLengths(g,[0..1000]:notify:=50000);
n = 736: after 50000 steps, the iterate has 157 binary digits.
n = 736: after 100000 steps, the iterate has 135 binary digits.
n = 736: after 150000 steps, the iterate has 131 binary digits.
n = 736: after 200000 steps, the iterate has 507 binary digits.
n = 736: after 250000 steps, the iterate has 414 binary digits.
n = 736: after 300000 steps, the iterate has 457 binary digits.
n = 736: after 350000 steps, the iterate has 465 binary digits.
n = 736: after 400000 steps, the iterate has 325 binary digits.
n = 736: after 450000 steps, the iterate has 534 binary digits.
n = 896: after 50000 steps, the iterate has 359 binary digits.
n = 896: after 100000 steps, the iterate has 206 binary digits.
[ [ 1, 15 ], [ 2, 2 ], [ 16, 24 ], [ 22, 2 ], [ 26, 2 ], [ 32, 6296 ],
  [ 46, 2 ], [ 52, 8 ], [ 56, 296 ], [ 62, 2 ], [ 76, 8 ], [ 82, 2 ],
  [ 86, 2 ], [ 92, 8 ], [ 106, 2 ], [ 112, 104 ], [ 116, 8 ],
  [ 122, 2 ], [ 136, 440 ], [ 142, 2 ], [ 146, 2 ], [ 152, 40 ],
  [ 166, 2 ], [ 172, 8 ], [ 176, 24 ], [ 182, 2 ], [ 196, 8 ],
  [ 202, 2 ], [ 206, 2 ], [ 212, 8 ], [ 226, 2 ], [ 232, 24 ],
  [ 236, 8 ], [ 242, 2 ], [ 256, 56 ], [ 262, 2 ], [ 266, 2 ],
  [ 272, 408 ], [ 286, 2 ], [ 292, 8 ], [ 296, 104 ], [ 302, 2 ],
  [ 316, 8 ], [ 322, 2 ], [ 326, 2 ], [ 332, 8 ], [ 346, 2 ],
  [ 352, 264 ], [ 356, 8 ], [ 362, 2 ], [ 376, 1304 ], [ 382, 2 ],
  [ 386, 2 ], [ 392, 24 ], [ 406, 2 ], [ 412, 8 ], [ 416, 200 ],
  [ 422, 2 ], [ 436, 8 ], [ 442, 2 ], [ 446, 2 ], [ 452, 8 ],
  [ 466, 2 ], [ 472, 104 ], [ 476, 8 ], [ 482, 2 ], [ 496, 24 ],
  [ 502, 2 ], [ 506, 2 ], [ 512, 696 ], [ 526, 2 ], [ 532, 8 ],
```

```
[ 536, 3912 ], [ 542, 2 ], [ 556, 8 ], [ 562, 2 ], [ 566, 2 ],
[ 572, 8 ], [ 586, 2 ], [ 592, 888 ], [ 596, 8 ], [ 602, 2 ],
[ 616, 728 ], [ 622, 2 ], [ 626, 2 ], [ 632, 2776 ], [ 646, 2 ],
[ 652, 8 ], [ 656, 24 ], [ 662, 2 ], [ 676, 8 ], [ 682, 2 ],
[ 686, 2 ], [ 692, 8 ], [ 706, 2 ], [ 712, 24 ], [ 716, 8 ],
[ 722, 2 ], [ 736, 495448 ], [ 742, 2 ], [ 746, 2 ], [ 752, 1272 ],
[ 766, 2 ], [ 772, 8 ], [ 776, 376 ], [ 782, 2 ], [ 796, 8 ],
[ 802, 2 ], [ 806, 2 ], [ 812, 8 ], [ 826, 2 ], [ 832, 120 ],
[ 836, 8 ], [ 842, 2 ], [ 856, 2264 ], [ 862, 2 ], [ 866, 2 ],
[ 872, 24 ], [ 886, 2 ], [ 892, 8 ], [ 896, 132760 ], [ 902, 2 ],
[ 916, 8 ], [ 922, 2 ], [ 926, 2 ], [ 932, 8 ], [ 946, 2 ],
[ 952, 456 ], [ 956, 8 ], [ 962, 2 ], [ 976, 24 ], [ 982, 2 ],
[ 986, 2 ], [ 992, 1064 ] ]
```

So far the author has checked that all positive integers less than 173176 lie in finite cycles of g. Several of them are longer than 1000000, and the cycle containing 25952 has length 245719352. Whether the cycle containing 173176 is finite or infinite has not been checked so far – in any case it is longer than 5700000000, and it exceeds 10^{40000}. Presumably it is finite as well, but checking this may require a lot of computing time.

On the one hand the cycles of g seem to behave "randomly", perhaps as if they would ascend or descend from one point to the next by a certain factor depending on which side a thrown coin falls on. – In this "model", cycles would be finite with probability 1 since the simple random walk on $\mathbb{Z}$ is recurrent. On the other, there seems to be quite some structure on them, however little is known so far.

First we observe that each orbit under the action of G seems to split into two cycles of $h := abcacb$ of the same length (of course this has been checked for many more orbits than those shown here):

Example

```
gap> h := a*b*c*a*c*b;
<rcwa permutation of Z with modulus 360>
gap> List(CyclesOnFiniteOrbit(G,h,32),Length);
[ 3148, 3148 ]
gap> List(CyclesOnFiniteOrbit(G,h,736),Length);
[ 247724, 247724 ]
```

One cycle seems to contain the points at the odd positions and the other seems to contain the points at the even positions in the cycle of g:

Example

```
gap> cycle_g := Cycle(g,32);;
gap> positions1 := List(Cycle(h,32),n->Position(cycle_g,n));;
gap> Collected(positions1 mod 2);
[ [ 1, 3148 ] ]
gap> positions2 := List(Cycle(h,33),n->Position(cycle_g,n));;
gap> Collected(positions2 mod 2);
[ [ 0, 3148 ] ]
```

However the ordering in which these points are traversed looks pretty "scrambled":

Example

```
gap> positions1{[1..200]};
[ 1, 6271, 6291, 6281, 6285, 6287, 6283, 6289, 6273, 6275, 6277, 6279,
  6293, 5, 15, 17, 19, 6259, 6261, 6263, 6265, 21, 23, 25, 41, 6227,
  6229, 6231, 6233, 6235, 6237, 6239, 43, 53, 55, 57, 63, 59, 61, 65,
  45, 47, 49, 51, 67, 6223, 6221, 69, 6163, 6215, 6205, 6209, 6211,
  6207, 6213, 6165, 6171, 6177, 6179, 6181, 6183, 6175, 6173, 6185,
  6189, 6191, 6187, 6193, 6169, 6167, 6195, 6199, 6201, 6197, 6203,
  6217, 73, 83, 85, 87, 103, 113, 115, 117, 4357, 4361, 4363, 4359,
  4365, 4371, 4373, 4375, 4377, 4369, 4367, 4379, 119, 121, 123, 125,
  129, 131, 127, 133, 139, 141, 143, 145, 137, 135, 147, 149, 151, 153,
  155, 159, 161, 157, 163, 169, 175, 4283, 4281, 177, 4271, 4273, 4275,
  4277, 181, 4255, 4257, 4259, 4261, 4263, 4265, 4267, 183, 2161, 2163,
  4195, 4199, 4201, 4197, 4203, 4209, 4211, 4213, 4215, 4207, 4205,
  4217, 2165, 2167, 2169, 2171, 2175, 2177, 2173, 2179, 2185, 2187,
  2189, 2191, 2183, 2181, 2193, 2195, 2197, 2199, 2201, 2467, 2469,
  4117, 4121, 4123, 4119, 4125, 4131, 4133, 4135, 4137, 4129, 4127,
  4139, 2471, 2473, 2475, 2477, 2487, 2489, 2491, 2507, 2517, 2519,
  2521, 2537, 3923, 3925, 3941, 3943 ]
```

7.5 A group which acts 4-transitively on the positive integers

In this section, we would like to show that the group G generated by the two permutations

Example

```
gap> a := RcwaMapping([[3,0,2],[3,1,4],[3,0,2],[3,-1,4]]);;
gap> u := RcwaMapping([[3,0,5],[9,1,5],[3,-1,5],[9,-2,5],[9,4,5]]);;
gap> SetName(a,"a"); SetName(u,"u"); G := Group(a,u);;
```

which we have already investigated in earlier examples acts 4-transitively on the set of positive integers. Obviously, it acts on the set of positive integers. First we show that this action is transitive. We start by checking in which residue classes sufficiently large positive integers are mapped to smaller ones by a suitable group element:

Example

```
gap> List([a,a^-1,u,u^-1],DecreasingOn);
[ 1(2), 0(3), 0(5) U 2(5), 2(3) ]
gap> Union(last);
Z \ 4(30) U 16(30) U 28(30)
```

We see that we cannot always choose such a group element from the set of generators and their inverses – otherwise the union would be `Integers`.

Example

```
gap> List([a,a^-1,u,u^-1,a^2,a^-2,u^2,u^-2],DecreasingOn);
[ 1(2), 0(3), 0(5) U 2(5), 2(3), 1(8) U 7(8), 0(3) U 2(9) U 7(9),
```

```
  0(25) U 12(25) U 17(25) U 20(25), 2(3) U 1(9) U 3(9) ]
gap> Union(last); # Still not enough ...
Z \ 4(90) U 58(90) U 76(90)
gap> List([a,a^-1,u,u^-1,a^2,a^-2,u^2,u^-2,a*u,u*a,(a*u)^-1,(u*a)^-1],
>          DecreasingOn);
[ 1(2), 0(3), 0(5) U 2(5), 2(3), 1(8) U 7(8), 0(3) U 2(9) U 7(9),
  0(25) U 12(25) U 17(25) U 20(25), 2(3) U 1(9) U 3(9),
  3(5) U 0(10) U 7(20) U 9(20), 0(5) U 2(5), 2(3), 3(9) U 4(9) U 8(9) ]
gap> Union(last); # ... but that's it!
Integers
```

Finally, we have to deal with "small" integers. We use the notation for the coefficients of rcwa mappings introduced at the beginning of this manual. Let $c_{r(m)} > a_{r(m)}$. Then we easily see that $(a_{r(m)}n + b_{r(m)})/c_{r(m)} > n$ implies $n < b_{r(m)}/(c_{r(m)} - a_{r(m)})$. Thus we can restrict our considerations to integers $n < b_{\max}$, where $b_{\max}$ is the largest second entry of a coefficient triple of one of the group elements in our list:

Example

```
gap> List([a,a^-1,u,u^-1,a^2,a^-2,u^2,u^-2,a*u,u*a,(a*u)^-1,(u*a)^-1],
>          f->Maximum(List(Coefficients(f),c->c[2])));
[ 1, 1, 4, 2, 7, 7, 56, 28, 25, 17, 17, 11 ]
gap> Maximum(last);
56
```

Thus this upper bound is 56. The rest is easy – all we have to do is to check that the orbit containing 1 contains also all other positive integers less than or equal to 56:

Example

```
gap> S := [1];;
gap> while not IsSubset(S,[1..56]) do
>      S := Union(S,S^a,S^u,S^(a^-1),S^(u^-1));
>    od;
gap> IsSubset(S,[1..56]);
true
```

Checking 2-transitivity is computationally harder, and in the sequel we will omit some steps which are in practice needed to find out "what to do". The approach taken here is to show that the stabilizer of 1 in G acts transitively on the set of positive integers greater than 1. We do this by similar means as used above for showing the transitivity of the action of G on the positive integers. We start by determining all products of at most 5 generators and their inverses, which stabilize 1 (taking at most 4-generator products would not suffice!):

Example

```
gap> gens := [a,u,a^-1,u^-1];;
gap> tups := Concatenation(List([1..5],k->Tuples([1..4],k)));;
gap> Length(tups);
1364
gap> tups := Filtered(tups,tup->ForAll([[1,3],[3,1],[2,4],[4,2]],
```

```
>                                        l->PositionSublist(tup,l)=fail));;
gap> Length(tups);
484
gap> stab := [];;
gap> for tup in tups do
>      n := 1;
>      for i in tup do n := n^gens[i]; od;
>      if n = 1 then Add(stab,tup); fi;
>    od;
gap> Length(stab);
118
gap> stabelm := List(stab,tup->Product(List(tup,i->gens[i])));;
gap> ForAll(stabelm,elm->1^elm=1); # Check.
true
```

The resulting products have various different not quite small moduli:

Example

```
gap> List(stabelm,Modulus);
[ 4, 3, 16, 25, 9, 81, 64, 100, 108, 100, 25, 75, 27, 243, 324, 243,
  256, 400, 144, 400, 100, 432, 324, 400, 80, 400, 625, 25, 75, 135,
  150, 75, 225, 81, 729, 486, 729, 144, 144, 81, 729, 1296, 729, 6561,
  1024, 1600, 192, 1600, 400, 576, 432, 1600, 320, 1600, 2500, 100, 100,
  180, 192, 192, 108, 972, 1728, 972, 8748, 1600, 400, 320, 80, 1600,
  2500, 300, 2500, 625, 625, 75, 675, 75, 75, 135, 405, 600, 120, 600,
  1875, 75, 225, 405, 225, 225, 675, 243, 2187, 729, 2187, 216, 216,
  243, 2187, 1944, 2187, 19683, 576, 144, 576, 432, 81, 81, 729, 2187,
  5184, 324, 8748, 243, 2187, 19683, 26244, 19683 ]
gap> Lcm(last);
12597120000
gap> Collected(Factors(last));
[ [ 2, 10 ], [ 3, 9 ], [ 5, 4 ] ]
```

Similar as before, we determine for any of the above mappings the residue classes whose elements larger than the largest $b_{r(m)}$ - coefficient of the respective mapping are mapped to smaller integers:

Example

```
gap> decs := List(stabelm,DecreasingOn);;
gap> List(decs,Modulus);
[ 2, 3, 8, 25, 9, 9, 16, 100, 12, 50, 25, 75, 27, 81, 54, 81, 64, 400,
  48, 200, 100, 72, 108, 400, 80, 200, 625, 25, 75, 45, 75, 75, 225, 81,
  243, 81, 243, 144, 144, 81, 243, 216, 243, 243, 128, 1600, 64, 400,
  400, 48, 144, 1600, 320, 400, 2500, 100, 100, 60, 96, 192, 108, 324,
  144, 324, 972, 400, 400, 80, 80, 400, 2500, 100, 1250, 625, 625, 25,
  75, 75, 75, 45, 135, 600, 120, 150, 1875, 75, 225, 135, 225, 225, 675,
  243, 729, 243, 729, 108, 216, 243, 729, 162, 729, 2187, 144, 144, 144,
  144, 81, 81, 243, 729, 1296, 324, 972, 243, 729, 2187, 1458, 2187 ]
gap> Lcm(last);
174960000
```

Since the least common multiple of the moduli of these unions of residue classes is as large as 174960000, directly forming their union and checking whether it is equal to the set of integers would take relatively much time and memory. However, starting with the set of integers and subtracting the above sets one-by-one in a suitably chosen order is cheap:

Example

```
gap> SortParallel(decs,stabelm,
>      function(S1,S2)
>        return First([1..100],k->Factorial(k) mod Modulus(S1)=0)
>             < First([1..100],k->Factorial(k) mod Modulus(S2)=0);
>      end);
gap> S := Integers;;
gap> for i in [1..Length(decs)] do
>      S_old := S; S := Difference(S,decs[i]);
>      if S <> S_old then ViewObj(S); Print("\n"); fi;
>      if S = [] then maxind := i; break; fi;
>    od;
0(2)
2(6) U 4(6)
<union of 8 residue classes (mod 30)>
<union of 19 residue classes (mod 90) (9 classes)>
<union of 114 residue classes (mod 720)>
<union of 99 residue classes (mod 720)>
<union of 57 residue classes (mod 720)>
<union of 54 residue classes (mod 720)>
<union of 41 residue classes (mod 720)>
<union of 35 residue classes (mod 720)>
<union of 8 residue classes (mod 720) (6 classes)>
4(720) U 94(720) U 148(720) U 238(720)
<union of 24 residue classes (mod 5760)>
<union of 72 residue classes (mod 51840)>
<union of 48 residue classes (mod 51840)>
<union of 192 residue classes (mod 259200)>
<union of 168 residue classes (mod 259200)>
<union of 120 residue classes (mod 259200)>
<union of 96 residue classes (mod 259200)>
<union of 72 residue classes (mod 259200)>
<union of 60 residue classes (mod 259200)>
<union of 48 residue classes (mod 259200)>
<union of 24 residue classes (mod 259200)>
<union of 12 residue classes (mod 259200) (6 classes)>
<union of 24 residue classes (mod 777600)>
<union of 12 residue classes (mod 777600) (6 classes)>
111604(194400) U 14404(777600) U 208804(777600)
[  ]
```

Similar as above, it remains to check that the “small” integers all lie in the orbit containing 2. Obviously, it is sufficient to check that any integer greater than 2 is mapped to a smaller one by some suitably chosen element of the stabilizer under consideration:

Example

```
gap> Maximum(List(stabelm{[1..maxind]},
```

```
>                  f->Maximum(List(Coefficients(f),c->c[2]))));
6581
gap> Filtered([3..6581],n->Minimum(List(stabelm,elm->n^elm))>=n);
[ 4 ]
```

We have to treat 4 separately:

Example

