Iterated monodromy groups

Version 0.0.0

19/11/2012

Laurent Bartholdi

Groups and dynamical systems

Laurent Bartholdi Email: laurent.bartholdi@gmail.com
Homepage: http://www.uni-math.gwdg.de/laurent/

Address: Mathematisches Institut
Bunsenstraße 3-5
D-37073 Göttingen
Germany

Abstract

This document describes the package IMG, which implements in GAP the iterated monodromy groups of Nekrashevych. It depends on the package FR.

For comments or questions on IMG please contact the author; this package is still under development.

Copyright

Acknowledgements

Part of this work is/was supported by the "German Science Foundation".

Colophon

This project started in the mid-1990s, when, as a PhD student I did many calculations with groups generated by automata, and realized the similarities between all calculations; it quickly became clear that these calculations could be done much better by a computer than by a human.

The first routines I wrote constructed finite representations of the groups considered, so as to get insight from fast calculations within GAP. The results then had to be proved correct within the infinite group under consideration, and this often involved guessing appropriate words in the infinite group with a given image in the finite quotient.

Around 2000, I had developed quite a few routines, which I assembled in a GAP package, that dealt directly with infinite groups. This package was primitive at its core, but was extended with various routines as they became useful.

I decided in late 2005 to start a new package from scratch, that would encorporate as much functionality as possible in a uniform manner; that would handle semigroups as well as groups; that could be easily extended; and with a complete, understandable documentation. I hope I am not too far from these objectives.

Contents

Chapter 1

Licensing

Chapter 2

IMG package

2.1 A brief mathematical introduction

This chapter assumes that you have no familiarity with groups generated by automata. If you do, and wish to see their usage within GAP through a sample session, please skip to Section 2.2. For a more thourough introduction on self-similar groups see [BGN03] or [BGŠ03].

We shall here be interested in groups G defined by their action on a regular rooted tree. Let X be a finite set; and let X^* denote the set of words (free monoid) over X. Then X^* naturally has the structure of a regular rooted tree: the root is the empty word, and vertex $v \in X^*$ is connected to vertex vx for all choices of $x \in X$. Each vertex except the root therefore has $\#X + 1$ neighbours.

Let W denote the automorphism group of the graph X^*. Given $a \in W$, we may restrict its action to $X \subset X^*$, and obtain a permutation π_a on X, called the *activity* of a. We may also obtain, for all $x \in X$, a tree automorphism $a_x \in W$, called the *state of a at x*, by the formula

$$(v)a_x = w \quad \text{if} \quad (xv)a = x^{\pi_a} w.$$

The data (a_x, π_a) determine uniquely the automorphism a, and any choice of a_x and π_a defines a tree isometry. We therefore have a group isomorphism

$$\phi : W \to W \wr Sym(X),$$

called the *Wreath recursion*. The image of ϕ is the permutational wreath product $W^X \rtimes Sym(X)$.

The state a_x should be interpreted as the restriction of the action of a on the subtree xX^*; the automorphism a is defined by acting first on each of the subtrees of the form xX^* by its respective state, and then permuting these subtrees according to π_a. The wreath recursion can be iterated on the states of a, to define states a_v for any $v \in X^*$.

The automorphism $a \in W$ may be represented by a graph, as follows. There is one vertex for each state a_v of a, labeled π_{a_v}; and for each $x \in X$ there is one edge from state a_v to state a_{vx}, labeled x. This graph is nothing but a quotient of the regular rooted tree X^*, where vertices v and w are identified if $a_v = a_w$. Again, this graph, with a choice of initial vertex, determines uniquely the automorphism a.

This graph may be conveniently encoded in what is called a *Moore machine*: it consists of a set Q, the vertex set of the graph; an alphabet, X; a 'transition' function $\phi : Q \times X \to Q$, where $\phi(q,x)$ is the endpoint of the edge starting at q and labeled x; and a labeling π of X by the symmetric group on X. We will use the equivalent *Mealy machines*, given by a 'transition' function $\phi : Q \times X \to X \times Q$, encoding both ϕ and π together.

Of particular interest are *finite-state automorphisms*: these are automorphisms whose Mealy machine has finitely many states. The product and inverse of finite-state automorphisms is again finite-state.

A subgroup $G \leq W$ is *self-similar* if $G^{\phi} \subset G \wr Sym(X)$. This is equivalent to asking, for every $a \in G$, that all of its states a_x also belong to G.

The following important properties have also been considered. A subgroup $G \leq W$ is *level-transitive* if its action is transitive on all the G-subsets X^n. It is *weakly branched* if it is level-transitive, and for every $v \in X^*$ there is a non-trivial $a_v \in G$ that fixes $X^* \setminus vX^*$. It is *branched* if furthermore for each $n \in \mathbb{N}$ the group generated by all such a_v for all v of length n has finite index in G.

A self-similar finitely generated group $G \leq W$ is *contracting* if there are constants $K, n \in \mathbb{N}$ and $\lambda < 1$ such that $|a_v| \leq \lambda |a| + K$ for all $a \in G$ and $v \in X^n$; here $|a|$ denotes the minimal number of generators needed to express a. It then follows that there exists a finite set $N \subset G$ such that for all $a \in G$, all but finitely many of the states of a belong to N. The minimal such N is called the *nucleus* of G. Since the states of elements of the nucleus are again in the nucleus, we see that the nucleus is naturally a Mealy machine. By considering all elements of W obtained from this Mealy machine by choosing all possible initial states, we obtain a generating set for G made of all states of a single machine; this is the *group generated* by the machine.

In this package, we are mainly interested in self-similar groups of finite-state automorphisms. The reason is historical: Aleshin [Ale83], and later Grigorchuk [Gri80] and Gupta and Sidki [GS83] constructed peculiar examples of groups using self-similar finite-state automorphisms. All these groups can be defined by drawing a small machine (at most five vertices) and considering the group that they generate.

We assumed for simplicity that the elements a were invertible. Actually, in the definition of Mealy machines it makes sense to accept arbitrary maps, and not necessarily bijections of X as a label at each vertex. One may in this way define peculiar semigroups.

2.2 An example session

This is a brief introduction describing some of the simpler features of the FR package. It assumes you have some familiarity with the theory of groups defined by automata; if not, a brief mathematical introduction may be found in Section 2.1. We show here and comment a typical use of the package.

The package is installed by unpacking the archive in the `pkg/` directory of your GAP installation. It can also be placed in a local directory, which must be added to the load-path by invoking `gap` with the `-l` option.

Example

```
gap> LoadPackage("fr");
--------------------------------------------------------------
Loading FR 0.857142p5 (Functionally recursive and automata groups)
by Laurent Bartholdi (http://www.uni-math.gwdg.de/laurent)
--------------------------------------------------------------
true
```

Many FR groups are predefined by FR, see Chapter **??**. We consider here the *Basilica group*, considered in [GŻ02] and [BV05].

We may start by defining a group: it has two generators a and b, satisfying the specified recursions.

Example

```
gap> B := FRGroup("a=<1,b>(1,2)","b=<1,a>",IsFRMealyElement);
<self-similar group over [ 1 .. 2 ] with 2 generators>
```

```
gap> AssignGeneratorVariables(B);
#I  Assigned the global variables [ a, b ]
```

We have just created the group $B = \langle a, b \rangle$.

Note that this group is predefined as `BasilicaGroup`. We now compute the decompositions of the generators:

Example

```
gap> DecompositionOfFRElement(a); DecompositionOfFRElement(b);
[ [ <2|identity ...>, <2|b> ], [ 2, 1 ] ]
[ [ <2|identity ...>, <2|a> ], [ 1, 2 ] ]
```

Elements are described as words in the generators; they are printed as `<2|a>`, where the 2 reminds of the degree of the tree on which a acts.

The optional argument `IsFRElement` (**??**) tells FR to store elements in this way. This representation is always possible, but it is usually inefficient for calculations. The argument `IsMealyElement` (**??**) forces FR to use a more efficient representation, which in some cases may take an infinite time to set up. With no extra argument, FR does what it thinks is best. The advantages of both representations are sometimes obtained by the argument `IsFRMealyElement` (**??**), which stores both representations.

Elements act on sequences over $\{1,2\}$. The action is computed in the standard manner:

Example

```
gap> 1^a; [1]^a; [1,1]^a;
2
[ 2 ]
[ 2, 1 ]
```

Periodic sequences are also implemented in FR; they are constructed by giving the period and preperiod. The period is printed by preceding it with a "/":

Example

```
gap> v := PeriodicList([1],[2]);
[ 1, / 2 ]
gap> v^a; v^(a^2);
[/ 2 ]
[/ 1, 2 ]
gap> last{[1..10]};
[ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ]
```

Most computations are much more efficient if B's elements are converted to *Mealy representation*,

Example

```
gap> Bm := Image(IsomorphismMealyGroup(B));
<recursive group over [ 1 .. 2 ] with 2 generators>
gap> a := Bm.1; b := Bm.2;
<Mealy element on alphabet [ 1, 2 ] with 3 states>
<Mealy element on alphabet [ 1, 2 ] with 3 states>
```

This could have been done automatically by specifying `IsMealyElement` as last argument in the call to `FRGroup`.

The group B is torsion-free, and its elements are bounded automata. Although torsion-freeness is difficult to check for FR, it can be checked on individual elements:

Example

```
gap> IsBoundedFRSemigroup(Bm);
true
gap> Order(a); Order(b);
infinity
infinity
gap> g := PseudoRandom(B);; Length(InitialState(g));
4679
gap> Order(g); time;
infinity
2599
```

The group B is weakly branched; more precisely, the derived subgroup B' contains $B' \times B'$. To prove that, it suffices to check $[a,b] \times 1 \in B'$ and $1 \times [a,b] \in B'$. These elements are constructed using `VertexElement` (**??**):

Example

```
gap> c := Comm(a,b);
<Mealy element on alphabet [ 1, 2 ] with 9 states>
gap> K := NormalClosure(Bm,Group(c));
<self-similar group over [ 1 .. 2 ] with 3 generators>
gap> VertexElement(1,c) in K; VertexElement(1,c) in K;
true
true
gap> DecompositionOfFRElement(VertexElement(1,c))=[[c,One(Bm)],[1,2]];
true
gap> VertexElement(2,c)=Comm(b,a^2);
true
```

Note that we had to guess the form of the element `VertexElement(2,c)` above. This could have been found out by GAP using `ShortGroupWordInSet` (**??**).

We may also check the relations $[b^p,(b^p)^{a^p}] = 1$ and $[a^{2p},(a^{2p})^{b^p}]$ for p any power of 2:

Example

```
gap> ForAll([0..10],i->IsOne(Comm(b^(2^i),(b^(2^i))^((a^(2^i)))))); time;
true
1361
```

Since the group B is bounded, it is contracting. We compute its nucleus:

Example

```
gap> NucleusOfFRSemigroup(B);
[ <2|identity ...>, <2|b>, <2|b^-1>, <2|a>, <2|a^-1>, <2|b^-1*a>, <2|a^-1*b> ]
```

We then compute the Mealy machine with stateset this nucleus, and draw it graphically (this requires the external programs graphviz and imagemagick):

Example

```
gap> N := NucleusMachine(B);
<Mealy machine on alphabet [ 1, 2 ] with 7 states>
gap> Draw(N);
```

We may also draw powers of the dual automaton: these are approximations to the Schreier graph of B. However, we also construct a smaller Mealy machine with states only a and b, which give better images:

Example

```
gap> Draw(DualMachine(N)^3);
gap> M := AsMealyMachine(FRMachine(a))[1];
<Mealy machine on alphabet [ 1, 2 ] with 3 states>
gap> Draw(DualMachine(M)^4);
```

These Schreier graphs are orbits of the group; they can be displayed as follows:

Example

```
gap> WordGrowth(B:point:=[1,1,1,1],draw);
```

More properties of B can be checked, or experimented with, on its finite quotients obtained by truncating the tree on which B acts at a given length. `PermGroup(B,n)` constructs a permutation group which is the natural quotient of B acting on 2^n points:

Example

```
gap> G := PermGroup(B,7);
<permutation group with 2 generators>
gap> Size(G); LogInt(last,2);
309485009821345068724781056
88
```

We may "guess" the structure of the Lie algebra of B by examining the ranks of the successive quotients along its Jennings series:

Example

```
gap> J := JenningsLieAlgebra(G); time;
<Lie algebra of dimension 88 over GF(2)>
18035
gap> List([1..15],i->Dimension(Grading(J).hom_components(i)));
[ 2, 3, 1, 4, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1 ]
```

The "4" in position 8 of that list should really be a "5"; computations on finite quotients of B usually give lower bounds for invariants of B. In that case, we guess that the ranks behave like a "ruler" function, i.e. that the rank of the homogeneous component of degree i is $2+\nu_2(i)$ if i is a power of 2 and is $1+\nu_2(i)$ otherwise; here $\nu_2(i)$ is the number of times 2 divides i.

