VKCURVE
Version 1.2

D. Bessis, J. Michel

March 2009

Contents

Chapter 1

The VKCURVE package

The main function of the VKCURVE package computes the fundamental group of the complement of a complex algebraic curve in $\mathbb{C}^2$, using an implementation of the Van Kampen method (see for example [Che73] for a clear and modernized account of this method).

```
gap> FundamentalGroup(x^2-y^3);
# I there are 2 generators and 1 relator of total length 6
1: bab=aba

gap> FundamentalGroup((x+y)*(x-y)*(x+2*y));
# I there are 3 generators and 2 relators of total length 12
1: cab=abc
2: bca=abc
```

The input is a polynomial in the two variables `x` and `y`, with rational coefficients. Though approximate calculations are used at various places, they are controlled and the final result is exact.

The output is a record which contains lots of information about the computation, including a presentation of the computed fundamental group, which is what is displayed when printing the record.

Our motivation for writing this package was to find explicit presentations for generalized braid groups attached to certain complex reflection groups. Though presentations were known for almost all cases, six exceptional cases were missing (in the notations of Shephard and Todd, these cases are G_{24}, G_{27}, G_{29}, G_{31}, G_{33} and G_{34}). Since the a priori existence of nice presentations for braid groups was proved in [Bes01], it was upsetting not to know them explicitly. In the absence of any good grip on the geometry of these six examples, brute force was a way to get an answer. Using VKCURVE, we have obtained presentations for all of them.

This package was developed thanks to computer resources of the Institut de Mathématiques de Jussieu in Paris. We thank the computer support team, especially Joël Marchand, for the stability and the efficiency of the working environment.

We have tried to design this package with the novice GAP3 user in mind. The only steps required to use it are

- Run GAP33 (the package is not compatible with GAP34).
- Make sure the packages CHEVIE and VKCURVE are loaded (beware that we require the development version of CHEVIE, `http://www.math.jussieu.fr/~jmichel/chevie.html` and not the one in the GAP3.3.3.4 distribution)
- Use the function `FundamentalGroup`, as demonstrated in the above examples.

If you are not interested in the details of the algorithm, and if `FundamentalGroup` gives you satisfactory answers in a reasonable time, then you do not need to read this manual any further.

We use our own package for multivariate polynomials which is more effective, for our purposes, than the default in GAP33 (see `Mvp`). When VKCURVE is loaded, the variables `x` and `y` are pre-defined as `Mvp`s; one can also use GAP3 polynomials (which will be converted to `Mvp`s).

The implementation uses `Decimal` numbers, `Complex` numbers and braids as implemented in the (development version of the) package CHEVIE, so VKCURVE is dependent on this package.

To implement the algorithms, we needed to write auxiliary facilities, for instance find zeros of complex polynomials, or work with piecewise linear braids, which may be useful on their own. These various facilities are documented in this manual.

Before discussing our actual implementation, let us give an informal summary of the mathematical background. Our strategy is adapted from the one originally described in the 1930's by Van Kampen. Let C be an affine algebraic curve, given as the set of zeros in $\mathbb{C}^2$ of a non-zero reduced polynomial $P(x,y)$. The problem is to compute a presentation of the fundamental group of $\mathbb{C}^2 - C$. Consider P as a polynomial in x, with coefficients in the ring of polynomials in y

$$P = \alpha_0(y)x^n + \alpha_1(y)x^{n-1} + \ldots + \alpha_{n-1}(y)x + \alpha_n(y),$$