```
gap> 1^(u*a*u^2*a^-1*u);
1
gap> 4^(u*a*u^2*a^-1*u);
3
```

Now we know that any positive integer greater than 1 lies in the same orbit under the action of the stabilizer of 1 in G as 2, thus that this stabilizer acts transitively on $\mathbb{N} \setminus \{1\}$. But this means that we have established the 2-transitivity of the action of G on $\mathbb{N}$.

In the following, we essentially repeat the above steps to show that this action is indeed 3-transitive:

Example

```
gap> tups := Concatenation(List([1..6],k->Tuples([1..4],k)));;
gap> tups := Filtered(tups,tup->ForAll([[1,3],[3,1],[2,4],[4,2]],
>                                        l->PositionSublist(tup,l)=fail));;
gap> stab := [];;
gap> for tup in tups do
>      l := [1,2];
>      for i in tup do l := List(l,n->n^gens[i]); od;
>      if l = [1,2] then Add(stab,tup); fi;
>    od;
gap> Length(stab);
212
gap> stabelm := List(stab,tup->Product(List(tup,i->gens[i])));;
gap> decs := List(stabelm,DecreasingOn);;
gap> SortParallel(decs,stabelm,function(S1,S2)
>      return First([1..100],k->Factorial(k) mod Mod(S1)=0)
>           < First([1..100],k->Factorial(k) mod Mod(S2)=0); end);
gap> S := Integers;;
gap> for i in [1..Length(decs)] do
>      S_old := S; S := Difference(S,decs[i]);
>      if S <> S_old then ViewObj(S); Print("\n"); fi;
>      if S = [] then break; fi;
>    od;
Z \ 1(8) U 7(8)
<union of 151 residue classes (mod 240)>
<union of 208 residue classes (mod 720)>
<union of 51 residue classes (mod 720)>
<union of 45 residue classes (mod 720)>
<union of 39 residue classes (mod 720)>
<union of 33 residue classes (mod 720)>
<union of 23 residue classes (mod 720)>
```

```
<union of 19 residue classes (mod 720) (7 classes)>
<union of 17 residue classes (mod 720) (6 classes)>
<union of 16 residue classes (mod 720) (7 classes)>
<union of 14 residue classes (mod 720) (9 classes)>
<union of 8 residue classes (mod 720) (6 classes)>
<union of 7 residue classes (mod 720) (6 classes)>
238(360) U 4(720) U 148(720) U 454(720)
<union of 38 residue classes (mod 5760)>
<union of 37 residue classes (mod 5760)>
<union of 25 residue classes (mod 5760)>
<union of 21 residue classes (mod 5760)>
<union of 17 residue classes (mod 5760) (13 classes)>
<union of 16 residue classes (mod 5760) (12 classes)>
<union of 138 residue classes (mod 51840)>
<union of 48 residue classes (mod 51840)>
<union of 32 residue classes (mod 51840)>
<union of 20 residue classes (mod 51840) (14 classes)>
<union of 16 residue classes (mod 51840) (12 classes)>
<union of 68 residue classes (mod 259200)>
<union of 42 residue classes (mod 259200)>
<union of 32 residue classes (mod 259200)>
<union of 26 residue classes (mod 259200)>
<union of 25 residue classes (mod 259200)>
<union of 11 residue classes (mod 259200) (10 classes)>
<union of 10 residue classes (mod 259200) (9 classes)>
<union of 7 residue classes (mod 259200) (6 classes)>
13414(129600) U 2164(259200) U 66964(259200) U 228964(259200)
2164(259200) U 66964(259200) U 228964(259200)
[  ]
gap> Maximum(List(stabelm,f->Maximum(List(Coefficients(f),c->c[2]))));
515816
gap> smallnum := [4..515816];;
gap> for i in [1..Length(stabelm)] do
>      smallnum := Filtered(smallnum,n->n^stabelm[i]>=n);
>    od;
gap> smallnum;
[  ]
```

The same for 4-transitivity:

Example

```
gap> tups := Concatenation(List([1..8],k->Tuples([1..4],k)));;
gap> tups := Filtered(tups,tup->ForAll([[1,3],[3,1],[2,4],[4,2]],
>                                l->PositionSublist(tup,l)=fail));;
gap> stab := [];;
gap> for tup in tups do
>      l := [1,2,3];
>      for i in tup do l := List(l,n->n^gens[i]); od;
>      if l = [1,2,3] then Add(stab,tup); fi;
>    od;
gap> Length(stab);
528
```

```
gap> stabelm := [];;
gap> for i in [1..Length(stab)] do
>      elm := One(G);
>      for j in stab[i] do
>        if Modulus(elm) > 10000 then elm := fail; break; fi;
>        elm := elm * gens[j];
>      od;
>      if elm <> fail then Add(stabelm,elm); fi;
>    od;
gap> Length(stabelm);
334
gap> decs := List(stabelm,DecreasingOn);;
gap> SortParallel(decs,stabelm,
>      function(S1,S2)
>        return First([1..100],k->Factorial(k) mod Modulus(S1) = 0)
>             < First([1..100],k->Factorial(k) mod Modulus(S2) = 0);
>      end);
gap> S := Integers;;
gap> for i in [1..Length(decs)] do
>      S_old := S; S := Difference(S,decs[i]);
>      if S <> S_old then ViewObj(S); Print("\n"); fi;
>      if S = [] then maxind := i; break; fi;
>    od;
Z \ 1(8) U 7(8)
<union of 46 residue classes (mod 72)>
<union of 20 residue classes (mod 72) (8 classes)>
4(18)
<union of 28 residue classes (mod 576)>
<union of 22 residue classes (mod 576)>
<union of 21 residue classes (mod 576)>
40(72) U 4(144) U 94(144) U 346(576) U 418(576)
<union of 16 residue classes (mod 576) (6 classes)>
<union of 15 residue classes (mod 576) (6 classes)>
4(144) U 94(144) U 346(576) U 418(576)
<union of 30 residue classes (mod 5184)>
<union of 26 residue classes (mod 5184)>
<union of 6 residue classes (mod 1296)>
<union of 504 residue classes (mod 129600)>
<union of 324 residue classes (mod 129600)>
<union of 282 residue classes (mod 129600)>
<union of 239 residue classes (mod 129600)>
<union of 218 residue classes (mod 129600)>
<union of 194 residue classes (mod 129600)>
<union of 154 residue classes (mod 129600)>
<union of 97 residue classes (mod 129600)>
<union of 85 residue classes (mod 129600)>
<union of 77 residue classes (mod 129600)>
<union of 67 residue classes (mod 129600)>
<union of 125 residue classes (mod 259200)>
<union of 108 residue classes (mod 259200)>
<union of 107 residue classes (mod 259200)>
<union of 101 residue classes (mod 259200)>
<union of 100 residue classes (mod 259200)>
```

```
<union of 84 residue classes (mod 259200)>
<union of 80 residue classes (mod 259200)>
<union of 76 residue classes (mod 259200)>
<union of 70 residue classes (mod 259200)>
<union of 66 residue classes (mod 259200)>
<union of 54 residue classes (mod 259200)>
<union of 53 residue classes (mod 259200)>
<union of 47 residue classes (mod 259200)>
<union of 43 residue classes (mod 259200)>
<union of 31 residue classes (mod 259200)>
<union of 24 residue classes (mod 259200)>
<union of 23 residue classes (mod 259200)>
<union of 13 residue classes (mod 259200) (8 classes)>
57406(129600) U 115006(129600) U 192676(259200) U 250276(259200)
57406(129600) U 192676(259200) U 250276(259200) U 374206(388800)
57406(129600) U 192676(259200) U 250276(259200)
250276(259200) U 57406(388800) U 316606(388800) U 451876(777600)
316606(388800) U 451876(777600) U 509476(777600) U 768676(777600)
<union of 18 residue classes (mod 3110400) (6 classes)>
451876(777600) U 509476(777600) U 705406(777600) U 768676(777600)
 U 2649406(3110400)
451876(777600) U 705406(777600) U 768676(777600) U 2649406(3110400)
451876(777600) U 705406(777600) U 2649406(3110400)
705406(777600) U 2007076(3110400) U 2649406(3110400) U 2784676(3110400)
<union of 14 residue classes (mod 9331200) (8 classes)>
2260606(2332800) U 5759806(9331200) U 5895076(9331200) U 8227876(9331200)
4593406(6998400) U 15091006(27993600) U 17559076(27993600)
 U 24557476(27993600)
<union of 14 residue classes (mod 83980800) (8 classes)>
18590206(20995200) U 24557476(83980800) U 45552676(83980800)
 U 71078206(83980800)
[  ]
gap> Maximum(List(stabelm{[1..maxind]},
>                  f->Maximum(List(Coefficients(f),c->c[2]))));
58975
gap> smallnum := [5..58975];;
gap> for i in [1..maxind] do
>      smallnum := Filtered(smallnum,n->n^stabelm[i]>=n);
>    od;
gap> smallnum;
[  ]
```

There is even some evidence that the degree of transitivity of the action of G on the positive integers is higher than 4:

Example

```
gap> phi := EpimorphismFromFreeGroup(G);
[ a, u ] -> [ a, u ]
gap> F := Source(phi);
<free group on the generators [ a, u ]>
gap> List([5..20],
>         n->RepresentativeActionPreImage(G,[1,2,3,4,5],
```

```
>                                    [1,2,3,4,n],OnTuples,F));
[ <identity ...>, a^-3*u^4*a*u^-2*a^2, a^-1*(a^-1*u)^4*a^-1*u^-1*a,
  a^4*u^-2*a^-4, a^-1*u^-4*a, (u^2*a^-1)^2*u^-2, u^-2*a^-2*u^4,
  a^-1*u^2*a, a^-1*u^-6*a, a^2*u^4*a^2*u^2, u^-4*a*u^-2*a^-3,
  a^-1*u^-2*a^-3*u^4*a^2, a^2*(a*u^2)^2, (a*u^-4)^2*a^-2,
  u^-2*a*u^2*a*u^-2, u^-4*a^2*u^2 ]
```

Enter `AssignGlobals(LoadRCWAExamples().CollatzlikePerms);` in order to assign the global variables defined in this section.

7.6 A group which acts 3-transitively, but not 4-transitively on $\mathbb{Z}$

In this section, we would like to show that the group G generated by the two permutations $n \mapsto n+1$ and $\tau_{1(2),0(4)}$ acts 3-transitively, but not 4-transitively on the set of integers.

Example

```
gap> G := Group(ClassShift(0,1),ClassTransposition(1,2,0,4));
<rcwa group over Z with 2 generators>
gap> IsTame(G);
false
gap> (G.1^-2*G.2)^3*(G.1^2*G.2)^3; # G <> the free product C_infty * C_2.
IdentityMapping( Integers )
gap> Display(G:CycleNotation:=false);

Wild rcwa group over Z, generated by

[
Tame rcwa permutation of Z: n -> n + 1

Rcwa permutation of Z with modulus 4, of order 2

        /
        | 2n-2    if n in 1(2)
 n |-> <  (n+2)/2 if n in 0(4)
        | n       if n in 2(4)
        \

]
```

This group acts transitively on $\mathbb{Z}$, since already the cyclic group generated by the first of the two generators does so. Next we have to show that it acts 2-transitively. We essentially proceed as in the example in the previous section, by checking that the stabilizer of 0 acts transitively on $\mathbb{Z} \setminus \{0\}$.

Example

```
gap> gens := [ClassShift(0,1)^-1,ClassTransposition(1,2,0,4),
>              ClassShift(0,1)];;
gap> tups := Concatenation(List([1..6],k->Tuples([-1,0,1],k)));;
gap> tups := Filtered(tups,tup->ForAll([[0,0],[-1,1],[1,-1]],
>                                     l->PositionSublist(tup,l)=fail));;
```

```
gap> Length(tups);
189
gap> stab := [];;
gap> for tup in tups do
>      n := 0;
>      for i in tup do n := n^gens[i+2]; od;
>      if n = 0 then Add(stab,tup); fi;
>    od;
gap> stabelm := List(stab,tup->Product(List(tup,i->gens[i+2])));;
gap> Collected(List(stabelm,Modulus));
[ [ 4, 6 ], [ 8, 4 ], [ 16, 3 ] ]
gap> decs := List(stabelm,DecreasingOn);
[ 0(4), 3(4), 0(4), 3(4), 2(4), 0(4), 4(8), 2(4), 2(4), 0(4), 1(4),
  0(8), 3(8) ]
gap> Union(decs);
Integers
```

Similar as in the previous section, it remains to check that the integers with "small" absolute value all lie in the orbit containing 1 under the action of the stabilizer of 0:

Example

```
gap> Maximum(List(stabelm,f->Maximum(List(Coefficients(f),
>                                     c->AbsInt(c[2])))));
21
gap> S := [1];;
gap> for elm in stabelm do S := Union(S,S^elm,S^(elm^-1)); od;
gap> IsSubset(S,Difference([-21..21],[0])); # Not yet ..
false
gap> for elm in stabelm do S := Union(S,S^elm,S^(elm^-1)); od;
gap> IsSubset(S,Difference([-21..21],[0])); # ... but now!
true
```

Now we have to check for 3-transitivity. Since we cannot find for every residue class an element of the pointwise stabilizer of $\{0,1\}$ which properly divides its elements, we also have to take additions and subtractions into consideration. Since the moduli of all of our stabilizer elements are quite small, simply looking at sets of representatives is cheap:

Example

```
gap> tups := Concatenation(List([1..10],k->Tuples([-1,0,1],k)));;
gap> tups := Filtered(tups,tup->ForAll([[0,0],[-1,1],[1,-1]],
>                                     l->PositionSublist(tup,l)=fail));;
gap> Length(tups);
3069
gap> stab := [];;
gap> for tup in tups do
>      l := [0,1];
>      for i in tup do l := List(l,n->n^gens[i+2]); od;
>      if l = [0,1] then Add(stab,tup); fi;
>    od;
gap> Length(stab);
```

```
10
gap> stabelm := List(stab,tup->Product(List(tup,i->gens[i+2])));;
gap> Maximum(List(stabelm,Modulus));
8
gap> Maximum(List(stabelm,
>                f->Maximum(List(Coefficients(f),c->AbsInt(c[2])))));
8
gap> decsp := List(stabelm,elm->Filtered([9..16],n->n^elm<n));
[ [ 9, 13 ], [ 10, 12, 14, 16 ], [ 12, 16 ], [ 9, 13 ], [ 12, 16 ],
  [ 9, 11, 13, 15 ], [ 9, 11, 13, 15 ], [ 12, 16 ], [ 12, 16 ],
  [ 9, 11, 13, 15 ] ]
gap> Union(decsp);
[ 9, 10, 11, 12, 13, 14, 15, 16 ]
gap> decsm := List(stabelm,elm->Filtered([-16..-9],n->n^elm>n));
[ [ -15, -13, -11, -9 ], [ -16, -12 ], [ -16, -12 ], [ -15, -11 ],
  [ -16, -14, -12, -10 ], [ -15, -11 ], [ -15, -11 ],
  [ -16, -14, -12, -10 ], [ -16, -14, -12, -10 ], [ -15, -11 ] ]
gap> Union(decsm);
[ -16, -15, -14, -13, -12, -11, -10, -9 ]
gap> S := [2];;
gap> for elm in stabelm do S := Union(S,S^elm,S^(elm^-1)); od;
gap> IsSubset(S,Difference([-8..8],[0,1]));
true
```

At this point we have established 3-transitivity. It remains to check that the group G does not act 4-transitively. We do this by checking that it is not transitive on 4-tuples (mod 4). Since n mod 8 determines the image of n under a generator of G (mod 4), it suffices to compute (mod 8):

Example

```
gap> orb := [[0,1,2,3]];;
gap> extend := function ()
>      local gen;
>      for gen in gens do
>        orb := Union(orb,List(orb,l->List(l,n->n^gen) mod 8));
>      od;
>    end;;
gap> repeat
>      old := ShallowCopy(orb);
>      extend(); Print(Length(orb),"\n");
>    until orb = old;
7
27
97
279
573
916
1185
1313
1341
1344
1344
gap> Length(Set(List(orb,l->l mod 4)));
```

```
120
gap> last < 4^4;
true
```

This shows that G acts not 4-transitively on $\mathbb{Z}$. The corresponding calculation for 3-tuples looks as follows:

Example

```
gap> orb := [[0,1,2]];;
gap> repeat
>      old := ShallowCopy(orb);
>      extend(); Print(Length(orb),"\n");
>    until orb = old;
7
27
84
207
363
459
503
512
512
gap> Length(Set(List(orb,l->l mod 4)));
64
gap> last = 4^3;
true
```

Needless to say that the latter kind of argumentation is not suitable for proving, but only for disproving k-transitivity.

7.7 An rcwa mapping which seems to be contracting, but very slow

The iterates of an integer under the Collatz mapping T seem to approach its contraction centre – this is the finite set where all trajectories end up after a finite number of steps – rather quickly and do not get very large before doing so (of course this is a purely heuristic statement as the $3n+1$ conjecture has not been proved so far!):

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);;
gap> S0 := LikelyContractionCentre(T,100,1000);
#I  Warning: `LikelyContractionCentre' is highly probabilistic.
The returned result can only be regarded as a rough guess.
See ?LikelyContractionCentre for more information.
[ -136, -91, -82, -68, -61, -55, -41, -37, -34, -25, -17, -10, -7, -5,
  -1, 0, 1, 2 ]
gap> S0^T = S0; # This holds by definition of the contraction centre.
true
gap> List([1..30],n->Length(Trajectory(T,n,S0)));
[ 1, 1, 5, 2, 4, 6, 11, 3, 13, 5, 10, 7, 7, 12, 12, 4, 9, 14, 14, 6, 6,
```

```
  11, 11, 8, 16, 8, 70, 13, 13, 13 ]
gap> Maximum(List([1..1000],n->Length(Trajectory(T,n,S0))));
113
gap> Maximum(List([1..1000],n->Maximum(Trajectory(T,n,S0))));
125252
```

The following mapping seems to be contracting as well, but its trajectories are much longer:

Example

```
gap> f6 := RcwaMapping([[ 1,0,6],[ 5, 1,6],[ 7,-2,6],
>                        [11,3,6],[11,-2,6],[11,-1,6]]);;
gap> Display(f6);

Rcwa mapping of Z with modulus 6

        /
        | n/6       if n in 0(6)
        | (5n+1)/6  if n in 1(6)
        | (7n-2)/6  if n in 2(6)
 n |-> <  (11n+3)/6 if n in 3(6)
        | (11n-2)/6 if n in 4(6)
        | (11n-1)/6 if n in 5(6)
        |
        \

gap> S0 := LikelyContractionCentre(f6,1000,100000);;
#I  Warning: `LikelyContractionCentre' is highly probabilistic.
The returned result can only be regarded as a rough guess.
See ?LikelyContractionCentre for more information.
gap> Trajectory(f6,25,S0);
[ 25, 21, 39, 72, 12, 2 ]
gap> List([1..100],n->Length(Trajectory(f6,n,S0)));
[ 1, 1, 3, 4, 1, 2, 3, 2, 1, 5, 7, 2, 8, 17, 3, 16, 1, 4, 17, 6, 5, 2,
  5, 5, 6, 1, 4, 2, 15, 1, 1, 3, 2, 5, 13, 3, 2, 3, 4, 1, 8, 4, 4, 2, 7,
  19, 23517, 3, 9, 3, 1, 18, 14, 2, 20, 23512, 14, 2, 6, 6, 1, 4, 19,
  12, 23511, 8, 23513, 10, 1, 13, 13, 3, 1, 23517, 7, 20, 7, 9, 9, 6,
  12, 8, 6, 18, 14, 23516, 31, 12, 23545, 4, 21, 19, 5, 1, 17, 17, 13,
  19, 6, 23515 ]
gap> Maximum(Trajectory(f6,47,S0));
7363391777762473304431877054771075818733690108051469808715809256737742295\
45698886054
```

Computing the trajectory of 3224 takes quite a while – this trajectory ascends to about $3 \cdot 10^{2197}$, before it approaches the fixed point 2 after 19949562 steps.