Chapter 3

Iterated monodromy groups

Iterated monodromy machines are a special class of group FR machines (see Section **??**) with attribute `IMGRelator` (3.1.4). This attribute records a cyclic ordering of the generators of the machine whose product is trivial.

The interpretation is the following: the machine encodes a *Thurston map*, i.e. a post-critically finite topological branched self-covering of the sphere S^2. Generators of the machine correspond to loops in the fundamental group of the sphere (punctured at post-critical points), that circle once counter-clockwise around a post-critical point. For more details on the connection between self-similar groups and Thurston maps, see [Nek05].

IMG elements are a bit different from group FR elements: while we said a group FR element is trivial if and only if its action on sequences is trivial, we say that an IMG element g is trivial if there exists an integer N such that unfolding N times the recursion for g yields only trivial states (as elements of the underlying free group).

3.1 Creators and operations for IMG machines

3.1.1 IsIMGMachine

▷ `IsIMGMachine(`*m*`)` (filter)
▷ `IsPolynomialFRMachine(`*m*`)` (filter)
▷ `IsPolynomialIMGMachine(`*m*`)` (filter)

The categories of *IMG* and *polynomial* machines. IMG machines are group FR machines with an additional element, their attribute `IMGRelator` (3.1.4); see `AsIMGMachine` (3.1.3).

A polynomial machine is a group FR machine with a distinguished state (which must be a generator of the stateset), stored as the attribute `AddingElement` (**??**); see `AsPolynomialFRMachine` (3.1.9). If it is normalized, in the sense that the wreath recursion of the adding element `a` is `[[a,1,...,1],[d,1,...,d-1]]`, then the basepoint is assumed to be at $+\infty$; the element `a` describes a clockwise loop around infinity; the kth preimage of the basepoint is at $\exp(2i\pi(k-1)/d)\infty$, for $k = 1,\ldots,d$; and there is a direct connection from basepoint k to $k+1$ for all $k = 1,\ldots,d-1$.

The last category is the intersection of the first two.

3.1.2 IMGMachineNC

▷ IMGMachineNC(*fam, group, trans, out, rel*) (operation)

Returns: An IMG FR machine.

This function creates, without checking its arguments, a new IMG machine in family *fam*, stateset *group*, with transitions and output *trans,out*, and IMG relator *rel*.

3.1.3 AsIMGMachine

▷ AsIMGMachine(*m[, w]*) (operation)

Returns: An IMG FR machine.

This function creates a new IMG FR machine, starting from a group FR machine *m*. If a state w is specified, and that state defines the trivial FR element, then it is used as IMGRelator (3.1.4); if the state w is non-trivial, then a new generator f is added to *m*, equal to the inverse of w; and the IMG relator is chosen to be w*f. Finally, if no relator is specified, and the product (in some ordering) of the generators is trivial, then that product is used as IMG relator. In other cases, the method returns fail.

Note that IMG elements and FR elements are compared differently (see the example below); namely, an FR element is trivial precisely when it acts trivially on sequences. An IMG element is trivial precisely when a finite number of applications of free cancellation, the IMG relator, and the decomposition map, result in trivial elements of the underlying free group.

A standard FR machine can be recovered from an IMG FR machine by AsGroupFRMachine (**??**), AsMonoidFRMachine (**??**), and AsSemigroupFRMachine (**??**).

Example

```
gap> m := UnderlyingFRMachine(BasilicaGroup);
<Mealy machine on alphabet [ 1 .. 2 ] with 3 states>
gap> g := AsGroupFRMachine(m);
<FR machine with alphabet [ 1 .. 2 ] on Group( [ f1, f2 ] )>
gap> AsIMGMachine(g,Product(GeneratorsOfFRMachine(g)));
<FR machine with alphabet [ 1 .. 2 ] on Group( [ f1, f2, t ] )/[ f1*f2*t ]>
gap> Display(last);
 G  |              1         2
----+-----------------+---------+
 f1 |          <id>,2      f2,1
 f2 |          <id>,1      f1,2
  t | f2^-1*f1*f2*t,2   f1^-1,1
----+-----------------+---------+
Relator: f1*f2*t
gap> g := AsGroupFRMachine(GuptaSidkiMachine);
<FR machine with alphabet [ 1 .. 3 ] on Group( [ f1, f2 ] )>
gap> m := AsIMGMachine(g,GeneratorsOfFRMachine(g)[1]);
<FR machine with alphabet [ 1 .. 3 ] on Group( [ f1, f2, t ] )/[ f1*t ]>
gap> x := FRElement(g,2)^3; IsOne(x);
<3|identity ...>
true
gap> x := FRElement(m,2)^3; IsOne(x);
<3#f2^3>
false
```

3.1.4 IMGRelator

▷ `IMGRelator(m)` (attribute)

Returns: The relator of the IMG FR machine.

This attribute stores the product of generators that is trivial. In essence, it records an ordering of the generators whose product is trivial in the punctured sphere's fundamental group.

3.1.5 CleanedIMGMachine

▷ `CleanedIMGMachine(m)` (attribute)

Returns: A cleaned-up version of `m`.

This command attempts to shorten the length of the transitions in `m`, and ensure (if possible) that the product along every cycle of the states of a generator is a conjugate of a generator. It returns the new machine.

3.1.6 NewSemigroupFRMachine

▷ `NewSemigroupFRMachine(...)` (attribute)
▷ `NewMonoidFRMachine(...)` (attribute)
▷ `NewGroupFRMachine(...)` (attribute)
▷ `NewIMGMachine(...)` (attribute)

Returns: A new FR machine, based on string descriptions.

This command constructs a new FR or IMG machine, in a format similar to `FRGroup` (**??**); namely, the arguments are strings of the form "gen=<word-1,...,word-d>perm"; each `word-i` is a word in the generators; and `perm` is a transformation, either written in disjoint cycle or in images notation.

Except in the semigroup case, `word-i` is allowed to be the empty string; and the "<...>" may be skipped altogether. In the group or IMG case, each `word-i` may also contain inverses.

In the IMG case, an extra final argument is allowed, which is a word in the generators, and describes the IMG relation. If absent, FR will attempt to find such a relation.

The following examples construct realizable foldings of the polynomial $z^3 + i$, following Cui's arguments.

Example

```
gap> fold1 := NewIMGMachine("a=<,,b,,,B>(1,2,3)(4,5,6)","b=<,,b*a/b,,,B*A/B>",
     "A=<,,b*a,,,B*A>(3,6)","B=(1,6,5,4,3,2)");
gap> <FR machine with alphabet [ 1, 2, 3, 4, 5, 6 ] on Group( [ a, b, A, B ] )/[ a*B*A*b ]>
gap> fold2 := NewIMGMachine("a=<,,b,,,B>(1,2,3)(4,5,6)","b=<,,b*a/b,,,B*A/B>",
     "A=(1,6)(2,5)(3,4)","B=<B*A,,,b*a,,>(1,4)(2,6)(3,5)");;
gap> RationalFunction(fold1); RationalFunction(fold2);
...
```

3.1.7 AsIMGElement

▷ `AsIMGElement(e)` (operation)
▷ `IsIMGElement(e)` (filter)

The category of *IMG elements*, namely FR elements of an IMG machine. See `AsIMGMachine` (3.1.3) for details.

3.1.8 IsKneadingMachine

▷ `IsKneadingMachine(`*m*`)` (property)
▷ `IsPlanarKneadingMachine(`*m*`)` (property)

Returns: Whether *m* is a (planar) kneading machine.

A *kneading machine* is a special kind of Mealy machine, used to describe postcritically finite complex polynomials. It is a machine such that its set of permutations is "treelike" (see [Nek05, §6.7]) and such that each non-trivial state occurs exactly once among the outputs.

Furthermore, this set of permutations is *treelike* if there exists an ordering of the states that their product in that order t is an adding machine; i.e. such that t's activity is a full cycle, and the product of its states along that cycle is conjugate to t. This element t represents the Carathéodory loop around infinity.

Example

```
gap> M := BinaryKneadingMachine("0");
BinaryKneadingMachine("0*")
gap> Display(M);
   |  1     2
---+-----+-----+
 a | c,2   b,1
 b | a,1   c,2
 c | c,1   c,2
---+-----+-----+
gap> IsPlanarKneadingMachine(M);
true
gap> IsPlanarKneadingMachine(GrigorchukMachine);
false
```

3.1.9 AsPolynomialFRMachine

▷ `AsPolynomialFRMachine(`*m*`[, `*adder*`])` (operation)
▷ `AsPolynomialIMGMachine(`*m*`[, `*adder*`[, `*relator*`]])` (operation)

Returns: A polynomial FR machine.

The first function creates a new polynomial FR machine, starting from a group or Mealy machine. A *polynomial* machine is one that has a distinguished adding element, `AddingElement` (**??**).

If the argument is a Mealy machine, it must be planar (see `IsPlanarKneadingMachine` (3.1.8)). If the argument is a group machine, its permutations must be treelike, and its outputs must be such that, up to conjugation, each non-trivial state appears exactly once as the product along all cycles of all states.

If a second argument *adder* is supplied, it is checked to represent an adding element, and is used as such.

The second function creates a new polynomial IMG machine, i.e. a polynomial FR machine with an extra relation among the generators. the optional second argument may be an adder (if *m* is an IMG machine) or a relator (if *m* is a polynomial FR machine). Finally, if *m* is a group FR machine, two arguments, an adder and a relator, may be specified.

A machine without the extra polynomial / IMG information may be recovered using `AsGroupFRMachine` (**??**).

Example

```
gap> M := PolynomialIMGMachine(2,[1/7],[];; SetName(StateSet(M),"F"); M;
<FR machine with alphabet [ 1, 2 ] and adder f4 on F/[ f4*f3*f2*f1 ]>
```

```
gap> Mi := AsIMGMachine(M);
<FR machine with alphabet [ 1, 2 ] on F/[ f4*f3*f2*f1 ]>
gap> Mp := AsPolynomialFRMachine(M);
<FR machine with alphabet [ 1, 2 ] and adder f4 on F>
gap> Mg := AsGroupFRMachine(M);
<FR machine with alphabet [ 1, 2 ] on F>
gap>
gap> AsPolynomialIMGMachine(Mg);
<FR machine with alphabet [ 1, 2 ] and adder f4 on F/[ f4*f3*f2*f1 ]>
gap> AsPolynomialIMGMachine(Mi);
<FR machine with alphabet [ 1, 2 ] and adder f4 on F/[ f4*f3*f2*f1 ]>
gap> AsPolynomialIMGMachine(Mp);
<FR machine with alphabet [ 1, 2 ] and adder f4 on F/[ f4*f3*f2*f1 ]>
gap> AsIMGMachine(Mg);
<FR machine with alphabet [ 1, 2 ] on F4/[ f1*f4*f3*f2 ]>
gap> AsPolynomialFRMachine(Mg);
<FR machine with alphabet [ 1, 2 ] and adder f4 on F4>
```

3.1.10 AddingElement (FR machine)

▷ `AddingElement(`*m*`)` (attribute)

Returns: The relator of the IMG FR machine.