where the α_i are polynomials in y. Let $\Delta(y)$ be the discriminant of P or, in other words, the resultant of P and $\frac{\partial P}{\partial x}$. Since P is reduced, Δ is non-zero. For a generic value of y, the polynomial in x given by $P(x,y)$ has n distinct roots. When $y = y_j$, with j in $1,\ldots,d$, we are in exactly one of the following situations: either $P(x,y_j) = 0$ (we then say that y_j is bad), or $P(x,y_j)$ has a number of roots in x strictly smaller than n. Fix y_0 in $\mathbb{C} - \{y_1,\ldots,y_d\}$. Consider the projection $p : \mathbb{C}^2 \to \mathbb{C}, (x,y) \mapsto y$. It restricts to a locally trivial fibration with base space $B = \mathbb{C} - \{y_1,\ldots,y_d\}$ and fibers homeomorphic to the complex plane with n points removed. We denote by E the total space $p^{-1}(B)$ and by F the fiber over y_0. The fundamental group of F is isomorphic to the free group on n generators. Let $\gamma_1,\ldots,\gamma_d$ be loops in the pointed space (B,y_0) representing a generating system for $\pi_1(B,y_0)$. By trivializing the pullback of p along γ_i, one gets a (well-defined up to isotopy) homeomorphism of F, and a (well-defined) automorphism ϕ_i of the fundamental group of F, identified with the free group F_n by the choice of a generating system $f_1,\ldots,f_n$. An effective way of computing ϕ_i is by following the solutions in x of $P(x,y) = 0$, when y moves along ϕ_i. This defines a loop in the space of configuration of n points in a plane, hence an element b_i of the braid group B_n (via an identification of B_n with the fundamental group of this configuration space). Let ϕ be the Hurwitz action of B_n on F_n. All choices can be made in such a way that $\phi_i = \phi(b_i)$. The theorem of Van Kampen asserts that, if there are no bad roots of the discriminant, a presentation for the fundamental group of $\mathbb{C}^2 - C$ is

$$< f_1, \ldots, f_n \mid \forall i, j, \phi_i(f_j) = f_j >$$

A variant of the above presentation (see `VKQuotient`) can be used to deal with bad roots of the discriminant.

This algorithm is implemented in the following way.

- As input, we have a polynomial P. The polynomial is reduced if it was not.
- The discriminant Δ of P with respect to x is computed. It is a polynomial in y.
- The roots of Δ are approximated, via the following procedure. First, we reduce Δ and get Δ_{red} (generating the radical of the ideal generated by Δ). The roots $\{y_1, \ldots, y_d\}$ of Δ_{red} are separated by `SeparateRoots` (which implements Newton's method).
- Loops around these roots are computed by `LoopsAroundPunctures`. This function first computes some sort of honeycomb, consisting of a set S of affine segments, isolating the y_i. Since it makes the computation of the monodromy more effective, each inner segment is a fragment of the mediatrix of two roots of Δ. Then a vertex of one the segments is chosen as a basepoint, and the function returns a list of lists of oriented segments in S: each list of segment encodes a piecewise linear loop γ_i circling one of y_i.
- For each segment in S, we compute the monodromy braid obtained by following the solutions in x of $P(x, y) = 0$ when y moves along the segment. By default, this monodromy braid is computed by `FollowMonodromy`. The strategy is to compute a piecewise-linear braid approximating the actual monodromy geometric braid. The approximations are controlled. The piecewise-linear braid is constructed step-by-step, by computations of linear pieces. As soon as new piece is constructed, it is converted into an element of B_n and multiplied; therefore, though the braid may consist of a huge number of pieces, the function `FollowMonodromy` works with constant memory. The packages also contains a variant function `ApproxFollowMonodromy`, which runs faster, but without guarantee on the result (see below).
- The monodromy braids b_i corresponding to the loops γ_i are obtained by multiplying the corresponding monodromy braids of segments. The action of these elements of B_n on the free group F_n is computed by `BnActsOnFn` and the resulting presentation of the fundamental group is computed by `VKQuotient`. It happens for some large problems that the whole fundamental group process fails here, because the braids b_i obtained are too long and the computation of the action on F_n requires thus too much memory. We have been able to solve such problems when they occur by calling on the b_i at this stage our function `ShrinkBraidGeneratingSet` which finds smaller generators for the subgroup of B_n generated by the b_i (see the description in the third chapter). This function is called automatically at this stage if `VKCURVE.shrinkBraid` is set to `true` (the default for this variable is `false`).
- Finally, the presentation is simplified by `ShrinkPresentation`. This function is a heuristic adaptation and refinement of the basic GAP3 functions for simplifying presentations. It is non-deterministic.

From the algorithmic point of view, memory should not be an issue, but the procedure may take a lot of CPU time (the critical part being the computation of the monodromy braids by `FollowMonodromy`). For instance, an empirical study with the curves $x^2 - y^n$ suggests that the needed time grows exponentially with n. Two solutions are offered to deal with curves for which the computation time becomes unreasonable.