When constructing the mapping `f6`, the denominators of the partial mappings have been chosen to be equal and the numerators have been chosen to be numbers coprime to the common denominator, whose product is just a little bit smaller than the `Modulus(f6)`th power of the denominator. In the example we have $5 \cdot 7 \cdot 11^3 = 46585$ and $6^6 = 46656$.

Although the trajectories of `T` are much shorter than those of `f6`, it seems likely that this does not make the problem of deciding whether the mapping `T` is contracting essentially easier – even for mappings with much shorter trajectories than `T` the problem seems to be equally hard. A solution can usually only be found in trivial cases, i.e. for example when there is some k such that applying the kth power of the respective mapping to any integer decreases its absolute value.

Enter `AssignGlobals(LoadRCWAExamples().SlowlyContractingMappings);` in order to assign the global variables defined in this section.

7.8 Checking a result by P. Andaloro

In [And00], P. Andaloro has shown that proving that trajectories of integers $n \in 1(16)$ under the Collatz mapping always contain 1 would be sufficient to prove the $3n+1$ conjecture. In the sequel, this result is verified by RCWA. Checking that the union of the images of the residue class $1(16)$ under powers of the Collatz mapping T contains $\mathbb{Z} \setminus 0(3)$ is obviously enough. Thus we put $S := 1(16)$, and successively unite the set S with its image under T:

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);
<rcwa mapping of Z with modulus 2>
gap> S := ResidueClass(Integers,16,1);
1(16)
gap> S := Union(S,S^T);
1(16) U 2(24)
gap> S := Union(S,S^T);
1(12) U 2(24) U 17(48) U 33(48)
gap> S := Union(S,S^T);
<union of 30 residue classes (mod 144)>
gap> S := Union(S,S^T);
<union of 42 residue classes (mod 144)>
gap> S := Union(S,S^T);
<union of 172 residue classes (mod 432)>
gap> S := Union(S,S^T);
<union of 676 residue classes (mod 1296)>
gap> S := Union(S,S^T);
<union of 810 residue classes (mod 1296)>
gap> S := Union(S,S^T);
<union of 2638 residue classes (mod 3888)>
gap> S := Union(S,S^T);
<union of 33 residue classes (mod 48)>
gap> S := Union(S,S^T);
<union of 33 residue classes (mod 48)>
gap> Union(S,ResidueClass(Integers,3,0)); # Et voila ...
Integers
```

Further similar computations are shown in Section 7.17.

Enter `AssignGlobals(LoadRCWAExamples().CollatzMapping);` in order to assign the global variables defined in this section.

7.9 Two examples by Matthews and Leigh

In [ML87], K. R. Matthews and G. M. Leigh have shown that two trajectories of the following (surjective, but not injective) mappings are acyclic (mod x) and divergent:

Example

```
gap> x := Indeterminate(GF(4),1);; SetName(x,"x");
gap> R := PolynomialRing(GF(2),1);
GF(2)[x]
gap> ML1 := RcwaMapping(R,x,[[1,0,x],[(x+1)^3,1,x]]*One(R));;
gap> ML2 := RcwaMapping(R,x,[[1,0,x],[(x+1)^2,1,x]]*One(R));;
gap> Display(ML1);

Rcwa mapping of GF(2)[x] with modulus x

        /
        | P/x                       if P in 0(x)
 P |-> <  ((x^3+x^2+x+1)*P + 1)/x if P in 1(x)
        |
        \

gap> Display(ML2);

Rcwa mapping of GF(2)[x] with modulus x

        /
        | P/x                 if P in 0(x)
 P |-> <  ((x^2+1)*P + 1)/x if P in 1(x)
        |
        \

gap> List([ML1,ML2],IsSurjective);
[ true, true ]
gap> List([ML1,ML2],IsInjective);
[ false, false ]
gap> traj1 := Trajectory(ML1,One(R),16);
[ 1, x^2+x+1, x^4+x^2+x, x^3+x+1, x^5+x^4+x^2, x^4+x^3+x, x^3+x^2+1,
  x^5+x^2+1, x^7+x^6+x^5+x^3+1, x^9+x^7+x^6+x^5+x^3+x+1,
  x^11+x^10+x^8+x^7+x^6+x^5+x^2, x^10+x^9+x^7+x^6+x^5+x^4+x,
  x^9+x^8+x^6+x^5+x^4+x^3+1, x^11+x^8+x^7+x^6+x^4+x+1,
  x^13+x^12+x^11+x^8+x^7+x^6+x^4, x^12+x^11+x^10+x^7+x^6+x^5+x^3 ]
gap> traj2 := Trajectory(ML2,(x^3+x+1)*One(R),16);
[ x^3+x+1, x^4+x+1, x^5+x^3+x^2+x+1, x^6+x^3+1, x^7+x^5+x^4+x^2+x,
  x^6+x^4+x^3+x+1, x^7+x^4+x^3+x+1, x^8+x^6+x^5+x^4+x^3+x+1,
  x^9+x^6+x^3+x+1, x^10+x^8+x^7+x^5+x^4+x+1,
  x^11+x^8+x^7+x^5+x^4+x^3+x^2+x+1, x^12+x^10+x^9+x^8+x^7+x^5+1,
  x^13+x^10+x^7+x^4+x, x^12+x^9+x^6+x^3+1,
  x^13+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x,
  x^12+x^10+x^9+x^7+x^6+x^4+x^3+x+1 ]
```

The pattern which Matthews and Leigh used to show the divergence of the above trajectories can be recognized easily by looking at the corresponding Markov chains with the two states 0 mod x and 1 mod x:

Example

```
gap> traj1modx := Trajectory(ML1,One(R),400,x);;
gap> traj2modx := Trajectory(ML2,(x^3+x+1)*One(R),600,x);;
gap> List(traj1modx{[1..150]},val->Position([Zero(R),One(R)],val)-1);
[ 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1,
  1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
  1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
gap> List(traj2modx{[1..150]},val->Position([Zero(R),One(R)],val)-1);
[ 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
  1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
  0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 ]
```

What is important here are the lengths of the intervals between two changes from one state to the other:

Example

```
gap> ChangePoints := l->Filtered([1..Length(l)-1],pos->l[pos]<>l[pos+1]);;
gap> Diffs := l->List([1..Length(l)-1],pos->l[pos+1]-l[pos]);;
gap> Diffs(ChangePoints(traj1modx)); # The pattern in the first ...
[ 1, 1, 2, 4, 2, 2, 4, 8, 4, 4, 8, 16, 8, 8, 16, 32, 16, 16, 32, 64, 32,
  32, 64 ]
gap> Diffs(ChangePoints(traj2modx)); # ... and in the second example.
[ 1, 7, 1, 1, 1, 13, 1, 1, 1, 1, 1, 1, 1, 25, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 49, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 97, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 193, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
gap> Diffs(ChangePoints(last)); # Make this a bit more obvious.
[ 1, 3, 1, 7, 1, 15, 1, 31, 1, 63, 1 ]
```

This looks clearly acyclic, thus the trajectories diverge. Needless to say however that this computational evidence does not replace the proof along these lines given in the article cited above, but just sheds a light on the idea behind it.

Enter `AssignGlobals(LoadRCWAExamples().MatthewsLeigh);` in order to assign the global variables defined in this section.

7.10 Orders of commutators

We enter some wild rcwa permutation:

Example

```
gap> u := RcwaMapping([[3,0,5],[9,1,5],[3,-1,5],[9,-2,5],[9,4,5]]);;
gap> IsTame(u);;
gap> Display(u);

Wild rcwa permutation of Z with modulus 5

        /
        | 3n/5     if n in 0(5)
        | (9n+1)/5 if n in 1(5)
 n |-> <  (3n-1)/5 if n in 2(5)
        | (9n-2)/5 if n in 3(5)
        | (9n+4)/5 if n in 4(5)
        \
```

We would like to compute the order of $[u, n \mapsto n+k]$ and $[u^2, n \mapsto n+k]$ for different values of k:

Example

```
gap> nu := ClassShift(0,1);; # n -> n + 1
gap> l := Filtered([0..100],k->IsTame(Comm(u,nu^k)));
[ 0, 2, 3, 5, 6, 9, 10, 12, 13, 15, 17, 18, 20, 21, 24, 25, 27, 28, 30,
  32, 33, 35, 36, 39, 40, 42, 43, 45, 47, 48, 50, 51, 54, 55, 57, 58,
  60, 62, 63, 65, 66, 69, 70, 72, 73, 75, 77, 78, 80, 81, 84, 85, 87,
  88, 90, 92, 93, 95, 96, 99, 100 ]
gap> List(l,k->Order(Comm(u,nu^k)));
[ 1, 6, 5, 3, 5, 5, 3, infinity, 7, infinity, 7, 5, 3, infinity,
  infinity, 3, 5, 7, infinity, 7, infinity, 3, 5, 5, 3, 5, infinity,
  infinity, infinity, 5, 3, 5, 5, 3, infinity, 7, infinity, 7, 5, 3,
  infinity, infinity, 3, 5, 7, infinity, 7, infinity, 3, 5, 5, 3, 5,
  infinity, infinity, infinity, 5, 3, 5, 5, 3 ]
gap> u2 := u^2;
<wild rcwa permutation of Z with modulus 25>
gap> Filtered([1..16],k->IsTame(Comm(u2,nu^k))); # k<15->[u^2,nu^k] wild!
[ 15 ]
gap> Order(Comm(u2,nu^15));
infinity
gap> u2nu17 := Comm(u2,nu^17);
<rcwa permutation of Z with modulus 81>
gap> cycs := ShortCycles(u2nu17,[-100..100],100);;
gap> List(cycs,Length);
[ 72, 73, 72, 72, 72, 73, 72, 72, 73, 72, 72, 73, 72, 72, 73, 72, 72,
  73, 72, 72, 73, 72, 72 ]
gap> Lcm(last);
5256
```

```
gap> u2nu17^5256; # This element has indeed order 2^3*3^2*73 = 5256.
IdentityMapping( Integers )
gap> u2nu18 := Comm(u2,nu^18);
<rcwa permutation of Z with modulus 81>
gap> cycs := ShortCycles(u2nu18,[-100..100],100);;
gap> List(cycs,Length);
[ 21, 22, 22, 22, 21, 22, 22, 21, 22, 22, 21, 22, 21, 22, 22, 21, 22,
  22, 21, 22, 22, 21, 22 ]
gap> Lcm(last);
462
gap> u2nu18^462; # This is an element of order 2*3*7*11 = 462.
IdentityMapping( Integers )
gap> List([Comm(u2,nu^20),Comm(u2,nu^25),Comm(u2,nu^30)],Order);
[ 29, 9, 15 ]
```

We observe that our commutators have various different orders, and that the prime factors of these orders are not all “very small”.

Enter `AssignGlobals(LoadRCWAExamples().CollatzlikePerms);` in order to assign the global variables defined in this section.

7.11 An infinite subgroup of CT(GF(2)[x]) with many torsion elements

In this section, we have a look at the following subgroup of CT(GF(2)[x]):

Example

```
gap> x := Indeterminate(GF(2));; SetName(x,"x");
gap> R := PolynomialRing(GF(2),1);
GF(2)[x]
gap> a := ClassTransposition(0,x,1,x);;
gap> b := ClassTransposition(0,x^2+1,1,x^2+1);;
gap> c := ClassTransposition(1,x,0,x^2+x);;
gap> G := Group(a,b,c);
<rcwa group over GF(2)[x] with 3 generators>
gap> Display(G);

Rcwa group over GF(2)[x], generated by

[
Rcwa permutation of GF(2)[x]: P -> P + Z(2)^0

Rcwa permutation of GF(2)[x] with modulus x^2+1, of order 2

        /
        | P + 1 if P in 0(x^2+1) U 1(x^2+1)
 P |-> <  P     if P in x(x^2+1) U x+1(x^2+1)
        |
        \


Rcwa permutation of GF(2)[x] with modulus x^2+x, of order 2
```

```
         /
         | (x+1)*P + x+1   if P in 1(x)
 P |-> <  (P + x+1)/(x+1) if P in 0(x^2+x)
         | P               if P in x(x^2+x)
         \

]
```

We can easily find 2 normal subgroups of G:

Example
```
gap> N1 := Subgroup(G,[a*b,a*c]);
<rcwa group over GF(2)[x] with 2 generators>
gap> IsNormal(G,N1);
true
gap> Index(G,N1);
2
gap> G/N1;
Group([ (1,2), (1,2), (1,2) ])
gap> N2 := Subgroup(G,[a*b*c,a*c]);;
gap> IsNormal(G,N2);
true
gap> IsSubgroup(N1,N2);
false
```

Products of even numbers of generators of G may have infinite order. For example, we have

Example
```
gap> Order(a*b);
2
gap> Order(a*c);
infinity
gap> Order(b*c);
infinity
```

We would like to have a look at orders of products of odd numbers of generators. In order to restrict our considerations to “essentially different” products (as far as we can easily do this), we use the following auxiliary function:

GAP code
```
NormedWords := function ( F, lng )

  local  words, gens, tuples, w;

  gens   := GeneratorsOfGroup(F);
  tuples := EnumeratorOfTuples([1..3],lng);
  words  := [];
```

```
  for w in tuples do
    if    (w[1] = 1 or not 1 in w)
      and PositionSublist(w,[1,1]) = fail
      and PositionSublist(w,[2,2]) = fail
      and PositionSublist(w,[3,3]) = fail
      and PositionSublist(w,[2,1]) = fail
      and w[1] < w[lng]
      and w{[1,lng]} <> [1,2]
      and (w{[1..3]} = [1,2,3] or PositionSublist(w,[1,2,3]) = fail)
    then Add(words,w); fi;
  od;

  words := List(words,word->Product(List(word,i->gens[i])));
  return words;
end;
```

Now let's compute the possible orders of products of 3, 5, 7 or 9 generators:

Example

```
gap> F := FreeGroup("a","b","c");;
gap> phi := EpimorphismByGenerators(F,G);
[ a, b, c ] ->
[ ClassTransposition(0,x,1,x), ClassTransposition(0,x^2+1,1,x^2+1),
  ClassTransposition(1,x,0,x^2+x) ]
gap> B3 := NormedWords(F,3);
[ a*b*c ]
gap> B3 := List(B3,g->g^phi);
[ <rcwa permutation of GF(2)[x] with modulus x^3+x> ]
gap> List(B3,Order);
[ 20 ]
gap> B5 := NormedWords(F,5);
[ a*b*c*a*c, a*b*c*b*c ]
gap> B5 := List(B5,g->g^phi);
[ <rcwa permutation of GF(2)[x] with modulus x^3+x>,
  <rcwa permutation of GF(2)[x] with modulus x^4+x^3+x^2+x> ]
gap> List(B5,Order);
[ 12, 12 ]
gap> B7 := NormedWords(F,7);
[ a*b*c*a*c*a*c, a*b*c*a*c*b*c, a*b*c*b*c*a*c, a*b*c*b*c*b*c ]
gap> B7 := List(B7,g->g^phi);
[ <rcwa permutation of GF(2)[x] with modulus x^4+x^3+x^2+x>,
  <rcwa permutation of GF(2)[x] with modulus x^5+x>,
  <rcwa permutation of GF(2)[x] with modulus x^4+x^3+x^2+x>,
  <rcwa permutation of GF(2)[x] with modulus x^5+x> ]
gap> List(B7,Order);
[ 12, 12, 12, 30 ]
gap> B9 := NormedWords(F,9);
[ a*b*c*a*b*c*a*b*c, a*b*c*a*c*a*c*a*c, a*b*c*a*c*a*c*b*c, a*b*c*a*c*b*c*a*c,
  a*b*c*a*c*b*c*b*c, a*b*c*b*c*a*c*a*c, a*b*c*b*c*a*c*b*c, a*b*c*b*c*b*c*a*c,
  a*b*c*b*c*b*c*b*c ]
gap> B9 := List(B9,g->g^phi);;
gap> List(B9,Order);
```