This attribute stores the product of generators that is an adding machine. In essence, it records an ordering of the generators whose product corresponds to the Carathéodory loop around infinity.

The following example illustrates Wittner's shared mating of the airplane and the rabbit. In the machine `m`, an airplane is represented by `Group(a,b,c)` and a rabbit is represented by `Group(x,y,z)`; in the machine `newm`, it is the other way round. The effect of `CleanedIMGMachine` was to remove unnecessary instances of the IMG relator from `newm`'s recursion.

Example

```
gap> f := FreeGroup("a","b","c","x","y","z");;
gap> AssignGeneratorVariables(f);
gap> m := AsIMGMachine(FRMachine(f,[[a^-1,b*a],[One(f),c],[a,One(f)],[z*y*x,
       x^-1*y^-1],[One(f),x],[One(f),y]],[(1,2),(),(),(1,2),(),()]));;
gap> Display(m);
 G |      1             2
---+---------+-------------+
 a |  a^-1,2         b*a,1
 b |  <id>,1           c,2
 c |     a,1         <id>,2
 x | z*y*x,2   x^-1*y^-1,1
 y |  <id>,1           x,2
 z |  <id>,1           y,2
---+---------+-------------+
Relator: z*y*x*c*b*a
gap> iso := GroupHomomorphismByImages(f,f,[a,b^(y^-1),c^(x^-1*y^-1*a^-1),x^(b*a*z*a^-1),y,z^(a^-
gap> newm := CleanedIMGMachine(ChangeFRMachineBasis(m^iso,[a^-1*y^-1,y^-1*a^-1*c^-1]));;
gap> Display(newm);
 G |          1         2
---+-------------+---------+
 a | a^-1*c^-1,2   c*a*b,1
 b |      <id>,1       c,2
```

```
  c |         a,1    <id>,2
  x |       z*x,2    x^-1,1
  y |      <id>,1       x,2
  z |         y,1    <id>,2
----+-------------+---------+
Relator: c*a*b*y*z*x
```

3.1.11 PolynomialFRMachine

▷ `PolynomialFRMachine(d, per[, pre])` (operation)
▷ `PolynomialIMGMachine(d, per[, pre[, formal]])` (operation)
▷ `PolynomialMealyMachine(d, per[, pre])` (operation)

Returns: An IMG FR machine.

This function creates a group, IMG or Mealy machine that describes a topological polynomial. The polynomial is described symbolically in the language of *external angles*. For more details, see [DH84] and [DH85] (in the quadratic case), [BFH92] (in the preperiodic case), and [Poi] (in the general case).

`d` is the degree of the polynomial. `per` and `pre` are lists of angles or preangles. In what follows, angles are rational numbers, considered modulo 1. Each entry in `per` or `pre` is either a rational (interpreted as an angle), or a list of angles $[a_1,\ldots,a_i]$ such that $da_1 = \ldots = da_i$. The angles in `per` are angles landing at the root of a Fatou component, and the angles in `pre` land on the Julia set.

Note that, for IMG machines, the last generator of the machine produced is an adding machine, representing a loop going counterclockwise around infinity (in the compactification of $\mathbb{C}$ by a disk, this loop goes *clockwise* around that disk).

In constructing a polynomial IMG machine, one may specify a boolean flag `formal`, which defaults to `true`. In a *formal* recursion, distinct angles give distinct generators; while in a non-formal recursion, distinct angles, which land at the same point in the Julia set, give a single generator. The simplest example where this occurs is angle $5/12$ in the quadratic family, in which angles $1/3$ and $2/3$ land at the same point – see the example below.

The attribute `Correspondence(m)` records the angles landing on the generators: `Correspondence(m)[i]` is a list `[a,s]` where a is an angle landing on generator `i` and s is `"Julia"` or `"Fatou"`.

If only one list of angles is supplied, then FR guesses that all angles with denominator coprime to n are Fatou, and all the others are Julia.

The inverse operation, reconstructing the angles from the IMG machine, is `SupportingRays` (3.1.12).

Example

```
gap> PolynomialIMGMachine(2,[0],[]); # the adding machine
<FR machine with alphabet [ 1 .. 2 ] on Group( [ f1, f2 ] )/[ f2*f1 ]>
gap> Display(last);
 G  |     1        2
----+--------+--------+
 f1 | <id>,2     f1,1
 f2 |   f2,2   <id>,1
----+--------+--------+
Relator: f2*f1
gap> Display(PolynomialIMGMachine(2,[1/3],[])); # the Basilica
 G  |      1         2
----+---------+---------+
```

```
 f1 | f1^-1,2   f2*f1,1
 f2 |    f1,1    <id>,2
 f3 |    f3,2    <id>,1
----+---------+---------+
Relator: f3*f2*f1
gap> Display(PolynomialIMGMachine(2,[],[1/6])); # z^2+I
 G  |            1         2
----+---------------+---------+
 f1 | f1^-1*f2^-1,2   f2*f1,1
 f2 |          f1,1      f3,2
 f3 |          f2,1    <id>,2
 f4 |          f4,2    <id>,1
----+---------------+---------+
Relator: f4*f3*f2*f1
gap> PolynomialIMGMachine(2,[],[5/12]);
gap> PolynomialIMGMachine(2,[],[5/12]);
<FR machine with alphabet [ 1, 2 ] and adder f5 on Group( [ f1, f2, f3, f4, f5 ] )/[ f5*f4*f3*f2
gap> Correspondence(last);
[ [ 1/3, "Julia" ], [ 5/12, "Julia" ], [ 2/3, "Julia" ], [ 5/6, "Julia" ] ]
gap> PolynomialIMGMachine(2,[],[5/12],false);
<FR machine with alphabet [ 1, 2 ] and adder f4 on Group( [ f1, f2, f3, f4 ] )/[ f4*f3*f2*f1 ]>
gap> Correspondence(last);
[ [ [ 1/3, 2/3 ], "Julia" ], [ [ 5/12 ], "Julia" ], [ [ 5/6 ], "Julia" ] ]
```

The following construct the examples in Poirier's paper:

```
PoirierExamples := function(arg)
    if arg=[1] then
        return PolynomialIMGMachine(2,[1/7],[]);
    elif arg=[2] then
        return PolynomialIMGMachine(2,[],[1/2]);
    elif arg=[3,1] then
        return PolynomialIMGMachine(2,[],[5/12]);
    elif arg=[3,2] then
        return PolynomialIMGMachine(2,[],[7/12]);
    elif arg=[4,1] then
        return PolynomialIMGMachine(3,[[3/4,1/12],[1/4,7/12]],[]);
    elif arg=[4,2] then
        return PolynomialIMGMachine(3,[[7/8,5/24],[5/8,7/24]],[]);
    elif arg=[4,3] then
        return PolynomialIMGMachine(3,[[1/8,19/24],[3/8,17/24]],[]);
    elif arg=[5] then
        return PolynomialIMGMachine(3,[[3/4,1/12],[3/8,17/24]],[]);
    elif arg=[6,1] then
        return PolynomialIMGMachine(4,[],[[1/4,3/4],[1/16,13/16],[5/16,9/16]]);
    elif arg=[6,2] then
        return PolynomialIMGMachine(4,[],[[1/4,3/4],[3/16,15/16],[7/16,11/16]]);
    elif arg=[7] then
        return PolynomialIMGMachine(5,[[0,4/5],[1/5,2/5,3/5]],[[1/5,4/5]]);
    elif arg=[9,1] then
        return PolynomialIMGMachine(3,[[0,1/3],[5/9,8/9]],[]);
    elif arg=[9,2] then
        return PolynomialIMGMachine(3,[[0,1/3]],[[5/9,8/9]]);
```

```
    else
        Error("Unknown Poirier example ",arg);
    fi;
end;
```

3.1.12 SupportingRays

▷ SupportingRays(*m*) (attribute)

Returns: A [degree,fatou,julia] description of *m*.

This operation is the inverse of PolynomialIMGMachine (3.1.11): it computes a choice of angles, describing landing rays on Fatou/Julia critical points.

If there does not exist a complex realization, namely if the machine is obstructed, then this command returns an obstruction, as a record. The field minimal is set to false, and a proper submachine is set as the field submachine. The field homomorphism gives an embedding of the stateset of submachine into the original machine, and relation is the equivalence relation on the set of generators of *m* that describes the pinching.

Example

```
gap> r := PolynomialIMGMachine(2,[1/7],[]);
<FR machine with alphabet [ 1, 2 ] and adder f4 on Group( [ f1, f2, f3, f4 ] )/[ f4*f3*f2*f1 ]>
gap> F := StateSet(r);; SetName(F,"F");
gap> SupportingRays(r);
[ 2, [ [ 1/7, 9/14 ] ], [  ] ] # actually returns the angle 2/7
gap> # now CallFuncList(PolynomialIMGMachine,last) would return the machine r
gap> twist := GroupHomomorphismByImages(F,F,GeneratorsOfGroup(F),[F.1^(F.2*F.1),F.2^F.1,F.3,F.4]
[ f1, f2, f3, f4 ] -> [ f1*f2*f1^-1, f2*f1*f2*f1^-1*f2^-1, f3, f4 ]
gap> List([-5..5],i->2*SupportingRays(r*twist^i)[2][1][1]);
[ 4/7, 5/7, 4/7, 4/7, 5/7, 2/7, 4/7, 4/7, 2/7, 4/7, 4/7 ]
gap> r := PolynomialIMGMachine(2,[],[1/6]);;
gap> F := StateSet(r);;
gap> twist := GroupHomomorphismByImages(F,F,GeneratorsOfGroup(F),[F.1,F.2^(F.3*F.2),F.3^F.2,F.4]
gap> SupportingRays(r);
[ 2, [  ], [ [ 1/12, 7/12 ] ] ]
gap> SupportingRays(r*twist);
[ 2, [  ], [ [ 5/12, 11/12 ] ] ]
gap> SupportingRays(r*twist^2);
rec(
  transformation := [ [ f1, f2^-1*f3^-1*f2^-1*f3^-1*f2*f3*f2*f3*f2, f2^-1*f3^-1*f2^-1*f3*f2*f3*f
          f4 ] -> [ f1, f2, f3, f4 ],
      [ f1^-1*f2^-1*f1^-1*f2^-1*f1*f2*f1*f2*f1, f1^-1*f2^-1*f1^-1*f2*f1*f2*f1, f3, f4 ] ->
        [ f1, f2, f3, f4 ],
      [ f1^-1*f2^-1*f3^-1*f2*f1*f2^-1*f3*f2*f1, f2, f2*f1^-1*f2^-1*f3*f2*f1*f2^-1, f4 ] ->
        [ f1, f2, f3, f4 ], [ f1, f3*f2*f3^-1, f3, f4 ] -> [ f1, f2, f3, f4 ],
      [ f1, f2, f2*f3*f2^-1, f4 ] -> [ f1, f2, f3, f4 ],
      [ f1, f3*f2*f3^-1, f3, f4 ] -> [ f1, f2, f3, f4 ],
      [ f1, f2, f2*f3*f2^-1, f4 ] -> [ f1, f2, f3, f4 ],
      [ f1, f3*f2*f3^-1, f3, f4 ] -> [ f1, f2, f3, f4 ] ], machine := <FR machine with alphabet
    [ 1, 2 ] and adder f4 on Group( [ f1, f2, f3, f4 ] )/[ f4*f3*f2*f1 ]>, minimal := false,
  submachine := <FR machine with alphabet [ 1, 2 ] and adder f3 on Group( [ f1, f2, f3 ] )>,
  homomorphism := [ f1, f2, f3 ] -> [ f1, f2*f3, f4 ],
  relation := <equivalence relation on <object> >, niter := 8 )
```

3.1.13 AsGroupFRMachine (endomorphism)

▷ `AsGroupFRMachine(f)` (attribute)
▷ `AsMonoidFRMachine(f)` (attribute)
▷ `AsSemigroupFRMachine(f)` (attribute)

Returns: An FR machine.

