

RCWA

Residue-Class-Wise Affine Groups

Version 2.5.4

September 26, 2007

Stefan Kohl

Stefan Kohl — Email: kohl@mathematik.uni-stuttgart.de

— Homepage: <http://www.cip.mathematik.uni-stuttgart.de/~kohlsn/>

— Address: Institut für Geometrie und Topologie

Pfaffenwaldring 57

Universität Stuttgart

70550 Stuttgart

Germany

Abstract

RCWA is a package for GAP 4. It provides implementations of algorithms and methods for computing in certain infinite permutation groups. In principle, this package can deal at least with the following types of groups and their subgroups:

- Finite groups, and certain divisible torsion groups which they embed into.
- Free groups of finite rank.
- Free products of finitely many finite groups, thus in particular the modular group $\mathrm{PSL}(2, \mathbb{Z})$.
- Direct products of the above groups.
- Wreath products of the above groups with finite groups and with $(\mathbb{Z}, +)$.

With substantial help of this package, the author has found a countable simple group which has an uncountable series of simple subgroups. This simple group is generated by involutions which interchange disjoint residue classes of the integers. All the above groups embed into it.

Copyright

© 2003 - 2007 by Stefan Kohl. This package is distributed under the GNU General Public License.

Acknowledgements

I am very grateful to Bettina Eick for communicating this package and for her kind help in improving its documentation. Further I would like to thank the two anonymous referees for their constructive criticism and their helpful suggestions.

I am also very grateful to Laurent Bartholdi for his hint on how to construct wreath products of residue-class-wise affine groups with $(\mathbb{Z}, +)$. Last but not least I would like to thank all the people who have invited me so far to give talks on the subject in their seminars and on their conferences.

Contents

1	About the RCWA Package	6
1.1	Motivation	6
1.2	Purpose of this package	6
1.3	Groups which this package can deal with	7
1.4	Scope of this package	7
2	Residue-Class-Wise Affine Mappings	8
2.1	Basic definitions	8
2.2	Entering residue-class-wise affine mappings	9
2.2.1	ClassShift (r, m)	10
2.2.2	ClassReflection (r, m)	10
2.2.3	ClassTransposition (r1, m1, r2, m2)	11
2.2.4	ClassRotation (r, m, u)	12
2.2.5	RcwaMapping (the general constructor)	13
2.2.6	LocalizedRcwaMapping (for an rcwa mapping of \mathbb{Z} and a prime)	14
2.3	Basic arithmetic for residue-class-wise affine mappings	15
2.4	Attributes and properties of residue-class-wise affine mappings	17
2.4.1	LargestSourcesOfAffineMappings (for an rcwa mapping)	18
2.4.2	FixedPointsOfAffinePartialMappings (for an rcwa mapping)	18
2.4.3	Multpk (for an rcwa mapping, a prime and an exponent)	19
2.4.4	Determinant (of an rcwa mapping of \mathbb{Z})	19
2.4.5	Sign (of an rcwa permutation of \mathbb{Z})	20
2.5	Factoring residue-class-wise affine permutations	20
2.5.1	FactorizationIntoCSCRCT (for an rcwa permutation of \mathbb{Z})	20
2.5.2	PrimeSwitch (p)	21
2.5.3	mKnot (for an odd integer)	22
2.6	Extracting roots of residue-class-wise affine mappings	22
2.6.1	Root (k-th root of an rcwa mapping)	22
2.7	Special functions for non-bijective mappings	23
2.7.1	RightInverse (of an injective rcwa mapping)	23
2.7.2	CommonRightInverse (of two injective rcwa mappings)	23
2.7.3	ImageDensity (of an rcwa mapping)	23
2.8	On trajectories and cycles of residue-class-wise affine mappings	24
2.8.1	Trajectory (methods for rcwa mappings)	24
2.8.2	Trajectory (methods for rcwa mappings – “accumulated coefficients”)	24
2.8.3	IncreasingOn & DecreasingOn (for an rcwa mapping)	25

2.8.4	TransitionGraph (for an rcwa mapping and a modulus)	25
2.8.5	OrbitsModulo (for an rcwa mapping and a modulus)	26
2.8.6	FactorizationOnConnectedComponents (for an rcwa mapping and a modulus)	26
2.8.7	TransitionMatrix (for an rcwa mapping and a modulus)	26
2.8.8	Sources & Sinks (of an rcwa mapping)	27
2.8.9	Loops (of an rcwa mapping)	27
2.8.10	GluckTaylorInvariant (of a trajectory)	27
2.8.11	LikelyContractionCentre (of an rcwa mapping)	28
2.8.12	GuessedDivergence (of an rcwa mapping)	28
2.9	The categories and families of rcwa mappings	29
2.9.1	IsRcwaMapping	29
2.9.2	RcwaMappingsFamily (of a ring)	29
3	Residue-Class-Wise Affine Groups	30
3.1	Constructing residue-class-wise affine groups	30
3.1.1	RCWA (the group of all rcwa permutations of a ring)	30
3.1.2	CT (the group generated by all class transpositions of a ring)	31
3.1.3	IsomorphismRcwaGroup (for a group, over a given ring)	31
3.1.4	DirectProduct (for rcwa groups over \mathbb{Z})	32
3.1.5	WreathProduct (for an rcwa group over \mathbb{Z} , with a permutation group or $(\mathbb{Z}, +)$)	32
3.1.6	Restriction (of an rcwa mapping or -group, by an injective rcwa mapping)	33
3.1.7	Induction (of an rcwa mapping or -group, by an injective rcwa mapping)	33
3.2	Basic routines for investigating residue-class-wise affine groups	34
3.2.1	StructureDescription (for an rcwa group)	34
3.2.2	EpimorphismFromFpGroup (for an rcwa group and a search radius)	37
3.2.3	PreImagesRepresentative (for an epi. from a free group to an rcwa group)	38
3.3	The natural action of an rcwa group on the underlying ring	39
3.3.1	Orbit (for an rcwa group and either a point or a set)	39
3.3.2	DrawOrbitPicture (G, p0, r, h, w, colored, palette, filename)	40
3.3.3	ShortOrbits (for rcwa groups) & ShortCycles (for rcwa permutations)	40
3.3.4	Ball (for group, element and radius or group, point, radius and action)	41
3.3.5	RepresentativeAction (G, source, destination, action)	41
3.3.6	Projections (for an rcwa group and a modulus)	43
3.3.7	RepresentativeAction (for RCWA(R) and 2 partitions of R into residue classes)	43
3.4	Special attributes of tame residue-class-wise affine groups	44
3.4.1	RespectedPartition (of a tame rcwa group or -permutation)	44
3.4.2	ActionOnRespectedPartition & KernelOfActionOnRespectedPartition	45
3.5	Generating pseudo-random elements of RCWA(R) and CT(R)	46
3.6	The categories of residue-class-wise affine groups	47
3.6.1	IsRcwaGroup	47
4	Residue-Class-Wise Affine Monoids	48
4.1	Constructing residue-class-wise affine monoids	48
4.1.1	Rcwa (the monoid of all rcwa mappings of a ring)	49
4.2	Computing with residue-class-wise affine monoids	49
4.2.1	ShortOrbits (for rcwa monoid, set of points and bound on length)	50
4.2.2	Ball (for monoid, element and radius or monoid, point, radius and action)	51

5	Examples	52
5.1	Factoring Collatz' permutation of the integers	52
5.2	An rcwa mapping which seems to be contracting, but very slow	54
5.3	Checking a result by P. Andalaro	56
5.4	Two examples by Matthews and Leigh	57
5.5	Exploring the structure of a wild rcwa group	59
5.6	A wild rcwa mapping which has only finite cycles	61
5.7	An abelian rcwa group over a polynomial ring	65
5.8	A tame group generated by commutators of wild permutations	66
5.9	Checking for solvability	69
5.10	Some examples over (semi)localizations of the integers	70
5.11	Twisting 257-cycles into an rcwa mapping with modulus 32	73
5.12	The behaviour of the moduli of powers	74
5.13	Images and preimages under the Collatz mapping	75
5.14	A group which acts 4-transitively on the positive integers	77
5.15	A group which acts 3-transitively, but not 4-transitively on \mathbb{Z}	86
5.16	Grigorchuk groups	89
5.17	Forward orbits of a monoid with 2 generators	91
5.18	Representations of the free group of rank 2	92
5.19	Representations of the modular group $\mathrm{PSL}(2, \mathbb{Z})$	93
6	The Algorithms Implemented in RCWA	95
7	Installation and auxiliary functions	106
7.1	Requirements	106
7.2	Installation	106
7.3	The Info class of the package	106
7.3.1	InfoRCWA	106
7.4	The testing routine	106
7.4.1	RCWATest	106
7.5	Building the manual	107
7.5.1	RCWABuildManual	107
7.6	Loading and saving bitmap pictures	107
7.6.1	SaveAsBitmapPicture (picture, filename)	107
7.7	Running demonstrations	107
7.7.1	RunDemonstration (filename)	107
7.8	Some general utility functions	108

Chapter 1

About the RCWA Package

1.1 Motivation

The development of this package has originally been inspired by the famous $3n + 1$ -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. [Lag06]).

So far, no attempts have been made to investigate the structure of groups whose elements are permutations which are “similar to the Collatz mapping”, i.e. *residue-class-wise affine*.

After having investigated these groups for a couple of years, the author feels that this is a gap which is worth to be filled.

1.2 Purpose of this package

The present scope of computational group theory essentially comprises 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 introduces a new class of groups which are accessible to computational methods, and it therefore extends the current scope of computational group theory as outlined above.
- On the other, residue-class-wise affine groups are interesting mathematical objects in their own right, and this package is intended to serve as a tool for obtaining a better understanding of their rich and often complicated group theoretical and combinatorial structure.

1.3 Groups which this package can deal with

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, thus in particular the modular group $\mathrm{PSL}(2, \mathbb{Z})$.
- 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 here have not yet appeared in print. However they can be found in the preprint [Koh06a], which is available on the author’s homepage. Descriptions of many of the algorithms and methods which are implemented in this package can be found in [Koh07b].

1.4 Scope of this package

This package can be applied in various ways to various different problems, and it is just not possible to say what can be found out with its help about which groups. The best way to get an idea about this is likely to experiment with the examples discussed in this manual and included in the file `pkg/rcwa/examples/examples.g`.

Of course this package often does not provide an out-of-the-box solution for a given problem. Quite often it is possible to find an answer for a given question by using an interactive trial-and-error approach.

With substantial help of this package, the author has found the results mentioned in Section 1.3. Interactive sessions with this package have also lead 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

In this chapter we give the basic definitions, and describe 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 $f : \mathbb{Z} \rightarrow \mathbb{Z}$ 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

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

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 $\text{Sym}(\mathbb{Z})$. We denote the former by $\text{Rcwa}(\mathbb{Z})$, and the latter by $\text{RCWA}(\mathbb{Z})$.

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 an obvious way to suitable rings other than \mathbb{Z} . In fact, this package provides also some support for residue-class-wise affine groups over semilocalizations of \mathbb{Z} and over univariate polynomial rings over finite fields. The former of these rings have been chosen as examples of rings with only finitely prime elements, and the latter 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] = [ar(m), br(m), cr(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> SetName(T,"T"); Display(T);
```

Surjective rcwa mapping of Z with modulus 2

n mod 2		n ^T
0		n/2
1		(3n + 1)/2

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

Bijective rcwa mapping of Z with modulus 3

n mod 3		n ^a
0		2n/3
1		(4n - 1)/3
2		(4n + 1)/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(c1)`

(function)

Returns: The class shift $v_{r(m)}$.

The *class shift* $v_{r(m)}$ 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 `c1` stands for the residue class $r(m)$. Enclosing the argument list in list brackets is permitted.

Example

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

Tame bijective rcwa mapping of Z with modulus 12, of order infinity
```

n mod 12		n^ClassShift(5,12)
0 1 2 3 4 6 7 8 9 10 11		n
5		n + 12

2.2.2 ClassReflection (r, m)

◇ `ClassReflection(r, m)`

(function)

◇ `ClassReflection(c1)`

(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 `c1` stands for the residue class $r(m)$. Enclosing the argument list in list brackets is permitted.