```
[ 20, 4, 30, 12, 42, 30, 4, 42, 12 ]
```

Enter `AssignGlobals(LoadRCWAExamples().OddNumberOfGens_FiniteOrder);` in order to assign the global variables defined in this section.

7.12 An abelian rcwa group over a polynomial ring

We enter a 2-generated abelian wild rcwa group over GF(4)[x]:

Example

```
gap> x := Indeterminate(GF(4),1);; SetName(x,"x");
gap> R := PolynomialRing(GF(4),1);
GF(2^2)[x]
gap> e := One(GF(4));;
gap> p := x^2 + x + e;;    q := x^2 + e;;
gap> r := x^2 + x + Z(4);; s := x^2 + x + Z(4)^2;;
gap> cg := List( AllResidues(R,x^2), pol -> [ p, p * pol mod q, q ] );;
gap> ch := List( AllResidues(R,x^2), pol -> [ r, r * pol mod s, s ] );;
gap> g := RcwaMapping( R, q, cg );
<rcwa mapping of GF(2^2)[x] with modulus x^2+1>
gap> h := RcwaMapping( R, s, ch );
<rcwa mapping of GF(2^2)[x] with modulus x^2+x+Z(2^2)^2>
gap> List([g,h],IsTame);
[ false, false ]
gap> G := Group(g,h);
<rcwa group over GF(2^2)[x] with 2 generators>
gap> IsAbelian(G);
true
gap> IsTame(G);
false
```

It is easy to see that all orbits on GF(4)[x] under the action of `G` are finite.

Now we compute the action of the group `G` on one of its orbits, and make some statistics of the orbits of `G` containing polynomials of degree less than 4:

Example

```
gap> orb := Orbit(G,x^5);
[ x^5, x^5+x^4+x^2+1, x^5+x^3+x^2+Z(2^2)*x+Z(2)^0, x^5+x^3,
  x^5+x^4+x^3+x^2+Z(2^2)^2*x+Z(2^2)^2, x^5+x, x^5+x^4+x^3,
  x^5+x^2+Z(2^2)^2*x, x^5+x^4+x^2+x, x^5+x^3+x^2+Z(2^2)^2*x+Z(2)^0,
  x^5+x^4+Z(2^2)*x+Z(2^2), x^5+x^3+x, x^5+x^4+x^3+x^2+Z(2^2)*x+Z(2^2),
  x^5+x^4+x^3+x+1, x^5+x^2+Z(2^2)*x, x^5+x^4+Z(2^2)^2*x+Z(2^2)^2 ]
gap> H := Action(G,orb);
Group([ (1,2,4,7,6,9,12,14)(3,5,8,11,10,13,15,16),
  (1,3,6,10)(2,5,9,13)(4,8,12,15)(7,11,14,16) ])
gap> IsAbelian(H); # check ...
true
gap> IsCyclic(H);  # H, and therefore also G, is not cyclic
false
```

```
gap> Exponent(H);
8
gap> Collected(List(ShortOrbits(G,AllResidues(R,x^4),100),Length));
[ [ 1, 4 ], [ 2, 6 ], [ 4, 12 ], [ 8, 24 ] ]
```

Changing the generators a little changes the structure of the group and its action on the underlying ring a lot:

Example

```
gap> cg[1][2] := cg[1][2] + (x^2 + e) * p * q;;
gap> ch[7][2] := ch[7][2] + x * r * s;;
gap> g := RcwaMapping( R, q, cg );; h := RcwaMapping( R, s, ch );;
gap> G := Group(g,h);
<rcwa group over GF(2^2)[x] with 2 generators>
gap> IsAbelian(G);
false
gap> Support(G);
GF(2^2)[x] \ [ 1, Z(2^2), Z(2^2)^2 ]
gap> orb := Orbit(G,Zero(R));;
gap> Length(orb);
87
gap> StructureDescription(Action(G,orb));
"A87"
gap> Collected(List(orb,DegreeOfLaurentPolynomial));
[ [ -infinity, 1 ], [ 1, 2 ], [ 2, 4 ], [ 3, 16 ], [ 4, 64 ] ]
gap> S := AllResidues(R,x^6);;
gap> orbs := ShortOrbits(G,S,-1:finite);;
gap> List(orbs,Length);
[ 87, 1, 1, 1, 2, 2, 2, 2, 2, 4, 4, 4, 20, 4, 12, 4, 20, 4, 4, 12, 8, 8,
  48, 48, 16, 8, 8, 56, 8, 88, 8, 8, 8, 400, 16, 48, 16, 16, 16, 80, 16,
  16, 16, 96, 32, 192, 32, 16, 16, 416, 16, 48, 16, 16, 880, 16, 16, 16,
  16, 16, 16, 16, 16, 16, 848, 16, 16, 32, 16, 16, 16, 16, 16, 16, 16 ]
gap> Position(last,880);
55
gap> Set(orbs[55],DegreeOfLaurentPolynomial); # all elm's have same degree
[ 5 ]
gap> H := Action(G,orbs[55]);;
gap> IsPrimitive(H,MovedPoints(H));
false
gap> List(Blocks(H,MovedPoints(H)),Length);
[ 110, 110, 110, 110, 110, 110, 110, 110 ]
```

Enter `AssignGlobals(LoadRCWAExamples().AbelianGroupOverPolynomialRing);` in order to assign the global variables defined in this section.

7.13 Checking for solvability

Presently there is no general method available for testing wild rcwa groups for solvability. However, sometimes the question for solvability can be answered anyway. In the example below, the idea is to find a subgroup U which acts on a finite set S of integers, and which induces on S a non-solvable finite permutation group:

Example

```
gap> a := RcwaMapping([[3,0,2],[3, 1,4],[3,0,2],[3,-1,4]]);;
gap> b := RcwaMapping([[3,0,2],[3,13,4],[3,0,2],[3,-1,4]]);;
gap> G := Group(a,b);;
gap> ShortOrbits(Group(Comm(a,b)),[-10..10],100);
[ [ -10 ], [ -9 ], [ -30, -21, -14, -13, -11, -8 ], [ -7 ], [ -6 ],
  [ -12, -5, -4, -3, -2, 1 ], [ -1 ], [ 0 ], [ 2 ], [ 3 ],
  [ 4, 5, 6, 7, 10, 15 ], [ 8 ], [ 9 ] ]
gap> S := [ 4, 5, 6, 7, 10, 15 ];;
gap> Cycle(Comm(a,b),4);
[ 4, 7, 10, 15, 5, 6 ]
gap> elm := RepresentativeAction(G,S,Permuted(S,(1,4)),OnTuples);
<rcwa permutation of Z with modulus 81>
gap> List(S,n->n^elm);
[ 7, 5, 6, 4, 10, 15 ]
gap> U := Group(Comm(a,b),elm);
<rcwa group over Z with 2 generators>
gap> Action(U,S);
Group([ (1,4,5,6,2,3), (1,4) ])
gap> IsNaturalSymmetricGroup(last);
true
```

Thus the subgroup U induces on S a natural symmetric group of degree 6. Therefore the group G is not solvable. We conclude this example by factoring the group element `elm` into generators:

Example

```
gap> F := FreeGroup("a","b");
<free group on the generators [ a, b ]>
gap> RepresentativeActionPreImage(G,S,Permuted(S,(1,4)),OnTuples,F);
a^-2*b^-2*a*b*a^-1*b*a*b^-2*a
gap> a^-2*b^-2*a*b*a^-1*b*a*b^-2*a = elm;
true
```

Enter `AssignGlobals(LoadRCWAExamples().CheckingForSolvability);` in order to assign the global variables defined in this section.

7.14 Some examples over (semi)localizations of the integers

We start with something one can observe when trying to “transfer” an rcwa mapping from the ring of integers to one of its localizations:

Example

```
gap> a := RcwaMapping([[3,0,2],[3,1,4],[3,0,2],[3,-1,4]]);;
gap> IsBijective(a);
true
gap> a2 := LocalizedRcwaMapping(a,2);
```

```
<rcwa mapping of Z_( 2 ) with modulus 4>
gap> IsSurjective(a2); # As expected
true
gap> IsInjective(a2); # Why not??
false
gap> 0^a2;
0
gap> (1/3)^a2; # That's the reason!
0
```

The above can also be explained easily by pointing out that the modulus of the inverse of a is 3, and that 3 is a unit of $\mathbb{Z}_{(2)}$. Moving to $\mathbb{Z}_{(2,3)}$ solves this problem:

Example

```
gap> a23 := SemilocalizedRcwaMapping(a,[2,3]);
<rcwa mapping of Z_( 2, 3 ) with modulus 4>
gap> IsBijective(a23);
true
```

We get additional finite cycles, e.g.:

Example

```
gap> List(ShortOrbits(Group(a23),[0..50]/5,50),orb->Cycle(a23,orb[1]));
[ [ 0 ], [ 1/5, 2/5, 3/5 ],
  [ 4/5, 6/5, 9/5, 8/5, 12/5, 18/5, 27/5, 19/5, 13/5, 11/5, 7/5 ],
  [ 1 ], [ 2, 3 ], [ 14/5, 21/5, 17/5 ],
  [ 16/5, 24/5, 36/5, 54/5, 81/5, 62/5, 93/5, 71/5, 52/5, 78/5, 117/5,
      89/5, 68/5, 102/5, 153/5, 116/5, 174/5, 261/5, 197/5, 149/5,
      113/5, 86/5, 129/5, 98/5, 147/5, 109/5, 83/5, 61/5, 47/5, 34/5,
      51/5, 37/5, 29/5, 23/5 ], [ 4, 6, 9, 7, 5 ] ]
gap> List(last,Length);
[ 1, 3, 11, 1, 2, 3, 34, 5 ]
gap> List(ShortOrbits(Group(a23),[0..50]/7,50),orb->Cycle(a23,orb[1]));
[ [ 0 ], [ -1/7, 1/7 ], [ 2/7, 3/7, 4/7, 6/7, 9/7, 5/7 ], [ 1 ],
  [ 2, 3 ], [ 4, 6, 9, 7, 5 ] ]
gap> List(last,Length);
[ 1, 2, 6, 1, 2, 5 ]
```

However the structure of a group with prime set $\mathbb{P}$ remains invariant under the "transfer" from $\mathbb{Z}$ to $\mathbb{Z}_{(\mathbb{P})}$.

"Transferring" a non-invertible rcwa mapping from the ring of integers to some of its (semi)localizations can also turn it into an invertible one:

Example

```
gap> v := RcwaMapping([[6,0,1],[1,-7,2],[6,0,1],[1,-1,1],
>                      [6,0,1],[1, 1,2],[6,0,1],[1,-1,1]]);;
gap> Display(v);
```

```
Rcwa mapping of Z with modulus 8

        /
        | 6n      if n in 0(2)
        | n-1     if n in 3(4)
 n |-> <  (n-7)/2 if n in 1(8)
        | (n+1)/2 if n in 5(8)
        |
        \

gap> IsInjective(v);
true
gap> IsSurjective(v);
false
gap> Image(v);
Z \ 4(12) U 8(12)
gap> Difference(Integers,last);
4(12) U 8(12)
gap> v2 := LocalizedRcwaMapping(v,2);
<rcwa mapping of Z_( 2 ) with modulus 8>
gap> IsBijective(v2);
true
gap> Display(v2^-1);

Rcwa permutation of Z_( 2 ) with modulus 4

        /
        | 1/3 n / 2 if n in 0(4)
        | 2 n + 7   if n in 1(4)
 n |-> <  n + 1     if n in 2(4)
        | 2 n - 1   if n in 3(4)
        |
        \

gap> S := ResidueClass(Z_pi(2),2,0);; l := [S];;
gap> for i in [1..10] do Add(l,l[Length(l)]^v2); od;
gap> l; # Visibly v2 is wild ...
[ 0(2), 0(4), 0(8), 0(16), 0(32), 0(64), 0(128), 0(256), 0(512),
  0(1024), 0(2048) ]
gap> w2 := RcwaMapping(Z_pi(2),[[1,0,2],[2,-1,1],[1,1,1],[2,-1,1]]);;
gap> v2w2 := Comm(v2,w2);; v2w2^-1;;
gap> Display(v2w2);

Rcwa permutation of Z_( 2 ) with modulus 8

        /
        | 3 n   if n in 2(4)
        | n + 4 if n in 1(8)
 n |-> <  n - 4 if n in 5(8)
        | n     if n in 0(4) U 3(4)
        |
```

```
          \
```

Again, viewed as an rcwa mapping of the integers the commutator given at the end of the example would not be surjective.

Enter `AssignGlobals(LoadRCWAExamples().Semilocals);` in order to assign the global variables defined in this section.

7.15 Twisting 257-cycles into an rcwa mapping with modulus 32

We define an rcwa mapping x of order 257 with modulus 32. The easiest way to construct such a mapping is to prescribe a transition graph and then to assign suitable affine mappings to its vertices.

Example

```
gap> x_257 := RcwaMapping(
>      [[ 16,  2,  1], [ 16, 18,  1], [  1, 16,  1], [ 16, 18,  1],
>       [  1, 16,  1], [ 16, 18,  1], [  1, 16,  1], [ 16, 18,  1],
>       [  1, 16,  1], [ 16, 18,  1], [  1, 16,  1], [ 16, 18,  1],
>       [  1, 16,  1], [ 16, 18,  1], [  1, 16,  1], [ 16, 18,  1],
>       [  1,  0, 16], [ 16, 18,  1], [  1,-14,  1], [ 16, 18,  1],
>       [  1,-14,  1], [ 16, 18,  1], [  1,-14,  1], [ 16, 18,  1],
>       [  1,-14,  1], [ 16, 18,  1], [  1,-14,  1], [ 16, 18,  1],
>       [  1,-14,  1], [ 16, 18,  1], [  1,-14,  1], [  1,-31,  1]]);;
gap> Order(x_257);; Display(x_257:CycleNotation:=false);

Rcwa permutation of Z with modulus 32, of order 257

        /
        | 16n+18 if n in 1(2) \ 31(32)
        | n+16   if n in 2(32) U 4(32) U 6(32) U 8(32) U 10(32) U
        |                   12(32) U 14(32)
        | n-14   if n in 18(32) U 20(32) U 22(32) U 24(32) U 26(32) U
 n |-> <                    28(32) U 30(32)
        | 16n+2  if n in 0(32)
        | n/16   if n in 16(32)
        | n-31   if n in 31(32)
        |
        \

gap> Display(x_257);

Rcwa permutation of Z with modulus 32, of order 257

( 0(32), 2(512), 18(512), 4(512), 20(512), 6(512), 22(512),
  8(512), 24(512), 10(512), 26(512), 12(512), 28(512), 14(512),
  30(512), 16(512), 1(32), 34(512), 50(512), 36(512), 52(512),
  38(512), 54(512), 40(512), 56(512), 42(512), 58(512), 44(512),
  60(512), 46(512), 62(512), 48(512), 3(32), 66(512), 82(512),
  68(512), 84(512), 70(512), 86(512), 72(512), 88(512), 74(512),
  90(512), 76(512), 92(512), 78(512), 94(512), 80(512), 5(32),
  98(512), 114(512), 100(512), 116(512), 102(512), 118(512),
```

```
  104(512), 120(512), 106(512), 122(512), 108(512), 124(512),
  110(512), 126(512), 112(512), 7(32), 130(512), 146(512),
  132(512), 148(512), 134(512), 150(512), 136(512), 152(512),
  138(512), 154(512), 140(512), 156(512), 142(512), 158(512),
  144(512), 9(32), 162(512), 178(512), 164(512), 180(512),
  166(512), 182(512), 168(512), 184(512), 170(512), 186(512),
  172(512), 188(512), 174(512), 190(512), 176(512), 11(32),
  194(512), 210(512), 196(512), 212(512), 198(512), 214(512),
  200(512), 216(512), 202(512), 218(512), 204(512), 220(512),
  206(512), 222(512), 208(512), 13(32), 226(512), 242(512),
  228(512), 244(512), 230(512), 246(512), 232(512), 248(512),
  234(512), 250(512), 236(512), 252(512), 238(512), 254(512),
  240(512), 15(32), 258(512), 274(512), 260(512), 276(512),
  262(512), 278(512), 264(512), 280(512), 266(512), 282(512),
  268(512), 284(512), 270(512), 286(512), 272(512), 17(32),
  290(512), 306(512), 292(512), 308(512), 294(512), 310(512),
  296(512), 312(512), 298(512), 314(512), 300(512), 316(512),
  302(512), 318(512), 304(512), 19(32), 322(512), 338(512),
  324(512), 340(512), 326(512), 342(512), 328(512), 344(512),
  330(512), 346(512), 332(512), 348(512), 334(512), 350(512),
  336(512), 21(32), 354(512), 370(512), 356(512), 372(512),
  358(512), 374(512), 360(512), 376(512), 362(512), 378(512),
  364(512), 380(512), 366(512), 382(512), 368(512), 23(32),
  386(512), 402(512), 388(512), 404(512), 390(512), 406(512),
  392(512), 408(512), 394(512), 410(512), 396(512), 412(512),
  398(512), 414(512), 400(512), 25(32), 418(512), 434(512),
  420(512), 436(512), 422(512), 438(512), 424(512), 440(512),
  426(512), 442(512), 428(512), 444(512), 430(512), 446(512),
  432(512), 27(32), 450(512), 466(512), 452(512), 468(512),
  454(512), 470(512), 456(512), 472(512), 458(512), 474(512),
  460(512), 476(512), 462(512), 478(512), 464(512), 29(32),
  482(512), 498(512), 484(512), 500(512), 486(512), 502(512),
  488(512), 504(512), 490(512), 506(512), 492(512), 508(512),
  494(512), 510(512), 496(512), 31(32) )

gap> Length(Cycle(x_257,0));
257
```

Enter `AssignGlobals(LoadRCWAExamples().LongCyclesOfPrimeLength);` in order to assign the global variables defined in this section.

7.16 The behaviour of the moduli of powers

We give some examples of how the series of the moduli of powers of a given rcwa mapping of the integers can look like.

Example