A global variable `VKCURVE.monodromyApprox` controls which monodromy function is used. The default value of this variable is `false`, which means that `FollowMonodromy` will be used. If the variable is set by the user to `true` then the function `ApproxFollowMonodromy` will be used instead. This function runs faster than `FollowMonodromy`, but the approximations are no longer controlled. Therefore presentations obtained while `VKCURVE.monodromyApprox` is set to `true` are not certified. However, though it is likely that there exists examples for which `ApproxFollowMonodromy` actually returns incorrect answers, we still have not seen one.

The second way of dealing with difficult examples is to parallelize the computation. Since the computations of the monodromy braids for each segment are independent, they can be performed simultaneously on different computers. The functions `PrepareFundamentalGroup`, `Segments` and `FinishFundamentalGroup` provide basic support for parallel computing.

1.1 FundamentalGroup

`FundamentalGroup(`*curve* `[,` *printlevel*`])`

curve should be an `Mvp` in x and y, or a `GAP3` polynomial in two variables (which means a polynomial in a variable which is assumed to be `y` over the polynomial ring $\mathbb{Q}[x]$) representing an equation $f(x, y)$ for a curve in $\mathbb{C}^2$. The coefficients should be rationals, gaussian rationals or `Complex` rationals. The result is a record with a certain number of fields which record steps in the computation described in this introduction:

```
gap> r:=FundamentalGroup(x^2-y^3);
# I there are 2 generators and 1 relator of total length 6
1: bab=aba

gap> RecFields(r);
[ "curve", "discy", "roots", "dispersal", "points", "segments", "loops",
  "zeros", "B", "monodromy", "basepoint", "dispersal", "braids",
  "presentation","operations" ]
gap> r.curve;
x^2-y^3
gap> r.discy;
X(Rationals)
gap> r.roots;
[ 0 ]
gap> r.points;
[ -I, -1, 1, I ]
gap> r.segments;
[ [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ]
gap> r.loops;
```

```
[ [ 4, -3, -1, 2 ] ]
gap> r.zeros;
[ [ 707106781187/1000000000000+707106781187/1000000000000I,
    -707106781187/1000000000000-707106781187/1000000000000I ],
  [ I, -I ], [ 1, -1 ],
  [ -707106781187/1000000000000+707106781187/1000000000000I,
  707106781187/1000000000000-707106781187/1000000000000I ] ]
gap> r.monodromy;
[ (w0)^-1, w0, , w0 ]
gap> r.braids;
[ w0.w0.w0 ]
gap> DisplayPresentation(r.presentation);
1: bab=aba
```

Here `r.curve` records the entered equation, `r.discy` its discriminant with respect to x, `r.roots` the roots of this discriminant, `r.points`, `r.segments` and `r.loops` describes loops around these zeros as explained in the documentation of `LoopsAroundPunctures`; `r.zeros` records the zeros of $f(x, y_i)$ when y_i runs over the various `r.points`; `r.monodromy` records the monodromy along each of `r.segments`, and `r.braids` is the resulting monodromy along the loops. Finally `r.presentation` records the resulting presentation (which is what is printed by default when `r` is printed).

The second optional argument triggers the display of information on the progress of the computation. It is recommended to set the *printlevel* at 1 or 2 when the computation seems to take a long time without doing anything. *printlevel* set at 0 is the default and prints nothing; set at 1 it shows which segment is currently active, and set at 2 it traces the computation inside each segment.

```
gap> FundamentalGroup(x^2-y^3,1);
# There are 4 segments in 1 loops
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[1]:=B(-1);
# segment 1/4 Time=0sec
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[2]:=B(1);
# segment 2/4 Time=0sec
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[3]:=B();
# segment 3/4 Time=0sec
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[4]:=B(1);
# segment 4/4 Time=0sec
# Computing monodromy braids
# loop[1]=w0.w0.w0
# I there are 2 generators and 1 relator of total length 6
1: bab=aba
```

