Iterated monodromy groups

Version 0.1.1

02/01/2014

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 as a part of the GAP package FR. It expanded so much that I decided in 2012 to split it off, so as to keep more cleanly separated the group theory on one side, and the complex analysis, on the other side.

Contents

Chapter 1

Licensing

Chapter 2

IMG package

2.1 A brief mathematical introduction

This chapter assumes that you have no familiarity with iterated monodromy groups. 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 and iterated monodromy groups, see [BGN03] or [Nek05].

Basic knowledge of the package FR is assumed, however. Please refer to its online documentation, or the same references as above.

The fundamental objects are *Thurston maps*: branched self-coverings of the sphere S^2. These are continuous maps $f : S^2 \to S^2$ that, locally, are modelled on the complex map $z \mapsto z^n$. The *critical points* are those points z at which the map is modelled on $z \mapsto z^n$ for some $n > 1$. The *post-critical set* is the union P of strictly forward orbits of critical points. It is assumed finite.

Two Thurston maps $f : (S^2, P) \to (S^2, P)$ and $g : (S^2, Q) \to (S^2, Q)$ are *combinatorially equivalent* if they are isotopic through isotopies $(S^2, P) \to (S^2, Q)$ that are constant on P.

Denote by G the fundamental group $\pi_1(S^2 \setminus P, *)$ at a basepoint $*$. Then combinatorial equivalence classes of Thurston maps are classified by wreath recursions for G, namely homomorphisms $\phi : G \to G \wr Sym_d$. They are encoded, in IMG, by a new kind of FR machine, called a *sphere machine*. 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.

By a fundamental theorem of Thurston, every Thurston map (apart from a few low-complexity examples) is either combinatorially equivalent to a rational map, or is *obstructed*: there exists a system of curves on $S^2 \setminus P$ with some appropriate non-contraction property.

The operations in this package let one manipulate Thurston maps, and in particular

- compute the rational map, or the obstruction, associated with a sphere machine;
- compute the sphere machine associated with a rational map;
- construct sphere machines algebraically, and identify their combinatorial parameters, in the case of polynomial maps;
- compute combinations of sphere machines, such as matings.

2.2 An example session

This is a brief introduction describing some of the simpler features of the IMG package. It assumes you have some familiarity with the theory of groups defined by automata, and of holomorphic dynamical systems; 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("img");
true
```

Many maps and sphere machines are predefined by the IMG, see Chapter 5.

We may start by defining a machine by its polynomial:

Example

```
gap> basilica := PolynomialSphereMachine(2,[1/3]);
<sphere machine with alphabet [ 1 .. 2 ] and adder FRElement(...,f3) on Group( [ f1, f2, f3 ] )/
3*f2*f1 ]>
gap> Display(basilica);
 G  |      1         2
----+---------+---------+
 f1 | f1^-1,2   f3^-1,1
 f2 |    f1,1    <id>,2
 f3 |    f3,2    <id>,1
----+---------+---------+
Adding element: FRElement(...,f3)
Relators: [ f3*f2*f1 ]
gap> P1MapBySphereMachine(basilica);
<z^2-1._z>
```

We have just created the basilica machine, with group $G = \langle f1, f2, f3 | f3f2f1 \rangle$, and computed the corresponding rational map.

We now compute the mating of the basilica with itself: that is the Thurston map which acts as $z^2 - 1$ on the upper hemisphere, by angle doubling on the equator, and also as $z^2 - 1$ on the lower hemisphere. It turns out that this map is obstructed, i.e. does not have a realization as a rational map. The obstruction consists of a curve separating the points -1 and 0 on both hemispheres:

Example

```
gap> basilica2 := Mating(basilica,basilica);
<sphere machine with alphabet [ 1 .. 2 ] on Group( [ f1, f2, g1, g2 ] ) / [ f2*f1*g2*g1 ]>
gap> P1MapBySphereMachine(last);
rec(
  machine := <sphere machine with alphabet [ 1, 2 ] on Group( [ f1, f2, g1, g2 ] ) / [ f2*f1*g2*
]>, matrix := [ [ 1 ] ], multicurve := [ f1*g1^G ] )
```

Chapter 3

Sphere groups and machines

3.1 Sphere groups

Fundamental groups of punctured spheres, and of sphere orbifolds, are defined in IMG as a special class of finitely presented groups. The generators of the group represent the punctures of the sphere, or more generally its orbispace points.

3.1.1 IsSphereGroup

▷ `IsSphereGroup` (filter)

A sphere group is a special kind of finitely presented group, in which exactly one relation is a product, in some order, of all the generators, and all the other relations (possibly none) are powers of generators.

Sphere groups are used to represent the fundamental groups of punctured spheres, or more generally orbifolds whose underlying space is a sphere.

3.1.2 IsomorphismSphereGroup

▷ `IsomorphismSphereGroup(`g`)` (attribute)
▷ `AsSphereGroup(`g`)` (attribute)

These functions compute an isomorphism from g to a sphere group; the first form returns the isomorphism, while the second one returns its image.

3.1.3 EulerCharacteristic

▷ `EulerCharacteristic(`g`)` (attribute)

Returns: The Euler characteristic of g.

The Euler characteristic of a free group of rank n is $1-n$; and it multiplies by the index on subgroups. A sphere group is finite if and only if its Euler characteristic is positive, and is virtually abelian if and only if its Euler characteristic is 0.

3.1.4 RankOfSphereGroup

▷ `RankOfSphereGroup(`*g*`)` (attribute)

Returns: The number of generators of *g*.

3.1.5 OrderingOfSphereGroup

▷ `OrderingOfSphereGroup(`*g*`)` (attribute)

Returns: The list of the orders of the generators.

This attribute has the property that `Product(GeneratorsOfGroup(g){OrderingOfSphereGroup(g)})` is the identity.

3.1.6 ExponentsOfSphereGroup

▷ `ExponentsOfSphereGroup(`*g*`)` (attribute)

Returns: The list of exponents of the generators.

This attribute has the property that `GeneratorsOfGroup(g)[i]^ExponentsOfSphereGroup(g)[i]` is the identity for all `i`. If an element has infinite order, the value stored is `0`.

3.1.7 IsomorphismFreeGroup

▷ `IsomorphismFreeGroup(`*g*`)` (attribute)

Returns: An isomorphism to a free group, if it exists.

If *g* was created as a sphere group with all exponents infinity, then *g* is isomorphic to a free group on all the generators but one; this attribute stores such an isomorphism.

3.1.8 SphereGroup

▷ `SphereGroup(`*ordering[, exponent]*`)` (function)

Returns: A new sphere group.

ordering is either a list of integers, describing the order of the generators that is to be trivial; or an integer `m`, in which case the ordering is `[m,m-1,..,1]`.

The optional second argument *exponent* is a list of integers describing the exponents of the generators. The value `0` specifies a generator of infinite order.

3.1.9 IsSphereConjugacyClass

▷ `IsSphereConjugacyClass` (filter)

Elements of a sphere group represent based loops on a punctured sphere. Loops (without specified basepoint) are represented by conjugacy classes. A *multicurve* is a collection of non-intersecting loops.

Conjugacy classes may be raised to integer powers; the *n*th power of a conjugacy class is the conjugacy class of the *n*th power of an element.

3.1.10 IsPeripheral

▷ `IsPeripheral(c)` (property)

Returns: Whether the conjugacy class `c` is peripheral.

A conjugacy class is *peripheral* if it contains a generator of the sphere group.

3.1.11 PeripheralClasses

▷ `PeripheralClasses(g)` (attribute)

Returns: The peripheral conjugacy classes of `g`.

3.1.12 IntersectionNumber

▷ `IntersectionNumber(c, d)` (attribute)

Returns: The intersection number of the conjugacy classes `c` and `d`.

The *geometric intersection number* of two loops is the minimal number of intersections they may have. The *self-intersection number* of a loop is the intersection number of the loop with a small translate.

3.1.13 AutomorphismGroup

▷ `AutomorphismGroup(g)` (attribute)

▷ `EpimorphismToOut(a)` (attribute)

This function computes the *pure* automorphism group of the sphere group `g`, namely the group of automorphisms that preserves all the peripheral conjugacy classes (conjugacy classes of generators).

The attribute `EpimorphismToOut` stores an epimorphism from the automorphism group to the group of outer automorphisms.

3.1.14 AmalgamateFreeProduct

▷ `AmalgamateFreeProduct(g, h, x, y)` (operation)

This function computes the amalgamated free product of two sphere groups `g` and `h`, along the cyclic subgroups $\langle x \rangle$ of `g` and $\langle y \rangle$ of `h`.

The attribute `EmbeddingsOfAmalgamatedFreeProduct` is a list of length two, storing the embeddings of `g` and `h` respectively into the amalgam.

3.2 Sphere machines

Sphere machines are simply group FR machines (see Section **fr: FRMachineNC (family,free,listlist,list)**) whose underlying `StateSet` (**fr: StateSet (FR machine)**) is a sphere group. DeclareProperty("IsSphereMachine", IsFRMachine);

3.2.1 IsSphereMachine

▷ `IsSphereMachine(m)` (filter)
▷ `IsPolynomialSphereMachine(m)` (filter)

The categories of *Sphere* and *polynomial* machines. Sphere machines are group FR machines whose underlying group is a sphere group, see `SphereGroup` (3.1.8).

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` (3.3.7); see `AsPolynomialSphereMachine` (3.3.8). 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.

DeclareAttribute("AsSphereMachine", IsGroupFRMachine); DeclareOperation("AsSphereMachine", [IsGroupFRMachine, IsWord]); DeclareOperation("AsSphereMachine", [IsGroupFRMachine, IsSphereGroup]);

3.2.2 AsSphereMachine

▷ `AsSphereMachine(m[, w])` (operation)

Returns: A sphere machine.

This function creates a new sphere 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 relator; if `w` is a sphere group, then it is used as the new stateset. Finally, if no relator and no group is specified, and the product (in some ordering) of the generators is trivial, then that product is used as relator. In other cases, the method returns `fail`.

A standard FR machine can be recovered from a sphere machine by `AsGroupFRMachine` (**fr: AsGroupFRMachine**), `AsMonoidFRMachine` (**fr: AsMonoidFRMachine**), and `AsSemigroupFRMachine` (**fr: 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> AsSphereMachine(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
```