```
gap> a := RcwaMapping([[3,0,2],[3, 1,4],[3,0,2],[3,-1,4]]);;
gap> List([0..4],i->Modulus(a^i));
[ 1, 4, 16, 64, 256 ]
```

```
gap> e1 := RcwaMapping([[1,4,1],[2,0,1],[1,0,2],[2,0,1]]);;
gap> e2 := RcwaMapping([[1,4,1],[2,0,1],[1,0,2],[1,0,1],
>                       [1,4,1],[2,0,1],[1,0,1],[1,0,1]]);;
gap> List([e1,e2],Order);
[ infinity, infinity ]
gap> List([1..20],i->Modulus(e1^i));
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
gap> List([1..20],i->Modulus(e2^i));
[ 8, 4, 8, 4, 8, 4, 8, 4, 8, 4, 8, 4, 8, 4, 8, 4, 8, 4, 8, 4 ]
gap> Display(e2);

Rcwa permutation of Z with modulus 8, of order infinity

        /
        | n+4 if n in 0(4)
        | 2n  if n in 1(4)
 n |-> <  n/2 if n in 2(8)
        | n   if n in 3(4) U 6(8)
        |
        \

gap> e2^2 = Restriction(RcwaMapping([[1,2,1]]),RcwaMapping([[4,0,1]]));
true
gap> g:=RcwaMapping([[2,2,1],[1, 4,1],[1,0,2],[2,2,1],[1,-4,1],[1,-2,1]]);;
gap> h:=RcwaMapping([[2,2,1],[1,-2,1],[1,0,2],[2,2,1],[1,-1,1],[1, 1,1]]);;
gap> List([0..7],i->Modulus(g^i));
[ 1, 6, 12, 12, 12, 12, 6, 1 ]
gap> List([1..18],i->Modulus((g^3*h)^i));
[ 12, 6, 12, 12, 12, 6, 12, 6, 12, 12, 12, 6, 12, 6, 12, 12, 12, 6 ]
gap> u := RcwaMapping([[3,0,5],[9,1,5],[3,-1,5],[9,-2,5],[9,4,5]]);;
gap> List([0..3],i->Modulus(u^i));
[ 1, 5, 25, 125 ]
gap> v6 := RcwaMapping([[-1,2,1],[1,-1,1],[1,-1,1]]);;
gap> List([0..6],i->Modulus(v6^i));
[ 1, 3, 3, 3, 3, 3, 1 ]
gap> w8 := RcwaMapping([[-1,3,1],[1,-1,1],[1,-1,1],[1,-1,1]]);;
gap> List([0..8],i->Modulus(w8^i));
[ 1, 4, 4, 4, 4, 4, 4, 4, 1 ]
gap> z := RcwaMapping([[2,1,1],[1, 1,1],[2,-1,1],[2, -2,1],
>                      [1,6,2],[1, 1,1],[1,-6,2],[2,  5,1],
>                      [1,6,2],[1, 1,1],[1, 1,1],[2, -5,1],
>                      [1,0,1],[1,-4,1],[1, 0,1],[2,-10,1]]);;
gap> IsBijective(z);
true
gap> List([0..25],i->Modulus(z^i));
[ 1, 16, 32, 64, 64, 128, 128, 128, 128, 128, 128, 256, 256, 256, 256,
  256, 256, 512, 512, 512, 512, 512, 512, 1024, 1024, 1024 ]
```

Enter `AssignGlobals(LoadRCWAExamples().ModuliOfPowers);` in order to assign the global variables defined in this section.

7.17 Images and preimages under the Collatz mapping

We have a look at the images of the residue class 1(2) under powers of the Collatz mapping.

Example

```
gap> T := RcwaMapping([[1,0,2],[3,1,2]]);;
gap> S0 := ResidueClass(Integers,2,1);;
gap> S1 := S0^T;
2(3)
gap> S2 := S1^T;
1(3) U 8(9)
gap> S3 := S2^T;
2(3) U 4(9)
gap> S4 := S3^T;
Z \ 0(3) U 5(9)
gap> S5 := S4^T;
Z \ 0(3) U 7(9)
gap> S6 := S5^T;
Z \ 0(3)
gap> S7 := S6^T;
Z \ 0(3)
```

Thus the image gets stable after applying the mapping T for the 6th time. Hence T^6 maps the residue class 1(2) surjectively onto the union of the residue classes 1(3) and 2(3), which T stabilizes setwise. Now we would like to determine the preimages of 1(3) and 2(3) in 1(2) under T^6. The residue class 1(2) has to be the disjoint union of these sets.

Example

```
gap> U := Intersection(PreImage(T^6,ResidueClass(Integers,3,1)),S0);
<union of 11 residue classes (mod 64)>
gap> V := Intersection(PreImage(T^6,ResidueClass(Integers,3,2)),S0);
<union of 21 residue classes (mod 64)>
gap> AsUnionOfFewClasses(U);
[ 1(64), 5(64), 7(64), 9(64), 21(64), 23(64), 29(64), 31(64), 49(64),
  51(64), 59(64) ]
gap> AsUnionOfFewClasses(V);
[ 3(32), 11(32), 13(32), 15(32), 25(32), 17(64), 19(64), 27(64), 33(64),
  37(64), 39(64), 41(64), 53(64), 55(64), 61(64), 63(64) ]
gap> Union(U,V) = S0 and Intersection(U,V) = [];  # consistency check
true
```

The images of the residue class 0(3) under powers of T look as follows:

Example

```
gap> S0 := ResidueClass(Integers,3,0);
0(3)
gap> S1 := S0^T;
0(3) U 5(9)
gap> S2 := S1^T;
0(3) U 5(9) U 7(9) U 8(27)
```

```
gap> S3 := S2^T;
<union of 20 residue classes (mod 27) (6 classes)>
gap> S4 := S3^T;
<union of 73 residue classes (mod 81)>
gap> S5 := S4^T;
Z \ 10(81) U 37(81)
gap> S6 := S5^T;
Integers
gap> S7 := S6^T;
Integers
```

Thus every integer is the image of a multiple of 3 under T^6. This means that it would be sufficient to prove the $3n+1$ conjecture for multiples of 3. We can obtain the corresponding result for multiples of 5 as follows:

Example

```
gap> S := [ResidueClass(Integers,5,0)];
[ 0(5) ]
gap> for i in [1..12] do Add(S,S[i]^T); od;
gap> for s in S do View(s); Print("\n"); od;
0(5)
0(5) U 8(15)
0(5) U 4(15) U 8(15)
0(5) U 2(15) U 4(15) U 8(15) U 29(45)
<union of 73 residue classes (mod 135)>
<union of 244 residue classes (mod 405)>
<union of 784 residue classes (mod 1215)>
<union of 824 residue classes (mod 1215)>
<union of 2593 residue classes (mod 3645)>
<union of 2647 residue classes (mod 3645)>
<union of 2665 residue classes (mod 3645)>
<union of 2671 residue classes (mod 3645)>
1(3) U 2(3) U 0(15)
gap> Union(S[13],ResidueClass(Integers,3,0));
Integers
gap>  List(S,Si->Float(Density(Si)));
[ 0.2, 0.266667, 0.333333, 0.422222, 0.540741, 0.602469, 0.645267,
  0.678189, 0.711385, 0.7262, 0.731139, 0.732785, 0.733333 ]
```

Enter `AssignGlobals(LoadRCWAExamples().CollatzMapping);` in order to assign the global variables defined in this section.

7.18 An extension of the Collatz mapping T to a permutation of $\mathbb{Z}^2$

The Collatz mapping T is surjective, but not injective:

Example

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

```
Rcwa mapping of Z with modulus 2

        /
        | n/2       if n in 0(2)
 n |-> <  (3n+1)/2 if n in 1(2)
        |
        \

gap> IsInjective(T); IsSurjective(T);
false
true
gap> PreImages(T,2);
[ 1, 4 ]
```

Often, dealing with rcwa permutations is easier. Indeed the Collatz mapping T can be extended in natural ways to permutations of $\mathbb{Z}^2$. For example, the following permutation acts on the second coordinate just like T:

Example

```
gap> Sigma_T := RcwaMapping( Integers^2, [[1,0],[0,6]],
>                            [[[[2,0],[0,1]],[0,0],2],
>                             [[[4,0],[0,3]],[2,1],2],
>                             [[[2,0],[0,1]],[0,0],2],
>                             [[[4,0],[0,3]],[2,1],2],
>                             [[[4,0],[0,1]],[0,0],2],
>                             [[[4,0],[0,3]],[2,1],2]] );
<rcwa mapping of Z^2 with modulus (1,0)Z+(0,6)Z>
gap> IsBijective(Sigma_T);
true
gap> Display(Sigma_T);

Rcwa permutation of Z^2 with modulus (1,0)Z+(0,6)Z

            /
            | (2m+1,(3n+1)/2) if (m,n) in (0,1)+(1,0)Z+(0,2)Z
            | (m,n/2)         if (m,n) in (0,0)+(1,0)Z+(0,6)Z U
 (m,n) |-> <                              (0,2)+(1,0)Z+(0,6)Z
            | (2m,n/2)        if (m,n) in (0,4)+(1,0)Z+(0,6)Z
            |
            \

gap> Display(Sigma_T^-1);

Rcwa permutation of Z^2 with modulus (2,0)Z+(0,3)Z

            /
            | (m,2n)             if (m,n) in (0,0)+(1,0)Z+(0,3)Z U
            |                                (0,1)+(1,0)Z+(0,3)Z
 (m,n) |-> <  (m/2,2n)           if (m,n) in (0,2)+(2,0)Z+(0,3)Z
            | ((m-1)/2,(2n-1)/3) if (m,n) in (1,2)+(2,0)Z+(0,3)Z
            |
```

```
        \
```

Now, the $3n+1$ conjecture is equivalent to the assertion that the line $n = 4$ is a set of representatives for the cycles of Sigma_T on the half plane $n > 0$.

Let's have a look at a part of a cycle of Sigma_T:

Example

```
gap> Trajectory(Sigma_T,[0,27],75);
[ [ 0, 27 ], [ 1, 41 ], [ 3, 62 ], [ 3, 31 ], [ 7, 47 ], [ 15, 71 ],
  [ 31, 107 ], [ 63, 161 ], [ 127, 242 ], [ 127, 121 ], [ 255, 182 ],
  [ 255, 91 ], [ 511, 137 ], [ 1023, 206 ], [ 1023, 103 ],
  [ 2047, 155 ], [ 4095, 233 ], [ 8191, 350 ], [ 8191, 175 ],
  [ 16383, 263 ], [ 32767, 395 ], [ 65535, 593 ], [ 131071, 890 ],
  [ 131071, 445 ], [ 262143, 668 ], [ 262143, 334 ], [ 524286, 167 ],
  [ 1048573, 251 ], [ 2097147, 377 ], [ 4194295, 566 ], [ 4194295, 283 ],
  [ 8388591, 425 ], [ 16777183, 638 ], [ 16777183, 319 ],
  [ 33554367, 479 ], [ 67108735, 719 ], [ 134217471, 1079 ],
  [ 268434943, 1619 ], [ 536869887, 2429 ], [ 1073739775, 3644 ],
  [ 1073739775, 1822 ], [ 2147479550, 911 ], [ 4294959101, 1367 ],
  [ 8589918203, 2051 ], [ 17179836407, 3077 ], [ 34359672815, 4616 ],
  [ 34359672815, 2308 ], [ 68719345630, 1154 ], [ 68719345630, 577 ],
  [ 137438691261, 866 ], [ 137438691261, 433 ], [ 274877382523, 650 ],
  [ 274877382523, 325 ], [ 549754765047, 488 ], [ 549754765047, 244 ],
  [ 1099509530094, 122 ], [ 1099509530094, 61 ], [ 2199019060189, 92 ],
  [ 2199019060189, 46 ], [ 4398038120378, 23 ], [ 8796076240757, 35 ],
  [ 17592152481515, 53 ], [ 35184304963031, 80 ], [ 35184304963031, 40 ],
  [ 70368609926062, 20 ], [ 70368609926062, 10 ], [ 140737219852124, 5 ],
  [ 281474439704249, 8 ], [ 281474439704249, 4 ], [ 562948879408498, 2 ],
  [ 562948879408498, 1 ], [ 1125897758816997, 2 ],
  [ 1125897758816997, 1 ], [ 2251795517633995, 2 ],
  [ 2251795517633995, 1 ] ]
gap> Trajectory(Sigma_T^-1,[0,27],20);
[ [ 0, 27 ], [ 0, 54 ], [ 0, 108 ], [ 0, 216 ], [ 0, 432 ], [ 0, 864 ],
  [ 0, 1728 ], [ 0, 3456 ], [ 0, 6912 ], [ 0, 13824 ], [ 0, 27648 ],
  [ 0, 55296 ], [ 0, 110592 ], [ 0, 221184 ], [ 0, 442368 ],
  [ 0, 884736 ], [ 0, 1769472 ], [ 0, 3538944 ], [ 0, 7077888 ],
  [ 0, 14155776 ] ]
```

While it seems easy to make conjectures regarding the behaviour of cycles of Sigma_T, obtaining results on it is apparently hard. We observe however that Sigma_T can be written as a product of two permutations of $\mathbb{Z}^2$ whose cycles can be described easily:

Example

```
gap> a := RcwaMapping(Integers^2,[[1,0],[0,2]],[[[[4,0],[0,1]],[0, 0],2],
>                                               [[[4,0],[0,1]],[2,-1],2]]);
<rcwa mapping of Z^2 with modulus (1,0)Z+(0,2)Z>
gap> b := a^-1*Sigma_T;
<rcwa permutation of Z^2 with modulus (2,0)Z+(0,3)Z>
gap> Display(a);
```

```
Rcwa permutation of Z^2 with modulus (1,0)Z+(0,2)Z

              /
              | (2m,n/2)       if (m,n) in (0,0)+(1,0)Z+(0,2)Z
 (m,n) |-> <  (2m+1,(n-1)/2) if (m,n) in (0,1)+(1,0)Z+(0,2)Z
              |
              \

gap> Display(b);

Rcwa permutation of Z^2 with modulus (2,0)Z+(0,3)Z

              /
              | (m,3n+2) if (m,n) in (1,0)+(2,0)Z+(0,1)Z
              | (m/2,n)  if (m,n) in (0,0)+(2,0)Z+(0,3)Z U
 (m,n) |-> <                        (0,1)+(2,0)Z+(0,3)Z
              | (m,n)    if (m,n) in (0,2)+(2,0)Z+(0,3)Z
              |
              \
```

It is easy to see that both a and b have infinite order. The cycles of a have roughly hyperbolic shape and run, so to speak, from $(0,\pm\infty)$ to $(\pm\infty,0)$. A given cycle contains only finitely many points both of whose coordinates are nonzero. The fixed points of a are (0,0) and (-1,-1). We have a look at an example of a cycle of a:

Example

```
gap> Trajectory(a,[1000,1000],15);
[ [ 1000, 1000 ], [ 2000, 500 ], [ 4000, 250 ], [ 8000, 125 ],
  [ 16001, 62 ], [ 32002, 31 ], [ 64005, 15 ], [ 128011, 7 ],
  [ 256023, 3 ], [ 512047, 1 ], [ 1024095, 0 ], [ 2048190, 0 ],
  [ 4096380, 0 ], [ 8192760, 0 ], [ 16385520, 0 ] ]
gap> Trajectory(a^-1,[1000,1000],15);
[ [ 1000, 1000 ], [ 500, 2000 ], [ 250, 4000 ], [ 125, 8000 ],
  [ 62, 16001 ], [ 31, 32002 ], [ 15, 64005 ], [ 7, 128011 ],
  [ 3, 256023 ], [ 1, 512047 ], [ 0, 1024095 ], [ 0, 2048190 ],
  [ 0, 4096380 ], [ 0, 8192760 ], [ 0, 16385520 ] ]
```

It is left as an easy exercise to the reader to find out how the cycles of b look like.

Enter `AssignGlobals(LoadRCWAExamples().ZxZ);` in order to assign the global variables defined in this section.

7.19 Finite quotients of Grigorchuk groups

In this section, we show how to construct finite quotients of the two infinite periodic groups introduced by Rostislav Grigorchuk in [Gri80] with the help of RCWA. The first of these, nowadays known as "Grigorchuk group", is investigated in an example given on the GAP website – see `http://www.gap-system.org/Doc/Examples/grigorchuk.html`. The RCWA package permits a simpler and more elegant construction of the finite quotients of this group: The function `TopElement` given on the mentioned webpage gets unnecessary, and the function SequenceElement can be simplified as follows:

```
SequenceElement := function ( r, level )

  return Permutation(Product(Filtered([1..level-1],k->k mod 3 <> r),
                          k->ClassTransposition(    2^(k-1)-1,2^(k+1),
                                                 2^k+2^(k-1)-1,2^(k+1))),
                     [0..2^level-1]);
end;
```

The actual constructors for the generators are modified as follows:

```
a := level -> Permutation(ClassTransposition(0,2,1,2),[0..2^level-1]);
b := level -> SequenceElement(0,level);
c := level -> SequenceElement(2,level);
d := level -> SequenceElement(1,level);
```

All computations given on the webpage can now be done just as with the “original” construction of the quotients of the Grigorchuk group. In the sequel, we construct finite quotients of the second group introduced in [Gri80]:

Example

```
gap> FourCycle := RcwaMapping((4,5,6,7),[4..7]);
( 0(4), 1(4), 2(4), 3(4) )
gap> GrigorchukGroup2Generator := function ( level )
>      if level = 1 then return FourCycle; else
>        return   Restriction(FourCycle, RcwaMapping([[4,1,1]]))
>               * Restriction(FourCycle, RcwaMapping([[4,3,1]]))
>               * Restriction(GrigorchukGroup2Generator(level-1),
>                             RcwaMapping([[4,0,1]]));
>      fi;
>    end;;
gap> GrigorchukGroup2 := level -> Group(FourCycle,
>                                   GrigorchukGroup2Generator(level));;
```

We can do similar things as shown in the example on the GAP webpage for the “first” Grigorchuk group:

Example

```
gap> G := List([1..4],lev->GrigorchukGroup2(lev)); # The first 4 quotients.
[ <rcwa group over Z with 2 generators>,
  <rcwa group over Z with 2 generators>,
  <rcwa group over Z with 2 generators>,
  <rcwa group over Z with 2 generators> ]
gap> H := List([1..4],lev->Action(G[lev],[0..4^lev-1])); # Isom. perm.-gps.
[ Group([ (1,2,3,4), (1,2,3,4) ]),
```

```
  Group([ (1,2,3,4)(5,6,7,8)(9,10,11,12)(13,14,15,16),
      (1,5,9,13)(2,6,10,14)(4,8,12,16) ]),
  <permutation group with 2 generators>,
  <permutation group with 2 generators> ]
gap> List(H,Size);
[ 4, 1024, 4294967296, 1329227995784915872903807060280344576 ]
gap> List(last,n->Collected(Factors(n)));
[ [ [ 2, 2 ] ], [ [ 2, 10 ] ], [ [ 2, 32 ] ], [ [ 2, 120 ] ] ]
gap> List(H,NilpotencyClassOfGroup);
[ 1, 6, 14, 40 ]
```

Enter `AssignGlobals(LoadRCWAExamples().GrigorchukQuotients);` in order to assign the global variables defined in this section.

7.20 Forward orbits of a monoid with 2 generators

The $3n+1$ conjecture asserts that the forward orbit of any positive integer under the Collatz mapping T contains 1. In contrast, it seems likely that "most" trajectories of the two mappings

$$T_5^{\pm} : \mathbb{Z} \longrightarrow \mathbb{Z}, \quad n \longmapsto \begin{cases} \frac{n}{2} & \text{if } n \text{ even}, \\ \frac{5n\pm 1}{2} & \text{if } n \text{ odd} \end{cases}$$

diverge. However we can show by means of computation that the forward orbit of any positive integer under the action of the monoid generated by the two mappings T_5^- and T_5^+ indeed contains 1. First of all, we enter the generators:

Example

```
gap> T5m := RcwaMapping([[1,0,2],[5,-1,2]]);;
gap> T5p := RcwaMapping([[1,0,2],[5, 1,2]]);;
```

We look for a number k such that for any residue class $r(2^k)$ there is a product f of k mappings $T_5^{\pm}$ whose restriction to $r(2^k)$ is given by $n \mapsto (an+b)/c$ where $c > a$:

Example

```
gap> k := 1;;
gap> repeat
>      maps := List(Tuples([T5m,T5p],k),Product);
>      decr := List(maps,DecreasingOn);
>      decreasable := Union(decr);
>      Print(k,": "); View(decreasable); Print("\n");
>      k := k + 1;
>    until decreasable = Integers;
1: 0(2)
2: 0(4)
3: Z \ 1(8) U 7(8)
4: 0(4) U 3(16) U 6(16) U 10(16) U 13(16)
5: Z \ 7(32) U 25(32)
6: <union of 48 residue classes (mod 64)>
```

```
7: Integers
```

Thus $k = 7$ serves our purposes. To be sure that for any positive integer n our monoid contains a mapping f such that $n^f < n$, we still need to check this condition for "small" n. Since in case $c > a$ we have $(an+b)/c \geq n$ if only if $n \leq b/(c-a)$, we only need to check those n which are not larger than the largest coefficient $b_{r(m)}$ occurring in any of the products under consideration:

Example

```
gap> maxb := Maximum(List(maps,f->Maximum(List(Coefficients(f),t->t[2]))));
25999
gap> small := Filtered([1..maxb],n->ForAll(maps,f->n^f>=n));
[ 1, 7, 9, 11 ]
```

This means that except of 1, only for $n \in \{7,9,11\}$ there is no product of 7 mappings $T_5^{\pm}$ which maps n to a smaller integer. We check that also the forward orbits of these three integers contain 1 by successively computing preimages of 1:

Example

```
gap> S := [1];; k := 0;;
gap> repeat
>      S := Union(S,PreImage(T5m,S),PreImage(T5p,S));
>      k := k+1;
>    until IsSubset(S,small);
gap> k;
17
```

Enter `AssignGlobals(LoadRCWAExamples().CollatzMapping);` in order to assign the global variables defined in this section.

7.21 The free group of rank 2 and the modular group PSL(2,$\mathbb{Z}$)

The free group of rank 2 embeds into RCWA($\mathbb{Z}$) – in fact it embeds even in the subgroup which is generated by all class transpositions. An explicit embedding can be constructed by transferring the construction of the so-called "Schottky groups" (cf. [dlH00], page 27) from PSL(2,$\mathbb{C}$) to RCWA($\mathbb{Z}$) (we use the notation from the cited book):

Example

```
gap> D := AllResidueClassesModulo(4);
[ 0(4), 1(4), 2(4), 3(4) ]
gap> gamma1 := RepresentativeAction(RCWA(Integers),
>                                    Difference(Integers,D[1]),D[2]);;
gap> gamma2 := RepresentativeAction(RCWA(Integers),
>                                    Difference(Integers,D[3]),D[4]);;
gap> F2 := Group(gamma1,gamma2);
<rcwa group over Z with 2 generators>
```

We can do some checks:

Example

```
gap> X1 := Union(D{[1,2]});; X2 := Union(D{[3,4]});;
gap>     IsSubset(X1,X2^gamma1) and IsSubset(X1,X2^(gamma1^-1))
>    and IsSubset(X2,X1^gamma2) and IsSubset(X2,X1^(gamma2^-1));
true
```

The generators are products of 3 class transpositions, each:

Example

```
gap> Factorization(gamma1);
[ ( 0(2), 1(2) ), ( 3(4), 5(8) ), ( 0(2), 1(8) ) ]
gap> Factorization(gamma2);
[ ( 0(2), 1(2) ), ( 1(4), 7(8) ), ( 0(2), 3(8) ) ]
```

The above construction is used by `IsomorphismRcwaGroup` (3.1.1) to embed free groups of any rank ≥ 2.

We give another only slightly different representation of the free group of rank 2. We verify that it really is one by applying the so-called *Table-Tennis Lemma* (see e.g. [dlH00], Section II.B.) to the infinite cyclic groups generated by the two generators and to the same two sets `X1` and `X2` as above:

Example

```
gap> r1 := ClassTransposition(0,2,1,2)*ClassTransposition(0,2,1,4);;
gap> r2 := ClassTransposition(0,2,1,2)*ClassTransposition(0,2,3,4);;
gap> F2 := Group(r1^2,r2^2);;
gap> List(GeneratorsOfGroup(F2),IsTame);
[ false, false ]
gap>     IsSubset(X1,X2^F2.1) and IsSubset(X1,X2^(F2.1^-1))
>    and IsSubset(X2,X1^F2.2) and IsSubset(X2,X1^(F2.2^-1));
true
gap> [Sources(r1),Sinks(r1),Loops(r1)]; # compare with X1
[ [ 0(4) ], [ 1(4) ], [ 0(4), 1(4) ] ]
gap> [Sources(r2),Sinks(r2),Loops(r2)]; # compare with X2
[ [ 2(4) ], [ 3(4) ], [ 2(4), 3(4) ] ]
gap>    IsSubset(X1,Union(Sinks(r1))) and IsSubset(X1,Union(Sinks(r1^-1)))
>   and IsSubset(X2,Union(Sinks(r2))) and IsSubset(X2,Union(Sinks(r2^-1)));
true
gap> IsSubset(Union(Sinks(r1)),X2^F2.1) and
>    IsSubset(Union(Sinks(r1^-1)),X2^(F2.1^-1));
true
gap> IsSubset(Union(Sinks(r2)),X1^F2.2) and
>    IsSubset(Union(Sinks(r2^-1)),X1^(F2.2^-1));
true
```

Drawing the transition graphs of `r1` and `r2` for modulus 4 may help to understand what is actually done in this calculation. It is easy to see that the group generated by `r1` and `r2` is *not* free:

Example

```
gap> Order(r1/r2);
3
```

The modular group PSL(2,$\mathbb{Z}$) embeds into CT($\mathbb{Z}$) as well. We give an embedding, and check that it really is one by applying the Table Tennis Lemma as above:

Example

```
gap> PSL2Z :=
>      Group(ClassTransposition(0,3,1,3) * ClassTransposition(0,3,2,3),
>            ClassTransposition(1,3,0,6) * ClassTransposition(2,3,3,6));;
gap> List(GeneratorsOfGroup(PSL2Z),Order);
[ 3, 2 ]
gap> X1 := Difference(Integers,ResidueClass(0,3));
Z \ 0(3)
gap> X2 := ResidueClass(0,3);
0(3)
gap> IsSubset(X1,X2^PSL2Z.1) and IsSubset(X1,X2^(PSL2Z.1^2));
true
gap> IsSubset(X2,X1^PSL2Z.2);
true
```

A slightly different representation of PSL(2,$\mathbb{Z}$) can be obtained by using RCWA's general method for `IsomorphismRcwaGroup` for free products of finite groups:

Example