Example

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

Bijective rcwa mapping of Z with modulus 9, of order 2
```

n mod 9		n^ClassReflection(5,9)
0 1 2 3 4 6 7 8		n
5		-n + 10

2.2.3 ClassTransposition (r1, m1, r2, m2)

◇ `ClassTransposition(r1, m1, r2, m2)` (function)

◇ `ClassTransposition(c11, c12)` (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* $\tau_{r_1(m_1), r_2(m_2)} \in \text{RCWA}(\mathbb{Z})$ 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 `c11` and `c12` 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 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));

Bijective rcwa mapping of Z with modulus 10, of order 2
```

n mod 10		n^ClassTransposition(1,2,8,10)
0 2 4 6		n
1 3 5 7 9		5n + 3
8		(n - 3)/5

The set of all class transpositions of the ring of integers generates the simple group $\text{CT}(\mathbb{Z})$ mentioned in Chapter 1. This group has a representation as a GAP object – see `CT` (3.1.2). The set of all generalized class transpositions of \mathbb{Z} generates a simple group as well, cf. [Koh06a].

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(c1, 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 `c1` 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 bijective rcwa mapping of \mathbb{Z}_2 with modulus 2, of order infinity

$n \bmod 2$	$n^{\text{ClassRotation}(1, 2, 1/3)}$
0	n
1	$1/3 n + 2/3$

```
gap> x := Indeterminate(GF(8), 1);; SetName(x, "x");
gap> R := PolynomialRing(GF(8), 1);;
gap> Display(ClassRotation(1, x, Z(8)*One(R)));
```

Bijective rcwa mapping of $\text{GF}(2^3)[x]$ with modulus x , of order 7

$P \bmod x$	$P^{\text{ClassRotation}(\mathbb{Z}(2)^0, x, \mathbb{Z}(2^3))}$
$0 * \mathbb{Z}(2)$	$\mathbb{Z}(2^3)$
$\mathbb{Z}(2^3)^2$	$\mathbb{Z}(2^3)^3$
$\mathbb{Z}(2^3)^4$	$\mathbb{Z}(2^3)^5$
$\mathbb{Z}(2^3)^6$	P
$\mathbb{Z}(2)^0$	$\mathbb{Z}(2^3) * P + \mathbb{Z}(2^3)^3$

There are properties `IsClassShift`, `IsClassReflection`, `IsClassRotation`, `IsClassTransposition` and `IsGeneralizedClassTransposition`, which indicate whether a given rcwa mapping belongs to the corresponding series. By default, class shifts, class reflections, class transpositions and class rotations are given descriptive names of the form `Class...`. They can be given user-defined names upon creation via the option `Name`. Setting the `Name` attribute can be avoided by passing the empty string.

In the sequel, a description of the general-purpose constructor for rcwa mappings is given. This might 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)

\diamond RcwaMapping(<i>R</i> , <i>m</i> , <i>coeffs</i>)	(method)
\diamond RcwaMapping(<i>R</i> , <i>coeffs</i>)	(method)
\diamond RcwaMapping(<i>coeffs</i>)	(method)
\diamond RcwaMapping(<i>perm</i> , <i>range</i>)	(method)
\diamond RcwaMapping(<i>m</i> , <i>values</i>)	(method)
\diamond RcwaMapping(<i>pi</i> , <i>coeffs</i>)	(method)
\diamond RcwaMapping(<i>q</i> , <i>m</i> , <i>coeffs</i>)	(method)
\diamond RcwaMapping(<i>P1</i> , <i>P2</i>)	(method)
\diamond RcwaMapping(<i>cycles</i>)	(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.

In case 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 taken. 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 $\text{Length}(\text{coeffs})$ and coefficients *coeffs*,
- (c) of $R = \mathbb{Z}$ with modulus $\text{Length}(\text{coeffs})$ and coefficients *coeffs*,
- (d) of $R = \mathbb{Z}$, permuting any set $\text{range} + k * \text{Length}(\text{range})$ 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 $\text{Length}(\text{coeffs})$ and coefficients *coeffs* (the set of primes π which denotes the underlying ring is passed as argument *pi*),
- (g) of $R = \text{GF}(q)[x]$ with modulus *m* and coefficients *coeffs*,
- (h) a bijective rcwa mapping which induces a bijection between the partitions *P1* and *P2* of R into residue classes and which is affine on the elements of *P1*, or
- (i) a bijective rcwa mapping with “residue class cycles” given by a list *cycles* of lists of pairwise disjoint residue classes which are to be permuted cyclically, each, 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+Z(2)^0>
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)
<bijective rcwa mapping of Z with modulus 4, of order 3>
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+Z(2)^0>
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)
<bijective rcwa mapping of Z with modulus 16, of order 4>
gap> Cycle(last,ResidueClass(0,2));
[ 0(2), 1(4), 3(8), 7(16) ]

```

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 \mathbb{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. The output of the methods for Display and Print describe 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 a method for LaTeX, respectively LaTeXObj. The output of this method 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(LaTeXObj(T));
n \ \longmapsto \
\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 the LaTeXObj - method described above 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 *. Bijective rcwa mappings 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 – runtime 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;
<bijective rcwa mapping 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 \wedge , 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`. There is also a method for `RestrictedPerm` for computing the restriction of a bijective rcwa mapping 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 $\text{RCWA}(R_1)$ into $\text{RCWA}(R_2)$, 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`. An rcwa mapping is called *integral* if its divisor equals 1. An rcwa mapping is called *balanced* if its multiplier and its divisor have the same prime divisors. 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 $\text{CT}(\mathbb{Z})$ 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);
```

Bijective rcwa mapping of \mathbb{Z} with modulus 5

n mod 5		n^f
0		3n/5
1		(9n + 1)/5
2		(3n - 1)/5
3		(9n - 2)/5
4		(9n + 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(\mathcal{F})

(attribute)

Returns: The coarsest partition of $\text{Source}(\mathcal{F})$ on whose elements the rcwa mapping \mathcal{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);
<bijective rcwa mapping 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(\mathcal{F})

(attribute)

Returns: A list of the sets of fixed points of the affine partial mappings of the rcwa mapping \mathcal{F} in the quotient field of its source.

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

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(\mathcal{F} , p , k)`

(operation)

Returns: The union of the residue classes $r(m)$ such that $p^k \mid a_{r(m)}$ if $k \geq 0$, and the union of the residue classes $r(m)$ such that $p^k \mid c_{r(m)}$ if $k \leq 0$. In this context, m denotes the modulus of \mathcal{F} , and $a_{r(m)}$ and $c_{r(m)}$ denote the coefficients of \mathcal{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 $\text{Mod}(\mathcal{F})$) on which an rcwa mapping \mathcal{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) ]
```

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

2.4.4 Determinant (of an rcwa mapping of \mathbb{Z})

◇ `Determinant(\mathcal{F})`

(method)

Returns: The determinant of the rcwa mapping \mathcal{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 bijective rcwa mappings 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 \mathbb{Z})

◇ **Sign(g)**

(attribute)

Returns: The sign of the bijective rcwa mapping 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

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

(−1)

The sign mapping is an epimorphism from $\text{RCWA}(\mathbb{Z})$ 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 $\text{RCWA}(\mathbb{Z})$ of index 2. The simple group $\text{CT}(\mathbb{Z})$ 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 $\text{RCWA}(\mathbb{Z})$. 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 \mathbb{Z})

◇ **FactorizationIntoCSCRCT(g)**

(attribute)

◇ **Factorization(g)**

(method)

Returns: A factorization of the bijective rcwa mapping 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` (7.3.1) to 2.

By default, prime switches (\rightarrow `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, ClassTransposition(0,6,2,6),
  ClassTransposition(0,6,5,6) ]
```

For purposes of demonstrating the capabilities of the factorization routine, in Section 5.1 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)

(function)

◇ PrimeSwitch(p, k)

(function)

Returns: In the one-argument form the *prime switch* $\sigma_p := \tau_{0(8),1(2p)} \cdot \tau_{4(8),-1(2p)} \cdot \tau_{0(4),1(2p)} \cdot \tau_{2(4),-1(2p)} \cdot \tau_{2(2p),1(4p)} \cdot \tau_{4(2p),2p+1(4p)}$, and in the two-argument form the restriction of σ_p by $n \mapsto kn$.

For an odd prime p , the prime switch σ_p is a bijective rcwa mapping of \mathbb{Z} with modulus $4p$, multiplier p and divisor 2. The key mathematical property of a prime switch is that it is a product of class transpositions, but that its multiplier and its divisor are coprime anyway. Prime switches can be distinguished from other rcwa mappings by their GAP property `IsPrimeSwitch`.

Example

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

Wild bijective rcwa mapping of Z with modulus 12

      n mod 12      |      n^PrimeSwitch(3)
-----+-----
    0              |  n/2
    1  7           |  n + 1
    2  6 10        |  (3n + 4)/2
    3  9           |  n
    4              |  n - 3
    5  8 11        |  n - 1

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

Obtaining a factorization of a bijective rcwa mapping 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 introduced in [Kel99].

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

Example

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

Wild bijective rcwa mapping of \mathbb{Z} with modulus 5

$n \bmod 5$		$n^{\text{mKnot}(5)}$
0		$6n/5$
1		$(4n + 1)/5$
2		$(6n - 2)/5$
3		$(4n + 3)/5$
4		$(6n - 4)/5$

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);
<bijective rcwa mapping of Z with modulus 8>
gap> Display(last);
```

Bijective rcwa mapping of \mathbb{Z} with modulus 8

$n \bmod 8$		n^f
0 1 2 3 4 5		$n + 2$
6		$n - 5$
7		$n - 7$

```
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
```

$n \bmod 2$		n^f
0		$n/2$
1		$(n - 1)/2$

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)>,
  <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 $n^{\wedge}(f^{\wedge}(k-1)) = (c[k][1]*n + c[k][2])/c[k][3]$ 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.

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 [Soi02].

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(\mathcal{F} , m)`

(operation)

Returns: The partition of `AllResidues(Source(\mathcal{F}), m)` corresponding to the weakly connected components of the transition graph of the rcwa mapping \mathcal{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(\mathcal{F} , m)`

(operation)

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

The product of the returned mappings is \mathcal{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(\mathcal{F} , m)`

(operation)

Returns: The transition matrix of the rcwa mapping \mathcal{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

$$M_{r_1(m), r_2(m)} := \frac{|R/mR|}{|R/\hat{m}R|} \cdot |\{r(\hat{m}) \in R/\hat{m}R \mid r \in r_1(m) \wedge r^{\mathcal{F}} \in r_2(m)\}|,$$

where \hat{m} is the product of m and the square of the modulus of \mathcal{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(\mathcal{F}), m)`.

The transition matrix is a weighted adjacency matrix of the corresponding transition graph `TransitionGraph(\mathcal{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 $\text{Mod}(f)$ 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 $\text{Mod}(f)$ 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.

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 introduced in [GT02]. This is $(\sum_{i=1}^l a_i \cdot a_{i \bmod l+1}) / (\sum_{i=1}^l a_i^2)$, 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 $n \mapsto (n/2 \text{ if } n \text{ even}, 3n+1 \text{ if } n \text{ odd})$ 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 nontrivially 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 The categories and families of rcwa mappings

2.9.1 IsRcwaMapping

- ◇ IsRcwaMapping(\mathcal{F}) (filter)
- ◇ IsRcwaMappingOfZ(\mathcal{F}) (filter)
- ◇ IsRcwaMappingOfZ_pi(\mathcal{F}) (filter)
- ◇ IsRcwaMappingOfGFqx(\mathcal{F}) (filter)

Returns: `true` if \mathcal{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`.

2.9.2 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 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
```

There are methods for the operations `View`, `Display`, `Print` and `String` which are applicable to rcwa groups. All rcwa groups over a ring R are subgroups of $\text{RCWA}(R)$. The group $\text{RCWA}(R)$ itself is not finitely generated, thus cannot be constructed as described above. It is handled as a special case:

3.1.1 RCWA (the group of all rcwa permutations of a ring)

◇ **RCWA(R)**

(function)

Returns: The group $\text{RCWA}(R)$ 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.

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

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

◇ $\text{CT}(R)$

(function)

Returns: The group $\text{CT}(R)$ which is generated by all class transpositions of the ring R .

Example

```
gap> CT_Z := CT(Integers);
CT(Z)
gap> IsSimple(CT_Z); # One of a longer list of stored attributes/properties.
true
gap> IsSubgroup(CT_Z, G);
false
```

Another way of constructing an rcwa group is taking the image of an rcwa representation:

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

◇ $\text{IsomorphismRcwaGroup}(G, R)$

(attribute)

◇ $\text{IsomorphismRcwaGroup}(G)$

(attribute)

Returns: A monomorphism from the group G to $\text{RCWA}(R)$ or to $\text{RCWA}(\mathbb{Z})$, 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 $\text{PSL}(2, \mathbb{C})$ to $\text{RCWA}(\mathbb{Z})$.