1.2 PrepareFundamentalGroup

`PrepareFundamentalGroup(`*curve*`, `*name*`)`

`VKCURVE.Segments(`*name*`[,`*range*`])`

`FinishFundamentalGroup(`*r*`)`

These functions provide a means of distributing a fundamental group computation over several machines. The basic strategy is to write to a file the startup-information necessary to compute the monodromy along a segment, in the form of a partially-filled version of the record returned by `FundamentalGroup`. Then the monodromy along each segment can be done in a separate process, writing again the result to files. These results are then gathered and processed by `FinishFundamentalGroup`. The whole process is illustrated in an example below. The extra argument *name* to `PrepareFundamentalGroup` is a prefix used to name intermediate files. One does first :

```
gap> PrepareFundamentalGroup(x^2-y^3,"a2");
    ----------------------------------
Data saved in a2.tmp
You can now compute segments 1 to 4
in different GAP sessions by doing in each of them:
    a2:=rec(name:="a2");
    VKCURVE.Segments(a2,[1..4]);
(or some other range depending on the session)
Then when all files a2.xx have been computed finish by
    a2:=rec(name:="a2");
    FinishFundamentalGroup(a2);
```

Then one can compute in separate sessions the monodromy along each segment. The second argument of `Segments` tells which segments to compute in the current session (the default is all). An example of such sessions may be:

```
gap> a2:=rec(name:="a2");
rec(
  name := "a2" )
gap> VKCURVE.Segments(a2,[2]);
# The following braid was computed by FollowMonodromy in 8 steps.
a2.monodromy[2]:=a2.B(1);
# segment 2/4 Time=0.1sec
gap> a2:=rec(name:="a2");
rec(
  name := "a2" )
gap> VKCURVE.Segments(a2,[1,3,4]);
# The following braid was computed by FollowMonodromy in 8 steps.
a2.monodromy[2]:=a2.B(1);
# segment 2/4 Time=0.1sec
```

When all segments have been computed the final session looks like

```
gap> a2:=rec(name:="a2");
rec(
  name := "a2" )
gap> FinishFundamentalGroup(a2);
1: bab=aba
```

Chapter 2

Multivariate polynomials and rational fractions

The functions described in this file were written to alleviate the deficiency of GAP3 3 in manipulating multi-variate polynomials. In GAP3 3 one can only define one-variable polynomials over a given ring; this allows multi-variate polynomials by taking this ring to be a polynomial ring; but, in addition to providing little flexibility in the choice of coefficients, this "full" representation makes for somewhat inefficient computation. The use of the `Mvp` (MultiVariate Polynomials) described here is faster than GAP3 polynomials as soon as there are two variables or more. What is implemented here is actually "Puiseux polynomials", i.e. linear combinations of monomials of the type $x_1^{a_1} \dots x_n^{a_n}$ where x_i are variables and a_i are exponents which can be arbitrary rational numbers. Some functions described below need their argument to involve only variables to integral powers; we will refer to such objects as "Laurent polynomials"; some functions require further that variables are raised only to positive powers: we refer then to "true polynomials". Rational fractions (`RatFrac`) have been added, thanks to work of Gwenaëlle Genet (the main difficulty there was to write an algorithm for the Gcd of multivariate polynomials, a non-trivial task). The coefficients of our polynomials can in principle be elements of any ring, but some algorithms like division or Gcd require the coefficients of their arguments to be invertible.

2.1 Mvp

`Mvp(` *string s* `[,` *coeffs v*`] )`

Defines an indeterminate with name s suitable to build multivariate polynomials.

```
gap> x:=Mvp("x");y:=Mvp("y");(x+y)^3;
x
y
3xy^2+3x^2y+x^3+y^3
```

If a second argument (a vector of coefficients v) is given, returns `Sum([1..Length(v)],i->Mvp(s)^(i-1)*v[i])`.

```
gap> Mvp("a",[1,2,0,4]);
1+2a+4a^3
```

`Mvp(` *polynomial x*`)`

Converts the GAP3 polynomial x to an `Mvp`. It is an error if `x.baseRing.indeterminate.name` is not bound; otherwise this is taken as the name of the `Mvp` variable.

```
gap> q:=Indeterminate(Rationals);
X(Rationals)
gap> Mvp(q^2+q);
Error, X(Rationals) should have .name bound in
Mvp( q ^ 2 + q ) called from
main loop
brk>
gap> q.name:="q";;
gap> Mvp(q^2+q);
q+q^2
```

`Mvp(` *FracRat x*`)`

Returns `false` if the argument rational fraction is not in fact a Laurent polynomial. Otherwise returns that polynomial.

```
gap> Mvp(x/y);
xy^-1
gap> Mvp(x/(y+1));
false
```