DeclareAttribute("CleanedSphereMachine",IsSphereMachine);

3.2.3 CleanedSphereMachine

▷ `CleanedSphereMachine(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.

DeclareGlobalFunction("NewSphereMachine");

3.2.4 NewSphereMachine

▷ `NewSphereMachine(...)` (attribute)

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

This command constructs a new sphere machine, in a format similar to `FRGroup` (**fr: 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. The underlying group of the machine is a sphere group.

`word-i` is allowed to be the empty string; and the "<...>" may be skipped altogether. Each `word-i` may also contain inverses.

The extra final arguments describe relations in the underlying sphere group; at least one relation is required, the product of the generators in an appropriate order.

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

Example

```
gap> fold1 := NewSphereMachine("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)","a*B*A*b");
gap> <FR machine with alphabet [ 1, 2, 3, 4, 5, 6 ] on Group( [ a, b, A, B ] )/[ a*B*A*b ]>
gap> fold2 := NewSphereMachine("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)","a*B*A*b");;
gap> P1MapBySphereMachine(fold1); P1MapBySphereMachine(fold2);
...
```

3.3 Polynomial sphere machines

Polynomial sphere machines have a special extra attribute, an `AddingElement` (3.3.7). This is an element of the underlying FR group, which acts as an adding element on the machine's alphabet. It represents a fixed point of a Thurston map of maximal ramification; typically, the point ∞ of a polynomial. DeclareOperation("PolynomialMealyMachine",[IsPosInt,IsList,IsList]); DeclareOperation("PolynomialMealyMachine",[IsPosInt,IsList]); DeclareOperation("PolynomialSphereMachine",[IsPosInt,IsList,IsList,IsRecord]); DeclareOperation("PolynomialSphereMachine",[IsPosInt,IsList,IsList]); DeclareOperation("PolynomialSphereMachine",[IsPosInt,IsList,IsRecord]); DeclareOperation("PolynomialSphereMachine",[IsPosInt,IsList]);

3.3.1 PolynomialSphereMachine

▷ `PolynomialSphereMachine(d, per[, pre][, options])` (operation)

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

Returns: A sphere or Mealy machine.

This function creates a sphere 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 sphere 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 sphere machine, one may specify a record `options`, which may contain the following fields: `mealy` (boolean, default `false`) specifies if a formal construction is required; `adding` specifying that the adding machine should have the most compact representation; and `orbispace` (boolean, default `false`) asking the constructed group to have orbispace points of minimal degree.

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 IMG 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 sphere machine, is `SupportingRays` (3.3.2).

Example

```
gap> PolynomialSphereMachine(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(PolynomialSphereMachine(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(PolynomialSphereMachine(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> PolynomialSphereMachine(2,[],[5/12]);
<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" ] ]
gap> PolynomialSphereMachine(2,[],[5/12],rec(formal:=true));
<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" ] ]
```

The following construct the examples in Poirier's paper:

```
PoirierExamples := function(arg)
    if arg=[1] then
        return PolynomialSphereMachine(2,[1/7],[]);
    elif arg=[2] then
        return PolynomialSphereMachine(2,[],[1/2]);
    elif arg=[3,1] then
        return PolynomialSphereMachine(2,[],[5/12]);
    elif arg=[3,2] then
        return PolynomialSphereMachine(2,[],[7/12]);
    elif arg=[4,1] then
        return PolynomialSphereMachine(3,[[3/4,1/12],[1/4,7/12]],[]);
    elif arg=[4,2] then
        return PolynomialSphereMachine(3,[[7/8,5/24],[5/8,7/24]],[]);
    elif arg=[4,3] then
        return PolynomialSphereMachine(3,[[1/8,19/24],[3/8,17/24]],[]);
    elif arg=[5] then
        return PolynomialSphereMachine(3,[[3/4,1/12],[3/8,17/24]],[]);
    elif arg=[6,1] then
        return PolynomialSphereMachine(4,[],[[1/4,3/4],[1/16,13/16],[5/16,9/16]]);
    elif arg=[6,2] then
        return PolynomialSphereMachine(4,[],[[1/4,3/4],[3/16,15/16],[7/16,11/16]]);
    elif arg=[7] then
        return PolynomialSphereMachine(5,[[0,4/5],[1/5,2/5,3/5]],[[1/5,4/5]]);
    elif arg=[9,1] then
        return PolynomialSphereMachine(3,[[0,1/3],[5/9,8/9]],[]);
    elif arg=[9,2] then
        return PolynomialSphereMachine(3,[[0,1/3]],[[5/9,8/9]]);
    else
        Error("Unknown Poirier example ",arg);
    fi;
end;
```

DeclareAttribute("SupportingRays",IsFRMachine);

3.3.2 SupportingRays

▷ `SupportingRays(`*m*`)` (attribute)

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

This operation is the inverse of `PolynomialSphereMachine` (3.3.1): 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 := PolynomialSphereMachine(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(PolynomialSphereMachine,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 := PolynomialSphereMachine(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 )
```

DeclareAttribute("NormalizedPolynomialSphereMachine",IsSphereMachine);

3.3.3 NormalizedPolynomialSphereMachine

▷ `NormalizedPolynomialSphereMachine(`m`)` (attribute)

Returns: A polynomial sphere machine.

This function returns a new sphere 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 $i \mapsto i-1$.

DeclareAttribute("SimplifiedSphereMachine",IsSphereMachine);

3.3.4 SimplifiedSphereMachine

▷ `SimplifiedSphereMachine(`*m*`)` (attribute)

Returns: A simpler sphere machine.

This function returns a new sphere 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=SimplifiedSphereMachine(m)`, then `m^Product(Correspondence(n))=n`.

Example