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

The class of groups which can faithfully be represented as rcwa groups over the integers is closed under taking direct products, under taking wreath products with finite groups and under taking wreath products with the infinite cyclic group $(\mathbb{Z}, +)$. Therefore these operations can be used to build rcwa groups as well:

3.1.4 DirectProduct (for rcwa groups over \mathbb{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 $\text{RCWA}(\mathbb{Z})$. This method chooses to embed the groups $G_1, G_2, G_3 \dots$ via restrictions by $n \mapsto mn$, $n \mapsto mn+1$, $n \mapsto mn+2 \dots$ (\rightarrow 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
```

3.1.5 WreathProduct (for an rcwa group over \mathbb{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 $\text{DegreeAction}(P)$ 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 $\text{DegreeAction}(P)$. The second-mentioned method restricts (\rightarrow Restriction (3.1.6)) the group G to the residue class 3(4), and maps the generator of the infinite cyclic group Z to $\text{ClassTransposition}(0,2,1,2) * \text{ClassTransposition}(0,2,1,4)$.

Example

```
gap> F2 := Image(IsomorphismRcwaGroup(FreeGroup(2)));;
gap> F2wrA5 := WreathProduct(F2,AlternatingGroup(5));;
gap> Embedding(F2wrA5,1);
[ <wild bijective rcwa mapping of Z with modulus 8>,
  <wild bijective rcwa mapping of Z with modulus 8> ] ->
[ <wild bijective rcwa mapping of Z with modulus 40>,
  <wild bijective rcwa mapping of Z with modulus 40> ]
gap> Embedding(F2wrA5,2);
[ (1,2,3,4,5), (3,4,5) ] ->
[ <bijective rcwa mapping of Z with modulus 5, of order 5>,
  <bijective rcwa mapping of Z with modulus 5, of order 3> ]
gap> ZwrZ := WreathProduct(Group(ClassShift(0,1)),Group(ClassShift(0,1)));
<wild rcwa group over Z with 2 generators>
gap> Embedding(ZwrZ,1);
[ ClassShift(0,1) ] ->
[ <tame bijective rcwa mapping of Z with modulus 4, of order infinity> ]
gap> Embedding(ZwrZ,2);
[ ClassShift(0,1) ] ->
[ <wild bijective rcwa mapping of Z with modulus 4> ]
```


Many of the above group constructions are based on certain monomorphisms from the group $\text{RCWA}(R)$ into itself. 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 them:

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 $\text{Induction}(\text{Restriction}(g, f), f) = g$ and $\text{Induction}(\text{Restriction}(G, f), f) = G$. 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
```

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 *integral* respectively *class-wise order-preserving* if all of its elements are so. There are corresponding methods available for `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 $\text{CT}(\mathbb{Z})$.

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,IsIntegral,
>          IsClassWiseOrderPreserving,IsSignPreserving],f->f(G));
[ 24, 2, 2, [ 2, 3 ], false, false, false ]
```

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 some 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 describes the structure of the rcwa group G to some extent.

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 \mathbb{Z} , the infinite dihedral group is denoted by D0 and free groups of rank 2, 3, 4, ... 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 $\langle \text{unknown} \rangle$. In general, the structure of a section denoted by $\langle \text{unknown} \rangle$ can be very complicated and very difficult to exhibit. While for isomorphic finite groups always the same structure description is computed, this cannot be guaranteed for isomorphic rcwa groups.

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"

```

However the extent to which the structure of an rcwa group can be exhibited automatically is certainly 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);
[ ClassTransposition(0, 2, 1, 2), ClassTransposition(0, 3, 1, 3) ] ->
[ (1, 2) (3, 4) (5, 6), (1, 2) (4, 5) ]

```

Next we say a few words about the membership test for rcwa groups. For tame rcwa groups, membership or non-membership can always be decided. For wild groups, membership or non-membership can very often be decided quite quick as well, but not always. 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 direct product of two free groups of rank 2 can faithfully be represented as an rcwa group. According to [Mih58] this implies that in general the membership problem for rcwa groups is algorithmically undecidable.

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));
<bijective rcwa mapping of Z with modulus 48>
gap> ClassTransposition(0,2,1,6)^g = ClassTransposition(1,4,0,8);
true
```

The number of conjugacy classes of $\text{RCWA}(\mathbb{Z})$ 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 ]
```

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)

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.

Both the performance and the returned epimorphism depend on whether the package FR [Bar07] is present or not.

— Example —

```
gap> a := ClassTransposition(2,4,3,4 :Name:="a");;
gap> b := ClassTransposition(4,6,8,12:Name:="b");;
gap> c := ClassTransposition(3,4,4,6 :Name:="c");;
gap> G := Group(a,b,c);
<rcwa group over Z with 3 generators>
gap> phi := EpimorphismFromFpGroup(G,6);
[ a, b, c ] -> [ a, b, c ]
gap> RelatorsOfFpGroup(Source(phi));
[ a^2, b^2, c^2, c*b*c*b*c*b, c*b*c*a*c*b*c*a*c*b*c*a,
  b*a*b*a*b*a*b*a*b*a*b*a ]
```

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 ϕ from a free group to an rcwa group.

The epimorphism ϕ 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` (3.3.5).

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 ϕ .

— Example —

```
gap> a := RcwaMapping([[2,0,3],[4,-1,3],[4,1,3]]);; SetName(a,"a");
gap> b := ClassShift(0,1:Name:="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
<bijjective rcwa mapping of Z with modulus 8>
gap> PreImagesRepresentative(phi,g); # -> a factorization of g
b^-4*a^-1*b^-1*a^-1*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));
<bijjective rcwa mapping of Z with modulus 8>
gap> pre := PreImagesRepresentatives(phi,g);
[ b^-1*a^-1*b^-1*a^-1*b^3*a*b*a*b^-2, b^-1*a^-1*b*a^-1*b^3*a*b^-1*a*b^-2 ]
gap> rel := CyclicallyReducedWord(pre[1]/pre[2]); # -> a nontriv. relation
b^-1*a^-1*b^3*a*b^2*a^-1*b^-3*a*b^-1
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 is often feasible:

Example

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

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> 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
```

RCWA permits drawing 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 produced pictures can be processed further with many common graphics programs:

3.3.2 DrawOrbitPicture (G, p0, r, h, w, colored, palette, filename)

◇ DrawOrbitPicture(*G, p0, r, 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 *r* denotes the radius of the ball around *p0* to be computed. The size of the created 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 drawn. In the former case, *palette* must be a list of triples of integers in the range 0, ..., 255, denoting the RGB values of 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 (7.6.1).

Example

```
gap> PSL2Z := Image(IsomorphismRcwaGroup(FreeProduct(CyclicGroup(2),
>                                                CyclicGroup(3))));
gap> DrawOrbitPicture(PSL2Z, [0,1], 20, 512, 512, false, fail, "example1.bmp");
gap> DrawOrbitPicture(PSL2Z, Combinations([1..4], 2), 20, 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:

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

◇ ShortOrbits(*G, S, maxlen*) (operation)

◇ ShortCycles(*g, S, maxlen*) (operation)

◇ ShortCycles(*g, maxlen*) (operation)

Returns: In the first form a list of all finite orbits of the rcwa group *G* of length at most *maxlen* which intersect nontrivially with the set *S*.

In the second form a list of all cycles of the rcwa permutation *g* of length at most *maxlen* which intersect nontrivially with the set *S*.

In the third form a list of all cycles of the rcwa permutation *g* of length at most *maxlen* which do not correspond to cycles consisting of residue classes.

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 ] ]
```

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:

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

◇ `Ball(G, g, r)` (method)

◇ `Ball(G, p, r, action)` (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*.

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 splitted up and returned as a list of spheres.

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) ] ]
```

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:

3.3.5 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 nontrivially.

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);

Bijjective rcwa mapping of Z with modulus 12

-----+-----
n mod 12 | n^f
-----+-----
0 2 3 6 8 11 | n
1 7 10 | n - 3
4 | n + 1
5 9 | n + 4

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);

Bijjective rcwa mapping of Z with modulus 12

-----+-----
n mod 12 | n^f
-----+-----
0 3 6 | 2n/3
1 | (2n - 8)/3
2 8 11 | (4n + 1)/3
4 7 10 | (4n - 1)/3
5 | (4n - 17)/3
9 | (4n - 15)/3

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*b^-1*a*b*a*b^-2*a*b*a^-1*b
gap> elm := w^phi;
<bijjective rcwa mapping 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:

3.3.6 Projections (for an rcwa group and a modulus)

◇ `Projections(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> Projections(G,4);
[ [ ClassTransposition(0,2,1,2), ClassShift(3,4) ] ->
  [ <bijective rcwa mapping of Z with modulus 4>,
    IdentityMapping( Integers ) ],
  [ ClassTransposition(0,2,1,2), ClassShift(3,4) ] ->
  [ <bijective rcwa mapping of Z with modulus 4>,
    <bijective rcwa mapping 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:

3.3.7 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);
<bijective rcwa mapping of Z with modulus 3>
gap> P1^elm = P2;
true
gap> IsTame(elm);
false
gap> elm := RepresentativeAction(RCWA(Integers),P1,P2:IsTame);
<tame bijective rcwa mapping 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])]);
<bijective rcwa mapping 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) ]

```

3.4 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 respected partition of the rcwa group G / of the rcwa permutation g .

A tame element $g \in \text{RCWA}(R)$ permutes a partition of R into finitely many residue classes on all of which it is affine. Given a tame group $G < \text{RCWA}(R)$, 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.

Related attributes are `RespectedPartitionShort` and `RespectedPartitionLong`. The first of these denotes a respected partition consisting of residue classes $r(m)$ where m divides the modulus of G or g , respectively. The second denotes a respected partition consisting of residue classes $r(m)$ where the modulus of G (respectively g) divides m .

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($g, P, \text{OnPoints}$)`.

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);
[ 3(6), 5(6), 0(8), 2(8), 4(8), 6(8), 1(12), 7(12) ]

```

3.4.2 ActionOnRespectedPartition & KernelOfActionOnRespectedPartition

◇ ActionOnRespectedPartition(G) (attribute)

◇ KernelOfActionOnRespectedPartition(G) (attribute)

Returns: The action of the tame rcwa group G on $\text{RespectedPartition}(G)$ or the kernel of this action, respectively.

The method for `KernelOfActionOnRespectedPartition` uses the package `Polycyclic` [EN06]. The rank of the largest free abelian subgroup of the kernel of the action of G on its stored respected partition can be computed by `RankOfKernelOfActionOnRespectedPartition(G)`.

Example

```
gap> G := Group(ClassTransposition(0,4,1,6),ClassShift(0,2));;
gap> H := ActionOnRespectedPartition(G);
Group([ (3,7)(5,8), (3,4,5,6) ])
gap> H = Action(G,P);
true
gap> Size(H);
48
gap> K := KernelOfActionOnRespectedPartition(G);
<rcwa group over Z with 3 generators>
gap> RankOfKernelOfActionOnRespectedPartition(G);
3
gap> Index(G,K);
48
gap> List(GeneratorsOfGroup(K),Factorization);
[ [ ClassShift(0,4)^2 ], [ ClassShift(2,4)^2 ], [ ClassShift(1,6)^2 ] ]
gap> Image(IsomorphismPcpGroup(K));
Pcp-group with orders [ 0, 0, 0 ]
```

Let G be a tame rcwa group over \mathbb{Z} , let \mathcal{P} be a respected partition of G and put $m := |\mathcal{P}|$. Then there is an rcwa permutation g which maps \mathcal{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);
[ 3(6), 5(6), 0(8), 2(8), 4(8), 6(8), 1(12), 7(12) ]
gap> RespectedPartition(G)^IntegralizingConjugator(G);
[ 0(8), 1(8), 2(8), 3(8), 4(8), 5(8), 6(8), 7(8) ]
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);
```

Bijjective rcwa mapping of \mathbb{Z} with modulus 12

$n \bmod 12$	n^f
0 2 4 6 8 10	$3n + 2$
1 5 9	$-n + 2$
3 7	$(n - 7)/2$
11	$(-n + 20)/3$

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(3,4,4,6),
  ClassShift(0,2)^-1, ClassReflection(3,4), ClassReflection(1,4) ]
```

There is an auxiliary function `ClassPairs([R ,] m)`, which is used in this context. In its one-argument form, this function returns a list of 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 $m_1, m_2 \leq m$. In its two-argument form, it does “the equivalent” for the ring R .