This function creates an FR machine on a 1-letter alphabet, that represents the endomorphism `f`. It is specially useful when combined with products of machines; indeed the usual product of machines corresponds to composition of endomorphisms.

Example

```
gap> f := FreeGroup(2);;
gap> h := GroupHomomorphismByImages(f,f,[f.1,f.2],[f.2,f.1*f.2]);
[ f1, f2 ] -> [ f2, f1*f2 ]
gap> m := AsGroupFRMachine(h);
<FR machine with alphabet [ 1 ] on Group( [ f1, f2 ] )>
gap> mm := TensorProduct(m,m);
<FR machine with alphabet [ 1 ] on Group( [ f1, f2 ] )>
gap> Display(mm);
 G  |         1
----+------------+
 f1 |    f1*f2,1
 f2 | f2*f1*f2,1
----+------------+
```

3.1.14 NormalizedPolynomialFRMachine

▷ `NormalizedPolynomialFRMachine(m)` (attribute)
▷ `NormalizedPolynomialIMGMachine(m)` (attribute)

Returns: A polynomial FR machine.

This function returns a new FR machine, in which the adding element has been put into a standard form $t = [t, 1, \ldots, 1]s$, where s is the long cycle $i \mapsto i - 1$.

For the first command, the machine returned is an FR machine; for the second, it is an IMG machine.

3.1.15 SimplifiedIMGMachine

▷ `SimplifiedIMGMachine(m)` (attribute)

Returns: A simpler IMG machine.

This function returns a new IMG machine, with hopefully simpler transitions. The simplified machine is obtained by applying automorphisms to the stateset. The sequence of automorphisms (in increasing order) is stored as a correspondence; namely, if `n=SimplifiedIMGMachine(m)`, then `m^Product(Correspondence(n))=n`.

Example

```
gap> r := PolynomialIMGMachine(2,[1/7],[]);;
gap> F := StateSet(r);; SetName(F,"F");
gap> twist := GroupHomomorphismByImages(F,F,GeneratorsOfGroup(F),[F.1^(F.2*F.1),F.2^F.1,F.3,F.4]
gap> m := r*twist;; Display(m);
 G  |                    1           2
----+------------------------+------------+
 f1 |          f1^-1*f2^-1,2   f3*f2*f1,1
```

```
 f2 | f1^-1*f2^-1*f1*f2*f1,1       <id>,2
 f3 |          f1^-1*f2*f1,1       <id>,2
 f4 |                   f4,2       <id>,1
----+------------------------+------------+
Adding element: f4
Relator: f4*f3*f2*f1
gap> n := SimplifiedIMGMachine(m);
<FR machine with alphabet [ 1, 2 ] and adder f4 on F>
gap> Display(n);
 G  |            1            2
----+---------------+------------+
 f1 | f2^-1*f1^-1,2   f1*f2*f3,1
 f2 |        <id>,1         f1,2
 f3 |        <id>,1         f2,2
 f4 |          f4,2       <id>,1
----+---------------+------------+
Adding element: f4
Relator: f4*f1*f2*f3
gap> n = m^Product(Correspondence(n));
true
```

3.1.16 Mating

▷ `Mating(`*m1, m2[, formal]*`)` (operation)

Returns: An IMG FR machine.

This function "mates" two polynomial IMG machines.

The mating is defined as follows: one removes a disc around the adding machine in *m1* and *m2*; one applies complex conjugation to *m2*; and one glues the hollowed spheres along their boundary circle.

The optional argument *formal*, which defaults to `true`, specifies whether a *formal* mating should be done; in a non-formal mating, generators of *m1* and *m2* which have identical angle should be treated as a single generator. A non-formal mating is of course possible only if the machines are realizable – see `SupportingRays` (3.1.12).

The attribute `Correspondence` is a pair of homomorphisms, from the statesets of *m1*,*m2* respectively to the stateset of the mating.

Example

```
gap> # the Tan-Shishikura examples
gap> z := Indeterminate(MPC_PSEUDOFIELD);;
gap> a := RootsFloat((z-1)*(3*z^2-2*z^3)+1);;
gap> c := RootsFloat((z^3+z)^3+z);;
gap> am := List(a,a->IMGMachine((a-1)*(3*z^2-2*z^3)+1));;
gap> cm := List(c,c->IMGMachine(z^3+c));;
gap> m := ListX(am,cm,Mating);;
gap> # m[2] is realizable
gap> RationalFunction(m[2]);
((1.66408+I*0.668485)*z^3+(-2.59772+I*0.627498)*z^2+(-1.80694-I*0.833718)*z
  +(1.14397-I*1.38991))/((-1.52357-I*1.27895)*z^3+(2.95502+I*0.234926)*z^2
  +(1.61715+I*1.50244)*z+1)
gap> # m[6] is obstructed
gap> RationalFunction(m[6]);
rec( matrix := [ [ 1/2, 1 ], [ 1/2, 0 ] ], machine := <FR machine with alphabet
```

```
    [ 1, 2, 3 ] on Group( [ f1, f2, f3, g1, g2, g3 ] )/[ f2*f3*f1*g1*g3*g2 ]>,
  obstruction := [ f1^-1*f3^-1*f2^-1*f1*f2*f3*f1*g2^-1*g3^-1*f1^-1*f3^-1*f2^-1,
      f2*f3*f1*f2*f3*f1*g2*f1^-1*f3^-1*f2^-1*f1^-1*f3^-1 ],
  spider := <spider on <triangulation with 8 vertices, 36 edges and
    12 faces> marked by GroupHomomorphismByImages( Group( [ f1, f2, f3, g1, g2, g3
     ] ), Group( [ f1, f2, f3, f4, f5 ] ), [ f1, f2, f3, g1, g2, g3 ],
    [ f1*f4*f2^-1*f1*f4^-1*f1^-1, f1*f4*f2^-1*f1*f4*f5^-1*f1^-1*f2*f4^-1*f1^-1,
      f1*f4*f2^-1*f1*f5*f1^-1*f2*f4^-1*f1^-1, f2*f4^-1*f1^-1*f2*f1*f4*f2^-1,
      f2*f4^-1*f3*f2^-1, f2*f4^-1*f1^-1*f3^-1*f4*f2^-1 ] )> )
```

3.1.17 AutomorphismVirtualEndomorphism

▷ `AutomorphismVirtualEndomorphism(v)` (attribute)

▷ `AutomorphismIMGMachine(m)` (attribute)

Returns: A description of the pullback map on Teichmüller space.

Let `m` be an IMG machine, thought of as a biset for the fundamental group G of a punctured sphere. Let M denote the automorphism of the surface, seen as a group of outer automorphisms of G that fixes the conjugacy classes of punctures.

Choose an alphabet letter `a`, and consider the virtual endomorphism $v : G_a \to G$. Let H denote the subgroup of M that fixes all conjugacy classes of G_a. then there is an induced virtual endomorphism $\alpha : H \to M$, defined by $t^\alpha = v^{-1}tv$. This is the homomorphism computed by the first command. Its source and range are in fact groups of automorphisms of range of `v`.

The second command constructs an FR machine associated with `\alpha`. Its stateset is a free group generated by elementary Dehn twists of the generators of `G`.

Example

```
gap> z := Indeterminate(COMPLEX_FIELD);;
gap> # a Sierpinski carpet map without multicurves
gap> m := IMGMachine((z^2-z^-2)/2/COMPLEX_I);
<FR machine with alphabet [ 1, 2, 3, 4 ] on Group( [ f1, f2, f3, f4 ] )/[ f3*f2*f1*f4 ]>
gap> AutomorphismIMGMachine(i);
<FR machine with alphabet [ 1, 2 ] on Group( [ x1, x2, x3, x4, x5, x6 ] )>
gap> Display(last);
 G  |     1        2
----+--------+--------+
 x1 | <id>,2   <id>,1
 x2 | <id>,1   <id>,2
 x3 | <id>,2   <id>,1
 x4 | <id>,2   <id>,1
 x5 | <id>,1   <id>,2
 x6 | <id>,2   <id>,1
----+--------+--------+
gap> # the original rabbit problem
gap> m := PolynomialIMGMachine(2,[1/7],[]);;
gap> v := VirtualEndomorphism(m,1);;
gap> a := AutomorphismVirtualEndomorphism(v);
MappingByFunction( <group with 20 generators>, <group with 6 generators>, function( a ) ... end
gap> Source(a).1;
[ f1, f2, f3, f4 ] -> [ f3*f2*f1*f2^-1*f3^-1, f2, f3, f3*f2*f1^-1*f2^-1*f3^-1*f2^-1*f3^-1 ]
gap> Image(a,last);
[ f1, f2, f3, f4 ] -> [ f1, f2, f2*f1*f3*f1^-1*f2^-1, f3^-1*f1^-1*f2^-1 ]
```

```
gap> # so last2*m is equivalent to m*last
```

3.1.18 DBRationalIMGGroup

▷ DBRationalIMGGroup(*sequence/map*) (function)

Returns: An IMG group from Dau's database.

This function returns the iterated monodromy group from a database of groups associated to quadratic rational maps. This database has been compiled by Dau Truong Tan [Tan02].

When called with no arguments, this command returns the database contents in raw form.

The argments can be a sequence; the first integer is the size of the postcritical set, the second argument is an index for the postcritical graph, and sometimes a third argument distinguishes between maps with same post-critical graph.

If the argument is a rational map, the command returns the IMG group of that map, assuming its canonical quadratic rational form form exists in the database.

Example

```
gap> DBRationalIMGGroup(z^2-1);
IMG((z-1)^2)
gap> DBRationalIMGGroup(z^2+1); # not post-critically finite
fail
gap> DBRationalIMGGroup(4,1,1);
IMG((z/h+1)^2|2h^3+2h^2+2h+1=0,h~-0.64)
```

3.1.19 PostCriticalMachine

▷ PostCriticalMachine(*f*) (function)

Returns: The Mealy machine of *f*'s post-critical orbit.

This function constructs a Mealy machine P on the alphabet [1], which describes the post-critical set of *f*. It is in fact an oriented graph with constant out-degree 1. It is most conveniently passed to Draw (**??**).

The attribute Correspondence(P) is the list of values associated with the stateset of P.

Example

```
gap> z := Indeterminate(Rationals,"z");;
gap> m := PostCriticalMachine(z^2);
<Mealy machine on alphabet [ 1 ] with 2 states>
gap> Display(m);
   |  1
---+-----+
 a | a,1
 b | b,1
---+-----+
gap> Correspondence(m);
[ 0, infinity ]
gap> m := PostCriticalMachine(z^2-1);; Display(m); Correspondence(m);
   |  1
---+-----+
 a | c,1
 b | b,1
 c | a,1
---+-----+
[ -1, infinity, 0 ]
```

3.1.20 Mandel

▷ `Mandel(`*`[map]`*`)` (function)

Returns: Calls the external program `mandel`.

This function starts the external program `mandel`, by Wolf Jung. The program is searched for along the standard PATH; alternatively, its location can be set in the string variable EXEC@FR.mandel.

When called with no arguments, this command returns starts `mandel` in its default mode. With a rational map as argument, it starts `mandel` pointing at that rational map.

More information on `mandel` can be found at `http://www.mndynamics.com`.

3.2 Spiders

FR contains an implementation of the Thurston-Hubbard-Schleicher "spider algorithm" [HS94] that constructs a rational map from an IMG recursion. This implementation does not give rigourous results, but relies of floating-point approximation. In particular, various floating-point parameters control the proper functioning of the algorithm. They are stored in a record, `EPS@fr`. Their meaning and default values are:

`EPS@fr.mesh := 10^-1`
: If points on the unit sphere are that close, the triangulation mesh should be refined.