```
gap> r := PolynomialSphereMachine(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 := SimplifiedSphereMachine(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
```

DeclareOperation("Mating",[IsPolynomialSphereMachine,IsPolynomialSphereMachine]); DeclareOperation("Mating",[IsPolynomialSphereMachine,IsPolynomialSphereMachine,IsBool]); DeclareAttribute("EquatorElement",IsSphereMachine); DeclareAttribute("EquatorTwist",IsSphereMachine);

3.3.5 Mating

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

▷ `EquatorElement(`*m*`)` (attribute)

▷ `EquatorTwist(`*m*`)` (attribute)

Returns: A sphere machine.

This function "mates" two polynomial sphere 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` (3.3.2).

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

The attribute `EquatorElement` is set, and records the original adding elements of *m1,m2*, which have become the equator of the mating.

Note that there are $d-1$ different matings between polynomials of degree d: each has $d-1$ fixed rays at angles $2\pi ik/(d-1)$. This command constructs the mating in which rays at angle 0 are matched to each other. To obtain the other matings, multiply the machine by a power of its `EquatorTwist`.

Example

```
gap> # the Tan-Shishikura examples
gap> SetP1Points(PMCOMPLEX);
gap> z := Indeterminate(@IMG.field);;
gap> a := RootsFloat((z-1)*(3*z^2-2*z^3)+1);;
gap> c := RootsFloat((z^2+1)^3*z^2+1);;
gap> am := List(a,a->SphereMachine((a-1)*(3*P1z^2-2*P1z^3)+1));;
gap> cm := List(c,c->SphereMachine(P1z^3+c));;
gap> m := ListX(am,cm,Mating);;
gap> # m[1] is realizable
gap> P1MapBySphereMachine(m[1]);
((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[29] is obstructed, and is the original Tan-Shishikura map
gap> P1MapBySphereMachine(m[29]);
rec(
  machine := <sphere machine with alphabet [ 1, 2, 3 ] on Group( [ f1, f2, f3, g1, g2,\
 g3 ] ) / [ f3*f2*f1*g3*g2*g1 ]>, matrix := [ [ 1, 1/2 ], [ 1, 0 ] ],
  multicurve := [ (f1*f3*f2)^2*f1*g1^-1*(g3*g2*g1)^3*f2^-1*f3*f2^G,
      f1^-1*f2^-1*(f3*f2*f1)^4*g2*(g3*g2*g1)^3^G ] )
gap> but the other mating of the same polynomials is not obstructed:
gap> P1MapBySphereMatrix(m[29]*EquatorTwist(m[29]));
<((-1.4495156808145406+0.44648102591936722i_z)*z^3+
(-1.1286550578708263-0.40162285610021786i_z)*z^2+
(1.0326873942952213-0.11770300021984977i_z)*z+
(1.0940864612174037+0.24650956710141259i_z))/
((0.85917327990384307-0.8755042485587835i_z)*z^3+
(0.9573881709899621-0.14875521653926685i_z)*z^2+
(-0.68923589444039035+0.48120812618585479i_z)*z+1._z)>
```

DeclareProperty("IsKneadingMachine",IsFRMachine); DeclareProperty("IsPlanarKneadingMachine",IsFRMachine); InstallTrueMethod(IsBoundedFRMachine,IsKneadingMachine); InstallTrueMethod(IsLevelTransitive,IsKneadingMachine);

3.3.6 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
```

DeclareAttribute("AddingElement", IsSphereMachine);

3.3.7 AddingElement

▷ `AddingElement(m)` (attribute)

Returns: The element generating the adding submachine.

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 `CleanedSphereMachine` 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 := AsSphereMachine(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 := CleanedSphereMachine(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
```

DeclareSynonym("IsPolynomialSphereMachine",IsSphereMachine and HasAddingElement); DeclareAttribute("AsPolynomialSphereMachine",IsFRMachine); DeclareOperation("AsPolynomialSphereMachine",[IsFRMachine,IsWord]);

3.3.8 AsPolynomialSphereMachine

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

Returns: A polynomial sphere machine.

The first function creates a new polynomial sphere machine, starting from a group or Mealy machine. A *polynomial* machine is one that has a distinguished adding element, `AddingElement` (3.3.7).

If the argument is a Mealy machine, it must be planar (see `IsPlanarKneadingMachine` (3.3.6)). 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.

Example

```
gap> M := PolynomialMealyMachine(2,[1/7],[]);
<Mealy machine on alphabet [ 1 .. 2 ] with 4 states>
gap> Mi := AsPolynomialSphereMachine(M);
!!!
```

DeclareOperation("LiftOfConjugacyClass", [IsGroupFRMachine,IsConjugacyClassGroupRep]);

3.3.9 LiftOfConjugacyClass

▷ `LiftOfConjugacyClass(`*`m, c`*`)` (operation)

Returns: A list of conjugacy classes and multiplicities.

This command computes the preimage of the conjugacy class *`c`* by the sphere machine *`m`*, namely, it applies the wreath recursion to a representative of *`c`* and collects the products on all cycles. It returns then a list of pairs `[cc,len]` where `cc` is the conjugacy class of a product on a cycle, and `len` is the length of the cycle.

DeclareAttribute("ComplexConjugate", IsFRMachine); # already declared for arithmetic objects

3.3.10 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 := PolynomialSphereMachine(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}
```

DeclareOperation("RotatedSpider", [IsPolynomialSphereMachine]); DeclareOperation("RotatedSpider", [IsPolynomialSphereMachine, IsInt]);

3.3.11 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 := PolynomialSphereMachine(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} ]
```

DeclareAttribute("KneadingSequence", IsRat);

3.3.12 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" ]
```

DeclareGlobalFunction("AllInternalAddresses");

3.3.13 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 ] ] ]
```

DeclareGlobalFunction("ExternalAnglesRelation");

3.3.14 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 ] ]
```

DeclareGlobalFunction("ExternalAngle");

3.3.15 ExternalAngle

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

Returns: The external angle identifying *machine*.

In case *machine* is the sphere 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` (3.3.14).

Example

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

3.4 Automorphisms of sphere machines

Consider a sphere group G and its automorphism group A. If M is a sphere machine for the group G, then pre- and post-composition by automorphisms in A gives new sphere machines. The set of such sphere machines is naturally described by a machine for the group A. DeclareOperation("AutomorphismVirtualEndomorphism",[IsGroupHomomorphism]); DeclareOperation("AutomorphismSphereMachine",[IsSphereMachine]);

3.4.1 AutomorphismVirtualEndomorphism

▷ `AutomorphismVirtualEndomorphism(`*v*`)` (attribute)

▷ `AutomorphismSphereMachine(`*m*`)` (attribute)