```
gap> G := Image(IsomorphismRcwaGroup(FreeProduct(CyclicGroup(3),
>                                                  CyclicGroup(2))));
<wild rcwa group over Z with 2 generators>
gap> List(GeneratorsOfGroup(G),Factorization);
[ [ ( 0(4), 2(4) ), ( 1(2), 0(4) ) ], [ ( 0(2), 1(2) ) ] ]
```

Enter `AssignGlobals(LoadRCWAExamples().F2_PSL2Z);` in order to assign the global variables defined in this section.

Chapter 8

The Algorithms Implemented in RCWA

This chapter lists brief descriptions of the algorithms and methods implemented in this package. These descriptions are kept very informal and terse, and some of them provide only rudimentary information. They are listed in alphabetical order. The word "trivial" as a description means that essentially nothing is done except of performing I/O operations, storing or recalling one or several values or doing very basic computations, and "straightforward" means that no sophisticated algorithm is used. Note that "trivial" and "straightforward" are to be read as *mathematically* trivial respectively straightforward, and that the code of a function or method attributed in this way can still be reasonably long and complicated. Longer and better descriptions of *some* of the algorithms and methods can be found in [Koh08].

`ActionOnRespectedPartition(G)`
: "Straightforward" after having computed a respected partition by `RespectedPartition`.

`AllElementsOfCTZWithGivenModulus(m)`
: This function first determines a list of all unordered partitions $\mathcal{P}$ of $\mathbb{Z}$ into `m` residue classes. Then for any such partition $\mathcal{P}$ it runs a loop over the elements of the symmetric group of degree `m`. For any $\sigma \in \mathrm{S}_m$ and any partition $\mathcal{P}$ it constructs the element of $\mathrm{CT}(\mathbb{Z})$ with modulus dividing m which maps the ordered partition $\{0(m), 1(m), \dots, m-1(m)\}$ to the ordered partition obtained from $\mathcal{P}$ by permuting the residue classes with σ. Finally it discards the elements whose modulus is a proper divisor of `m`, and returns the "rest".

`Ball(G,g,r)`
: "Straightforward".

`Ball(G,p,r,act)`
: "Straightforward".

`ClassPairs(m)`
: Runs a loop over all 4-tuples of nonnegative integers less than `m`, and filters by congruence criteria and ordering of the entries.

`ClassReflection(r,m)`
: "Trivial".

`ClassRotation(r,m,u)`
: "Trivial".

`ClassShift(r,m)`
"Trivial".

`ClassTransposition(r1,m1,r2,m2)`
"Trivial".

`ClassWiseOrderPreservingOn(f)`**, etc.**
Forms the union of the residue classes modulo the modulus of `f` in whose corresponding coefficient triple the first entry is positive, zero or negative, respectively.

`Coefficients(f)`
"Trivial".

`CommonRightInverse(l,r)`
See `RightInverse`.

`CT(R)`
Attributes and properties are set according to [Koh10].

`CycleRepresentativesAndLengths(g,S)`
"Straightforward".

`CyclesOnFiniteOrbit(G,g,n)`
"Straightforward".

`DecreasingOn(f)`
Forms the union of the residue classes which are determined by the coefficients as indicated.

`DerivedSubgroup(G)`
No genuine method – GAP Library methods already work for tame groups.

`Determinant(g)`
Evaluation of the given expression. For the mathematical meaning (epimorphism!), see Theorem 2.11.9 in [Koh05].

`DifferencesList(l)`
"Trivial".

`DirectProduct(G1,G2, ... )`
Restricts the groups `G1`, `G2`, ... to disjoint residue classes. See `Restriction` and Corollary 2.3.3 in [Koh05].

`Display(f)`
"Trivial".

`DistanceToNextSmallerPointInOrbit(G,n)`
"Straightforward" – computes balls of radius r about `n` for $r = 1,2,\dots$ until a point smaller than n is found.

`Divisor(f)`
Lcm of coefficients, as indicated.

`DrawGrid(U,range_y,range_x,filename)`
"Straightforward".

`DrawOrbitPicture`
Compute spheres of radius $1,\ldots,r$ around the given point(s). Choose the origin either in the lower left corner of the picture (if all points lie in the first quadrant) or in the middle of the picture (if they don't). Mark points of the ball with black pixels in case of a monochrome picture. Choose colors from the given palette depending on the distance from the starting points in case of a colored picture.

`EpimorphismFromFpGroup(G,r)`
Computes orders of elements in the ball of radius `r` about 1 in `G`, and uses the corresponding relations if they affect the abelian invariants of `G`, `G'`, `G''`, etc..

`Exponent(G)`
Check whether `G` is finite. If it is, then use the GAP Library method, applied to `Image(IsomorphismPermGroup(G))`. Check whether `G` is tame. If yes, return `infinity`. If not, run a loop over `G` until finding an element of infinite order. Once one is found, return `infinity`.

The final loop to find a non-torsion element can be left away under the assumption that any finitely generated wild rcwa group has a wild element. It looks likely that this holds, but currently the author does not know a proof.

`ExtRepOfObj(f)`
"Trivial".

`FactorizationIntoCSCRCT(g)`, `Factorization(g)`
The method used here is rather sophisticated, and will likely some time be published elsewhere. At the moment termination is not guaranteed, but in case of termination the result is certain. The strategy is roughly first to make the mapping class-wise order-preserving and balanced, and then to remove all prime factors from multiplier and divisor one after the other in decreasing order by dividing by appropriate class transpositions. The remaining integral mapping can be factored in a similar way as a permutation of a finite set can be factored into transpositions.

`FactorizationOnConnectedComponents(f,m)`
Calls GRAPE to get the connected components of the transition graph, and then computes a partition of the suitably "blown up" coefficient list corresponding to the connected components.

`FixedPointsOfAffinePartialMappings(f)`
"Straightforward".

`FixedResidueClasses(g,maxmod)`, `FixedResidueClasses(G,maxmod)`
Runs a loop over all moduli $m \leq$ `maxmod` and all residues r modulo these moduli, and selects those residue classes $r(m)$ which are mapped to itself by `g`, respectively, by all generators of `G`.

`FloatQuotientsList(l)`
"Trivial".

`GluckTaylorInvariant(a)`
Evaluation of the given expression.

`GroupByResidueClasses(classes)`
: Finds all pairs of residue classes in the list `classes` which are disjoint, forms the corresponding class transpositions and returns the group generated by them.

`GuessedDivergence(f)`
: Numerical computation of the limit of some series, which seems to converge "often". Caution!!!

`Image(f)`, `Image(f,S)`
: "Straightforward" if one can compute images of residue classes under affine mappings and unite and intersect residue classes (Chinese Remainder Theorem). See Lemma 1.2.1 in [Koh05].

`ImageDensity(f)`
: Evaluation of the given expression.

`g in G` **(membership test for rcwa groups)**
: Test whether the mapping g or its inverse is in the list of generators of `G`. If it is, return `true`. Test whether its prime set is a subset of the prime set of `G`. If not, return `false`. Test whether the multiplier or the divisor of g has a prime factor which does not divide the multiplier of `G`. If yes, return `false`. Test if `G` is class-wise order-preserving, and g is not. If so, return `false`. Test if the sign of g is -1 and all generators of `G` have sign 1. If yes, return `false`. Test if `G` is class-wise order-preserving, all generators of `G` have determinant 0 and g has determinant $\neq 0$. If yes, return `false`. Test whether the support of g is a subset of the support of `G`. If not, return `false`. Test whether `G` fixes the nonnegative integers setwise, but g does not. If yes, return `false`.

If `G` is tame, proceed as follows: Test whether the modulus of g divides the modulus of `G`. If not, return `false`. Test whether `G` is finite and g has infinite order. If so, return `false`. Test whether g is tame. If not, return `false`. Compute a respected partition `P` of `G` and the finite permutation group `H` induced by `G` on it (see `RespectedPartition`). Check whether g permutes `P`. If not, return `false`. Let `h` be the permutation induced by g on `P`. Check whether `h` lies in `H`. If not, return `false`. Compute an element `g1` of `G` which acts on `P` like g. For this purpose, factor *h* into generators of `H` using `PreImagesRepresentative`, and compute the corresponding product of generators of `G`. Let `k := g/g1`. The mapping `k` is always integral. Compute the kernel `K` of the action of `G` on `P` using `KernelOfActionOnRespectedPartition`. Check whether `k` lies in `K`. This is done using the package Polycyclic [EHN13], and uses an isomorphism from a supergroup of `K` which is isomorphic to the `|P|`-fold direct product of the infinite dihedral group and which always contains `k` to a polycyclically presented group. If `k` lies in `K`, return `true`, otherwise return `false`.

If `G` is not tame, proceed as follows: Look for finite orbits of `G`. If some are found, test whether g acts on them, and whether the induced permutations lie in the permutation groups induced by `G`. If for one of the examined orbits one of the latter two questions has a negative answer, then return `false`. Look for a positive integer m such that g does not leave a partition of $\mathbb{Z}$ into unions of residue classes (mod m) invariant which is fixed by `G`. If successful, return `false`. If not, try to factor g into generators of `G` using `PreImagesRepresentative`. If successful, return `true`. If g is in `G`, this terminates after a finite number of steps. Both run time and memory requirements are exponential in the word length. If g is not in `G` at this stage, the method runs into an infinite loop.

`f in M` **(membership test for rcwa monoids)**

Test whether the mapping `f` is in the list of generators of `G`. If it is, return `true`. Test whether the multiplier of `f` is zero, but all generators of `M` have nonzero multiplier. If yes, return `false`. Test if neither `f` nor any generator of `M` has multiplier zero. If so, check whether the prime set of `f` is a subset of the prime set of `M`, and whether the set of prime factors of the multiplier of `f` is a subset of the union of the sets of prime factors of the multipliers of the generators of `M`. If one of these is not the case, return `false`. Check whether the set of prime factors of the divisor of `f` is a subset of the union of the sets of prime factors of the divisors of the generators of `M`. If not, return `false`. If the underlying ring is $\mathbb{Z}$ or a semilocalization thereof, then check whether `f` is not class-wise order-preserving, but `M` is. If so, return `false`.

If `f` is not injective, but all generators of `M` are, then return `false`. If `f` is not surjective, but all generators of `M` are, then return `false`. If the support of `f` is not a subset of the support of `M`, then return `false`. If `f` is not sign-preserving, but `M` is, then return `false`. Check whether `M` is tame. If so, then return `false` provided that one of the following three conditions hold: 1. The modulus of `f` does not divide the modulus of `M`. 2. `f` is not tame. 3. `M` is finite, and `f` is bijective and has infinite order. If membership has still not been decided, use `ShortOrbits` to look for finite orbits of `M`, and check whether `f` fixes all of them setwise. If a finite orbit is found which `f` does not map to itself, then return `false`.

Finally compute balls of increasing radius around 1 until `f` is found to lie in one of them. If that happens, return `true`. If `f` is an element of `M`, this will eventually terminate, but if at this stage `f` is not an element of `M`, this will run into an infinite loop.

`point in orbit` **(membership test for orbits)**

Uses the equality test for orbits: The orbit equality test computes balls of increasing radius around the orbit representatives until they intersect non-trivially. Once they do so, it returns `true`. If it finds that one or both of the orbits are finite, it makes use of that information, and returns `false` if appropriate. In between, i.e. after having computed balls to a certain extent depending on the properties of the group, it chooses a suitable modulus m and computes orbits (modulo m). If the representatives of the orbits to be compared belong to different orbits (mod m), it returns `false`. If this is not the case although the orbits are different, the equality test runs into an infinite loop.

`IncreasingOn(f)`

Forms the union of the residue classes which are determined by the coefficients as indicated.

`Index(G,H)`

In general, i.e. if the underlying ring is not $\mathbb{Z}$, proceed as follows: If both groups `G` and `H` are finite, return the quotient of their orders. If `G` is infinite, but `H` is finite, return `infinity`. Otherwise return the number of right cosets of `H` in `G`, computed by the GAP Library function `RightCosets`.

If the underlying ring is $\mathbb{Z}$, do additionally the following before attempting to compute the list of right cosets: If the group `G` is class-wise order-preserving, check whether one of its generators has nonzero determinant, and whether all generators of `H` have determinant zero. If so, then return `infinity`. Check whether `H` is tame, but `G` is not. If so, then return `infinity`. If `G` is tame, then check whether the rank of the largest free abelian subgroup of the kernel of the action of `G` on a respected partition is higher than the corresponding rank for `H`. For this check, use `RankOfKernelOfActionOnRespectedPartition`. If it is, then return `infinity`.

`Induction(g,f)`
Computes `f * g * RightInverse(f)`.

`Induction(G,f)`
Gets a set of generators by applying `Induction(g,f)` to the generators `g` of `G`.

`InjectiveAsMappingFrom(f)`
The function starts with the entire source of `f` as "preimage" `pre` and the empty set as "image" `im`. It loops over the residue classes (mod `Mod(f)`). For any such residue class `cl` the following is done: Firstly, the image of `cl` under `f` is added to `im`. Secondly, the intersection of the preimage of the intersection of the image of `cl` under `f` and `im` under `f` and `cl` is subtracted from `pre`.

`IntegralConjugate(f)`, `IntegralConjugate(G)`
Uses the algorithm described in the proof of Theorem 2.5.14 in [Koh05].

`IntegralizingConjugator(f)`, `IntegralizingConjugator(G)`
Uses the algorithm described in the proof of Theorem 2.5.14 in [Koh05].

`Inverse(f)`
Essentially inversion of affine mappings. See Lemma 1.3.1, Part (b) in [Koh05].

`IsBalanced(f)`
Checks whether the sets of prime factors of the multiplier and the divisor of `f` are the same.

`IsBijective(f)`
"Trivial", respectively, see `IsInjective` and `IsSurjective`.

`IsClassReflection(g)`
Computes the support of `g`, and compares `g` with the corresponding class reflection.

`IsClassRotation(g)`
Computes the support of `g`, extracts the possible rotation factor from the coefficients and compares `g` with the corresponding class rotation.

`IsClassShift(g)`
Computes the support of `g`, and compares `g` with the corresponding class shift.

`IsClassTransposition(g)`, `IsGeneralizedClassTransposition(g)`
Computes the support of `g`, writes it as a disjoint union of two residue classes and compares `g` with the class transposition which interchanges them.

`IsClassWiseOrderPreserving(f)`, `IsClassWiseTranslating(f)`
"Trivial".

`IsConjugate(RCWA(Integers),f,g)`
Test whether `f` and `g` have the same order, and whether either both or none of them is tame. If not, return `false`.

If the mappings are wild, use `ShortCycles` to search for finite cycles not belonging to an infinite series, until their numbers for a particular length differ. This may run into an infinite loop. If it terminates, return `false`.

If the mappings are tame, use the method described in the proof of Theorem 2.5.14 in [Koh05] to construct integral conjugates of *f* and *g*. Then essentially use the algorithm described in the proof of Theorem 2.6.7 in [Koh05] to compute "standard representatives" of the conjugacy classes which the integral conjugates of *f* and *g* belong to. Finally compare these standard representatives, and return `true` if they are equal and `false` if not.

`IsInjective(f)`
: See `Image`.

`IsIntegral(f)`
: "Trivial".

`IsNaturalCT(G), IsNaturalRCWA(G)`
: Only checks a set flag.

`IsomorphismMatrixGroup(G)`
: Uses the algorithm described in the proof of Theorem 2.6.3 in [Koh05].

`IsomorphismPermGroup(G)`
: If the group *G* is finite and class-wise order-preserving, use `ActionOnRespectedPartition`. If *G* is finite, but not class-wise order-preserving, compute the action on the respected partition which is obtained by splitting any residue class $r(m)$ in `RespectedPartition(G)` into three residue classes $r(3m), r+m(3m), r+2m(3m)$. If *G* is infinite, there is no isomorphism to a finite permutation group, thus return `fail`.

`IsomorphismRcwaGroup(G)`
: The method for finite groups uses `RcwaMapping`, Part (d).

The method for free products of finite groups uses the Table-Tennis Lemma (which is also known as *Ping-Pong Lemma*, cf. e.g. Section II.B. in [dlH00]). It uses regular permutation representations of the factors G_r $(r = 0,\dots,m-1)$ of the free product on residue classes modulo $n_r := |G_r|$. The basic idea is that since point stabilizers in regular permutation groups are trivial, all non-identity elements map any of the permuted residue classes into their complements. To get into a situation where the Table-Tennis Lemma is applicable, the method computes conjugates of the images of the mentioned permutation representations under rcwa permutations σ_r which satisfy $0(n_r)^{\sigma_r} = \mathbb{Z} \setminus r(m)$.

The method for free groups uses an adaptation of the construction given on page 27 in [dlH00] from PSL(2,$\mathbb{C}$) to RCWA($\mathbb{Z}$). As an equivalent for the closed discs used there, the method takes the residue classes modulo two times the rank of the free group.

`IsOne(f)`
: "Trivial".

`IsPerfect(G)`
: If the group *G* is trivial, then return `true`. Otherwise if it is abelian, then return `false`.

If the underlying ring is $\mathbb{Z}$, then do the following: If one of the generators of *G* has sign -1, then return `false`. If *G* is class-wise order-preserving and one of the generators has nonzero determinant, then return `false`.

If `G` is wild, and perfectness has not been decided so far, then give up. If `G` is finite, then check the image of `IsomorphismPermGroup(G)` for perfectness, and return `true` or `false` accordingly.

If the group `G` is tame and if it acts transitively on its stored respected partition, then return `true` or `false` depending on whether the finite permutation group `ActionOnRespectedPartition(G)` is perfect or not. If `G` does not act transitively on its stored respected partition, then give up.

`IsPrimeSwitch(g)`
: Checks whether the multiplier of g is an odd prime, and compares g with the corresponding prime switch.

`IsSignPreserving(f)`
: If `f` is not class-wise order-preserving, then return `false`. Otherwise let $c \geq 1$ be greater than or equal to the maximum of the absolute values of the coefficients $b_{r(m)}$ of the affine partial mappings of `f`, and check whether the minimum of the image of $\{0,\dots,c\}$ under `f` is nonnegative and whether the maximum of the image of $\{-c,\dots,-1\}$ under `f` is negative. If both is the case, then return `true`, otherwise return `false`.

`IsSolvable(G)`
: If `G` is abelian, then return `true`. If `G` is tame, then return `true` or `false` depending on whether `ActionOnRespectedPartition(G)` is solvable or not. If `G` is wild, then give up.

`IsSubset(G,H)` **(checking for a subgroup relation)**
: Check whether the set of stored generators of `H` is a subset of the set of stored generators of `G`. If so, return `true`. Check whether the prime set of `H` is a subset of the prime set of `G`. If not, return `false`. Check whether the support of `H` is a subset of the support of `G`. If not, return `false`. Check whether `G` is tame, but `H` is wild. If so, return `false`.

If `G` and `H` are both tame, then proceed as follows: If the multiplier of `H` does not divide the multiplier of `G`, then return `false`. If `H` does not respect the stored respected partition of `G`, then return `false`. Check whether the finite permutation group induced by `H` on `RespectedPartition(G)` is a subgroup of `ActionOnRespectedPartition(G)`. If yes, return `true`. Check whether the order of `H` is greater than the order of `G`. If so, return `false`.

Finally use the membership test to check whether all generators of `H` lie in `G`, and return `true` or `false` accordingly.

`IsSurjective(f)`
: See `Image`.

`IsTame(G)`
: Checks whether the modulus of the group is nonzero.

`IsTame(f)`
: Application of the criteria given in Corollary 2.5.10 and 2.5.12 and Theorem A.8 and A.11 in [Koh05], as well as of the criteria given in [Koh07a]. The criterion "surjective, but not injective means wild" (Theorem A.8 in [Koh05]) is the subject of [Koh07b]. The package GRAPE is needed for the application of the criterion which says that an rcwa permutation is wild if a transition graph has a weakly-connected component which is not strongly-connected (cf. Theorem A.11 in [Koh05]).

`IsTransitive(G,Integers)`

Look for finite orbits, using `ShortOrbits` on a couple of intervals. If a finite orbit is found, return `false`. Test if G is finite. If yes, return `false`.