`EPS@fr.prec := 10^-6`
: If points on the unit sphere are that close, they are considered equal.

`EPS@fr.obst := 10^-1`
: If points on the unit sphere are that close, they are suspected to form a Thurston obstruction.

`EPS@fr.juliaiter := 10^3`
: In computing images of the Julia set, never recur deeper than that.

`EPS@fr.fast := 10^-1`
: If the spider moved less than that amount in the last iteration, try speeding up by only wiggling the spider's legs, without recomputing it.

`EPS@fr.ratprec := 10^-10`
: The desired precision on the coefficients of the rational function.

3.2.1 DelaunayTriangulation

▷ `DelaunayTriangulation(`*`points[, quality]`*`)` (operation)

Returns: A Delaunay triangulation of the sphere.

If *`points`* is a list of points on the unit sphere, represented by their 3D coordinates, this function creates a triangulation of the sphere with these points as vertices. This triangulation is such that the angles are as equilateral as possible.

This triangulation is a recursive collection of records, one for each vertex, oriented edge or face. Each such object has a `pos` component giving its coordinates; and an `index` component identifying it uniquely. Additionally, vertices and faces have a `n` component which lists their neighbours in CCW order, and edges have `from,to,left,right,reverse` components.

If all points are aligned on a great circle, or if all points are in a hemisphere, some points are added so as to make the triangulation simplicial with all edges of length $< \pi$. These vertices additionally have a `fake` component set to `true`.

A triangulation may be plotted with `Draw`; this requires appletviewer to be installed. The command `Draw(t:detach)` detaches the subprocess after it is started. The extra arguments `Draw(t:lower)` or `Draw(t:upper)` stretch the triangulation to the lower, respectively upper, hemisphere.

If the second argument *quality*, which must be a floatean, is present, then all triangles in the resulting triangulation are guaranteed to have circumcircle ratio / minimal edge length at most *quality*. Of course, additional vertices may need to be added to ensure that.

Example

```
gap> octagon := Concatenation(IdentityMat(3),-IdentityMat(3))*1.0;
gap> dt := DelaunayTriangulation(octagon);
<triangulation with 6 vertices, 24 edges and 8 faces>
gap> dt!.v;
[ <vertex 1>, <vertex 2>, <vertex 3>, <vertex 4>, <vertex 5>, <vertex 6> ]
gap> last[1].n;
[ <edge 17>, <edge 1>, <edge 2>, <edge 11> ]
gap> last[1].from;
<vertex 1>
```

3.2.2 LocateInTriangulation

▷ `LocateInTriangulation(`*t[, seed], point*`)` (operation)

Returns: The face in *t* containing *point*.

This command locates the face in *t* that contains *point*; or, if *point* lies on an edge or a vertex, it returns that edge or vertex.

The optional second argument specifies a starting vertex, edge, face, or vertex index from which to start the search. Its only effect is to speed up the algorithm.

Example

```
gap> cube := Tuples([-1,1],3)/Sqrt(3.0);;
gap> dt := DelaunayTriangulation(cube);
<triangulation with 8 vertices, 36 edges and 12 faces>
gap> LocateInTriangulation(dt,dt!.v[1].pos);
<vertex 1>
gap> LocateInTriangulation(dt,[3/5,0,4/5]*1.0);
<face 9>
```

3.2.3 IsSphereTriangulation

▷ `IsSphereTriangulation` (filter)

▷ `IsMarkedSphere` (filter)

▷ `Spider(`*ratmap*`)` (attribute)

▷ `Spider(`*machine*`)` (attribute)

The category of triangulated spheres (points in Moduli space), or of marked, triangulated spheres (points in Teichmüller space).

Various commands have an attribudte `Spider`, which records this point in Teichmüller space.

3.2.4 RationalFunction

▷ `RationalFunction([z, ]m)` (operation)

Returns: A rational function.

This command runs a modification of Hubbard and Schleicher's "spider algorithm" [HS94] on the IMG FR machine `m`. It either returns a rational function `f` whose associated machine is `m`; or a record describing the Thurston obstruction to realizability of `f`.

This obstruction record `r` contains a list `r.multicurve` of conjugacy classes in `StateSet(m)`, which represent short multicurves; a matrix `r.mat`, and a spider `r.spider` on which the obstruction was discovered.

If a rational function is returned, it has preset attributes `Spider(f)` and `IMGMachine(f)` which is a simplified version of `m`. This rational function is also normalized so that its post-critical points have barycenter=0 and has two post-critical points at infinity and on the positive real axis. Furthermore, if `m` is polynomial-like, then the returned map is a polynomial.

The command accepts the following options, to return a map in a given normalization:

`RationalFunction(m:param:=IsPolynomial)`
returns $f = z^d + A_{d-2}z^{d-2} + \cdots + A_0$;

`RationalFunction(m:param:=IsBicritical)`
returns $f = ((pz+q)/(rz+s)^d$, with 1postcritical;

`RationalFunction(m:param:=n)`
returns $f = 1 + a/z + b/z^2$ or $f = a/(z^2+2z)$ if `n`=2.

Example

```
gap> m := PolynomialIMGMachine(2,[1/3],[]);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1, f2, f3 ] )/[ f3*f2*f1 ]>
gap> RationalFunction(m);
0.866025*z^2+(-1)*z+(-0.288675)
```

3.2.5 Draw (spider)

▷ `Draw(s)` (operation)

This command plots the spider `s` in a separate X window. It displays the complex sphere, big dots at the post-critical set (feet of the spider), and the arcs and dual arcs of the triangulation connecting the feet.

If the option `julia:=<gridsize>` (if no grid size is specified, it is 500 by default), then the Julia set of the map associated with the spider is also displayed. Points attracted to attracting cycles are coloured in pastel tones, and unattracted points are coloured black.

If the option `noarcs` is specified, the printing of the arcs and dual arcs is disabled.

The options `upper`, `lower` and `detach` also apply.

3.2.6 FRMachine (rational function)

▷ `FRMachine(f)` (operation)
▷ `IMGMachine(f)` (operation)

Returns: An IMG FR machine.

This function computes a triangulation of the sphere, on the post-critical set of *f*, and lifts it through the map *f*. the action of the fundamental group of the punctured sphere is then read into an IMG fr machine m, which is returned.

This machine has a preset attribute Spider(m).

An approximation of the Julia set of *f* can be computed, and plotted on the spider, with the form IMGMachine(f:julia) or IMGMachine(f:julia:=gridsize).

Example

```
gap> z := Indeterminate(COMPLEX_FIELD);;
gap> IMGMachine(z^2-1);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1, f2, f3 ] )/[ f2*f1*f3 ]>
gap> Display(last);
 G  |            1        2
----+---------------+--------+
 f1 |          f2,2   <id>,1
 f2 | f3^-1*f1*f3,1   <id>,2
 f3 |        <id>,2     f3,1
----+---------------+--------+
Relator: f2*f1*f3
```

3.2.7 FindThurstonObstruction

▷ FindThurstonObstruction(*list*) (operation)

Returns: A description of the obstruction corresponding to *list*, or fail.

This method accepts a list of IMG elements on the same underlying machine, and treats these as representatives of conjugacy classes defining (part of) a multicurve. It computes whether these curves, when supplemented with their lifts under the recursion, constitute a Thurston obstruction, by computing its transition matrix.

The method either returns fail, if there is no obstruction, or a record with as fields matrix,machine,obstruction giving respectively the transition matrix, a simplified machine, and the curves that constitute a minimal obstruction.

Example

```
gap> r := PolynomialIMGMachine(2,[],[1/6]);;
gap> F := StateSet(r);;
gap> twist := GroupHomomorphismByImages(F,F,GeneratorsOfGroup(F),[F.1,F.2^(F.3*F.2),F.3^F.2,F.4]
gap> SupportingRays(r*twist^-1);
rec( machine := <FR machine with alphabet [ 1, 2 ] on F/[ f4*f1*f2*f3 ]>,
     twist := [ f1, f2, f3, f4 ] -> [ f1, f3^-1*f2*f3, f3^-1*f2^-1*f3*f2*f3, f4 ],
     obstruction := "Dehn twist" )
gap> FindThurstonObstruction([FRElement(last.machine,[2,3])]);
rec( matrix := [ [ 1 ] ], machine := <FR machine with alphabet [ 1, 2 ] on F/[ f4*f1*f2*f3 ]>, o
```

3.2.8 HurwitzMap

▷ HurwitzMap(*spider, monodromy*) (operation)

Returns: A record describing a Hurwitz map.

If *spider* is a spider, marked by a group G, and *monodromy* is a homomorphism from G to a permutation group, this function computes a rational map whose critical values are the vertices of *spider* and whose monodromy about these critical values is given by *monodromy*.

The returned data are in a record with a field `degree`, the degree of the map; two fields `map` and `post`, describing the desired $\mathbb{P}^1$-map — `post` is a Möbius transformation, and the composition of `map` and `post` is the desired map; and lists `zeros`, `poles` and `cp` describing the zeros, poles and critical points of the map. Each entry in these lists is a record with entries `degree`, `pos` and `to` giving, for each point in the source of `map`, the local degree and the vertex in *`spider`* it maps to.

This function requires external programs in the subdirectory "hurwitz" to have been compiled.