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

Let `m` be a sphere 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> SetP1Points(PMCOMPLEX);
gap> z := Indeterminate(@IMG.field);;
gap> # a Sierpinski carpet map without multicurves
gap> m := SphereMachine((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> AutomorphismSphereMachine(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 := PolynomialSphereMachine(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
```

Chapter 4

Holomorphic maps

A large part of IMG consists of code that manipulates rational maps and complex coordinates on spheres.

4.1 $\mathbb{P}^1$ points

Points on the sphere are represented by complex numbers, possibly infinity. These complex numbers are encapsulated in `P1Point` (4.1.1) objects. DeclareCategory("IsP1Point",IsObject); DeclareCategoryCollections("IsP1Point"); DeclareCategoryCollections("IsP1PointCollection"); DeclareSynonym("IsP1PointList",IsP1PointCollection and IsList); DeclareCategory("IsIEEE754P1Point",IsP1Point); BindGlobal("P1PointsFamily",NewFamily("P1PointsFamily",IsP1Point)); BindGlobal("TYPE_P1POINT",NewType(P1PointsFamily,IsP1Point and IsPositionalObjectRep)); BindGlobal("TYPE_IEEE754P1POINT",NewType(P1PointsFamily,IsIEEE754P1Point and IsDataObjectRep)); DeclareOperation("P1Point",[IsFloat]); DeclareOperation("P1Point",[IsRat]); DeclareOperation("P1Point",[IsInfinity]); DeclareOperation("P1Point",[IsFloat,IsFloat]);

4.1.1 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.

DeclareAttribute("P1Coordinate",IsP1Point);

4.1.2 P1Coordinate

▷ `P1Coordinate(`*`p`*`)` (function)

Returns: The complex number represented by *p*.

DeclareOperation("CleanedP1Point",[IsP1Point,IsFloat]); DeclareOperation("CleanedP1Point",[IsP1Point]);

4.1.3 CleanedP1Point

▷ `CleanedP1Point(p[, prec])` (function)

Returns: *p*, rounded towards 0/1/infinity/reals at precision *prec*.

DeclareGlobalVariable("P1infinity"); DeclareOperation("P1INFINITY@",[IsP1Point]); DeclareGlobalVariable("P1one"); DeclareGlobalVariable("P1zero");

4.1.4 P1infinity

▷ `P1infinity` (global variable)

▷ `P1one` (global variable)

▷ `P1zero` (global variable)

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

DeclareAttribute("P1Antipode",IsP1Point);

4.1.5 P1Antipode

▷ `P1Antipode(p)` (function)

Returns: The antipode of *p* on the Riemann sphere.

DeclareOperation("P1Barycentre",[IsP1PointList]); DeclareOperation("P1Barycentre",[IsP1Point]); DeclareOperation("P1Barycentre",[IsP1Point,IsP1Point]); DeclareOperation("P1Barycentre",[IsP1Point,IsP1Point,IsP1Point]);

4.1.6 P1Barycentre

▷ `P1Barycentre(points, ...)` (function)

Returns: The barycentre of its arguments (which can also be a list of P1 points).

DeclareOperation("P1Circumcentre",[IsP1Point,IsP1Point,IsP1Point]);

4.1.7 P1Circumcentre

▷ `P1Circumcentre(p, q, r)` (function)

Returns: The centre of the smallest disk containing *p,q,r*.

DeclareOperation("P1Distance",[IsP1Point,IsP1Point]);

4.1.8 P1Distance

▷ `P1Distance(p, q)` (function)

Returns: The spherical distance from *p* to *q*.

DeclareOperation("P1Midpoint",[IsP1Point,IsP1Point]);

4.1.9 P1Midpoint

▷ `P1Midpoint(p, q)` (function)

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

DeclareAttribute("P1Sphere",IsList);

4.1.10 P1Sphere

▷ `P1Sphere(v)` (function)

Returns: The P1 point corresponding to `v` in $\mathbb{R}^3$.

DeclareAttribute("SphereP1",IsP1Point);

4.1.11 SphereP1

▷ `SphereP1(p)` (function)

Returns: The coordinates in $\mathbb{R}^3$ of `p`.

DeclareAttribute("SphereP1Y",IsP1Point);

4.1.12 SphereP1Y

▷ `SphereP1Y(p)` (function)

Returns: The Y coordinate in $\mathbb{R}^3$ of `p`.

DeclareOperation("P1XRatio",[IsP1Point,IsP1Point,IsP1Point,IsP1Point]); DeclareOperation("XRatio",[IsP1Point,IsP1Point,IsP1Point,IsP1Point]);

4.1.13 P1XRatio

▷ `P1XRatio(p, q, r, s)` (function)

▷ `XRatio(p, q, r, s)` (function)

Returns: The cross ratio of `p, q, r, s`.

The cross ratio of four points `p,q,r,s` is defined as `(p-r)(q-s)/(p-s)(q-r)`. The values `P1zero,P1one,P1infinity` correspond respectively to the special cases `(p=r or q=s)`, `(p=q or r=s)`, `(p=s or q=r)`.

In the first form, the result is a P1 point. In the second form, it is a complex number.

DeclareOperation("CollectedP1Points",[IsP1PointList]); DeclareOperation("CollectedP1Points",[IsP1PointList,IsFloat]);

4.1.14 CollectedP1Points

▷ `CollectedP1Points(p1points[, precision])` (operation)

Returns: A list of pairs `[point,multiplicity]`.

Collects the points in *p1points*; points at distance at most *precision* are considered equal, and the barycentre of the clustered points is returned.

If the argument *precision* is not supplied, `@IMG.p1eps` is taken.

DeclareOperation("MatchP1Points",[IsP1PointList,IsP1PointCollColl,IsFloat]); DeclareOperation("MatchP1Points",[IsP1PointList,IsP1PointCollColl]);

4.1.15 MatchP1Points

▷ MatchP1Points(*p1pointsA, p1pointsB[, separation]*) (operation)

Returns: A list of giving closest point in *p1pointsB* to points in *p1pointsA*, or fail.

Finds for each point *p1pointsA[i]* the closest point *p1pointB[p[i]]*. If the next-closest is at least *separation* further away for all i, then the list p is returned. Otherwise, fail is returned.

If the argument *separation* is not supplied, 2 is taken.

DeclareOperation("ClosestP1Point",[IsP1PointList,IsP1Point]);

4.1.16 ClosestP1Point

▷ ClosestP1Point(*p1points, p1point*) (operation)

Returns: The point in *p1points* closest to *p1point*.

DeclareSynonym("IsP1Map",IsUnivariateRationalFunction and IsFloatRationalFunction); DeclareCategory("IsIEEE754P1Map",IsP1Map); BindGlobal("TYPE_IEEE754P1MAP", NewType(RationalFunctionsFamily(PMCOMPLEX_PSEUDOFIELD), IsIEEE754P1Map and IsDataObjectRep));

4.1.17 IsP1Map

▷ IsP1Map (filter)

P1 maps are stored more efficiently than rational functions, but are otherwise equivalent.

DeclareOperation("MoebiusMap",[IsP1Point]); DeclareOperation("MoebiusMap",[IsP1Point,IsP1Point]); DeclareOperation("MoebiusMap",[IsP1Point,IsP1Point,IsP1Point]); DeclareOperation("MoebiusMap",[IsP1Point,IsP1Point,IsP1Point,IsP1Point,IsP1Point,IsP1Point]); DeclareOperation("MoebiusMap",[IsP1PointList]); DeclareOperation("MoebiusMap",[IsP1PointList,IsP1PointList]);

4.1.18 MoebiusMap

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

Returns: A new Möbius transformation.

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 0,1,P1infinity to *p,q,r* respectively; in the third case, the map sending 0,P1infinity to *p,q* respectively, and of the form $(z-p)/(z-q)$; and in the fourth case, a rotation sending *P1infinity* to *p*.

DeclareOperation("P1ROTATION@",[IsP1Point,IsP1PointList,IsP1PointList]); DeclareGlobalFunction("P1MapRotatingP1Points");

4.1.19 P1MapRotatingP1Points

▷ P1MapRotatingP1Points(*points[, oldpoints]*) (operation)

Returns: A Möbius rotation sending the last of *points* to P1infinity.

A Möbius rotation is computed that sends the last of *points* to P1infinity and, assuming the last point in *oldpoints* is also P1infinity, matches the points in *points* and *oldpoints* as closely as possible.

DeclareOperation("P1MapNormalizingP1Points",[IsP1PointList]); DeclareOperation("P1MapNormalizingP1Points",[IsP1PointList,IsP1PointList]);

4.1.20 P1MapNormalizingP1Points

▷ P1MapNormalizingP1Points(*points[, oldpoints]*) (operation)

Returns: A Möbius transformation sending the last of *points* to P1infinity.

A Möbius transformation is computed that sends the last of *points* to P1infinity and makes the barycentre of the points in $\mathbb{R}^3$ as close as possible to the origin. If a list *oldpoints* is also given, the Möbius transformation computed rotates about infinity so as to match the points in *points* and *oldpoints* as closely as possible; this then determines the transformation uniquely.

DeclareGlobalVariable("P1z");

4.1.21 P1z

▷ P1z (global variable)

The identity Möbius transformation.

DeclareGlobalFunction("P1Monomial");

4.1.22 P1Monomial

▷ P1Monomial(*n*) (function)

Returns: The rational function z^n.

DeclareOperation("CleanedP1Map",[IsP1Map,IsFloat]); DeclareOperation("CleanedP1Map",[IsP1Map]);

4.1.23 CleanedP1Map

▷ CleanedP1Map(*map[, prec]*) (operation)

Returns: *map*, with coefficients rounded using *prec*.

DeclareSynonym("CompositionP1Map",CompositionMapping2);

4.1.24 CompositionP1Map

▷ CompositionP1Map(*map1, ...*) (operation)

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

DeclareSynonym("InverseP1Map",InverseGeneralMapping);

4.1.25 InverseP1Map

▷ InverseP1Map(*map*) (operation)

Returns: The functional inverse of the Möbius transformation *map*.

DeclareAttribute("ComplexConjugate",IsP1Map); # already there for numerical objects

4.1.26 ComplexConjugate (P1 map)

▷ `ComplexConjugate(`*`map`*`)` (operation)

Returns: The complex conjugated map.

DeclareOperation("ConjugatedP1Map",[IsP1Map,IsP1Map]);

4.1.27 ConjugatedP1Map

▷ `ConjugatedP1Map(`*`map, mobius`*`)` (operation)