`Mvp(` *elm*, *coeff*`)`

Build efficiently an `Mvp` from the given list of coefficients and the list *elm* describing the corresponding monomials. A monomial is itself described by a record with a field `.elm` containing the list of involved variable names and a field `.coeff` containing the list of corresponding exponents.

```
gap> Mvp([rec(elm:=["y","x"],coeff:=[1,-1])],[1]);
x^-1y
```

`Mvp(` *scalar x*`)`

A scalar is anything which is not one of the previous types (like a cyclotomic, or a finite-field-element, etc). Returns the constant multivariate polynomial whose constant term is x.

```
gap> Degree(Mvp(1));
0
```

2.2 Operations for Mvp

The arithmetic operations `+`, `-`, `*`, `/` and `^` work for `Mvp`s. They also have `Print` and `String` methods. The operations `+`, `-`, `*` work for any inputs. `/` works only for Laurent polynomials, and may return a rational fraction (see below); if one is sure that the division is exact, one should call `MvpOps.ExactDiv` (see below).

```
gap> x:=Mvp("x");y:=Mvp("y");
x
y
gap> a:=x^(-1/2);
```

```
x^(-1/2)
gap> (a+y)^4;
x^-2+4x^(-3/2)y+6x^-1y^2+4x^(-1/2)y^3+y^4
gap> (x^2-y^2)/(x-y);
x+y
gap> (x-y^2)/(x-y);
(x-y^2)/(x-y)
gap> (x-y^2)/(x^(1/2)-y);
Error, x^(1/2)-y is not a polynomial with respect to x
 in
V.operations.Coefficients( V, v ) called from
Coefficients( q, var ) called from
MvpOps.ExactDiv( x, q ) called from
fun( arg[1][i] ) called from
List( p, function ( x ) ... end ) called from
...
brk>
```

Only monomials can be raised to a non-integral power; they can be raised to a fractional power of denominator `b` only if `GetRoot(x,b)` is defined where `x` is their leading coefficient. For an `Mvp` m, the function `GetRoot(m,n)` is equivalent to `m^(1/n)`. Raising a non-monomial Laurent polynomial to a negative power returns a rational fraction.

```
gap> (2*x)^(1/2);
ER(2)x^(1/2)
gap> (evalf(2)*x)^(1/2);
1.4142135624x^(1/2)
gap> GetRoot(evalf(2)*x,2);
1.4142135624x^(1/2)
```

The `Degree` of a monomial is the sum of the exponent of the variables. The `Degree` of an `Mvp` is the largest degree of a monomial.

```
gap> a;
x^(-1/2)
gap> Degree(a);
-1/2
gap> Degree(a+x);
1
gap> Degree(Mvp(0));
-1
```

The `Valuation` of an `Mvp` is the minimal degree of a monomial.

```
gap> a;
x^(-1/2)
gap> Valuation(a);
-1/2
gap> Valuation(a+x);
-1/2
gap> Valuation(Mvp(0));
-1
```

The `Format` routine formats `Mvp`s in such a way that they can be read back in by GAP3 or by some other systems, by giving an appropriate option as a second argument, or using the functions `FormatGAP`, `FormatMaple` or `FormatTeX`. The `String` method is equivalent to `Format`, and gives a compact display.

```
gap> p:=7*x^5*y^-1-2;
-2+7x^5y^-1
gap> Format(p);
"-2+7x^5y^-1"
gap> FormatGAP(p);
"-2+7*x^5*y^-1"
gap> FormatMaple(p);
"-2+7*x^5*y^(-1)"
gap> FormatTeX(p);
"-2+7x^5y^{-1}"
```

The `Value` method evaluates an `Mvp` by fixing simultaneously the value of several variables. The syntax is `Value(x, [` *string1* `,` *value1* `,` *string2* `,` *value2* `,` ... `])`.

```
gap> p;
-2+7x^5y^-1
gap> Value(p,["x",2]);
-2+224y^-1
gap> Value(p,["y",3]);
-2+7/3x^5
gap> Value(p,["x",2,"y",3]);
218/3
```

One should pay attention to the fact that the last value is not a rational number, but a constant `Mvp` (for consistency). See the function `ScalMvp` below for how to convert such constants to their base ring.

```
gap> Value(p,["x",y]);
-2+7y^4
gap> Value(p,["x",y,"y",x]);
-2+7x^-1y^5
```