Example

```
gap> # we'll construct 2d-2 points on the equator, and permutations
gap> # in order (1,2),...,(d-1,d),(d-1,d),...,(1,2) for these points.
gap> # first, the spider.
gap> d := 20;;
gap> z := List([0..2*d-3], i->P1Point(Exp(i*PMCOMPLEX.constants.2IPI/(2*d-2))));;
gap> spider := TRIVIALSPIDER@FR(z);;
gap> g := FreeGroup(2*d-2);;
gap> IMGMARKING@FR(spider,g);
gap> # next, the permutation representation
gap> perms := List([1..d-1],i->(i,i+1));;
gap> Append(perms,Reversed(perms));
gap> perms := GroupHomomorphismByImages(g,SymmetricGroup(d),GeneratorsOfGroup(g),perms);;
gap> # now compute the map
gap> HurwitzMap(spider,perms);
rec( cp := [ rec( degree := 2, pos := <1.0022-0.0099955i>, to := <vertex 19[ 9, 132, 13, 125 ]>
      rec( degree := 2, pos := <1.0022-0.0099939i>, to := <vertex 20[ 136, 128, 129, 11 ]> ),
      rec( degree := 2, pos := <1.0039-0.0027487i>, to := <vertex 10[ 73, 74, 16, 82 ]> ),
      rec( degree := 2, pos := <1.0006-0.0027266i>, to := <vertex 29[ 185, 20, 179, 21 ]> ),
      rec( degree := 2, pos := <1.0045-7.772e-05i>, to := <vertex 9[ 24, 77, 17, 72 ]> ),
      rec( degree := 2, pos := <1.1739+0.33627i>, to := <vertex 2[ 31, 32, 41, 28 ]> ),
      rec( degree := 2, pos := <1.0546+0.12276i>, to := <vertex 3[ 37, 38, 33, 46 ]> ),
      rec( degree := 2, pos := <1.026+0.061128i>, to := <vertex 4[ 43, 39, 52, 45 ]> ),
      rec( degree := 2, pos := <1.0148+0.03305i>, to := <vertex 5[ 49, 44, 58, 51 ]> ),
      rec( degree := 2, pos := <1.0098+0.018122i>, to := <vertex 6[ 55, 50, 64, 57 ]> ),
      rec( degree := 2, pos := <1.0071+0.0093947i>, to := <vertex 7[ 61, 62, 71, 59 ]> ),
      rec( degree := 2, pos := <1.0055+0.0037559i>, to := <vertex 8[ 67, 68, 63, 69 ]> ),
      rec( degree := 2, pos := <1.0035-0.0047633i>, to := <vertex 11[ 79, 75, 88, 81 ]> ),
      rec( degree := 2, pos := <1.0031-0.0062329i>, to := <vertex 12[ 85, 80, 94, 87 ]> ),
      rec( degree := 2, pos := <1.0029-0.0073311i>, to := <vertex 13[ 91, 86, 100, 93 ]> ),
      rec( degree := 2, pos := <1.0027-0.008187i>, to := <vertex 14[ 97, 92, 106, 99 ]> ),
      rec( degree := 2, pos := <1.0026-0.008824i>, to := <vertex 15[ 103, 98, 112, 105 ]> ),
      rec( degree := 2, pos := <1.0025-0.0092966i>, to := <vertex 16[ 109, 104, 118, 111 ]> ),
      rec( degree := 2, pos := <1.0024-0.0096345i>, to := <vertex 17[ 115, 110, 124, 117 ]> ),
      rec( degree := 2, pos := <1.0023-0.0098698i>, to := <vertex 18[ 121, 116, 122, 123 ]> ),
      rec( degree := 2, pos := <1.0021-0.0098672i>, to := <vertex 21[ 133, 127, 142, 135 ]> ),
      rec( degree := 2, pos := <1.002-0.0096298i>, to := <vertex 22[ 139, 134, 148, 141 ]> ),
      rec( degree := 2, pos := <1.002-0.0092884i>, to := <vertex 23[ 145, 140, 154, 147 ]> ),
      rec( degree := 2, pos := <1.0019-0.0088147i>, to := <vertex 24[ 151, 146, 160, 153 ]> ),
      rec( degree := 2, pos := <1.0017-0.008166i>, to := <vertex 25[ 157, 152, 166, 159 ]> ),
      rec( degree := 2, pos := <1.0016-0.0073244i>, to := <vertex 26[ 163, 158, 172, 165 ]> ),
      rec( degree := 2, pos := <1.0014-0.0061985i>, to := <vertex 27[ 169, 164, 178, 171 ]> ),
      rec( degree := 2, pos := <1.0011-0.0047031i>, to := <vertex 28[ 175, 170, 176, 177 ]> ),
      rec( degree := 2, pos := <0.99908+0.0038448i>, to := <vertex 31[ 187, 183, 196, 189 ]> ),
      rec( degree := 2, pos := <0.99759+0.0094326i>, to := <vertex 32[ 193, 188, 202, 195 ]> ),
      rec( degree := 2, pos := <0.99461+0.018114i>, to := <vertex 33[ 199, 194, 208, 201 ]> ),
```

```
      rec( degree := 2, pos := <0.98944+0.032796i>, to := <vertex 34[ 205, 200, 214, 207 ]> ),
      rec( degree := 2, pos := <0.9772+0.058259i>, to := <vertex 35[ 211, 206, 220, 213 ]> ),
      rec( degree := 2, pos := <0.94133+0.11243i>, to := <vertex 36[ 217, 212, 226, 219 ]> ),
      rec( degree := 2, pos := <0.79629+0.23807i>, to := <vertex 37[ 223, 224, 225, 221 ]> ),
      rec( degree := 2, pos := <1+0i>, to := <vertex 30[ 181, 182, 6, 190 ]> ) ], degree := 20,
  map := <((-0.32271060393507572-4.3599244721894763i_z)*z^20+(3.8941736874493795+78.415744809040
i_z)*z^19+(-16.808157937605603-665.79436908026275i_z)*z^18+(2.6572296014719168+3545.861245383101
)*z^17+(316.57668022762243-13273.931613611372i_z)*z^16+(-1801.6631038749117+37090.818733740503i_
z^15+(5888.6033008259928-80172.972599556582i_z)*z^14+(-13500.864941314803+137069.100158838256i_z)
13+(23251.436304923012-187900.36507913063i_z)*z^12+(-31048.192131502536+208077.63047409133i_z)*z
+(32639.349270133433-186578.17493860485i_z)*z^10+(-27155.791223040047+135145.40893002271i_z)*z^9
7836.343164500577-78489.005444299968i_z)*z^8+(-9153.842142530224+36053.895961137248i_z)*z^7+(359
408777659944-12810.65497539577i_z)*z^6+(-1047.541279063196+3397.470068169695i_z)*z^5+(212.906725
0024-633.29691376653466i_z)*z^4+(-26.989372105307872+74.040615571896637i_z)*z^3+(1.6073346640110
-4.0860733899027055i_z)*z^2)/(z^18+(-18.034645372692019-0.45671993287358581i_z)*z^17+(153.540499
49956+7.7811506405054889i_z)*z^16+(-819.9344323563339-62.384270590463998i_z)*z^15+(3077.71530771
75+312.59552100187739i_z)*z^14+(-8623.1225834872057-1096.4398001099003i_z)*z^13+(18689.343968250
2856.8568878158458i_z)*z^12+(-32038.568184053798-5725.9186424029094i_z)*z^11+(44038.148375498437
17.0162876593004i_z)*z^10+(-48898.555649389084-11295.156285052604i_z)*z^9+(43964.579894637543+11
.997395732025i_z)*z^8+(-31931.403449371515-9074.2344933443364i_z)*z^7+(18595.347261301522+5786.6
424805825i_z)*z^6+(-8565.0823844971637-2899.3353634270734i_z)*z^5+(3051.6919509143086+1117.44496
99487i_z)*z^4+(-811.56293104533825-319.93036282549667i_z)*z^3+(151.69784956523344+64.11787684283
5i_z)*z^2+(-17.785127700028404-8.0311759305108268i_z)*z+(0.98427999507354302+0.47338721325094818
))>, poles := [ rec( degree := 1, pos := <0.99517+0.30343i>, to := <vertex 1[ 26, 27, 1, 34 ]> )
      rec( degree := 1, pos := <1.0021+0.11512i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0028+0.05702i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0026+0.030964i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0025+0.016951i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0024+0.0085784i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0024+0.003208i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0023-0.00046905i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0023-0.0030802i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0023-0.0049913i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0023-0.0064163i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0022-0.0074855i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0022-0.0082954i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0022-0.0089048i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0022-0.0093543i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0022-0.0096742i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0022-0.0098869i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 1, pos := <1.0022-0.0099988i>, to := <vertex 1[ 26, 27, 1, 34 ]> ),
      rec( degree := 2, pos := <P1infinity>, to := <vertex 1[ 26, 27, 1, 34 ]> ) ],
  post := <((-0.91742065452766763+0.99658449300666985i_z)*z+(0.74087581626192234-1.1339948562200
i_z))/((-0.75451451285920013+0.96940026593933015i_z)*z+(0.75451451285920013-0.96940026593933015i
)>, zeros := [ rec( degree := 1, pos := <0.92957+0.28362i>, to := <vertex 38[ 30, 3, 228, 7 ]> )
      rec( degree := 1, pos := <0.99173+0.11408i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
      rec( degree := 1, pos := <0.99985+0.056874i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
      rec( degree := 1, pos := <1.0014+0.030945i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
      rec( degree := 1, pos := <1.002+0.016938i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
      rec( degree := 1, pos := <1.0022+0.0085785i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
      rec( degree := 1, pos := <1.0022+0.0032076i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
      rec( degree := 1, pos := <1.0022-0.00046827i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
      rec( degree := 1, pos := <1.0022-0.0030802i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
```

```
        rec( degree := 1, pos := <1.0022-0.0049908i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.006416i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.0074855i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.0082953i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.0089047i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.0093542i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.0096742i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.0098869i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 1, pos := <1.0022-0.0099988i>, to := <vertex 38[ 30, 3, 228, 7 ]> ),
        rec( degree := 2, pos := <0+0i>, to := <vertex 38[ 30, 3, 228, 7 ]> ) ] )
```

3.2.9 DessinByPermutations

▷ `DessinByPermutations(s0, s1[, sinf])` (operation)

Returns: A rational map (see `HurwitzMap` (3.2.8)) with monodromies s,t.

This command computes the Hurwitz map associated with the spider $[0,1]\cup[1,\infty]$; the monodromy representation is by the permutation *s0* at 0 and *s1* at 1. The optional third argument *sinf* is the monodromy at ∞, and must equal $s_0^{-1}s_1^{-1}$.

The data is returned as a record, with entries `degree`, `map`, `post`, and lists `poles`, `zeros`, and `above1`. Each entry in the list is a record with entries `pos` and `degree`.

Example

```
gap> DessinByPermutations((1,2),(2,3));
rec( above1 := [ rec( degree := 2, pos := <1+0i> ),
                 rec( degree := 1, pos := <-0.5-1.808e-14i> ) ],
     degree := 3,
     map := <(-1.9999999999946754+2.1696575743432764e-13i_z)*z^3+(2.9999999999946754-2.169657574
     poles := [ rec( degree := 3, pos := <P1infinity> ) ],
     post := <z>,
     zeros := [ rec( degree := 1, pos := <1.5+5.4241e-14i> ),
               rec( degree := 2, pos := <0+0i> ) ] )
gap> # the Cui example
gap> DessinByPermutations((1,3,12,4)(5,9)(6,7)(10,13,11)(2,8),
           (1,5,13,6)(7,10)(2,3)(8,11,12)(4,9),
           (1,7,11,2)(3,8)(4,5)(9,12,13)(6,10));
rec(
  above1 := [ rec( degree := 2, pos := <1.9952-0.79619i> ),
      rec( degree := 2, pos := <0.43236-0.17254i> ), rec( degree := 2, pos := <-0.9863-0.16498i>
      rec( degree := 3, pos := <-0.12749-0.99184i> ), rec( degree := 4, pos := <1+0i> ) ],
  degree := 13,
  map := <((-6.9809917616400366e-12+0.13002709490708636i_z)*z^13+(-0.68172329304137969-0.8451761
2062078i_z)*z^12+(4.0903397584184269+0.30979932084028583i_z)*z^11+(-6.3643009040280925+7.5930410
99336i_z)*z^10+(-5.1732765988942884-16.738910009700096i_z)*z^9+(21.528087032174511+6.1135459901O
5i_z)*z^8+(-15.258776392407746+13.657687016998921i_z)*z^7+(-1.6403496019814323-13.45331629709422
z)*z^6+(4.4999999996894351+3.3633290741375781i_z)*z^5+(-0.99999999990279009+2.5538451239904557e-
_z)*z^4)/(z^9+(-4.4999999999400613+3.3633290744267983i_z)*z^8+(1.6403496020557891-13.45331629745
7i_z)*z^7+(15.258776391831654+13.657687016903173i_z)*z^6+(-21.528087030670253+6.1135459892162567
)*z^5+(5.1732765986730511-16.738910007513041i_z)*z^4+(6.3643009027133139+7.593041020468557i_z)*z
(-4.0903397575324512+0.30979932067785648i_z)*z^2+(0.68172329288354727-0.8451761166966415i_z)*z+(
734454343833585e-12+0.13002709487107747i_z))>,
  poles := [ rec( degree := 2, pos := <1.6127-0.49018i> ),
      rec( degree := 2, pos := <0.5-0.04153i> ), rec( degree := 2, pos := <-0.61269-0.49018i> ),
```

```
        rec( degree := 3, pos := <0.5-0.43985i> ), rec( degree := 4, pos := <P1infinity> ) ],
    post := <-z+1._z>,
    zeros := [ rec( degree := 2, pos := <1.9863-0.16498i> ),
        rec( degree := 3, pos := <1.1275-0.99184i> ), rec( degree := 2, pos := <0.56764-0.17254i>
        rec( degree := 2, pos := <-0.99516-0.79619i> ), rec( degree := 4, pos := <0+0i> ) ] )
```

3.2.10 KneadingSequence (angle)

▷ `KneadingSequence(`*`angle`*`)` (attribute)

Returns: The kneading sequence associated with *angle*.

This function converts a rational angle to a kneading sequence, to describe a quadratic polynomial.

If *angle* is in $[1/7, 2/7]$ and the option `marked` is set, the kneading sequence is decorated with markings in A,B,C.