Returns: The map `CompositionP1Map(InverseP1Map(mobius),map,mobius)`.

DeclareAttribute("CoefficientsOfP1Map",IsP1Map);

4.1.28 CoefficientsOfP1Map

▷ `CoefficientsOfP1Map(`*`map`*`)` (operation)

Returns: Coefficients of numerator and denominator of *`map`*, lowest degree first.

DeclareGlobalFunction("P1MapByCoefficients"); DeclareOperation("P1MAPBYCOEFFICIENTS2@",[IsObject,IsList,IsList]);

4.1.29 P1MapByCoefficients

▷ `P1MapByCoefficients(`*`numer, denom`*`)` (operation)

Returns: The P1 map with numerator coefficients *`numer`* and denominator *`denom`*, lowest degree first.

DeclareAttribute("NumeratorP1Map",IsP1Map); DeclareAttribute("DenominatorP1Map",IsP1Map);

4.1.30 NumeratorP1Map

▷ `NumeratorP1Map(`*`map`*`)` (attribute)

▷ `DenominatorP1Map(`*`map`*`)` (attribute)

Returns: The numerator/denominator of *`map`*.

DeclareOperation("P1MapByZerosPoles",[IsP1PointList,IsP1PointList,IsP1Point,IsP1Point]);

4.1.31 P1MapByZerosPoles

▷ `P1MapByZerosPoles(`*`zeros, poles, src, dest`*`)` (operation)

Returns: The P1 map with specified zeros and poles, and sending *`src`* to *`dest`*.

DeclareOperation("P1Path",[IsP1Point,IsP1Point]);

4.1.32 P1Path

▷ `P1Path(`*`p, q`*`)` (operation)

Returns: The P1 map sending `0` to *`p`* and `1` to *`q`*.

DeclareAttribute("DegreeOfP1Map",IsP1Map);

4.1.33 DegreeOfP1Map

▷ DegreeOfP1Map(*map*) (operation)

Returns: The degree of *map*.

DeclareSynonym("P1Image",ImageElm); DeclareOperation("ImageElm",[IsP1Map,IsP1Point]);

4.1.34 P1Image

▷ P1Image(*map, p1point*) (operation)

Returns: The image of *p1point* under *map*.

DeclareSynonym("P1PreImage",PreImageElm); DeclareOperation("PreImageElm",[IsP1Map,IsP1Point]);

4.1.35 P1PreImage

▷ P1PreImage(*map, p1point*) (operation)

Returns: The preimage of *p1point* under *map*.

DeclareSynonym("P1PreImages",PreImagesElm); DeclareOperation("PreImagesElm",[IsP1Map,IsP1Point]);

4.1.36 P1PreImages

▷ P1PreImages(*map, p1point*) (operation)

Returns: The preimages of *p1point* under *map*.

DeclareAttribute("Primitive",IsP1Map); DeclareAttribute("Derivative",IsP1Map);

4.1.37 Primitive

▷ Primitive(*map*) (operation)

▷ Derivative(*map*) (operation)

Returns: The [anti]derivative of the rational map *map*.

DeclareOperation("P1MapScaling",[IsP1Map,IsP1Point]);

4.1.38 P1MapScaling

▷ P1MapScaling(*map, p1point*) (operation)

Returns: The scaling factor from *point* to *map(point)*.

DeclareAttribute("CriticalPointsOfP1Map",IsP1Map);

4.1.39 CriticalPointsOfP1Map

▷ CriticalPointsOfP1Map(*map*) (attribute)

Returns: The critical points of *map*.

DeclareAttribute("AsP1Map",IsScalar);

4.1.40 AsP1Map

▷ `AsP1Map(rat)` (operation)

Returns: The P1 map given by the rational function `rat`.

DeclareOperation("P1MapSL2",[IsMatrix]);

4.1.41 P1MapSL2

▷ `P1MapSL2(mat)` (operation)

Returns: The Möbius P1 map given by the 2x2 matrix `mat`.

DeclareAttribute("SL2P1Map",IsP1Map);

4.1.42 SL2P1Map

▷ `SL2P1Map(map)` (operation)

Returns: The matrix of the Möbius P1 map `map`.

DeclareGlobalFunction("SetP1Points");

4.1.43 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);`.

4.2 Triangulations

The next objects are finite triangulations of the sphere. They are represented by lists of points, edges and faces, and all adjacency relations between them. Each point, edge and face has a `P1Point` (4.1.1) as position, and furthermore edges come with a parametrization $\gamma([0,1])$ for a Möbius transformation γ. DeclareCategory("IsSphereTriangulation", IsObject); BindGlobal("TRIANGULATION_FAMILY", NewFamily("SphereTriangulations", IsSphereTriangulation)); BindGlobal("TYPE_TRIANGULATION", NewType(TRIANGULATION_FAMILY, IsSphereTriangulation)); DeclareRepresentation("IsTriangulationObjectRep", IsComponentObjectRep and IsAttributeStoringRep,[]); DeclareCategory("IsTriangulationObject",IsTriangulationObjectRep); DeclareCategory("IsTriangulationVertex",IsTriangulationObject); DeclareCategory("IsTriangulationEdge",IsTriangulationObject); DeclareCategory("IsTriangulationFace",IsTriangulationObject); BindGlobal("TRIANGULATIONOBJECT_FAMILY", NewFamily("TriangulationFamily",IsTriangulationObject,CanEasilySortElements,CanEasilySortElements)); BindGlobal("TYPE_VERTEX", NewType(TRIANGULATIONOBJECT_FAMILY,IsTriangulationVertex)); BindGlobal("TYPE_EDGE", NewType(TRIANGULATIONOBJECT_FAMILY,IsTriangulationEdge));

BindGlobal("TYPE_FACE", NewType(TRIANGULATIONOBJECT_FAMILY,IsTriangulationFace)); DeclareAttribute("Neighbour", IsTriangulationVertex); DeclareOperation("Neighbours", [IsTriangulationVertex]); DeclareOperation("Neighbours", [IsTriangulationVertex,IsTriangulationEdge]); DeclareOperation("Valency", [IsTriangulationVertex]); DeclareAttribute("Pos", IsTriangulationVertex); DeclareOperation("ClosestFace", [IsTriangulationObject]); DeclareOperation("ClosestFaces", [IsTriangulationObject]); DeclareOperation("ClosestVertex", [IsTriangulationObject]); DeclareOperation("ClosestVertices", [IsTriangulationObject]); DeclareProperty("IsFake", IsTriangulationVertex); DeclareAttribute("Left", IsTriangulationEdge); DeclareAttribute("Right", IsTriangulationEdge); DeclareAttribute("To", IsTriangulationEdge); DeclareAttribute("From", IsTriangulationEdge); DeclareAttribute("Next", IsTriangulationEdge); DeclareAttribute("Prevopp", IsTriangulationEdge); DeclareAttribute("Opposite", IsTriangulationEdge); DeclareAttribute("Pos", IsTriangulationEdge); DeclareAttribute("FromPos", IsTriangulationEdge); DeclareAttribute("ToPos", IsTriangulationEdge); DeclareAttribute("Length", IsTriangulationEdge); DeclareAttribute("Map", IsTriangulationEdge); DeclareAttribute("GroupElement", IsTriangulationEdge, "mutable"); DeclareAttribute("Neighbour", IsTriangulationFace); DeclareOperation("Neighbours", [IsTriangulationFace]); DeclareOperation("Neighbours", [IsTriangulationFace,IsTriangulationEdge]); DeclareAttribute("Pos", IsTriangulationFace); DeclareAttribute("Radius", IsTriangulationFace); DeclareAttribute("Centre", IsTriangulationFace); DeclareOperation("Valency", [IsTriangulationFace]); DeclareOperation("Draw", [IsSphereTriangulation]);

4.2.1 IsSphereTriangulation

▷ `IsSphereTriangulation` (filter)

The category of triangulated spheres (points in Moduli space).

This triangulation is a collection of vertices, edges and faces. These are new GAP objects. The attributes for vertices are:

`Neighbour`
: any edge starting at the vertex

`Neighbours`
: a list of neighbours, in counterclockwise order (an optional adugument lets one specify the starting edge)

`Valency`
: the number of neighbours

`Pos` The P1 point where the vertex is located

`ClosestVertex`
: The vertex itself

`ClosestVertices`
: A list containing the vertex itself

`ClosestFace`
: The face left of the first neighbour

`ClosestFaces`
: The faces that contain the vertex

`IsFake`
: whether the vertex was added for refinement

The edges come in opposite pairs, and are thought of as having a face on their left. Their possible attributes are:

`Left`, `Right`
: The adjacent faces

`To`, `From`
: The vertices that the edge goes to/from

`Next`
: The edge after on the left face (starting where the present edge ends)

`Prevopp`
: The opposite of the edge before on the left face (starting where the present edge starts)

`Opposite`
: The opposite edge (with reversed orientation)

`Pos` The position of the midpoint. `FromPos` and `ToPos` are shortcuts

`Length`

`Map` A P1 map sending $[0, 1]$ to the edge

`GroupElement`
: A group element describing "crossing through the edge from the left to the right"

`ClosestVertex`
: The from vertex

`ClosestVertices`
: The two endpoints

`ClosestFace`
: The left neighbour

`ClosestFaces`
: The two adjacent faces

The faces have the following possible attributes:

`Neighbour`
: Some edge with this face on its left

`Neighbours`
: The neighbours of the face, in counterclockwise order around the face (an optional argument lets one specify the starting edge)

`Pos` The position of the face's barycentre

`Radius`, `Centre`
The circumradius and circumcentre of the face (assumed to be a triangle)

`Valency`
The number of neighbouring edges

`ClosestVertex`
The from of the first neighbour

`ClosestVertices`
The vertices that the face contains

`ClosestFace`
The face itself

`ClosestFaces`
A list containing the face itself

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.

DeclareOperation("EdgePath", [IsSphereTriangulation,IsTriangulationFace,IsTriangulationFace]);

4.2.2 EdgePath

▷ `EdgePath(`*`t, f0, f1`*`)` (operation)

Returns: A sequence of edges taking *`f0`* to *`f1`*.

DeclareOperation("DelaunayTriangulation", [IsList]); DeclareOperation("DelaunayTriangulation", [IsList, IsFloat]);

4.2.3 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 satisfies the *Delaunay* condition that no point lies in the circumcircle of any face.

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 the `IsFake` property set to `true`.

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>
```