Example

```
gap> List(ClassPairs(4), ClassTransposition);
[ ClassTransposition(0,2,1,2), ClassTransposition(0,2,1,4),
  ClassTransposition(0,2,3,4), ClassTransposition(0,3,1,3),
  ClassTransposition(0,3,2,3), ClassTransposition(0,4,1,4),
  ClassTransposition(0,4,2,4), ClassTransposition(0,4,3,4),
  ClassTransposition(1,2,0,4), ClassTransposition(1,2,2,4),
  ClassTransposition(1,3,2,3), ClassTransposition(1,4,2,4),
  ClassTransposition(1,4,3,4), ClassTransposition(2,4,3,4) ]
gap> List(last, TransposedClasses);
[ [ 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) ] ]
```

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 $\text{RCWA}(R)$ and $\text{CT}(R)$ 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 $\text{Rcwa}(R)$. The monoid $\text{Rcwa}(R)$ 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 of all rcwa mappings of a ring)

◇ Rcwa(R)

(function)

Returns: The monoid $\text{Rcwa}(R)$ 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 $\text{Rcwa}(R)$, 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, maxlmg)`

(method)

Returns: A list of finite forward orbits of the rcwa monoid M of length at most $maxlmg$ 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> Display(Action(M,last[1]));
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> quotes := List(orbs,orb->Action(N,orb));
gap> List(quotes,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 splitted 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

Examples

This chapter discusses a number of “nice” 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 interestingness or difficulty.

Please note that now since quite a while no further examples have been added to this chapter, and that the capabilities of this package have been extended considerably in the meantime.

The rcwa mappings defined in this chapter (and in fact many more) can be found in the file `pkg/rcwa/examples/examples.g`, so there is no need to extract them from the manual files. This file can be read into the current GAP session by issuing `RCWAReadExamples()`;

The examples are typically far from discussing the respective aspects exhaustively. 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!

5.1 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> SetName(Collatz,"Collatz"); Display(Collatz);

Rcwa mapping of Z with modulus 3

-----+-----
n mod 3 | n^Collatz
-----+-----
0       | 2n/3
1       | (4n - 1)/3
2       | (4n + 1)/3

gap> ShortCycles(Collatz,[-50..50],50); # There are some finite cycles:
[ [-111, -74, -99, -66, -44, -59, -79, -105, -70, -93, -62, -83 ],
  [-9, -6, -4, -5, -7 ], [-3, -2 ], [-1 ], [ 0 ], [ 1 ], [ 2, 3 ],
  [ 4, 5, 7, 9, 6 ], [ 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);
[ ClassTransposition(0,6,4,6), ClassTransposition(0,6,5,6),
  ClassTransposition(0,6,3,6), ClassTransposition(0,6,1,6),
  ClassTransposition(0,6,2,6), ClassTransposition(2,3,4,6),
  ClassTransposition(0,3,4,6), ClassTransposition(2,3,1,6),
  ClassTransposition(0,3,1,6), ClassTransposition(0,36,35,36),
  ClassTransposition(0,36,22,36), ClassTransposition(0,36,18,36),
  ClassTransposition(0,36,17,36), ClassTransposition(0,36,14,36),
  ClassTransposition(0,36,20,36), ClassTransposition(0,36,4,36),
  ClassTransposition(2,36,8,36), ClassTransposition(2,36,16,36),
  ClassTransposition(2,36,13,36), ClassTransposition(2,36,9,36),
  ClassTransposition(2,36,7,36), ClassTransposition(2,36,6,36),
  ClassTransposition(2,36,3,36), ClassTransposition(2,36,10,36),
  ClassTransposition(2,36,15,36), ClassTransposition(2,36,12,36),
  ClassTransposition(2,36,5,36), ClassTransposition(21,36,28,36),
  ClassTransposition(21,36,33,36), ClassTransposition(21,36,30,36),
  ClassTransposition(21,36,23,36), ClassTransposition(21,36,34,36),
  ClassTransposition(21,36,31,36), ClassTransposition(21,36,27,36),
  ClassTransposition(21,36,25,36), ClassTransposition(21,36,24,36),
```

```

ClassTransposition(26,36,32,36), ClassTransposition(26,36,29,36),
ClassTransposition(10,18,35,36), ClassTransposition(5,18,35,36),
ClassTransposition(10,18,17,36), ClassTransposition(5,18,17,36),
ClassTransposition(8,12,14,24), ClassTransposition(6,9,17,18),
ClassTransposition(3,9,17,18), ClassTransposition(0,9,17,18),
ClassTransposition(6,9,16,18), ClassTransposition(3,9,16,18),
ClassTransposition(0,9,16,18), ClassTransposition(6,9,11,18),
ClassTransposition(3,9,11,18), ClassTransposition(0,9,11,18),
ClassTransposition(6,9,4,18), ClassTransposition(3,9,4,18),
ClassTransposition(0,9,4,18), ClassTransposition(0,6,14,24),
ClassTransposition(0,6,2,24), ClassTransposition(8,12,17,18),
ClassTransposition(7,12,17,18), ClassTransposition(8,12,11,18),
ClassTransposition(7,12,11,18), PrimeSwitch(3)^-1,
ClassTransposition(7,12,17,18), ClassTransposition(2,6,17,18),
ClassTransposition(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)). At the end of Section 5.6, a much smaller factorization task is performed “manually” for purposes of illustration.

5.2 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> SetName(f6,"f6");
gap> Display(f6);

Rcwa mapping of Z with modulus 6

-----+-----
      n mod 6      |      n^f6
-----+-----
      0            | n/6
      1            | (5n + 1)/6
      2            | (7n - 2)/6
      3            | (11n + 3)/6
      4            | (11n - 2)/6
      5            | (11n - 1)/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 f_6 , 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 $\text{Modulus}(f_6)$ 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 f_6 , 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 k th power of the respective mapping to any integer decreases its absolute value.

5.3 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> 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 5.13.

5.4 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> SetName(ML1,"ML1"); SetName(ML2,"ML2");
gap> Display(ML1);
```

Rcwa mapping of $GF(2)[x]$ with modulus x

P mod x		P^{ML1}
0*Z(2)		P/x
Z(2)^0		((x^3+x^2+x+Z(2)^0)*P + Z(2)^0)/x

```
gap> Display(ML2);
```

Rcwa mapping of $GF(2)[x]$ with modulus x

P mod x		P^{ML2}
0*Z(2)		P/x
Z(2)^0		((x^2+Z(2)^0)*P + Z(2)^0)/x

```
gap> List([ML1,ML2],IsSurjective);
[ true, true ]
gap> List([ML1,ML2],IsInjective);
[ false, false ]
gap> traj1 := Trajectory(ML1,One(R),16);
[ Z(2)^0, x^2+x+Z(2)^0, x^4+x^2+x, x^3+x+Z(2)^0, x^5+x^4+x^2, x^4+x^3+x,
  x^3+x^2+Z(2)^0, x^5+x^2+Z(2)^0, x^7+x^6+x^5+x^3+Z(2)^0,
  x^9+x^7+x^6+x^5+x^3+x+Z(2)^0, 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+Z(2)^0,
  x^11+x^8+x^7+x^6+x^4+x+Z(2)^0, 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+Z(2)^0, x^4+x+Z(2)^0, x^5+x^3+x^2+x+Z(2)^0, x^6+x^3+Z(2)^0,
  x^7+x^5+x^4+x^2+x, x^6+x^4+x^3+x+Z(2)^0, x^7+x^4+x^3+x+Z(2)^0,
  x^8+x^6+x^5+x^4+x^3+x+Z(2)^0, x^9+x^6+x^3+x+Z(2)^0,
  x^10+x^8+x^7+x^5+x^4+x+Z(2)^0, x^11+x^8+x^7+x^5+x^4+x^3+x^2+x+Z(2)^0,
  x^12+x^10+x^9+x^8+x^7+x^5+Z(2)^0, x^13+x^10+x^7+x^4+x,
  x^12+x^9+x^6+x^3+Z(2)^0, 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+Z(2)^0 ]
```

Example

Example

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.

5.5 Exploring the structure of a wild rcwa group

In this example, a simple attempt to should be made to investigate the structure of a given wild group by finding orders of torsion elements. In general, determining the structure of a given wild group seems to be a very hard task. First of all, the group in question has to be defined:

Example

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

Rcwa mapping of \mathbb{Z} with modulus 5

$n \bmod 5$	n^u
0	$3n/5$
1	$(9n + 1)/5$
2	$(3n - 1)/5$
3	$(9n - 2)/5$
4	$(9n + 4)/5$

```
gap> nu := ClassShift(0,1);
gap> G := Group(u,nu);
<rcwa group over  $\mathbb{Z}$  with 2 generators>
gap> IsTame(G);
false
```

Now we would like to know which orders torsion elements of G can have – taking a look at the above generators it seems to make sense to try commutators:

Example

```
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 ]
```

Example

```

gap> Display(Comm(u,nu^13));

Bijjective rcwa mapping of Z with modulus 9

-----+-----
n mod 9 | n^f
-----+-----
0 3 6   | n + 5
1 4 7   | 3n - 9
2 8     | n - 11
5       | (n + 16)/3

gap> Order(Comm(u,nu^13));
7
gap> u2 := u^2;
<wild bijective rcwa mapping of Z with modulus 25>
gap> Filtered([1..16],k->IsTame(Comm(u2,nu^k))); # k < 15 -> commutator wild!
[ 15 ]
gap> Order(Comm(u2,nu^15));
infinity
gap> u2nu17 := Comm(u2,nu^17);
<bijjective rcwa mapping of Z with modulus 81>
gap> cycs := ShortCycles(u2nu17,[-100..100],100);;
gap> List(cycs,Length);
[ 72, 72, 73, 72, 73, 72, 72, 73, 72, 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);
<bijjective rcwa mapping of Z with modulus 81>
gap> cycs := ShortCycles(u2nu18,[-100..100],100);;
gap> List(cycs,Length);
[ 22, 22, 22, 21, 22, 22, 22, 21, 21, 22, 22, 21, 22, 21, 22, 22, 21,
  22, 22, 21, 22, 22, 21 ]
gap> Lcm(last);
462
gap> u2nu18^462; # This is an element of order 2*3*7*11 = 462.
IdentityMapping( Integers )
gap> Order(Comm(u2,nu^20));
29
gap> Order(Comm(u2,nu^25));
9
gap> Order(Comm(u2,nu^30));
15

```

Thus even this rather simple-minded approach reveals various different orders of torsion elements, and the involved primes are also not all very “small”.

5.6 A wild rcwa mapping which has only finite cycles

Some wild rcwa mappings of \mathbb{Z} have only finite cycles. In this section, a permutation is examined which can be shown to be such a mapping and which is likely to be something like a “minimal” example.

Over $R = \text{GF}(q)[x]$, the degree function gives rise to a partition of R into finite sets which is left invariant by suitable wild rcwa mappings. Over $R = \mathbb{Z}$ the situation looks different – there is no such “natural” partition into finite sets which can be fixed by a wild rcwa mapping.

Example

```
gap> kappa := RcwaMapping([[1,0,1],[1,0,1],[3,2,2],[1,-1,1],
> [2,0,1],[1,0,1],[3,2,2],[1,-1,1],
> [1,1,3],[1,0,1],[3,2,2],[2,-2,1]]);
gap> SetName(kappa,"kappa");
gap> List([-5..5],k->Modulus(kappa^k));
[ 7776, 1296, 432, 72, 24, 1, 12, 72, 144, 864, 1728 ]
gap> Display(kappa);
```

Bijjective rcwa mapping of \mathbb{Z} with modulus 12

n mod 12		n^kappa
0 1 5 9		n
2 6 10		$(3n + 2)/2$
3 7		$n - 1$
4		$2n$
8		$(n + 1)/3$
11		$2n - 2$