Example

```
gap> KneadingSequence(1/7);
[ 1, 1 ]
gap> KneadingSequence(1/5:marked);
[ "A1", "B1", "B0" ]
```

3.2.11 AllInternalAddresses

▷ `AllInternalAddresses(`*`n`*`)` (attribute)

Returns: Internal addresses of maps with period up to *n*.

This function returns internal addresses for all periodic points of period up to *n* under angle doubling. These internal addresses describe the prominent hyperbolic components along the path from the landing point to the main cardioid in the Mandelbrot set; this is a list of length `3k`, with at position `3i+1,3i+2` the left and right angles, respectively, and at position `3i+3` the period of that component. For example, `[ 3/7, 4/7, 3, 1/3, 2/3, 2 ]` describes the airplane: a polynomial with landing angles $[3/7, 4/7]$ of period 3; and such that there is a polynomial with landing angles $[1/3, 2/3]$ and period 2.

Example

```
gap> AllInternalAddresses(3);
[ [  ], [ [ 1/3, 2/3, 2 ] ],
[ [ 1/7, 2/7, 3 ], [ 3/7, 4/7, 3, 1/3, 2/3, 2 ], [ 5/7, 6/7, 3 ] ] ]
```

3.2.12 ExternalAnglesRelation

▷ `ExternalAnglesRelation(`*`degree, n`*`)` (function)

Returns: An equivalence relation on the rationals.

This function returns the equivalence relation on `Rationals` identifying all pairs of external angles that land at a common point of period up to *n* under angle multiplication by by *degree*.

Example

```
gap> ExternalAnglesRelation(2,3);
<equivalence relation on Rationals >
gap> EquivalenceRelationPartition(last);
[ [ 1/7, 2/7 ], [ 1/3, 2/3 ], [ 3/7, 4/7 ], [ 5/7, 6/7 ] ]
```

3.2.13 ExternalAngle

▷ `ExternalAngle(`*machine*`)` (function)

Returns: The external angle identifying *machine*.

In case *machine* is the IMG machine of a unicritical polynomial, this function computes the external angle landing at the critical value. More precisely, it computes the equivalence class of that external angle under `ExternalAnglesRelation` (3.2.12).

Example

```
gap> ExternalAngle(PolynomialIMGMachine(2,[1/7])); # the rabbit
{2/7}
gap> Elements(last);
[ 1/7, 2/7 ]
```

3.2.14 ChangeFRMachineBasis

▷ `ChangeFRMachineBasis(`*m[, l][, p]*`)` (attribute)

Returns: An equivalent FR machine, in a new basis.

This function constructs a new group FR machine, given a group FR machine *m* and, optionally, a list of states *l* (as elements of the free object `StateSet(m)`) and a permutation *p*, which defaults to the identity permutation.

The new machine has the following transitions: if alphabet letter `a` is mapped to `b` by state `s` in *m*, leading to state `t`, then, in the new machine, the input letter `a^p` is mapped to `b^p` by state `s`, leading to state `l[a]^-1*t*l[b]`.

The group generated by the new machine is isomorphic to the group generated by *m*. This command amounts to a change of basis of the associated bimodule (see [Nek05, Section 2.2]). It amounts to conjugation by the automorphism `c=FRElement("c",[l[1]*c,...,l[n]*c],[()],1)`.

If the second argument is absent, this command attempts to choose a list that makes many entries of the recursion trivial.

Example

```
gap> n := FRMachine(["tau","mu"],[[[],[1]],[[],[-2]]],[(1,2),(1,2)]);;
gap> Display(n);
 G   |     1         2
-----+--------+---------+
 tau | <id>,2     tau,1
  mu | <id>,2   mu^-1,1
-----+--------+---------+
gap> nt := ChangeFRMachineBasis(n,GeneratorsOfFRMachine(n){[1,1]});;
gap> Display(nt);
 G   |     1                    2
-----+--------+--------------------+
 tau | <id>,2                tau,1
  mu | <id>,2   tau^-1*mu^-1*tau,1
-----+--------+--------------------+
```

3.2.15 ComplexConjugate

▷ `ComplexConjugate(`*m*`)` (operation)

Returns: An FR machine with inverted states.

This function constructs an FR machine whose generating states are the inverses of the original states. If m came from a complex rational map $f(z)$, this would construct the machine of the conjugate map $\overline{f(\overline{z})}$.

Example

```
gap> a := PolynomialIMGMachine(2,[1/7]);
<FR machine with alphabet [ 1, 2 ] and adder FRElement(...,f4) on <object>/[ f4*f3*f2*f1 ]>
gap> Display(a);
 G  |            1            2
----+---------------+------------+
 f1 | f1^-1*f2^-1,2   f3*f2*f1,1
 f2 |          f1,1       <id>,2
 f3 |          f2,1       <id>,2
 f4 |          f4,2       <id>,1
----+---------------+------------+
Adding element: FRElement(...,f4)
Relator: f4*f3*f2*f1
gap> Display(ComplexConjugate(a));
 G  |            1                     2
----+---------------+---------------------+
 f1 | f1*f2*f3*f4,2   f4^-1*f2^-1*f1^-1,1
 f2 |          f1,1      <identity ...>,2
 f3 |          f2,1      <identity ...>,2
 f4 |          f4,2      <identity ...>,1
----+---------------+---------------------+
Adding element: FRElement(...,f4)
Relator: f1*f2*f3*f4
gap> ExternalAngle(a);
{2/7}
gap> ExternalAngle(ComplexConjugate(a));
{6/7}
```

3.2.16 BraidTwists

▷ `BraidTwists(m)` (operation)

Returns: Automorphisms of m's stateset that preserve its relator.

This function returns a list of (positive and negative) free group automorphisms, that are such that conjugating m by these automorphisms preserves its IMG relator.

These are in fact the generators of Artin's braid group.

3.2.17 RotatedSpider

▷ `RotatedSpider(m[, p])` (operation)

Returns: A polynomial FR machine with rotated spider at infinity.

This function constructs an isomorphic polynomial FR machine, but with a different numbering of the spider legs at infinity. This rotation is accomplished by conjugating by `adder^p`, where `adder` is the adding element of m, and p, the rotation parameter, is 1 by default.

Example

```
gap> a := PolynomialIMGMachine(3,[1/4]);
<FR machine with alphabet [ 1, 2, 3 ] and adder FRElement(...,f3) on <object>/[ f3*f2*f1 ]>
gap> Display(a);
 G  |      1        2        3
```

```
----+---------+--------+---------+
 f1 | f1^-1,2   <id>,3   f2*f1,1
 f2 |    f1,1   <id>,2    <id>,3
 f3 |    f3,3   <id>,1    <id>,2
----+---------+--------+---------+
Adding element: FRElement(...,f3)
Relator: f3*f2*f1
gap> Display(RotatedSpider(a));
 G  |     1            2              3
----+--------+-----------+--------------+
 f1 | <id>,2   f2*f1*f3,3   f3^-1*f1^-1,1
 f2 | <id>,1       <id>,2   f3^-1*f1*f3,3
 f3 |   f3,3       <id>,1          <id>,2
----+--------+-----------+--------------+
Adding element: FRElement(...,f3)
Relator: f3*f2*f1
gap> ExternalAngle(a);
{3/8}
gap> List([1..10],i->ExternalAngle(RotatedSpider(a,i)));
[ {7/8}, {1/4}, {7/8}, {1/4}, {7/8}, {1/4}, {7/8}, {1/4}, {7/8}, {1/4} ]
```

Chapter 4

Examples

IMG predefines a large collection of machines and groups. The groups are, whenever possible, defined as state closures of corresponding Mealy machines.

4.1 Examples of groups

4.1.1 PoirierExamples

▷ `PoirierExamples(...)` (function)

The examples from Poirier's paper [Poi]. See details under `PolynomialIMGMachine` (3.1.11); in particular, `PoirierExamples(1)` is the Douady rabbit map.

Chapter 5

IMG implementation details

IMG creates new categories for the various objects considered in the package. The first category is ...

5.1 Marked spheres

FR contains algorithms that convert a rational function (with floating-point complex coefficients) to an IMG machine and back.

Consider a rational map f of degree d. First, compute the post-critical set P_f; this is done by iterating f on critical points, and considering points at angle at most $EPS@.prec$ from each other as equal.

Then construct a spherical Delaunay triangulation T on P_f; and choose in it a minimal spanning tree; edges of that tree represent a generating set of the fundamental group G of $S^2 \setminus P_f$.

Lift first through f the edges of the dual tree in the dual tesselation of T; they will form d connected subgraphs, numbered $1,\dots,d$. Lift then the edges crossing the minimal spanning tree, and read the elements of F that their lifts represent, as well as the subgraphs they start and end in. This data describes an FR machine.

Choose then for each vertex in P_f an adjacent face in $T/$. This choice defines a generating family of F made up, for each vertex, of a path in the dual tree starting at a basepoint, a sequence of edges around a vertex starting at the chosen face, and a path back in the dual tree. The product, in an appropriate order, of these generators describes an IMG machine.

There is an epimorphism from the group G, generated by these loops around vertices, to F; and this epimorphism becomes an isomorphism if one adds to $G/$ the relation "product of generators in an appropriate order". Such a triangulation, with a given group G and a homomorphism from G to F, is called a *spider*.

The inverse algorithm is quite similar. Consider an IMG machine M, with stateset G. Start by a "standard" set of points on the sphere, one per generator of M, and construct a spider S on them. Find a rational map f with critical values at the vertices of S and monodromy given by the activities of M, and lift S to a spider T, marked by a group H. The lifting gives a homomorphism $f^* : G \to H \wr Sym(d)$.

By appropriately relabeling the alphabet, one can ensure that this homomorphism coincides with M's recursion at the first level. Furthermore, it identifies each vertex v_i of T either with a vertex w_j of S, if for some $g \in G$ a state in M of g is a conjugate g_j, while the corresponding entry of $f^*(g)$ is a conjugate of h_i.

Construct then a new triangulation S' by keeping only those vertices of T that were identified, and mapping them by a Möbius transformation μ to "standard position". This means that the barycenter

of the points is $0 \in \mathbb{R}^3$, that the last point goes to ∞, and that the next-to-last goes to $\mathbb{R}_+$. Letting F denote the group on the minimal spanning tree of S', there is a homomorphism $\mu_* : H \to F$.

The decomposition ϕ of M then produces a homomorphism $m : G \to F$ such that $\mu_* f^* = m\phi$. This turns S' into a spider on G. Iterate then this process with S'.

Either the spiders S converge, and then $\mu^{-1} f$ is the desired rational function; or there is a Thurston obstruction, which is a non-contracting multicurve. Seek therefore clusters of vertices that are very close from each other, and compute the curve going around them; this defines a conjugacy class in G. In M, compute the iterated decomposition of this curve, and its associated transition matrix. If it has spectrum at least 1, return the multicurve as an obstruction; otherwise, continue.

Chapter 6

Miscellanea

6.1 Complex numbers

6.1.1 IsPMComplex

▷ IsPMComplex (filter)
▷ PMCOMPLEX_FAMILY (family)
▷ PMCOMPLEX_PSEUDOFIELD (global variable)
▷ PMCOMPLEX (global variable)

A "poor man's" implementation of complex numbers, based on the underlying 64-bit floating-point numbers in GAP.

Strictly speaking, complex numbers do not form a field in GAP, because associativity etc. do not hold. Still, a field is defined, PMCOMPLEX_FIELD, making it possible to construct an indeterminate and rational functions, to be passed to FR's routines.

These complex numbers can be made the default floating-point numbers via SetFloats(PMCOMPLEX);. They may then be entered as standard floating-point numbers, with the suffix _z.