DeclareOperation("AddToTriangulation", [IsSphereTriangulation,IsP1Point]); DeclareOperation("AddToTriangulation", [IsSphereTriangulation,IsP1Point,IsBool]); DeclareOperation("AddToTriangulation", [IsSphereTriangulation,IsTriangulationFace,IsP1Point]); DeclareOperation("AddToTriangulation", [IsSphereTriangulation,IsTriangulationFace,IsP1Point,IsBool]);

4.2.4 AddToTriangulation

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

This command adds the P1 point *point* to the triangulation *t*. If a face *seed* is provided, it will speed up the search for the triangle in which the point is to be added. The other optional boolean argument *delaunay* specifies whether the Delaunay condition is to be fulfilled (by flipping diagonals of some quadrilaterals made of two neighbouring triangles) after the addition.

DeclareOperation("RemoveFromTriangulation", [IsSphereTriangulation,IsTriangulationVertex]);

4.2.5 RemoveFromTriangulation

▷ `RemoveFromTriangulation(`*`t, vertex`*`)` (operation)

This command removes the vertex *vertex* from the triangulation *t*.

DeclareOperation("LocateFaceInTriangulation", [IsSphereTriangulation,IsP1Point]); DeclareOperation("LocateFaceInTriangulation", [IsSphereTriangulation,IsObject,IsP1Point]); DeclareOperation("LocateInTriangulation", [IsSphereTriangulation,IsP1Point]); DeclareOperation("LocateInTriangulation", [IsSphereTriangulation,IsObject,IsP1Point]);

4.2.6 LocateFaceInTriangulation

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

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

This command locates the face in *t* that contains *point*; in the second form, 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>
```

DeclareOperation("WiggledTriangulation", [IsSphereTriangulation,IsObject]);

4.2.7 WiggledTriangulation (wiggle moebius)

▷ `WiggledTriangulation(t, moebiusmap)` (operation)
▷ `WiggledTriangulation(t, newpoints)` (operation)

Returns: A new triangulation, with moved vertices.

This command creates a new triangulation, in which only the P1 coordinates are changed. If the second argument `moebiusmap` is a Möbius transformation, then it is applied to the vertices and barycentres of faces and edges. If the second argument `newpoints` is a list of P1 points, then they are taken as new coordinates of the vertices.

DeclareGlobalFunction("EquidistributedP1Points");

4.2.8 EquidistributedP1Points

▷ `EquidistributedP1Points(N)` (function)

Returns: A list of `N` P1 points that are reasonably well spaced.

4.3 Marked spheres

Marked spheres should be thought of as punctured complex spheres, with a explicit identification of a sphere group with their fundamental group. They model points in Teichmüller space.

Marked spheres are given by a triangulation and a sphere group whose generators are in bijection with the triangulation's vertices. Internally, the marked sphere keeps track of a group element at each edge, stored as `GroupElement(e)`. This is the group element by which one should multiply as the edge is crossed transversally. In particular, the product of the edges going out of vertex v is conjugate to generator number v.

Given a marked sphere and a rational map (given by coefficients), a procedure computes the "lifted marked sphere" and a wreath recursion between their respective groups, i.e. a homomorphism $G_0 \to G_1 \wr Sym(d)$. The procedure will probably fail unless the critical values are close to the feet. There, the algorithm is already subtle: edges of the dual graph are represented by arcs of great circles.

One first computes the full preimage of the feet, and a triangulation spanning them on a sphere "above" the original marked sphere. Then, for each edge of the "down" dual graph, its preimage on the "up" sphere is an algebraic curve. One computes its intersections with all edges of the "up" dual graph, by finding zeroes of real polynomials, to determine from which triangle to which one the lifted edges go.

Because of rounding errors, one has to be careful as to when a real polynomial is supposed to have a zero, or when a point is supposed to be in a triangle. E.g., if T is a triangle and its sides are given by arcs of great circles, represented as $\gamma_i([0,1])$ for Möbius maps $\gamma_i, i = 1,2,3$, then the test $\Im(\gamma_i(z)) > 0 \forall i$ determines whether z is in the triangle. This test is really coded as $\Im(\gamma_i(z))/(1+|\gamma_i(z)|^2) > -@IMG.p1eps$ to take care of rounding errors. DeclareCategory("IsMarkedSphere", IsObject); BindGlobal("MARKEDSPHERES_FAMILY", NewFamily("MarkedSpheres", IsMarkedSphere)); BindGlobal("TYPE_MARKEDSPHERE", NewType(MARKEDSPHERES_FAMILY, IsMarkedSphere)); DeclareAttribute("MarkedSphere", IsSphereMachine); DeclareAttribute("MarkedSphere", IsP1Map); undocumented for now DeclareAttribute("VerticesOfMarkedSphere", IsMarkedSphere); DeclareAttribute("SpanningTreeBoundary", IsMarkedSphere);

4.3.1 IsMarkedSphere

▷ `IsMarkedSphere` (filter)

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

DeclareOperation("NewMarkedSphere", [IsP1PointCollection,IsSphereGroup]); DeclareOperation("NewMarkedSphere", [IsP1PointCollection]);

4.3.2 NewMarkedSphere

▷ `NewMarkedSphere(`*`points[, group]`*`)` (operation)

Returns: A new marked sphere on points *`points`*.

This function creates a new marked sphere, based on the Delaunay triangulation on *`points`*. If a sphere group *`group`* is specified, it is used to mark the sphere; otherwise a new sphere group is created.

DeclareOperation("Draw", [IsMarkedSphere]);

4.3.3 Draw (spider)

▷ `Draw(`*`s`*`)` (operation)

This command plots the marked sphere *`s`* in a separate 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.

DeclareOperation("WiggledMarkedSphere", [IsMarkedSphere,IsObject]);

4.3.4 WiggledMarkedSphere

▷ `WiggledMarkedSphere(`*`sphere, m`*`)` (operation)

Returns: A new marked sphere.

This operation moves the vertices of the marked sphere *`sphere`*, preserving its marking. The argument *`m`*, which specifies a movement of the vertices, is either a Möbius transformation (to be applied to all vertices) or a list of new positions for them.

DeclareOperation("SphereMachineOfBranchedCovering", [IsMarkedSphere,IsMarkedSphere,IsP1Map,IsBool]); DeclareOperation("SphereMachineOfBranchedCovering", [IsMarkedSphere,IsMarkedSphere,IsP1Map]); DeclareOperation("SphereMachineAndSphereOfBranchedCovering", [IsMarkedSphere,IsP1Map,IsBool]); DeclareOperation("SphereMachineAndSphereOfBranchedCovering", [IsMarkedSphere,IsP1Map]);

4.3.5 SphereMachineOfBranchedCovering

▷ `SphereMachineOfBranchedCovering(`*`down, up, map[, poly]`*`)` (operation)

▷ `SphereMachineAndSphereOfBranchedCovering(`*`down, map[, poly]`*`)` (operation)

Returns: A sphere machine or [machine,marked sphere].

The first function computes, out of a marked sphere *down* in the range of the P1 map *map* and a marked sphere *up* in its domain, the sphere machine representing the monodromy action of the map. Its input stateset is the model group of *down*, while its output stateset is the model group of *up*.

The second function first computes a marked sphere on the full preimage by *map* of the vertices of *down*, then computes the sphere machine, and finally returns a list containing the machine and the sphere at the source of *map*.

The optional parameter *poly* specifies that the map *map* is to be treated as a polynomial, and that the machine is to be normalized so that its last generator is an adding machine in standard form.

DeclareOperation("MonodromyOfP1Map", [IsMarkedSphere,IsP1Map]); DeclareOperation("MonodromyOfP1Map", [IsP1PointCollection,IsP1Map]); DeclareOperation("MonodromyOfP1Map", [IsP1Map]);

4.3.6 MonodromyOfP1Map

▷ `MonodromyOfP1Map(`*[marking,]map*`)` (operation)

Returns: The monodromy action of *map*.

This function computes the monodromy of the P1 map *map*; this is simply the activity of the sphere machine associated with the map.

The optional first argument *marking* may be a marked sphere, in which case the monodromy is returned as a homomorphism from the marked sphere's marking. It may also be a list of P1 points, in which case the monodromy is returned as a list of permutations, one per point. If the first argument is missing, it is assumed to be the list of critical values of *map*.

DeclareAttribute("SphereMachine", IsP1Map);

4.3.7 SphereMachine (rational function)

▷ `SphereMachine(`*f*`)` (operation)

Returns: A sphere 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 sphere machine `m`, which is returned.

This machine has a preset attribute `MarkedSphere(m)`.

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

Example