Search for an element g and a residue class $r(m)$ such that the restriction of g to $r(m)$ is given by $n \mapsto n+m$. Then the cyclic group generated by g acts transitively on $r(m)$. The element g is searched among the generators of G, its powers, its commutators, powers of its commutators and products of few different generators. The search for such an element may run into an infinite loop, as there is no guarantee that the group has a suitable element.

If suitable g and $r(m)$ are found, proceed as follows:

Put $S := r(m)$. Put $S := S \cup S^g$ for all generators g of G, and repeat this until S remains constant. This may run into an infinite loop.

If it terminates: If $S = \mathbb{Z}$, return `true`, otherwise return `false`.

`IsTransitiveOnNonnegativeIntegersInSupport(G)`

Computes balls about 1 with successively increasing radii, and checks whether the union of the sets where the elements of these balls are decreasing or shifting down equals the support of G. If a positive answer is found, transitivity on "small" points (nonnegative integers less than an explicit bound) is verified.

`IsZero(f)`

"Trivial".

`KernelOfActionOnRespectedPartition(G)`

First determine the abelian invariants of the kernel K. For this, compute sufficiently many quotients of orders of permutation groups induced by G on refinements of the stored respected partition P by the order of the permutation group induced by G on P itself. Then use a random walk through the group G. Compute powers of elements encountered along the way which fix P. Translate these kernel elements into elements of a polycyclically presented group isomorphic to the |P|-fold direct product of the infinite dihedral group (K certainly embeds into this group). Use Polycyclic [EHN13] to collect independent "nice" generators of K. Proceed until the permutation groups induced by K on the refined respected partitions all equal the initially stored quotients.

`LargestSourcesOfAffineMappings(f)`

Forms unions of residue classes modulo the modulus of the mapping, whose corresponding coefficient triples are equal.

`LaTeXStringRcwaMapping(f)`, `LaTeXAndXDVI(f)`

Collects residue classes those corresponding coefficient triples are equal.

`LikelyContractionCentre(f,maxn,bound)`

Computes trajectories with starting values from a given interval, until a cycle is reached. Aborts if the trajectory exceeds the prescribed bound. Form the union of the detected cycles.

`LoadDatabaseOf...()`, `LoadRCWAExamples()`

"Trivial". – These functions do nothing more than reading in certain files.

`LocalizedRcwaMapping(f,p)`

"Trivial".

`Log2HTML(logfilename)`
: Straightforward string operations.

`Loops(f)`
: Runs over the residue classes modulo the modulus of `f`, and selects those of them which `f` does not map to themselves, but which intersect non-trivially with their images under `f`.

`MaximalShift(f)`
: "Trivial".

`MergerExtension(G,points,point)`
: As described in `MergerExtension` (3.1.4).

`Mirrored(g)`, `Mirrored(G)`
: Conjugates with $n \mapsto -n-1$, as indicated in the definition.

`mKnot(m)`
: "Straightforward", following the definition given in [Kel99].

`Modulus(G)`
: Searches for a wild element in the group. If unsuccessful, tries to construct a respected partition (see `RespectedPartition`).

`Modulus(f)`
: "Trivial".

`MovedPoints(G)`
: Needs only forming unions of residue classes and determining fixed points of affine mappings.

`Multiplier(f)`
: Lcm of coefficients, as indicated.

`Multpk(f,p,k)`
: Forms the union of the residue classes modulo the modulus of the mapping, which are determined by the given divisibility criteria for the coefficients of the corresponding affine mapping.

`NrClassPairs(m)`
: Relatively straightforward. – Practical for values of `m` ranging up into the hundreds and corresponding counts of 10^9 and more.

`NrConjugacyClassesOfCTZOfOrder(ord)`,
: Evaluation of the expression `Length(Filtered(Combinations(DivisorsInt(ord)), l -> l <> [] and Lcm(l) = ord))`.

`NrConjugacyClassesOfRCWAZOfOrder(ord)`
: The class numbers are taken from Corollary 2.7.1 in [Koh05].

`ObjByExtRep(fam,l)`
: "Trivial".

`One(f)`, `One(G)`,
: "Trivial".

`Orbit(G,pnt,gens,acts,act)`

Check if the orbit has length less than a certain bound. If so, then return it as a list. Otherwise test whether the group `G` is tame or wild.

If `G` is tame, then test whether `G` is finite. If yes, then compute the orbit by the GAP Library method. Otherwise proceed as follows: Compute a respected partition $\mathscr{P}$ of `G`. Use $\mathscr{P}$ to find a residue class $r(m)$ which is a subset of the orbit to be computed. In general, $r(m)$ will not be one of the residue classes in $\mathscr{P}$, but a subset of one of them. Put $\Omega := r(m)$. Unite the set Ω with its images under all the generators of `G` and their inverses. Repeat that until Ω does not change any more. Return Ω.

If `G` is wild, then return an orbit object which stores the group `G`, the representative `rep` and the action `act`.

`OrbitsModulo(f,m)`

Uses GRAPE to compute the connected components of the transition graph.

`OrbitsModulo(G,m)`

"Straightforward".

`Order(f)`

Test for `IsTame`. If the mapping is not tame, then return `infinity`. Otherwise use Corollary 2.5.10 in [Koh05].

`PermutationOpNC(sigma,P,act)`

Several different methods for different types of arguments, which either provide straightforward optimizations via computing with coefficients directly, or just delegate to `PermutationOp`.

`PreImage(f,S)`

See `Image`.

`PreImagesRepresentative(phi,g)`, `PreImagesRepresentatives(phi,g)`

As described in the documentation of these methods. The underlying idea to successively compute two balls around 1 and g until they intersect non-trivially is standard in computational group theory. For rcwa groups it would mean wasting both memory and run time to actually compute group elements. Thus only images of tuples of points are computed and stored.

`PrimeSet(f)`, `PrimeSet(G)`

"Straightforward".

`PrimeSwitch(p)`

Multiplication of rcwa mappings as indicated.

`Print(f)`

"Trivial".

`f*g`

Essentially composition of affine mappings. See Lemma 1.3.1, Part (a) in [Koh05].

`ProjectionsToCoordinates(f)`

Straightforward coefficient operations.

`ProjectionsToInvariantUnionsOfResidueClasses(G,m)`
: Use `OrbitsModulo` to determine the supports of the images of the epimorphisms to be determined, and use `RestrictedPerm` to compute the images of the generators of G under these epimorphisms.

`QuotientsList(l)`
: "Trivial".

`Random(RCWA(Integers))`
: Computes a product of "randomly" chosen class shifts, class reflections and class transpositions. This seems to be suitable for generating reasonably good examples.

`RankOfKernelOfActionOnRespectedPartition(G)`
: Performs the first part of the computations done by `KernelOfActionOnRespectedPartition`.

`Rcwa(R)`
: "Trivial". – Attributes and properties set can be derived easily or hold by definition.

`RCWA(R)`
: Attributes and properties are set according to Theorem 2.1.1, Theorem 2.1.2, Corollary 2.1.6 and Theorem 2.12.8 in [Koh05].

`RCWABuildManual()`
: Consists of a call to a function from the GAPDoc package.

`RcwaGroupByPermGroup(G)`
: Uses `RcwaMapping`, Part (d).

`RCWAInfo(n)`
: "Trivial".

`RcwaMapping`
: (a)-(c): "trivial", (d): `n^perm - n` for determining the coefficients, (e): "affine mappings by values at two given points", (f) and (g): "trivial", (h) and (i): correspond to Lemma 2.1.4 in [Koh05], (j): uses a simple parser for the permitted expressions.

`RCWATestAll(), RCWATestInstall()`
: Just read in files running / containing the tests.

`RCWATestExamples()`
: Runs the example tester from the GAPDoc package.

`RepresentativeAction(G,src,dest,act), RepresentativeActionPreImage`
: As described in the documentation of these methods. The underlying idea to successively compute two balls around *src* and *dest* until they intersect non-trivially is standard in computational group theory. Words standing for products of generators of G are stored for every image of *src* or *dest*.

`RepresentativeAction(RCWA(Integers),P1,P2)`
: Arbitrary mapping: see Lemma 2.1.4 in [Koh05]. Tame mapping: see proof of Theorem 2.8.9 in [Koh05]. The former is almost trivial, while the latter is a bit complicated and takes usually also much more time.

`RepresentativeAction(RCWA(Integers),f,g)`
The algorithm used by `IsConjugate` constructs actually also an element `x` such that `f^x = g`.

`RespectedPartition(f)`, `RespectedPartition(G)`
There are presently two sophisticated algorithms implemented for finding respected partitions. One of them has evolved from the algorithm described in the proof of Theorem 2.5.8 in [Koh05]. The other one starts with the coarsest partition of the base ring such that every generator of `G` is affine on every part. This partition is then refined successively until a respected partition is obtained. The refinement step is basically as follows: Take the images of the partition under all generators of `G`. This way one obtains as many further partitions of the base ring as there are generators of `G`. Then the "new" partition is the coarsest common refinement of all these partitions.

`RespectsPartition(G,P)`
"Straightforward".

`RestrictedBall(G,g,r,modulusbound)`
"Straightforward".

`RestrictedPerm(g,S)`
"Straightforward".

`Restriction(g,f)`
Computes the action of `RightInverse(f) * g * f` on the image of `f`.

`Restriction(G,f)`
Gets a set of generators by applying `Restriction(g,f)` to the generators `g` of `G`.

`RightInverse(f)`
"Straightforward" if one knows how to compute images of residue classes under affine mappings, and how to compute inverses of affine mappings.

`Root(f,k)`
If `f` is bijective, class-wise order-preserving and has finite order:

Find a conjugate of `f` which is a product of class transpositions. Slice cycles $\prod_{i=2}^{l} \tau_{r_1(m_1),r_i(m_i)}$ of `f` a respected partition $\mathscr{P}$ into cycles $\prod_{i=1}^{l} \prod_{j=0}^{k-1} \tau_{r_1(km_1),r_i+jm_i(km_i)}$ of the `k`-fold length on the refined partition which one gets from $\mathscr{P}$ by decomposing any $r_i(m_i) \in \mathscr{P}$ into residue classes (mod km_i). Finally conjugate the resulting permutation back.

Other cases seem to be more difficult and are currently not covered.

`RotationFactor(g)`
"Trivial".

`RunDemonstration(filename)`
"Trivial" – only I/O operations.

`SemilocalizedRcwaMapping(f,pi)`
"Trivial".

`ShiftsDownOn(f)`, `ShiftsUpOn(f)`
Straightforward coefficient- and residue class operations.

`ShortCycles(g,maxlng)`
Looks for fixed points of affine partial mappings of powers of `g`.

`ShortCycles(g,S,maxlng)`, `ShortCycles(g,S,maxlng,maxn)`
"Straightforward".

`ShortOrbits(G,S,maxlng)`, `ShortOrbits(G,S,maxlng,maxn)`
"Straightforward".

`ShortResidueClassCycles(g,modulusbound,maxlng)`
Different methods – see source code in `pkg/rcwa/lib/rcwamap.gi`.

`ShortResidueClassOrbits(g,modulusbound,maxlng)`
Different methods – see source code in `pkg/rcwa/lib/rcwagrp.gi`.

`Sign(g)`
Evaluation of the given expression. For the mathematical meaning (epimorphism!), see Theorem 2.12.8 in [Koh05].

`Sinks(f)`
Computes the strongly connected components of the transition graph by the function `STRONGLY_CONNECTED_COMPONENTS_DIGRAPH`, and selects those which are proper subsets of their preimages and proper supersets of their images under `f`.

`Size(G)` **(order of an rcwa group)**
Test whether one of the generators of the group `G` has infinite order. If so, return `infinity`. Test whether the group `G` is tame. If not, return `infinity`. Test whether `RankOfKernelOfActionOnRespectedPartition(G)` is nonzero. If so, return `infinity`. Otherwise if `G` is class-wise order-preserving, return the size of the permutation group induced on the stored respected partition. If `G` is not class-wise order-preserving, return the size of the permutation group induced on the refinement of the stored respected partition which is obtained by splitting each residue class into three residue classes with equal moduli.

`Size(M)` **(order of an rcwa monoid)**
Check whether `M` is in fact an rcwa group. If so, use the method for rcwa groups instead. Check whether one of the generators of `M` is surjective, but not injective. If so, return `infinity`. Check whether for all generators f of `M`, the image of the union of the loops of f under f is finite. If not, return `infinity`. Check whether one of the generators of `M` is bijective and has infinite order. If so, return `infinity`. Check whether one of the generators of `M` is wild. If so, return `infinity`. Apply the above criteria to the elements of the ball of radius 2 around 1, and return `infinity` if appropriate. Finally attempt to compute the list of elements of `M`. If this is successful, return the length of the resulting list.

`SmallGeneratingSet(G)`
Eliminates generators g which can be found to be redundant *easily*, i.e. by checking whether the balls about 1 and g of some small radius r in the group generated by all generators of `G` except for g intersect nontrivially.

`Sources(f)`
Computes the strongly connected components of the transition graph by the function

STRONGLY_CONNECTED_COMPONENTS_DIGRAPH, and selects those which are proper supersets of their preimages and proper subsets of their images under *f*.

`SparseRep(f), StandardRep(f)`
Straightforward coefficient operations.

`SplittedClassTransposition(ct,k)`
"Straightforward".

`StructureDescription(G)`
This method uses a combination of techniques to obtain some basic information on the structure of an rcwa group. The returned description reflects the way the group has been built (DirectProduct, WreathProduct, etc.).

`f+g`
Pointwise addition of affine mappings.

`String(obj)`
"Trivial".

`Support(G)`
"Straightforward".

`Trajectory(f,n,...)`
Iterated application of an rcwa mapping. In the methods computing "accumulated coefficients", additionally composition of affine mappings.

`TransitionGraph(f,m)`
"Straightforward" – just check a sufficiently long interval.

`TransitionMatrix(f,m)`
Evaluation of the given expression.

`TransposedClasses(g)`
"Trivial".

`View(f)`
"Trivial".

`WreathProduct(G,P)`
Uses DirectProduct to embed the NrMovedPoints(*P*)th direct power of *G*, and RcwaMapping, Part (d) to embed the finite permutation group *P*.

`WreathProduct(G,Z)`
Restricts *G* to the residue class 3(4), and encodes the generator of *Z* as $\tau_{0(2),1(2)} \cdot \tau_{0(2),1(4)}$. It is used that the images of 3(4) under powers of this mapping are pairwise disjoint residue classes.

`Zero(f)`
"Trivial".

Chapter 9

Installation and Auxiliary Functions

9.1 Requirements

This version of RCWA needs at least GAP 4.8.7 in 64-bit mode, ResClasses 4.7.0, GRAPE 4.7 [Soi16], Polycyclic 2.11 [EHN13], FR 2.2.1 [Bar15], GAPDoc 1.5.1 [LN12], and Utils 0.40 [GKW16]. With possible exception of the most recent version of ResClasses, all needed packages are already present in an up-to-date standard GAP installation. The RCWA package is completely written in the GAP language and can be used on all platforms for which GAP is available.

9.2 Installation

Like any other GAP package, RCWA is usually installed in the `pkg` subdirectory of the GAP distribution. This is accomplished by extracting the distribution file in this directory. If you have done this, you can load the package as usual via `LoadPackage( "rcwa" );`.

9.3 Building the manual

The following routine is a development tool. As all files it generates are included in the distribution file anyway, users will not need it.

9.3.1 RCWABuildManual

▷ `RCWABuildManual()` (function)

Returns: nothing.

This function builds the manual of the RCWA package in the file formats LaTeX, PDF, HTML and ASCII text. This is accomplished using the GAPDoc package by Frank Lübeck and Max Neunhöffer. Building the manual is possible only on UNIX-type systems and requires PDFLaTeX.

9.4 The testing routines

9.4.1 RCWATestInstall

▷ `RCWATestInstall()` (function)

Returns: `true` if no errors were found, and `false` otherwise.

Performs a nontrivial computation to check whether an installation of RCWA appears to work. Errors, i.e. differences to the correct results of the test computation, are reported. The processed test file is `pkg/rcwa/tst/testinstall.tst`.

9.4.2 RCWATestAll

▷ `RCWATestAll()` (function)

Returns: `true` if no errors were found, and `false` otherwise.

Runs the full test suite of the RCWA package. Any differences to the supposed results of the test computations are reported. The processed test files are in the `tst` subdirectory of the package directory.

Please note that the test suite is a tool for developing. The tests are deliberately very volatile to allow to spot possible problems of any kind also in other packages or in the GAP Library. For this reason you may see reports of differences which simply reflect improved methods in other packages or in the GAP Library (for example an object may already know more of its attributes or properties than it is expected to, or an object may be represented in a better way), or which are caused by changes of the way certain objects are printed, and which are therefore harmless. However if the correct and the actual output look different mathematically or if you see error messages or if GAP crashes, then something went wrong. Also, reports about significantly increased run times of individual commands as well as run times of test files which are much longer than the predicted times shown may indicate a problem.

9.4.3 RCWATestExamples

▷ `RCWATestExamples()` (function)

Returns: nothing.

Runs all examples in the manual of the RCWA package, and reports any differences between the actual output and the output printed in the manual.

9.5 The Info class of the package

9.5.1 InfoRCWA

▷ `InfoRCWA` (info class)

This is the Info class of the RCWA package. See section *Info Functions* in the GAP Reference Manual for a description of the Info mechanism. For convenience: `RCWAInfo(n)` is a shorthand for `SetInfoLevel(InfoRCWA,n)`.

References

[And00] P. Andaloro. On total stopping times under $3x+1$ iteration. *Fibonacci Quarterly*, 38:73–78, 2000. 110

[Bar15] L. Bartholdi. *FR – Computations with functionally recursive groups. Version 2.2.1*, 2015. GAP package, `http://www.gap-system.org/Packages/fr.html`. 150

[dlH00] P. de la Harpe. *Topics in Geometric Group Theory*. Chicago Lectures in Mathematics, 2000. 32, 132, 133, 141

[EHN13] B. Eick, M. Horn, and W. Nickel. *Polycyclic – Computation with polycyclic groups (Version 2.11)*, 2013. GAP package, `http://www.gap-system.org/Packages/polycyclic.html`. 56, 138, 143, 150

[GKW16] S. Gutsche, S. Kohl, and C. Wensley. *Utils - Utility functions in GAP (Version 0.38)*, 2016. GAP package, `http://www.gap-system.org/Packages/utils.html`. 150

[Gri80] R. I. Grigorchuk. Burnside's problem on periodic groups. *Functional Anal. Appl.*, 14:41–43, 1980. 129, 130

[GT02] D. Gluck and B. D. Taylor. A new statistic for the $3x+1$ problem. *Proc. Amer. Math. Soc.*, 130(5):1293–1301, 2002. 28

[HEO05] D. F. Holt, B. Eick, and E. A. O'Brien. *Handbook of Computational Group Theory*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall / CRC, Boca Raton, FL, 2005. 5

[Hig74] G. Higman. *Finitely Presented Infinite Simple Groups*. Notes on Pure Mathematics. Department of Pure Mathematics, Australian National University, Canberra, 1974. 81, 82

[Kel99] T. P. Keller. Finite cycles of certain periodically linear permutations. *Missouri J. Math. Sci.*, 11(3):152–157, 1999. 22, 144

[Koh05] S. Kohl. *Restklassenweise affine Gruppen*. Dissertation, Universität Stuttgart, 2005. `http://d-nb.info/977164071`. 19, 20, 37, 53, 55, 56, 136, 138, 140, 141, 142, 144, 145, 146, 147, 148

[Koh07a] S. Kohl. Graph theoretical criteria for the wildness of residue-class-wise affine permutations, 2007. Preprint (short note), `http://www.gap-system.org/DevelopersPages/StefanKohl/preprints/graphcrit.pdf`. 142

[Koh07b] S. Kohl. Wildness of iteration of certain residue-class-wise affine mappings. *Adv. in Appl. Math.*, 39(3):322–328, 2007. DOI: 10.1016/j.aam.2006.08.003. 142

[Koh08] S. Kohl. Algorithms for a class of infinite permutation groups. *J. Symb. Comput.*, 43(8):545–581, 2008. DOI: 10.1016/j.jsc.2007.12.001. 6, 135

[Koh10] S. Kohl. A simple group generated by involutions interchanging residue classes of the integers. *Math. Z.*, 264(4):927–938, 2010. DOI: 10.1007/s00209-009-0497-8. 2, 6, 11, 40, 136

[Koh13] S. Kohl. Simple groups generated by involutions interchanging residue classes modulo lattices in $\mathbb{Z}^d$. *J. Group Theory*, 16(1):81–86, 2013. DOI: 10.1515/jgt-2012-0031. 66

[Lag03] J. C. Lagarias. The 3x+1 problem: An annotated bibliography, 2003+. `http://arxiv.org/abs/math.NT/0309224` (Part I), `http://arxiv.org/abs/math.NT/0608208` (Part II). 5

[LN12] F. Lübeck and M. Neunhöffer. *GAPDoc (Version 1.5.1)*. RWTH Aachen, 2012. GAP package, `http://www.gap-system.org/Packages/gapdoc.html`. 150

[ML87] K. R. Matthews and G. M. Leigh. A generalization of the Syracuse algorithm in GF(q)[x]. *J. Number Theory*, 25:274–278, 1987. 111

[Soi16] L. Soicher. *GRAPE – GRaph Algorithms using PErmutation groups (Version 4.7)*. Queen Mary, University of London, 2016. GAP package, `http://www.gap-system.org/Packages/grape.html`. 26, 150

Index