```
gap> List([-32..32],n->Length(Cycle(kappa,n)));
[ 4, 1, 4, 4, 7, 1, 10, 10, 1, 1, 4, 4, 7, 1, 10, 10, 4, 1, 7, 7, 1, 1,
  7, 7, 4, 1, 4, 4, 2, 1, 1, 2, 1, 1, 4, 4, 4, 1, 7, 7, 4, 1, 7, 7, 1,
  1, 10, 10, 7, 1, 4, 4, 7, 1, 10, 10, 1, 1, 4, 4, 4, 1, 13, 13, 7 ]
gap> List([2..14],k->Maximum(List([1..2^k],n->Length(Cycle(kappa,n)))));
[ 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40 ]
gap> List([2..14],k->Length(Cycle(kappa,2^k-2)));
[ 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40 ]
gap> Cycle(kappa,2^12-2);
[ 4094, 6142, 9214, 13822, 20734, 31102, 46654, 69982, 104974, 157462,
  236194, 354292, 708584, 236195, 472388, 157463, 314924, 104975,
  209948, 69983, 139964, 46655, 93308, 31103, 62204, 20735, 41468,
  13823, 27644, 9215, 18428, 6143, 12284, 4095 ]
gap> last mod 12;
[ 2, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 4, 8, 11, 8, 11, 8, 11, 8,
  11, 8, 11, 8, 11, 8, 11, 8, 11, 8, 11, 8, 11, 8, 3 ]
gap> lengthstats := Collected(List(ShortCycles(kappa,[1..12^4],100),
> Length));
[ [ 1, 6912 ], [ 4, 1728 ], [ 7, 864 ], [ 10, 432 ], [ 13, 216 ],
  [ 16, 108 ], [ 19, 54 ], [ 22, 27 ], [ 25, 13 ], [ 28, 7 ], [ 31, 3 ],
  [ 34, 2 ], [ 37, 1 ], [ 40, 1 ] ]
```

We would like to determine a partition of \mathbb{Z} into unions of cycles of equal length:

Example

```
gap> C := [Difference(Integers,MovedPoints(kappa))];; pow := [kappa^0];;
gap> rc := function(r,m) return ResidueClass(r,m); end;;
gap> for i in [1..3] do
>   Add(pow,kappa^i);
>   C[i+1] := Difference(rc(2,4),
>     Union(Union(C{[1..i]}),
>     Union(List([0..i],j->Intersection(
>       rc(2,4)^pow[j+1],
>       rc(2,4)^(pow[i-j+1]^(-1))))));
>   od;
gap> C;
[ 1(4) U 0(12) U [ -2 ], 2(24) U 18(24), 6(48) U 38(48) U 10(72) U 58(72)
, <union of 38 residue classes (mod 864)> ]
gap> List(C,S->Length(Cycle(kappa,S)));
[ 1, 4, 7, 10 ]
gap> Cycle(kappa,C[1]);
[ 1(4) U 0(12) U [ -2 ] ]
gap> Cycle(kappa,C[2]);
[ 2(24) U 18(24), 4(36) U 28(36), 8(72) U 56(72), 3(24) U 19(24) ]
gap> cycle7 := Cycle(kappa,C[3]);;
gap> for S in cycle7 do View(S); Print("\n"); od;
6(48) U 38(48) U 10(72) U 58(72)
10(72) U 58(72) U 16(108) U 88(108)
16(108) U 88(108) U 32(216) U 176(216)
11(72) U 59(72) U 32(216) U 176(216)
11(72) U 59(72) U 20(144) U 116(144)
7(48) U 39(48) U 20(144) U 116(144)
6(48) U 7(48) U 38(48) U 39(48)
gap> cycle10 := Cycle(kappa,C[4]);;
gap> for S in cycle10 do View(S); Print("\n"); od;
<union of 38 residue classes (mod 864)>
<union of 38 residue classes (mod 1296)>
<union of 12 residue classes (mod 648)>
<union of 12 residue classes (mod 648)>
<union of 22 residue classes (mod 1296)>
<union of 12 residue classes (mod 432)>
<union of 22 residue classes (mod 864)>
<union of 12 residue classes (mod 288)>
<union of 14 residue classes (mod 288)>
<union of 16 residue classes (mod 288)>
gap> List(cycle10,Density);
[ 19/432, 19/648, 1/54, 1/54, 11/648, 1/36, 11/432, 1/24, 7/144, 1/18 ]
gap> List(last,Float);
[ 0.0439815, 0.029321, 0.0185185, 0.0185185, 0.0169753, 0.0277778,
0.025463, 0.0416667, 0.0486111, 0.0555556 ]
gap> Sum(last2);
47/144
gap> Density(Union(cycle10));
47/432
```

Example

```

gap> P := List(C, S->Union(Cycle(kappa, S))));
gap> for S in P do View(S); Print("\n"); od;
1(4) U 0(12) U [ -2 ]
<union of 18 residue classes (mod 72)>
<union of 78 residue classes (mod 432)>
<union of 282 residue classes (mod 2592)>
gap> P2 := AsUnionOfFewClasses(P[2]);
[ 2(24), 3(24), 18(24), 19(24), 4(36), 28(36), 8(72), 56(72) ]
gap> Permutation(kappa, P2);
(1,5,7,2)(3,6,8,4)
gap> P3 := AsUnionOfFewClasses(P[3]);
[ 6(48), 7(48), 38(48), 39(48), 10(72), 11(72), 58(72), 59(72), 16(108),
  88(108), 20(144), 116(144), 32(216), 176(216) ]
gap> Permutation(kappa, P3);
(1,5,9,13,6,11,2)(3,7,10,14,8,12,4)
gap> P4 := AsUnionOfFewClasses(P[4]);
[ 14(96), 15(96), 78(96), 79(96), 22(144), 23(144), 118(144), 119(144),
  34(216), 35(216), 178(216), 179(216), 44(288), 236(288), 52(324),
  268(324), 68(432), 356(432), 104(648), 536(648) ]
gap> Permutation(kappa, P4);
(1,5,9,15,19,10,17,6,13,2)(3,7,11,16,20,12,18,8,14,4)
gap> List(P, S->Set(List(Intersection([1..12^4], S), n->Length(Cycle(kappa, n)))));
[ [ 1 ], [ 4 ], [ 7 ], [ 10 ] ]
gap> Set(List(Intersection([1..12^4], Difference(Integers, Union(P))),
  n->Length(Cycle(kappa, n))));
[ 13, 16, 19, 22, 25, 28, 31, 34, 37, 40 ]

```

Finally, the permutation `kappa` should be factored into involutions (this time “by hand”, for purposes of illustration):

Example

```

gap> elm1 := kappa;
kappa
gap> Multpk(elm1, 2, 1)^elm1;
8(12)
gap> Multpk(elm1, 2, -1)^elm1;
4(6)
gap> fact1 := ClassTransposition(4, 6, 8, 12);

```

Example

```
gap> elm2 := elm1/fact1;
<bijjective rcwa mapping of Z with modulus 12>
gap> Display(elm2);
```

Bijjective rcwa mapping of Z with modulus 12

n mod 12		n ^f
0 1 4 5 9		n
2 6 10		3n + 2
3 7 11		n - 1
8		(n + 1)/3

```
gap> Multpk(elm2,3,1)^elm2;
8(12)
gap> Multpk(elm2,3,-1)^elm2;
3(4)
gap> fact2 := ClassTransposition(3,4,8,12);;
gap> elm3 := elm2/fact2;
<bijjective rcwa mapping of Z with modulus 4>
gap> Display(elm3);
```

Bijjective rcwa mapping of Z with modulus 4

n mod 4		n ^f
0 1		n
2		n + 1
3		n - 1

```
gap> fact3 := ClassTransposition(2,4,3,4);;
gap> elm4 := elm3/fact3;
IdentityMapping( Integers )
gap> kappafacts := [ fact3, fact2, fact1 ];
[ ClassTransposition(2,4,3,4), ClassTransposition(3,4,8,12),
  ClassTransposition(4,6,8,12) ]
gap> kappa = Product(kappafacts);
true
```


5.7 An abelian rcwa group over a polynomial ring

In this section, a wild rcwa group over $\text{GF}(4)[x]$ should be investigated, which happens to be abelian. Of course in general, rcwa groups also over this ring are usually far from being abelian (see below). We start by defining this group:

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+Z(2)^0>
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],Order);
[ infinity, infinity ]
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
```

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+Z(2)^0, 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+Z(2)^0, 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> 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 causes the group structure to change 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> orb := Orbit(G,Zero(R));;
gap> Length(orb);
87
gap> Collected(List(orb,DegreeOfLaurentPolynomial));
[ [ 1, 2 ], [ 2, 4 ], [ 3, 16 ], [ 4, 64 ], [ infinity, 1 ] ]
gap> H := Action(G,orb);
<permutation group with 2 generators>
gap> IsNaturalAlternatingGroup(H);
true
gap> orb := Orbit(G,x^6);;
gap> Length(orb);
512
gap> H := Action(G,orb);
<permutation group with 2 generators>
gap> IsNaturalSymmetricGroup(H) or IsNaturalAlternatingGroup(H);
false
gap> blk := Blocks(H,[1..512]);;
gap> List(blk,Length);
[ 128, 128, 128, 128 ]
gap> Action(H,blk,OnSets);
Group([ (1,2)(3,4), (1,3)(2,4) ])
```

Thus the modified group has a quotient isomorphic to the alternating group of degree 87, and a quotient isomorphic to some wreath product or a subgroup thereof acting transitively, but not primitively on 512 points.

5.8 A tame group generated by commutators of wild permutations

In this section, we have a look at 3 wild rcwa mappings whose commutators generate tame groups:

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> c := RcwaMapping([[3,0,2],[3, 1,4],[3,0,2],[3,11,4]]);;
gap> SetName(a,"a"); SetName(b,"b"); SetName(c,"c");
gap> List([a,b,c],IsTame);
[ false, false, false ]
gap> ab := Comm(a,b);; ac := Comm(a,c);; bc := Comm(b,c);;
gap> SetName(ab,"[a,b]"); SetName(ac,"[a,c]"); SetName(bc,"[b,c]");
gap> List([ab,ac,bc],Order);
[ 6, 6, 12 ]
```

Now we would like to have a look at $[a,b]$...

Example

```
gap> Display(ab);
```

Bijjective rcwa mapping of \mathbb{Z} with modulus 18, of order 6

n mod 18	$n^{[a,b]}$
0 2 3 8 9 11 12 17	n
1 10	$2n - 5$
4 7 13 16	$n + 3$
5 14	$2n - 4$
6	$(n + 2)/2$
15	$(n - 5)/2$

... form the group generated by $[a,b]$ and $[a,c]$ and compute its action on one of its orbits:

Example

```
gap> G := Group(ab,ac);
<rcwa group over Z with 2 generators>
gap> orb := Orbit(G,1);
[ -15, -12, -7, -6, -5, -4, -3, -2, -1, 1 ]
gap> H := Action(G,orb);
Group([ (2,5,8,10,7,6), (1,3,6,9,4,5) ])
gap> Size(H);
3628800
gap> Size(G); # G acts faithfully on orb.
3628800
```

Hence the group G is isomorphic to the symmetric group on 10 points and acts faithfully on the orbit containing 1. Another question is which groups arise if we take as generators either ab , ac or bc and the involution which maps any integer to its additive inverse:

Example

```
gap> t := ClassReflection(0,1);;
gap> Display(t);
Bijjective rcwa mapping of Z: n -> -n
gap> G := Group(ab,t);
<rcwa group over Z with 2 generators>
gap> Size(G);
7257600
gap> phi := IsomorphismPermGroup(G);
[ [a,b], ClassReflection(0,1) ] ->
[ (1,36,12,27,9,15) (2,34,10,25,7,13) (3,35,11,26,8,14),
  (1,18) (2,17) (3,16) (4,15) (5,14) (6,13) (7,12) (8,11) (9,10) (20,21) (22,
    36) (23,35) (24,34) (25,33) (26,32) (27,31) (28,30) ]
gap> StructureDescription(Image(phi));
"C2 x S10"
```

Thus the group generated by ab and t is isomorphic to $C_2 \times S_{10}$. The next group is an extension of a perfect group of order 960:

Example

```
gap> G := Group(ac,t);;
gap> Size(G);
3840
gap> H := Image(IsomorphismPermGroup(G));;
gap> P := DerivedSubgroup(H);;
gap> Size(P);
960
gap> IsPerfect(P);
true
gap> PerfectGroup(PerfectIdentification(P));
A5 2^4'
```

The last group is infinite:

Example

```
gap> G := Group(bc,t);;
gap> Size(G);
infinity
gap> Order(bc*t);
infinity
gap> Modulus(G);
18
gap> RespectedPartition(G);
[ 1(9), 2(9), 4(9), 5(9), 7(9), 8(9), 0(18), 3(18), 6(18), 9(18),
  12(18), 15(18) ]
gap> ActionOnRespectedPartition(G);
Group([ (1,5,8,2,4,12)(3,9,6,11), (1,6)(2,5)(3,4)(8,12)(9,11) ])
gap> StructureDescription(last);
"S10"
gap> RankOfKernelOfActionOnRespectedPartition(G);
9
```

5.9 Checking for solvability

Is the group generated by the permutations a and b from the last paragraph solvable?