```
gap> SphereMachine(P1z^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
```

DeclareOperation("DistanceMarkedSpheres", [IsMarkedSphere, IsMarkedSphere]); DeclareOperation("DistanceMarkedSpheres", [IsMarkedSphere, IsMarkedSphere, IsBool]);

4.3.8 DistanceMarkedSpheres

▷ `DistanceMarkedSpheres(sphere1, sphere2[, fast])` (operation)

Returns: The approximate distance between the marked spheres.

This function approximates coarsely the Teichmüller distance between marked spheres with same model group. If the vertices of `sphere1` can be wiggled to the vertices of `sphere2` in such a manner that the markings coincide, then the distance is the sum of the movements of the vertices. Otherwise, it is $1+$ the sum of the lengths of the images of a sphere group automorphism that carries the marking of `sphere1` to that of `sphere2`.

4.4 The Hurwitz problem

Given a marked sphere and a permutation representation of its group, a procedure computes a rational map with that monodromy. If the representation has degree 2, or is bicritical, or a few other cases easily coded by hand, then the rational map is computed algebraically.

Otherwise, the "hard" part of the algorithm comes into play. A fresh marked sphere is constructed with only the feet with non-trivial permutation. The triangulation is then combinatorially lifted, using the permutation representation. In particular, the feet are now labeled by cycles of the permutations. It is a purely combinatorial triangulation, at this point; but one remembers that its edges have a length inherited from the sphere metric. The triangulation is then refined by repeatedly adding circumcentres of triangles, till every edge (say from v to w) has length $\leq @IMG.hurwitzmesh^{M}aximum(cyclelengthatv, cyclelengthatw)$.

Now an external C program, "layout", is called. It seeks a discrete conformal map, given by a function $u : \{feet\} \to \mathbb{R}$, such that if edge e (from v to w) has its length scaled by $u(v) \cdot u(w)$ then the sum-of-angles=2π condition holds at each vertex (with special treatment for a vertex at infinity; I skip details). Then the triangulation can be laid out on the plane, and projected back stereographically to the sphere. In this manner, we got good approximations of where the feet should be.

Now we run Newton's method on the feet positions, using as variables the lifted-feet positions. This is the external program "hsolve". It is assumed that the down-feet contain 0 and ∞, so that the rational map we are looking for is of the form $f(z) = C \cdot \prod (z - c_i)^{d_i}$ for known integers d_i. The c_i are the lifted-feet above 0 and ∞, and the equations in Newton's method say that the log-derivative of f must vanish at appropriate lifted-feet, and that f must map these lifted-feet to the down-feet. DeclareOperation("BranchedCoveringByMonodromy", [IsMarkedSphere,IsGroupHomomorphism]); DeclareOperation("BranchedCoveringByMonodromy", [IsMarkedSphere,IsGroupHomomorphism,IsRecord]);

4.4.1 BranchedCoveringByMonodromy

▷ `BranchedCoveringByMonodromy(sphere, monodromy[, last])` (operation)

Returns: A record describing a Hurwitz map.

If `sphere` is a marked sphere, 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 `sphere` 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 `sphere` it maps to.

If a third argument is supplied, it should be a record similar to the return value of the command. If the result is close enough to the supplied record, it will be used to speed up the calculation.

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 marked sphere
gap> d := 20;;
gap> z := List([0..2*d-3], i->P1Point(Exp(i*PMCOMPLEX.constants.2IPI/(2*d-2))));;
gap> g := SphereGroup(2*d-2);;
gap> sphere := NewMarkedSphere(z,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> BranchedCoveringByMonodromy(sphere,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+78.41574480904
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+137069.100158382256i_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+36053.8959611372481_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 ]> ) ] )
```

DeclareOperation("DessinByPermutations", [IsPerm,IsPerm]); DeclareOperation("DessinByPermutations", [IsPerm,IsPerm,IsPerm]);

4.4.2 DessinByPermutations

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

Returns: A rational map (see `BranchedCoveringByMonodromy` (4.4.1)) with monodromies `s,t`.