Evaluating an `Mvp` which is a Puiseux polynomial may cause calls to `GetRoot`

```
gap> p:=x^(1/2)*y^(1/3);
x^(1/2)y^(1/3)
gap> Value(p,["x",y]);
y^(5/6)
gap>  Value(p,["x",2]);
ER(2)y^(1/3)
gap>  Value(p,["y",2]);
Error, : unable to compute 3-th root of 2
 in
GetRoot( values[i], d[i] ) called from
f.operations.Value( f, x ) called from
Value( p, [ "y", 2 ] ) called from
main loop
```

```
brk>
```

The function `Derivative(p,v)` returns the derivative of `p` with respect to the variable given by the string `v`; if `v` is not given, with respect to the first variable in alphabetical order.

```
gap>  p:=7*x^5*y^-1-2;
-2+7x^5y^-1
gap> Derivative(p,"x");
35x^4y^-1
gap> Derivative(p,"y");
-7x^5y^-2
gap> Derivative(p);
35x^4y^-1
gap>  p:=x^(1/2)*y^(1/3);
x^(1/2)y^(1/3)
gap>  Derivative(p,"x");
1/2x^(-1/2)y^(1/3)
gap>  Derivative(p,"y");
1/3x^(1/2)y^(-2/3)
gap>  Derivative(p,"z");
0
```

The function `Coefficients(`*p*`, `*var*`)` is defined only for `Mvp`s which are polynomials in the variable *var* . It returns as a list the list of coefficients of *p* with respect to *var*.

```
gap> p:=x+y^-1;
y^-1+x
gap> Coefficients(p,"x");
[ y^-1, 1 ]
gap> Coefficients(p,"y");
Error, y^-1+x is not a polynomial with respect to y
 in
V.operations.Coefficients( V, v ) called from
Coefficients( p, "y" ) called from
main loop
brk>
```

The same caveat is applicable to `Coefficients` as to `Value`: the result is always a list of `Mvp`s. To get a list of scalars for univariate polynomials represented as `Mvp`s, one should use `ScalMvp`.

Finally we mention the functions `ComplexConjugate` and `evalf` which are defined using for coefficients the `Complex` and `Decimal` numbers of the CHEVIE package.

```
gap> p:=E(3)*x+E(5);
E5+E3x
gap> evalf(p);
0.3090169944+0.9510565163I+(-0.5+0.8660254038I)x
gap> p:=E(3)*x+E(5);
E5+E3x
gap> ComplexConjugate(p);
E5^4+E3^2x
```

```
gap> evalf(p);
0.3090169944+0.9510565163I+(-0.5+0.8660254038I)x
gap> ComplexConjugate(last);
0.3090169944-0.9510565163I+(-0.5-0.8660254038I)x
```

2.3 IsMvp

`IsMvp(` *p* `)`

Returns `true` if p is an `Mvp` and false otherwise.

```
gap> IsMvp(1+Mvp("x"));
true
gap> IsMvp(1);
false
```

2.4 ScalMvp

`ScalMvp(` *p* `)`

If p is an `Mvp` then if p is a scalar, return that scalar, otherwise return `false`. Or if p is a list, then apply `ScalMvp` recursively to it (but return false if it contains any `Mvp` which is not a scalar). Else assume p is already a scalar and thus return p.

```
gap> v:=[Mvp("x"),Mvp("y")];
[ x, y ]
gap> ScalMvp(v);
false
gap> w:=List(v,p->Value(p,["x",2,"y",3]));
[ 2, 3 ]
gap> Gcd(w);
Error, sorry, the elements of <arg> lie in no common ring domain in
Domain( arg[1] ) called from
DefaultRing( ns ) called from
Gcd( w ) called from
main loop
brk>
gap> Gcd(ScalMvp(w));
1
```

2.5 LaurentDenominator

`LaurentDenominator(` *p1* `,` *p2* `, ... )`

Returns the unique monomial `m` of minimal degree such that for all the Laurent polynomial arguments *p1*, *p2*, etc... the product $m * p_i$ is a true polynomial.

```
gap> LaurentDenominator(x^-1,y^-2+x^4);
xy^2
```