Example

```
gap> z := Indeterminate(PMCOMPLEX_FIELD,"z");
z
gap> (z+1/2)^5/(z-1/2);
(z^5+2.5*z^4+2.5*z^3+1.25*z^2+0.3125*z+0.03125)/(z+(-0.5))
gap> NewFloat(IsPMComplex,1,2);
1+2i
gap> last^2;
-3+4i
gap> RealPart(last);
-3
gap> Norm(last2);
25
gap> NewFloat(IsPMComplex,"1+2*I");
1+2i
gap> RootsFloat(z^2-5);
[ 2.23607, -2.23607 ]
gap> RootsFloat(ListWithIdenticalEntries(80,1.0_z));
[ 0.987688+0.156434i, 0.996917+0.0784591i, 0.996917-0.0784591i, 0.987688-0.156434i, 0.760406+0.6
```

```
  0.522499+0.85264i, 0.649448+0.760406i, 0.891007+0.45399i, 0.587785+0.809017i, 0.707107+0.70710
  0.382683+0.92388i, 0.85264+0.522499i, -0.59719-0.608203i, -0.867574-0.11552i, -0.186972-0.9902
  0.156434+0.987688i, 0.295424-0.953359i, 0.588289-0.808509i, 0.455128-0.893999i, 0.0951213-1.01
  0.97237-0.233445i, -0.233486+0.972416i, 0.379514-0.92918i, 3.09131e-07+1.i, 0.182752-0.984684i
  0.92388-0.382683i, -0.585832+0.81608i, 0.809018-0.587792i, -0.656055+0.770506i, 0.760385-0.649
  -0.15643+0.987703i, -0.307608-0.969002i, 0.649377-0.760134i, -0.382904+0.92328i, -0.857704+0.5
  -0.929824-0.488558i, -0.671579-0.790133i, -0.886052-0.560249i, -1.05047-0.0873829i, -0.496236-
  -0.585809-0.852796i, -0.518635+0.85364i, -1.04842+0.0255453i, -0.752485-0.724528i, -0.309225+0
 ]
gap> AsSortedList(List(last,AbsoluteValue));
[ 0.739812, 0.847513, 0.852377, 0.861109, 0.875231, 0.967092, 0.977534, 0.998083, 0.998317, 0.99
  1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.00001, 1.000
  1.01509, 1.01665, 1.01814, 1.01834, 1.02033, 1.0238, 1.02796, 1.02881, 1.03169, 1.03462, 1.035
  1.05036, 1.05155, 1.0541, 1.05442, 1.0672 ]
```

6.2 P1 points

6.2.1 IsP1Point

▷ `IsP1Point` (filter)
▷ `P1PointsFamily` (family)
▷ `P1Point(`*complex*`)` (function)
▷ `P1Point(`*real*`, `*imag*`)` (function)
▷ `P1Point(`*string*`)` (function)

P1 points are complex numbers or infinity; fast methods are implemented to compute with them, and to apply rational maps to them.

The first filter recognizes these objects. Next, the family they belong to. The next methods create a new P1 point.

6.2.2 CleanedP1Point

▷ `CleanedP1Point(`*p*`, `*prec*`)` (function)
Returns: *p*, rounded towards 0/1/infinity/reals at precision *prec*.

6.2.3 P1infinity

▷ `P1infinity` (global variable)
▷ `P1one` (global variable)
▷ `P1zero` (global variable)

The south, north and 'east' poles of the Riemann sphere.

6.2.4 P1Antipode

▷ `P1Antipode(`*p*`)` (function)
Returns: The antipode of *p* on the Riemann sphere.

6.2.5 P1Barycentre

▷ `P1Barycentre(`*`points, ...`*`)` (function)
Returns: The barycentre of its arguments (which can also be a list of P1 points).

6.2.6 P1Circumcentre

▷ `P1Circumcentre(p, q, r)` (function)
Returns: The centre of the smallest disk containing `p,q,r`.

6.2.7 P1Distance

▷ `P1Distance(p, q)` (function)
Returns: The spherical distance from `p` to `q`.

6.2.8 P1Midpoint

▷ `P1Midpoint(p, q)` (function)
Returns: The point between `p` to `q` (undefined if they are antipodes of each other).

6.2.9 P1Sphere

▷ `P1Sphere(v)` (function)
Returns: The P1 point corresponding to `v` in $\mathbb{R}^3$.

6.2.10 SphereP1

▷ `SphereP1(p)` (function)
Returns: The coordinates in $\mathbb{R}^3$ of `p`.

6.2.11 SphereP1Y

▷ `SphereP1Y(p)` (function)
Returns: The Y coordinate in $\mathbb{R}^3$ of `p`.

6.2.12 P1XRatio

▷ `P1XRatio(p, q, r, s)` (function)
Returns: The cross ratio of `p, q, r, s`.

6.2.13 IsP1Map

▷ `IsP1Map` (filter)
▷ `P1MapsFamily` (family)

P1 maps are stored more efficiently than rational functions, but are otherwise equivalent. The first filter recognizes these objects. Next, the family they belong to.

6.2.14 MoebiusMap

▷ MoebiusMap(*[sourcelist,]destlist*) (function)
▷ MoebiusMap(*p, q, r, s, t, u*) (function)
▷ MoebiusMap(*p, q, r*) (function)
▷ MoebiusMap(*p, q*) (function)

These methods create a new P1 map. In the first case, this is the Möbius transformation sending `p,q,r` to `P,Q,R` respectively; in the second case, the map sending `p,q,r` to `0,1,P1infinity` respectively; in the third case, the map sending `p,q` to `0,P1infinity` respectively, of the form $(z-p)/(z-q)$.

6.2.15 P1z

▷ P1z (global variable)

The identity Möbius transformation.

6.2.16 CleanedP1Map

▷ CleanedP1Map(*map, prec*) (operation)
Returns: *map*, with coefficients rounded using *prec*.

6.2.17 CompositionP1Map

▷ CompositionP1Map(*map1, ...*) (operation)
Returns: The composition of the maps passed as arguments, in the functional (*map1* last) order.

6.2.18 InverseP1Map

▷ InverseP1Map(*map*) (operation)
Returns: The functional inverse of the Möbius transformation *map*.

6.2.19 ConjugatedP1Map

▷ ConjugatedP1Map(*map, mobius*) (operation)
Returns: The map `CompositionP1Map(InverseP1Map(mobius),map,mobius)`.

6.2.20 CoefficientsOfP1Map

▷ CoefficientsOfP1Map(*map*) (operation)
Returns: Coefficients of numerator and denominator of *map*, lowest degree first.

6.2.21 P1MapByCoefficients

▷ P1MapByCoefficients(*numer, denom*) (operation)
Returns: The P1 map with numerator coefficients *numer* and denominator *denom*, lowest degree first.

6.2.22 P1Path

▷ `P1Path(p, q)` (operation)
Returns: The P1 map sending `0` to `p` and `1` to `q`.

6.2.23 DegreeOfP1Map

▷ `DegreeOfP1Map(map)` (operation)
Returns: The degree of `map`.

6.2.24 P1Image

▷ `P1Image(map, p1point)` (operation)
Returns: The image of `p1point` under `map`.

6.2.25 P1PreImages

▷ `P1PreImages(map, p1point)` (operation)
Returns: The preimages of `p1point` under `map`.

6.2.26 P1MapCriticalPoints

▷ `P1MapCriticalPoints(map)` (operation)
Returns: The critical points of `map`.

6.2.27 P1MapRational

▷ `P1MapRational(rat)` (operation)
Returns: The P1 map given by the rational function `rat`.

6.2.28 RationalP1Map

▷ `RationalP1Map(map)` (operation)
▷ `RationalP1Map(indeterminate, map)` (operation)
Returns: The rational function given by P1 map `map`.

6.2.29 P1MapSL2

▷ `P1MapSL2(mat)` (operation)
Returns: The Möbius P1 map given by the 2x2 matrix `mat`.

6.2.30 SL2P1Map

▷ `SL2P1Map(map)` (operation)
Returns: The matrix of the Möbius P1 map `map`.

6.2.31 SetP1Points

▷ `SetP1Points(`*`record[, prec]`*`)` (function)

Installs a default implementation for P1 points. Fundamentally, a P1 point is a complex number or infinity, with a few extra methods. The argument *`record`* is the record describing the floating-point implementation.

Currently, one implementation (the default) is based on pairs of IEEE754 floateans. It is fast, but is limited to 53 bits of precision. It is loaded via `SetP1Points(PMCOMPLEX);`.

Another implementation, in case the package Float is available, is based on MPC complex numbers. It offers unlimited precision, but is much slower. It is loaded via `SetP1Points(MPC);` or `SetP1Points(MPC,prec);`.

6.3 Miscellanea

6.4 User settings

6.4.1 InfoIMG

▷ `InfoIMG` (info class)

This is an `Info` class for the package IMG. The command `SetInfoLevel(InfoIMG,1);` switches on the printing of some information during the computations of certain IMG functions; in particular all automatic conversions between IMG machines and Mealy machines.

The command `SetInfoLevel(InfoIMG,2);` requests a little more information, and in particular prints intermediate results in potentially long calculations such as...

The command `SetInfoLevel(InfoIMG,3);` ensures that IMG will print information every few seconds or so. This is useful to gain confidence that the program is not stuck due to a programming bug by the author of IMG.

References

[Ale83] S. V. Aleshin. A free group of finite automata. *Vestnik Moskov. Univ. Ser. I Mat. Mekh.*, (4):12–14, 1983. 6

[BFH92] B. Bielefeld, Y. Fisher, and J. Hubbard. The classification of critically preperiodic polynomials as dynamical systems. *J. Amer. Math. Soc.*, 5(4):721–762, 1992. 15

[BGN03] L. Bartholdi, R. I. Grigorchuk, and V. Nekrashevych. From fractal groups to fractal sets. In *Fractals in Graz 2001*, Trends Math., pages 25–118. Birkhäuser, Basel, 2003. 5

[BGŠ03] L. Bartholdi, R. I. Grigorchuk, and Z. Šuniḱ. Branch groups. In *Handbook of algebra, Vol. 3*, pages 989–1112. North-Holland, Amsterdam, 2003. 5

[BV05] L. Bartholdi and B. Virág. Amenability via random walks. *Duke Math. J.*, 130(1):39–56, 2005. 6

[DH84] A. Douady and J. H. Hubbard. *Étude dynamique des polynômes complexes. Partie I*, volume 84 of *Publications Mathématiques d'Orsay [Mathematical Publications of Orsay]*. Université de Paris-Sud, Département de Mathématiques, Orsay, 1984. 15

[DH85] A. Douady and J. H. Hubbard. *Étude dynamique des polynômes complexes. Partie II*, volume 85 of *Publications Mathématiques d'Orsay [Mathematical Publications of Orsay]*. Université de Paris-Sud, Département de Mathématiques, Orsay, 1985. With the collaboration of P. Lavaurs, Tan Lei and P. Sentenac. 15

[Gri80] R. I. Grigorchuk. On Burnside's problem on periodic groups. *Funktsional. Anal. i Prilozhen.*, 14(1):53–54, 1980. 6

[GS83] N. Gupta and S. N. Sidki. On the Burnside problem for periodic groups. *Math. Z.*, 182(3):385–388, 1983. 6

[GŻ02] R. I. Grigorchuk and A. Żuk. On a torsion-free weakly branch group defined by a three state automaton. *Internat. J. Algebra Comput.*, 12(1-2):223–246, 2002. International Conference on Geometric and Combinatorial Methods in Group Theory and Semigroup Theory (Lincoln, NE, 2000). 6

[HS94] J. H. Hubbard and D. Schleicher. The spider algorithm. In *Complex dynamical systems (Cincinnati, OH, 1994)*, volume 49 of *Proc. Sympos. Appl. Math.*, pages 155–180. Amer. Math. Soc., Providence, RI, 1994. 22, 24

[Nek05] V. Nekrashevych. *Self-similar groups*, volume 117 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 2005. 10, 13, 30

[Poi] A. Poirier. On postcritically finite polynomials, part 1: critical portraits. Stony Brook IMS 1993/5. 15, 33

[Tan02] D. T. Tan. Quadratische morphismen. Diplomarbeit at ETHZ, under the supervision of R. Pink, 2002. 21

Index