This group is wild. Presently there is no general method available for testing wild rcwa groups for solvability. But nevertheless, for the given group we can obtain a negative answer to this question. 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]]);; SetName(a,"a");
gap> b := RcwaMapping([[3,0,2],[3,13,4],[3,0,2],[3,-1,4]]);; SetName(b,"b");
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);
<bijective rcwa mapping 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, as claimed. 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
```

5.10 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> a2 := LocalizedRcwaMapping(a,2);
<rcwa mapping of  $\mathbb{Z}_{(2)}$  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  $\mathbb{Z}_{(2,3)}$  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 ]
```

But the group structure remains invariant under the “transfer” of a group with prime set $\{2,3\}$ from \mathbb{Z} to $\mathbb{Z}_{(2,3)}$:

Example

```
gap> b23 := SemilocalizedRcwaMapping(b,[2,3]);;
gap> c23 := SemilocalizedRcwaMapping(c,[2,3]);;
gap> ab23 := Comm(a23,b23);
<rcwa mapping of Z_( 2, 3 ) with modulus 18>
gap> ac23 := Comm(a23,c23);
<rcwa mapping of Z_( 2, 3 ) with modulus 18>
gap> G := Group(ab23,ac23);
<rcwa group over Z_( 2, 3 ) with 2 generators>
gap> S := Intersection(Enumerator(Rationals){[1..128]},Z_pi([2,3]));
[ -10, -9, -8, -7, -6, -5, -4, -3, -2, -9/5, -8/5, -10/7, -7/5, -9/7,
  -6/5, -8/7, -1, -6/7, -4/5, -5/7, -3/5, -4/7, -3/7, -2/5, -2/7, -1/5,
  -1/7, 0, 1/11, 1/7, 1/5, 2/7, 2/5, 3/7, 4/7, 3/5, 5/7, 4/5, 6/7, 1,
   8/7, 6/5, 9/7, 7/5, 10/7, 8/5, 9/5, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
gap> orbs := ShortOrbits(G,S,50);;
gap> List(orbs,Length);
[ 10, 1, 10, 1, 10, 10, 1, 10, 10, 10, 10, 10, 10, 1, 10, 10,
  10, 1, 1, 10, 1 ]
gap> ForAll(orbs,orb->IsNaturalSymmetricGroup(Action(G,orb)));
true
```

“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> SetName(v,"v");
gap> Display(v);
```

Rcwa mapping of \mathbb{Z} with modulus 8

n mod 8		n ^v
-----	+	-----
0 2 4 6		6n
1		(n - 7)/2
3 7		n - 1
5		(n + 1)/2

Example

```

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);

Bijective rcwa mapping of Z_( 2 ) with modulus 4

      n mod 4      |      n^f
-----+-----
0      | 1/3 n / 2
1      | 2 n + 7
2      | n + 1
3      | 2 n - 1

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);; SetName(v2w2,"[v2,w2]"); v2w2^-1;;
gap> Display(v2w2);

```

Bijective rcwa mapping of Z_(2) with modulus 8

n mod 8		n^[v2,w2]
-----+-----		
0 3 4 7		n
1		n + 4
2 6		3 n
5		n - 4

Again, viewed as an rcwa mapping of the integers the commutator given at the end of the example would not be surjective.

5.11 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 := 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> SetName(x,"x"); Order(x); Display(x);
```

Bijjective rcwa mapping of \mathbb{Z} with modulus 32, of order 257

$n \bmod 32$	n^x
0	$16n + 2$
1 3 5 7 9 11 13 15 17 19 21	
23 25 27 29	$16n + 18$
2 4 6 8 10 12 14	$n + 16$
16	$n/16$
18 20 22 24 26 28 30	$n - 14$
31	$n - 31$

```
gap> Cycle(x,[1],0);
[ 0, 2, 18, 4, 20, 6, 22, 8, 24, 10, 26, 12, 28, 14, 30, 16, 1, 34, 50,
36, 52, 38, 54, 40, 56, 42, 58, 44, 60, 46, 62, 48, 3, 66, 82, 68, 84,
70, 86, 72, 88, 74, 90, 76, 92, 78, 94, 80, 5, 98, 114, 100, 116, 102,
118, 104, 120, 106, 122, 108, 124, 110, 126, 112, 7, 130, 146, 132,
148, 134, 150, 136, 152, 138, 154, 140, 156, 142, 158, 144, 9, 162,
178, 164, 180, 166, 182, 168, 184, 170, 186, 172, 188, 174, 190, 176,
11, 194, 210, 196, 212, 198, 214, 200, 216, 202, 218, 204, 220, 206,
222, 208, 13, 226, 242, 228, 244, 230, 246, 232, 248, 234, 250, 236,
252, 238, 254, 240, 15, 258, 274, 260, 276, 262, 278, 264, 280, 266,
282, 268, 284, 270, 286, 272, 17, 290, 306, 292, 308, 294, 310, 296,
312, 298, 314, 300, 316, 302, 318, 304, 19, 322, 338, 324, 340, 326,
342, 328, 344, 330, 346, 332, 348, 334, 350, 336, 21, 354, 370, 356,
372, 358, 374, 360, 376, 362, 378, 364, 380, 366, 382, 368, 23, 386,
402, 388, 404, 390, 406, 392, 408, 394, 410, 396, 412, 398, 414, 400,
25, 418, 434, 420, 436, 422, 438, 424, 440, 426, 442, 428, 444, 430,
446, 432, 27, 450, 466, 452, 468, 454, 470, 456, 472, 458, 474, 460,
476, 462, 478, 464, 29, 482, 498, 484, 500, 486, 502, 488, 504, 490,
506, 492, 508, 494, 510, 496, 31 ]
gap> Length(last);
257
```

5.12 The behaviour of the moduli of powers

In this section some examples are given, which illustrate how different the series of the moduli of powers of a given rcwa mapping of the integers can look like.

Example

```
gap> List([0..4], i->Modulus(a^i));
[ 1, 4, 16, 64, 256 ]
gap> List([0..6], i->Modulus(ab^i));
[ 1, 18, 18, 18, 18, 18, 1 ]
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, 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> SetName(z, "z");
gap> IsBijective(z);
true
gap> Display(z);
```

Bijective rcwa mapping of \mathbb{Z} with modulus 16

n mod 16		n^z
0		$2n + 1$
1 5 9 10		$n + 1$
2		$2n - 1$
3		$2n - 2$
4 8		$(n + 6)/2$
6		$(n - 6)/2$
7		$2n + 5$
11		$2n - 5$
12 14		n
13		$n - 4$
15		$2n - 10$

Example

```

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 ]
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> SetName(e1, "e1"); SetName(e2, "e2"); Display(e2);

```

Bijective rcwa mapping of \mathbb{Z} with modulus 8, of order infinity

n mod 8		n ^{e2}
0 4		n + 4
1 5		2n
2		n/2
3 6 7		n

```

gap> e2^2 = Restriction(RcwaMapping([[1,2,1]]), RcwaMapping([[4,0,1]]));
true

```

5.13 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)>
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;
```

Example

```

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 ]

```

5.14 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 wild mappings

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)>
<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)>
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)>
<union of 24 residue classes (mod 777600)>
<union of 12 residue classes (mod 777600)>
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

```

Example

```

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)>
<union of 17 residue classes (mod 720)>
<union of 16 residue classes (mod 720)>
<union of 14 residue classes (mod 720)>
<union of 8 residue classes (mod 720)>
<union of 7 residue classes (mod 720)>
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)>
<union of 16 residue classes (mod 5760)>
<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)>
<union of 16 residue classes (mod 51840)>
<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)>
<union of 10 residue classes (mod 259200)>
<union of 7 residue classes (mod 259200)>
13414(129600) U 2164(259200) U 66964(259200) U 228964(259200)
2164(259200) U 66964(259200) U 228964(259200)
[ ]

```

Example

```

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);

```

Example

```

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)>
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)>
<union of 15 residue classes (mod 576)>
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)>
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)>
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)>
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)>
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^-2*u*a^-1*u*a^-1*u*a^-1*u*a^-1*u^-1*a, a^4*u^-2*a^-4, a^-1*u^-4*a,
  u^2*a^-1*u^2*a^-1*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^3*u^2*a*u^2, a*u^-4*a*u^-4*a^-2, u^-2*a*u^2*a*u^-2, u^-4*a^2*u^2 ]

```

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

In this section, we would like to show that the wild group G generated by the two tame mappings $n \mapsto n + 1$ 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);
```

Wild rcwa group over \mathbb{Z} , generated by

[
Tame bijective rcwa mapping of \mathbb{Z} : $n \rightarrow n + 1$

Bijjective rcwa mapping of \mathbb{Z} with modulus 4, of order 2

$n \bmod 4$	$n^{\wedge}\text{ClassTransposition}(1,2,0,4)$
0	$(n + 2)/2$
1 3	$2n - 2$
2	n

]

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 ] ]
```

Example

```
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
```

— Example —

```
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 $n \bmod 8$ 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.

5.16 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]);
<bijective rcwa mapping of Z with modulus 4, of order 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 ]
```

5.17 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
```

5.18 Representations of the free group of rank 2

The free group of rank 2 embeds into $\text{RCWA}(\mathbb{Z})$ – 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 $\text{PSL}(2, \mathbb{C})$ to $\text{RCWA}(\mathbb{Z})$ (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);
[ ClassTransposition(0,2,1,2), ClassTransposition(3,4,5,8),
  ClassTransposition(0,2,1,8) ]
gap> Factorization(gamma2);
[ ClassTransposition(0,2,1,2), ClassTransposition(1,4,7,8),
  ClassTransposition(0,2,3,8) ]
```

The above construction is used by `IsomorphismRcwaGroup` (3.1.3) 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 X_1 and X_2 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);; SetName(F2,"F_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 r_1 and r_2 for modulus 4 may help understanding what is actually done in this calculation. It is easy to see that the group generated by r_1 and r_2 is *not* free:

Example

```
gap> Order(r1/r2);
3
```

5.19 Representations of the modular group $\mathrm{PSL}(2,\mathbb{Z})$

The modular group $\mathrm{PSL}(2,\mathbb{Z})$ embeds in the group generated by all class transpositions as well. We give an embedding, and check that it really is one by applying the Table Tennis Lemma as in the previous section:

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 ]
```

Example

```

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 $\text{PSL}(2, \mathbb{Z})$ can be obtained by using RCWA's general method for `IsomorphismRcwaGroup` for free products of finite groups:

Example

```

gap> Display(Image(IsomorphismRcwaGroup(FreeProduct(CyclicGroup(3),
>                                          CyclicGroup(2)))));

```

Wild rcwa group over \mathbb{Z} , generated by

[

Bijjective rcwa mapping of \mathbb{Z} with modulus 4

$n \bmod 4$	n^f
0	$n + 2$
1 3	$2n - 2$
2	$n/2$

Bijjective rcwa mapping of \mathbb{Z} with modulus 2

$n \bmod 2$	n^f
0	$n + 1$
1	$n - 1$

]

Chapter 6

The Algorithms Implemented in RCWA

This chapter lists brief descriptions of many of the algorithms and methods implemented in this package. These descriptions are kept very informal and short, 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 storing or recalling one or several values, 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 many of the algorithms and methods can be found in [Koh07b].

ActionOnRespectedPartition(G) “Straightforward” after having computed a respected partition by `RespectedPartition`. One only needs to know how to compute images of residue classes under affine mappings.

Ball(G, g, r) “Straightforward”.

Ball(G, p, r, act) “Straightforward”.

ClassPairs Run over all 4-tuples, and filter by divisibility criteria, size comparisons, ordering of the entries etc.

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 [Koh06a].

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].

DirectProduct($G1, G2, \dots$) Restricts the groups $G1, G2, \dots$ to disjoint residue classes. See Restriction and Corollary 2.3.3 in [Koh05].

Display(f) “Trivial”.

Divisor(f) Lcm of coefficients, as indicated.

DrawOrbitPicture Compute spheres of radius $1, \dots, r$ 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) If the package FR [Bar07] is loaded, then use its function `FindGroupRelations` to find relations. Otherwise proceed as follows: First compute the ball of radius r around 1 in the free group whose rank is the number of stored generators of G . Then compute the images of the elements of that ball under the natural projection onto the group G . Take pairs of elements of the ball whose images coincide, and add their quotients to the set of known relations. For images which have finite order, add the corresponding power relations. Finally, regardless of whether FR is present or not, simplify the finitely presented group with the determined relations by the operation `IsomorphismSimplifiedFpGroup` from the GAP Library, and return the natural epimorphism from it to G .