This command computes the Hurwitz map associated with the spanning tree $[0,1]\cup[1,\infty]$; 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> ) ] )
```

gap> # IV.5.2 in Granboulan's PhD, the automorphism group of the Mathieu group M_22 gap> autm22 := Group((1,2,3,4,5,6,7,8,9,10,11)(12,13,14,15,16,17,18,19,20,21,22), (1,9,3,2)(4,8,17,21)(5,20,19,6)(12,22,16,13)(7,18)(10,11)(14,15), (3,8)(4,20)(6,18)(7,17)(9,11)(13,15)(16,21));; gap> IsomorphismGroups(DerivedSubgroup(autm22),MathieuGroup(22))<>fail; true gap> DessinByPermutations(autm22.1,autm22.2,autm22.3); ... gap> # IV.5.3 in Granboulan's PhD, the "extraterrestrial" dessin with group M_24 gap> m24_ET := Group((1,2,3)(4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24), (1,7,5)(2,4,23)(3,22,8)(9,21,19)(10,18,12)(13,17,15), (1,4)(2,22)(3,7)(5,6)(8,21)(9,18)(10,11)(12,17)(13,14)(15,16)(19
gap> IsomorphismGroups(m24_ET,MathieuGroup(24))<>fail; true gap> @IMG.hurwitzmesh := 0.4;; # need finer precision gap> DessinByPermutations(m24_ET.1,m24_ET.2,m24_ET.3); ...

4.5 The spider algorithm

IMG implements an algorith, extending 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, `IMG@`. 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.p1eps := 10^-8`
: 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.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^-8`
: The minimal acceptable precision on the coefficients of the rational function.

For Thurston's algorithm, one starts by an arbitrary marked sphere (I chose its feet on the real axis, at equal small angles); computes a rational map using `BranchedCoveringByMonodromy` (4.4.1), computes its sphere machine using `SphereMachineOfBranchedCovering` (4.3.5), and matches the original sphere machine with the lifted one. This tells us which feet of the lifted marked sphere we should keep.

One then computes a normalized position for the sub-marked sphere: its last foot is put at ∞, another one (chosen cleverly) is on the positive real axis, and the center of mass of all feet (in $\mathbb{R}^3$) is $(0,0,0)$. This is much more stable, numerically, than putting three points at $0,1,\infty$. One has a Möbius transformation that puts the sub-marked sphere in normalized position, and again using

SphereMachineOfBranchedCovering (4.3.5) one computes its machine. One then composes the machines, and compares the product again to the original machine to determine the marking of the edges of the new marked sphere by group elements.

Then, one searches for all quadruples with large cross-ratio, and computes group-theoretically the curve separating the post-critical set in two parts that are well separated. One saturates the resulting curve into an invariant multicurve (aborting if there is an intersection between lifts), computes the Thurston matrix, and finds out (again algebraically) if there is an eigenvalue ≥ 1. There is no parameter in this part of the code, all quadruples are examined; this is a weakness of the current implementation, sometimes most of the computational time is spent on searching for obstructions when it's "clear" there are none.

The distance between two marked spheres (marked by the same group) is computed as follows: if their feet are close in the sense that the sum of the spherical distances between them is less that @IMG.fast, then wiggle one of the spiders to make its feet match that of the other; and check that the identity map gives, by SphereMachineOfBranchedCovering (4.3.5), the identity machine. In that case, the sum of the feet distances is the distance between the spheres. Otherwise, add to it some formula involving the entries in the biset, which gives large integer distances.

The Thurston algorithm stops when an obstruction is found or when the marked sphere moved less than @IMG.ratprec. Inside the main iteration of the algorithm, if the spider moved less that @IMG.fast, then don't compute the branched covering by monodromy, and don't compute the bisets; but just adjust the branched covering and the vertex positions of the spheres, guessing which ones should be kept. Check the guess: if it is incorrect, go back to the usual slow method. DeclareGlobalFunction("NormalizedP1Map"); DeclareProperty("IsBicritical", IsObject);

4.5.1 NormalizedP1Map

▷ NormalizedP1Map(*f, M, param*) (function)

Returns: [A canonical conjugate of *f*,the conjugator].

The last argument *param* is either IsPolynomial, IsBicritical or a positive integer.

In the first case, the map *f* is assumed conjugate to a polynomial. It is conjugated by a Möbius transformation that makes it a *centered* polynomial, namely a polynomial of the form $z^d + a_{d-2}z^{d-2} + \ldots + a_0$.

In the second case, the map *f* is assumed to have only two critical values; it is normalized as $(az^d + b)/(cz^d + e)$.

In the third case, the map *f* is assumed to have degree 2; it is normalized in the form $1 + a/z + b/z^2$, such that 0 is on a cycle of length *param*.

DeclareOperation("ThurstonAlgorithm", [IsSphereMachine]);

4.5.2 ThurstonAlgorithm

▷ ThurstonAlgorithm(*m*) (operation)

Returns: rec(map := f, machine := M, markedsphere := s).

This command runs Thurston's algorithm on the sphere machine *m*. It either returns a record with the P1 map f to which the algorithm converged, as well as the marked sphere with f's post-critical set and a simplified machine equivalent to *m*; or a record returned by ThurstonObstruction (4.5.5).

DeclareOperation("P1MapBySphereMachine", [IsSphereMachine]);

4.5.3 P1MapBySphereMachine

▷ P1MapBySphereMachine(*m*) (operation)

Returns: Either a map or an obstruction.

This command returs either the map computed by ThurstonAlgorithm (4.5.2) or a Thurston obstruction.

It runs a modification of Hubbard and Schleicher's "spider algorithm" [HS94] on the sphere machine *m*.

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

P1MapBySphereMachine(m:param:=IsPolynomial)
returns $f = z^d + A_{d-2}z^{d-2} + \cdots + A_0$;

P1MapBySphereMachine(m:param:=IsBicritical)
returns $f = ((pz+q)/(rz+s)^d$, with 1postcritical;

P1MapBySphereMachine(m:param:=n)
returns $f = 1 + a/z + b/z^2$ or $f = a/(z^2+2z)$ if n=2.

Example

```
gap> m := PolynomialSphereMachine(2,[1/3],[]);
<FR machine with alphabet [ 1, 2 ] on Group( [ f1, f2, f3 ] )/[ f3*f2*f1 ]>
gap> P1MapBySphereMachine(m);
0.866025*z^2+(-1)*z+(-0.288675)
```

DeclareOperation("ThurstonMatrix", [IsSphereMachine,IsMulticurve]);

4.5.4 ThurstonMatrix

▷ ThurstonMatrix(*m, multicurve*) (operation)

Returns: The transition matrix of the multicurve.

This command computes the iterated preimages of the multicurve *multicurve* till it obtains a backwards-invariant multicurve or some preimages intersect. In the latter case, fail returned, while in the former case the Thurston matrix of the multicurve is returned.

Example

```
gap> r := PolynomialSphereMachine(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> ThurstonMatrix(last.machine,[ConjugacyClass(F.2*F.3)]);
[ [ 1 ] ]
```

DeclareOperation("ThurstonObstruction", [IsSphereMachine,IsMarkedSphere]);

4.5.5 ThurstonObstruction

▷ ThurstonObstruction(*m, sphere*) (operation)

Returns: rec(multicurve := mc, matrix := m) or fail.

This command tries to find a Thurston obstruction (multicurve such that its Thurston matrix has spectral radius at least 1); it either returns `fail` if the search was inconclusive, or a record describing the obstruction.

The obstruction is searched for by considering small subtrees of the minimal spanning tree of *`sphere`*, computing loops surrounding these subtrees, and saturating them into a multicurve by taking their iterated preimages, see `ThurstonMatrix` (4.5.4).

Chapter 5

Examples

IMG predefines a large collection of sphere machines, as well as the generic constructions of polynomials.

5.1 Examples of groups

DeclareGlobalFunction("PoirierExamples");

5.1.1 PoirierExamples

▷ `PoirierExamples(...)` (function)

The examples from Poirier's paper [Poi]. See details under `PolynomialSphereMachine` (3.3.1); in particular, `PoirierExamples(1)` is the Douady rabbit map.

DeclareGlobalFunction("DBRationalIMGGroup");

5.1.2 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)
```

DeclareGlobalFunction("PostCriticalMachine");

5.1.3 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` (**fr: 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 ]
```

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 Helpers

DeclareGlobalFunction("Mandel");

6.2.1 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`.

DeclareOperation("NonContractingSubmatrix", [IsMatrix]);

6.2.2 NonContractingSubmatrix

▷ `NonContractingSubmatrix(mat)` (operation)

Returns: `fail` or a list of indices `l` such that `mat{l}{l}` is irreducible and non-contracting

This function computes a minimal submatrix whose spectral radius is ≥ 1. If none exists, it returns `fail`.

Example

```
gap> NonContractingSubmatrix([[2]]);
[ 1 ]
gap> NonContractingSubmatrix([[1/2]]);
fail
gap> NonContractingSubmatrix([[0,1],[1,0]]);
[ 1, 2 ]
gap> NonContractingSubmatrix([[0,1],[0,1]]);
[ 2 ]
```

6.3 User settings

6.3.1 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

[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. 12

[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

[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. 12

[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. 12

[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. 42, 44

[Nek05] V. Nekrashevych. *Self-similar groups*, volume 117 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 2005. 5, 17

[Poi] A. Poirier. On postcritically finite polynomials, part 1: critical portraits. Stony Brook IMS 1993/5. 12, 46

[Tan02] D. T. Tan. Quadratische morphismen. Diplomarbeit at ETHZ, under the supervision of R. Pink, 2002. 46

Index