The next functions have been provided by Gwenaëlle Genet

2.6 MvpGcd

`MvpGcd(` *p1* `,` *p2* `, ...)`

Returns the Gcd of the `Mvp` arguments. The arguments must be true polynomials.

```
gap> MvpGcd(x^2-y^2,(x+y)^2);
x+y
```

2.7 MvpLcm

`MvpLcm(` *p1* `,` *p2* `, ...)`

Returns the Lcm of the `Mvp` arguments. The arguments must be true polynomials.

```
gap> MvpLcm(x^2-y^2,(x+y)^2);
xy^2-x^2y-x^3+y^3
```

2.8 RatFrac

`RatFrac(` *num* `[,`*den*`] )`

Build the rational fraction (`RatFrac`) with numerator *num* and denominator *den* (when *den* is omitted it is taken to be 1).

```
gap> RatFrac(x,y);
x/y
gap> RatFrac(x*y^-1);
x/y
```

2.9 Operations for RatFrac

The arithmetic operations `+`, `-`, `*`, `/` and `^` work for `RatFrac`s. They also have `Print` and `String` methods.

```
gap> 1/(x+1)+y^-1;
(1+x+y)/(y+xy)
gap> 1/(x+1)*y^-1;
1/(y+xy)
gap> 1/(x+1)/y;
1/(y+xy)
gap> 1/(x+1)^-2;
1+2x+x^2
```

Similarly to `Mvp`s, `RatFrac`s hav `Format` and `Value` methods

```
gap> Format(1/(x*y+1));
"1/(1+xy)"
gap> FormatGAP(1/(x*y+1));
"1/(1+x*y)"
gap> Value(1/(x*y+1),["x",2]);
1/(1+2y)
```

2.10 IsRatFrac

`IsRatFrac(` *p* `)`

Returns `true` if p is an `Mvp` and false otherwise.

```
gap> IsRatFrac(1+RatFrac(x));
true
gap> IsRatFrac(x);
false
```

Chapter 3

The VKCURVE functions

We document here the various functions which are used in Van Kampen's algorithm as described in the introduction.

3.1 Discy

`Discy(` *Mvp p* `)`

The input should be an `Mvp` in `x` and `y`, with rational coefficients. The function returns the discriminant of *p* with respect to `x` (an `Mvp` in `y`); it uses interpolation to reduce the problem to discriminants of univariate polynomials, and works reasonably fast (not hundreds of times slower than MAPLE...).

```
gap> Discy(x+y^2+x^3+y^3);
4+27y^4+54y^5+27y^6
```

3.2 ResultantMat

`ResultantMat(` *v*, *w* `)`

v and *w* are vectors representing coefficients of two polynomials. The function returns Sylvester matrix for these two polynomials (whose determinant is the resultant of the two polynomials). It is used internally by Discy.

```
gap>  p:=x+y^2+x^3+y^3;
x+y^2+x^3+y^3
gap>  c:=Coefficients(p,"x");
[ y^2+y^3, 1, 0, 1 ]
gap> PrintArray(ResultantMat(c,Derivative(c)));
[[      1,       0,       1, y^2+y^3,       0],
 [      0,       1,       0,       1, y^2+y^3],
 [      3,       0,       1,       0,       0],
 [      0,       3,       0,       1,       0],
 [      0,       0,       3,       0,       1]]
gap> DeterminantMat(ResultantMat(c,Derivative(c)));
4+27y^4+54y^5+27y^6
```

3.3 NewtonRoot

`NewtonRoot(`*p*`,`*initial*`,`*precision*`)`

Here *p* is a list of `Complex` rationals representing the coefficients of a polynomial. The function computes a complex rational approximation to a root of *p*, guaranteed of distance closer than *precision* (a rational) to an actual root. The first approximation used is *initial*. If *initial* is in the attraction basin of a root of *p*, the one approximated. A possibility is that the Newton method starting from *initial* does not converge (the number of iterations after which this is decided is controlled by `VKCURVE.NewtonLim`); then the function returns `false`. Otherwise the function returns a pair: the approximation found, and an upper bound of the distance between that approximation and an actual root. The upper bound returned is a power of 10, and the approximation denominator's is rounded to a power of 10, in order to return smaller-sized rational result as much as possible. The point of returning an upper bound is that it is usually better than the asked-for *precision*. For the precision estimate a good reference is [HSS01].

```
gap> p:=List([1,0,1],Complex); # p=x^2+1
[ 1, 0, 1 ]
gap> NewtonRoot(p,Complex(1,1),10^-7);
[ I, 1/1000000000 ]
# obtained precision is actually 10^-9
gap> NewtonRoot(p,Complex(1),10^-7);
false
# here Newton does not converge
```