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.

FactorizationIntoCSRCT(g) This uses a rather sophisticated method which 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 almost similarly easily 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”.

GluckTaylorInvariant(a) Evaluation of the given expression.

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 g_1 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 $k := g/g_1$. 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` [EN06], 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 runtime 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 nontrivially. 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 $f * g * \text{RightInverse}(f)$.

`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 $\text{Mod}(f)$). 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.

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) 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) Tests whether the first entry of all coefficient triples is positive.

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”.

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 bijective rcwa mappings 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 $\text{PSL}(2, \mathbb{C})$ to $\text{RCWA}(\mathbb{Z})$. As an equivalent for the closed discs used there, the method takes the residue classes modulo two times the rank of the free group.

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 $\text{RespectedPartition}(G)$ is a subgroup of $\text{ActionOnRespectedPartition}(G)$. 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 [Koh06b]. 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`.

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` [EN06] 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.

- LaTeXObj(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.
- LocalizedRcwaMapping(f, p)** “Trivial”.
- 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 nontrivially with their images under f .
- 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.
- NrConjugacyClassesOfRCWAZOfOrder(ord)** The class numbers are taken from Corollary 2.7.1 in [Koh05].
- 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 \mathcal{P} of G . Use \mathcal{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 \mathcal{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].
- PreImage(f, S)** See Image.

PreImagesRepresentative(ϕ, g), PreImagesRepresentatives(ϕ, g) As indicated in the documentation of these methods. The underlying idea to successively compute two balls around 1 and g until they intersect nontrivially is standard in computational group theory. For rcwa groups it would mean wasting both memory and runtime 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].

Projections(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.

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) This performs basically the first part of the computations done by `KernelOfActionOnRespectedPartition`.

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].

RcwaGroupByPermGroup(G) Uses `RcwaMapping`, Part (d).

RcwaMapping (a)-(c): “trivial”, (d): $n^{\text{perm}} - n$ 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].

RepresentativeAction($G, src, dest, act$), RepresentativeActionPreImage
As indicated in the documentation of these methods. The underlying idea to successively compute two balls around src and $dest$ until they intersect nontrivially is standard in computational group theory. Words standing for products of generators of G are stored for any 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) Uses the algorithm described in the proof of Theorem 2.5.8 in [Koh05].

RespectsPartition(G, P) “Straightforward”.

RestrictedPerm(g, S) “Straightforward”.

Restriction(g, f) Computes the action of $\text{RightInverse}(f) * g * f$ on the image of f .

Restriction(G, f) Gets a set of generators by applying $\text{Restriction}(g, f)$ 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 \mathcal{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 \mathcal{P} by decomposing any $r_i(m_i) \in \mathcal{P}$ into residue classes $(\text{mod } km_i)$. Finally conjugate the resulting permutation back.

Other cases seem to be more difficult and are currently not covered.

RotationFactor(g) “Trivial”.

SemilocalizedRcwaMapping(f, pi) “Trivial”.

ShortCycles($f, maxlng$) Looks for fixed points of affine partial mappings of powers of f .

ShortOrbits($G, S, maxlng$) “Straightforward”.

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.

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 .

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.

Support(G) “Straightforward”.

Trajectory(f, n, \dots) 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 `DegreeAction(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.

Chapter 7

Installation and auxiliary functions

7.1 Requirements

The RCWA package needs at least GAP 4.4.7, ResClasses 2.5.1, GRAPE 4.0 [Soi02], Polycyclic 2.1 [EN06] and GAPDoc 1.0 [LN07]. 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 can be used under UNIX, under Windows and on the MacIntosh. It is completely written in the GAP language and does neither contain nor require external binaries. In particular, warnings concerning missing binaries issued by GRAPE or other packages can safely be ignored.

7.2 Installation

Like any other GAP package, RCWA must be 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");`.

7.3 The Info class of the package

7.3.1 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)`.

7.4 The testing routine

7.4.1 RCWATest

◇ RCWATest() (function)

Returns: Nothing.

Performs tests of the RCWA package. Errors, i.e. differences to the correct results of the test computations, are reported. The processed test files are in the directory `pkg/rcwa/tst`.

7.5 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.

7.5.1 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 systems and requires \PDF\LaTeX .

7.6 Loading and saving bitmap pictures

RCWA provides functions to create bitmap picture files from suitable pixel matrices and vice versa. The author has successfully tested this feature both under Linux and under Windows, and the produced pictures can be processed further with many common graphics programs:

7.6.1 SaveAsBitmapPicture (picture, filename)

◇ `SaveAsBitmapPicture(picture, filename)` (function)

Returns: Nothing.

Writes the pixel matrix *picture* to a bitmap- (bmp-) picture file named *filename*. The file-name should include the entire pathname. The argument *picture* can be a GF(2) matrix, in which case a monochrome picture file is generated. In this case, zeros stand for black pixels and ones stand for white pixels. The argument *picture* can also be an integer matrix, in which case a 24-bit True Color picture file is generated. In this case, the entries of the matrix are supposed to be integers $n = 65536 \cdot \text{red} + 256 \cdot \text{green} + \text{blue}$ in the range $0, \dots, 2^{24} - 1$ specifying the RGB values of the colors of the pixels.

The picture can be read back into GAP by the function `ReadFromBitmapPicture(filename)`.

Example

```
gap> color := n->32*(n mod 8)+256*32*(Int(n/8) mod 8)+65536*32*Int(n/64);;
gap> picture := List([1..512], y->List([1..512], x->color(Gcd(x,y)-1)));;
gap> SaveAsBitmapPicture(picture, "~/images/gcd.bmp");
```

7.7 Running demonstrations

RCWA provides a routine to run demonstrations of its functionality or of other features of GAP. It is intended for being used in talks.

7.7.1 RunDemonstration (filename)

◇ `RunDemonstration(filename)` (function)

Returns: Nothing.

This function executes the code in the file named *filename*. It shows a command and the corresponding output, waits for a keystroke, shows the next command and the corresponding output, waits again for a keystroke, and so on until the end of the file. The demonstration can be stopped by pressing `q`. The function is adapted from the function `Demonstration` in the file `lib/demo.g` of the main GAP distribution.

7.8 Some general utility functions

RCWA introduces a couple of small utility functions which can be used in a more general context: The function `GeneratorsAndInverses(G)` returns a list of generators of *G* and their inverses, `EpimorphismByGenerators(G,H)` is a shorthand for `GroupHomomorphismByImages(G,H, GeneratorsOfGroup(G), GeneratorsOfGroup(H))` (there is also an NC version of this), the function `ListOfPowers(g,exp)` returns the list $[g, g^2, \dots, g^{\text{exp}}]$ of powers of *g*, the function `AllProducts(l,k)` returns the list of all products of *k* entries of the list *l*, the function `DifferencesList(l)` returns the list of differences of consecutive entries of the list *l*, and the function `FloatQuotients(l)` returns the list of floating point approximations of quotients of consecutive entries of the list *l*.

There are also methods `EquivalenceClasses(l,inv)` and `EquivalenceClasses(l,rel)`, which decompose a list *l* into equivalence classes under an equivalence relation. The equivalence relation is given either as a function *inv* computing a class invariant of a given list entry or as a function *rel* which takes as arguments two list entries and returns either `true` or `false` depending on whether the arguments belong to the same equivalence class or not.

References

- [And00] P. Andaloro. On total stopping times under $3x + 1$ iteration. *Fibonacci Quarterly*, 38:73–78, 2000. [56](#)
- [Bar07] L. Bartholdi. *FR – Computations with functionally recursive groups. Version 0.714285*, 2007. GAP package, <http://mad.epfl.ch/~laurent/FR/>. [37](#), [96](#)
- [dlH00] P. de la Harpe. *Topics in Geometric Group Theory*. Chicago Lectures in Mathematics, 2000. [31](#), [92](#), [93](#), [100](#)
- [EN06] B. Eick and W. Nickel. *Polycyclic – Computation with polycyclic groups; Version 2.1*, 2006. GAP package, <http://www.gap-system.org/Packages/polycyclic.html>. [45](#), [97](#), [101](#), [106](#)
- [Gri80] R. I. Grigorchuk. Burnside’s problem on periodic groups. *Functional Anal. Appl.*, 14:41–43, 1980. [89](#), [90](#)
- [GT02] D. Gluck and B. D. Taylor. A new statistic for the $3x + 1$ problem. *Proc. Amer. Math. Soc.*, 130(5):1293–1301, 2002. [27](#)
- [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. [6](#)
- [Kel99] T. P. Keller. Finite cycles of certain periodically linear permutations. *Missouri J. Math. Sci.*, 11(3):152–157, 1999. [21](#), [22](#), [102](#)
- [Koh05] S. Kohl. *Restklassenweise affine Gruppen*. Dissertation, Universität Stuttgart, 2005. <http://deposit.ddb.de/cgi-bin/dokserv?idn=977164071>. [19](#), [20](#), [36](#), [43](#), [44](#), [45](#), [96](#), [97](#), [99](#), [101](#), [102](#), [103](#), [104](#)
- [Koh06a] S. Kohl. A simple group generated by involutions interchanging residue classes of the integers, 2006. <http://www.cip.mathematik.uni-stuttgart.de/~kohlsn/preprints/simplegp.pdf>. [7](#), [11](#), [95](#)
- [Koh06b] S. Kohl. Wildness of iteration of certain residue-class-wise affine mappings. *Adv. in Appl. Math.*, 2006. doi:10.1016/j.aam.2006.08.003. [101](#)
- [Koh07a] S. Kohl. Graph theoretical criteria for the wildness of residue-class-wise affine permutations, 2007. <http://www.cip.mathematik.uni-stuttgart.de/~kohlsn/preprints/graphcrit.pdf>. [101](#)

- [Koh07b] S. Kohl. A new class of groups which are accessible to computational methods, 2007. <http://www.cip.mathematik.uni-stuttgart.de/~kohlsn/preprints/compute.pdf>. 7, 95
- [Lag06] J. C. Lagarias. $3x+1$ problem annotated bibliography, 2006. <http://arxiv.org/abs/math.NT/0309224>. 6
- [LN07] F. Lübeck and M. Neunhöffer. *GAPDoc (Version 1.0)*. RWTH Aachen, 2007. GAP package, <http://www.gap-system.org/Packages/gapdoc.html>. 106
- [Mih58] K. A. Mihailova. The occurrence problem for direct products of groups. *Dokl. Acad. Nauk. SSSR*, 119:1103–1105, 1958. 35
- [ML87] K. R. Matthews and G. M. Leigh. A generalization of the Syracuse algorithm in $\text{GF}(q)[x]$. *J. Number Theory*, 25:274–278, 1987. 57
- [Soi02] L. Soicher. *GRAPE – GRaph Algorithms using PERmutation groups (Version 4.1)*. Queen Mary, University of London, 2002. GAP package, <http://www.gap-system.org/Packages/grape.html>. 25, 106

Index

- ActionOnRespectedPartition
 - for a tame rcwa group, [45](#)
- AllProducts, [108](#)
- balanced
 - definition, [17](#)
- Ball
 - for group, element and radius, [41](#)
 - for group, point, radius and action, [41](#)
 - for monoid, element and radius, [51](#)
 - for monoid, point, radius and action, [51](#)
- ClassPairs
 - m, [46](#)
 - R, m, [46](#)
- ClassReflection
 - cl, [10](#)
 - r, m, [10](#)
- ClassRotation
 - cl, u, [12](#)
 - r, m, u, [12](#)
- ClassShift
 - cl, [10](#)
 - r, m, [10](#)
- ClassTransposition
 - cl1, cl2, [11](#)
 - r1, m1, r2, m2, [11](#)
- ClassWiseConstantOn, [19](#)
- ClassWiseOrderPreservingOn, [19](#)
- ClassWiseOrderReversingOn, [19](#)
- Coefficients
 - of an rcwa mapping, [17](#)
- Collatz conjecture, [6](#)
- Collatz mapping, [6](#)
- CommonRightInverse
 - of two injective rcwa mappings, [23](#)
- CT
 - the group generated by all class transpositions of a ring, [31](#)
- DecreasingOn
 - for an rcwa mapping, [25](#)
- DerivedSubgroup
 - of an rcwa group, [36](#)
- Determinant
 - of an rcwa mapping of \mathbb{Z} , [19](#)
- DifferencesList, [108](#)
- DirectProduct
 - for rcwa groups over \mathbb{Z} , [32](#)
- Display
 - for an rcwa group, [30](#)
 - for an rcwa mapping, [14](#)
 - for an rcwa monoid, [48](#)
- Div
 - for an rcwa group, [33](#)
 - for an rcwa mapping, [17](#)
- Divisor
 - of an rcwa group, [33](#)
 - of an rcwa mapping, [17](#)
- divisor
 - definition, [8](#)
- DrawOrbitPicture
 - G, p0, r, h, w, colored, palette, filename, [40](#)
- EpimorphismByGenerators
 - for two groups, [108](#)
- EpimorphismFromFpGroup
 - for an rcwa group and a search radius, [37](#)
- EquivalenceClasses
 - for a list and a function computing a class invariant, [108](#)
 - for a list and a function describing an equivalence relation, [108](#)
- Exponent
 - of an rcwa group, [36](#)
- ExtRepOfObj, [29](#)
- Factorization

- for an rcwa permutation of Z , 20
- FactorizationIntoCSCRCT
 - for an rcwa permutation of Z , 20
- FactorizationOnConnectedComponents
 - for an rcwa mapping and a modulus, 26
- FixedPointsOfAffinePartialMappings
 - for an rcwa mapping, 18
- FloatQuotients, 108
- GeneratorsAndInverses
 - for a group, 108
- GluckTaylorInvariant
 - of a trajectory, 27
- Group, 30
- GroupByGenerators, 30
- GroupWithGenerators, 30
- GuessedDivergence
 - of an rcwa mapping, 28
- Image
 - of an rcwa mapping, 15
- ImageDensity
 - of an rcwa mapping, 23
- IncreasingOn
 - for an rcwa mapping, 25
- Index
 - for rcwa groups, 36
- Induction
 - of an rcwa group, by an injective rcwa mapping, 33
 - of an rcwa mapping, by an injective rcwa mapping, 33
- Induction
 - for an rcwa monoid, by an injective rcwa mapping, 49
- InfoRCWA, 106
- InjectiveAsMappingFrom
 - for an rcwa mapping, 24
- integral
 - definition, 17
- IntegralConjugate
 - of a tame rcwa group, 45
 - of a tame rcwa permutation, 45
- IntegralizingConjugator
 - of a tame rcwa group, 45
 - of a tame rcwa permutation, 45
- IsBalanced
 - for an rcwa mapping, 17
- IsBijective
 - for an rcwa mapping, 15
- IsClassReflection
 - for an rcwa mapping, 12
- IsClassRotation
 - for an rcwa mapping, 12
- IsClassShift
 - for an rcwa mapping, 12
- IsClassTransposition
 - for an rcwa mapping, 12
- IsClassWiseOrderPreserving
 - for an rcwa group, 33
 - for an rcwa mapping, 17
 - for an rcwa monoid, 49
- IsConjugate
 - for elements of $CT(R)$, 36
 - for elements of $RCWA(R)$, 36
- IsGeneralizedClassTransposition
 - for an rcwa mapping, 12
- IsInjective
 - for an rcwa mapping, 15
- IsIntegral
 - for an rcwa group, 33
 - for an rcwa mapping, 17
 - for an rcwa monoid, 49
- IsNaturalCT, 47
- IsNaturalRCWA, 47
- IsomorphismMatrixGroup
 - for an rcwa group, 36
- IsomorphismPermGroup
 - for a finite rcwa group, 35
- IsomorphismRcwaGroup
 - for a group, 31
 - for a group, over a given ring, 31
- IsPerfect
 - for an rcwa group, 36
- IsPrimeSwitch
 - for an rcwa mapping, 21
- IsRcwaGroup, 47
- IsRcwaGroupOverGFqx, 47
- IsRcwaGroupOverZ, 47
- IsRcwaGroupOverZOrZ_pi, 47
- IsRcwaGroupOverZ_pi, 47
- IsRcwaMapping, 29
- IsRcwaMappingOfGFqx, 29
- IsRcwaMappingOfZ, 29

- IsRcwaMappingOfZOrZ_pi, 29
- IsRcwaMappingOfZ_pi, 29
- IsRcwaMappingStandardRep, 29
- IsSignPreserving
 - for an rcwa group, 33
 - for an rcwa mapping, 17
 - for an rcwa monoid, 49
- IsSolvable
 - for an rcwa group, 36
- IsSubset
 - for two rcwa monoids, 49
- IsSurjective
 - for an rcwa mapping, 15
- IsTame
 - for an rcwa group, 36
 - for an rcwa mapping, 15
 - for an rcwa monoid, 49
- IsTransitive
 - for an rcwa group, on its underlying ring, 39
- KernelOfActionOnRespectedPartition
 - for a tame rcwa group, 45
- LargestSourcesOfAffineMappings
 - for an rcwa mapping, 18
- LaTeX
 - for an rcwa mapping, 14
- LaTeXAndXDVI
 - for an rcwa mapping, 15
- LaTeXObj
 - for an rcwa mapping, 14
- LikelyContractionCentre
 - of an rcwa mapping, 28
- ListOfPowers, 108
- LocalizedRcwaMapping
 - for an rcwa mapping of \mathbb{Z} and a prime, 14
- Loops
 - of an rcwa mapping, 27
- mKnot
 - for an odd integer, 22
- Mod
 - for an rcwa group, 33
 - for an rcwa mapping, 17
- Modulus
 - of an rcwa group, 33
 - of an rcwa mapping, 17
 - of an rcwa monoid, 49
- modulus
 - definition, 8
- ModulusOfRcwaMonoid
 - for an rcwa group, 33
- Monoid, 48
- MonoidByGenerators, 48
- MovedPoints
 - of an rcwa group, 39
 - of an rcwa mapping, 16
- Mult
 - for an rcwa group, 33
 - for an rcwa mapping, 17
- Multiplier
 - of an rcwa group, 33
 - of an rcwa mapping, 17
- multiplier
 - definition, 8
- Multpk
 - for an rcwa mapping, a prime and an exponent, 19
- Name
 - for cs / cr / ct, 12
- NrConjugacyClassesOfRCWAZOfOrder, 36
- ObjByExtRep, 29
- Orbit
 - for an rcwa group and a point, 39
 - for an rcwa group and a set, 39
- OrbitsModulo
 - for an rcwa mapping and a modulus, 26
- OrbitsModulo
 - for an rcwa group and a modulus, 43
- Order
 - of an rcwa permutation, 15
- PermutationOpNC
 - g, P, OnPoints, 44
- PreImage
 - of a residue class union under an rcwa mapping, 16
 - of a set of ring elements under an rcwa mapping, 16
- PreImageElm
 - of a ring element under an rcwa mapping, 16
- PreImagesElm

- of a ring element under an rcwa mapping, 16
- PreImagesRepresentative
 - for an epi. from a free group to an rcwa group, 38
- PreImagesRepresentatives
 - for an epi. from a free group to an rcwa group, 38
- PrimeSet
 - of an rcwa group, 33
 - of an rcwa mapping, 17
 - of an rcwa monoid, 49
- PrimeSwitch
 - p, 21
 - p, k, 21
- Print
 - for an rcwa group, 30
 - for an rcwa mapping, 14
 - for an rcwa monoid, 48
- Projections
 - for an rcwa group and a modulus, 43
- Random
 - CT(R), 46
 - RCWA(R), 46
- RCWA
 - the group of all rcwa permutations of a ring, 30
- Rcwa
 - the monoid of all rcwa mappings of a ring, 49
- rcwa group
 - class-wise order-preserving, 33
 - coercion, 17
 - conjugacy problem, 36
 - definition, 8
 - divisor, 33
 - integral, 33
 - membership test, 35
 - modulus, 33
 - multiplier, 33
 - prime set, 33
 - tame, 8
 - wild, 8
- rcwa mapping
 - arithmetic operations, 15
 - balanced, 17
 - class-wise order-preserving, 17
 - coercion, 17
 - definition, 8
 - divisor, 8
 - images under, 16
 - integral, 17
 - modulus, 8
 - multiplier, 8
 - prime set, 17
 - tame, 8
 - transition graph, 25
 - wild, 8
- rcwa monoid
 - class-wise order-preserving, 49
 - definition, 48
 - integral, 49
 - modulus, 49
 - prime set, 49
 - sign-preserving, 49
 - tame, 49
 - wild, 49
- rcwa monoids
 - membership test, 49
- RCWABuildManual, 107
- RCWAInfo, 106
- RcwaMapping
 - by finite field size, modulus and list of coefficients, 13
 - by list of coefficients, 13
 - by modulus and list of values, 13
 - by permutation and range, 13
 - by residue class cycles, 13
 - by ring and list of coefficients, 13
 - by ring, modulus and list of coefficients, 13
 - by set of noninvertible primes and list of coefficients, 13
 - by two partitions of a ring into residue classes, 13
- RcwaMappingsFamily
 - of a ring, 29
- RCWATest, 106
- ReadFromBitmapPicture
 - filename, 107
- RepresentativeAction
 - for RCWA(R) and 2 partitions of R into residue classes, 43

- G, source, destination, action, [41](#)
- RepresentativeActionPreImage
 - G, source, destination, action, F, [41](#)
- RespectedPartition
 - of a tame rcwa group, [44](#)
 - of a tame rcwa permutation, [44](#)
- RespectedPartitionLong
 - for a tame rcwa group, [44](#)
 - for a tame rcwa permutation, [44](#)
- RespectedPartitionShort
 - for a tame rcwa group, [44](#)
 - for a tame rcwa permutation, [44](#)
- RespectsPartition
 - for an rcwa group, [44](#)
 - for an rcwa permutation, [44](#)
- RestrictedPerm
 - for an rcwa permutation and a residue class union, [16](#)
- Restriction
 - of an rcwa group, by an injective rcwa mapping, [33](#)
 - of an rcwa mapping, by an injective rcwa mapping, [33](#)
- Restriction
 - for an rcwa monoid, by an injective rcwa mapping, [49](#)
- RightInverse
 - of an injective rcwa mapping, [23](#)
- Root
 - k-th root of an rcwa mapping, [22](#)
- RotationFactor
 - of a class rotation, [12](#)
- RunDemonstration
 - filename, [107](#)
- SaveAsBitmapPicture
 - picture, filename, [107](#)
- SemilocalizedRcwaMapping
 - for an rcwa mapping of \mathbb{Z} and a set of primes, [14](#)
- ShortCycles
 - for rcwa permutation and bound on length, [40](#)
 - for rcwa permutation, set of points and bound on length, [40](#)
- ShortOrbits
 - for rcwa group, set of points and bound on length, [40](#)
 - for rcwa monoid, set of points and bound on length, [50](#)
- Sign
 - of an rcwa permutation of \mathbb{Z} , [20](#)
- Sinks
 - of an rcwa mapping, [27](#)
- Size
 - for an rcwa group, [35](#)
 - for an rcwa monoid, [49](#)
- Sources
 - of an rcwa mapping, [27](#)
- SplittedClassTransposition
 - for a class transposition and a number of factors, [11](#)
- String
 - for an rcwa group, [30](#)
 - for an rcwa mapping, [14](#)
 - for an rcwa monoid, [48](#)
- StructureDescription
 - for an rcwa group, [34](#)
- Support
 - of an rcwa group, [39](#)
 - of an rcwa mapping, [16](#)
 - of an rcwa monoid, [49](#)
- tame
 - rcwa group, [8](#)
 - rcwa mapping, [8](#)
- Trajectory
 - for rcwa mapping, starting point, length, [24](#)
 - for rcwa mapping, starting point, length, coeff.-spec., [24](#)
 - for rcwa mapping, starting point, length, modulus, [24](#)
 - for rcwa mapping, starting point, set of end points, [24](#)
 - for rcwa mapping, starting point, set of end points, coeff.-spec., [24](#)
 - for rcwa mapping, starting point, set of end points, modulus, [24](#)
- TransitionGraph
 - for an rcwa mapping and a modulus, [25](#)
- TransitionMatrix
 - for an rcwa mapping and a modulus, [26](#)

TransposedClasses
of a class transposition, [11](#)

View
for an rcwa group, [30](#)
for an rcwa mapping, [14](#)
for an rcwa monoid, [48](#)

wild
rcwa group, [8](#)
rcwa mapping, [8](#)

WreathProduct
for an rcwa group over Z and a permutation group, [32](#)
for an rcwa group over Z and the infinite cyclic group, [32](#)