3.4 SeparateRootsInitialGuess

`SeparateRootsInitialGuess(`*p*`, `*v*`, `*safety*`)`

Here *p* is a list of complex rationals representing the coefficients of a polynomial, and *v* is a list of approximations to roots of *p* which should lie in different attraction basins for Newton's method. The result is a list *l* of complex rationals representing approximations to the roots of *p* (each element of *l* is the root in whose attraction basin the corresponding element of *v* lies), such that if *d* is the minimum distance between two elements of *l*, then there is a root of *p* within radius $d/(2*safety)$ of any element of *l*. When the elements of *v* do not lie in different attraction basins (which is necessarily the case if *p* has multiple roots), `false` is returned.

```
gap> p:=List([1,0,1],Complex);
[ 1, 0, 1 ]
gap> SeparateRootsInitialGuess(p,[Complex(1,1),Complex(1,-1)],100);
[ I, -I ]
gap> SeparateRootsInitialGuess(p,[Complex(1,1),Complex(2,1)],100);
false # 1+I and 2+I not in different attraction basins
```

3.5 SeparateRoots

`SeparateRoots(`*p*`, `*safety*`)`

Here p is a univariate `Mvp` with rational or complex rational coefficients, or a vector of rationals or complex rationals describing the coefficients of such a polynomial. The result is a list l of complex rationals representing approximations to the roots of p, such that if d is the minimum distance between two elements of l, then there is a root of p within radius $d/(2*safety)$ of any element of l. This is not possible when p has multiple roots, in which case `false` is returned.

```
gap> SeparateRoots(x^2+1,100);
[ I, -I ]
gap> SeparateRoots((x-1)^2,100);
false
gap> SeparateRoots(x^3-1,100);
[ -1/2-108253175473/125000000000I, 1, -1/2+108253175473/125000000000I]
```

3.6 LoopsAroundPunctures

`LoopsAroundPunctures(`*points*`)`

The input is a list of complex rational numbers. The function computes piecewise-linear loops representing generators of the fundamental group of the complement of *points* in the complex line.

```
gap> LoopsAroundPunctures([Complex(0,0)]);
rec(
  points := [ -I, -1, 1, I ],
  segments := [ [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  loops := [ [ 4, -3, -1, 2 ] ] )
```

The output is a record with three fields. The field `points` contains a list of complex rational numbers. The field `segments` contains a list of oriented segments, each of them encoded by the list of the positions in `points` of its two endpoints. The field `loops` contains a list of list of integers. Each list of integers represents a piecewise linear loop, obtained by concatenating the elements of `segments` indexed by the integers (a negative integer is used when the opposed orientation of the segment has to be taken).

3.7 FollowMonodromy

`FollowMonodromy(`*r*`,`*segno*`,`*print*`)`

This function computes the monodromy braid of the solution in x of an equation $P(x, y) = 0$ along a segment $[y_0, y_1]$. It is called by `FundamentalGroup`, once for each of the segments. The first argument is a global record, similar to the one produced by `FundamentalGroup` (see the documentation of this function) but only containing intermediate information. The second argument is the position of the segment in `r.segments`. The third argument is a print function, determined by the printlevel set by the user (typically, by calling `FundamentalGroup` with a second argument).

The function returns an element of the ambient braid group `r.B`.

This function has no reason to be called directly by the user, so we do not illustrate its behavior. Instead, we explain what is displayed on screen when the user sets the printlevel to 2.

What is quoted below is an excerpt of what is displayed on screen during the execution of
`gap>  FundamentalGroup((x+3*y)*(x+y-1)*(x-y),2);`

```
<1/16>    1 time=         0   ?2?1?3
<1/16>    2 time=     0.125   R2. ?3
<1/16>    3 time=   0.28125   R2. ?2
<1/16>    4 time=  0.453125   ?2R1?2
<1/16>    5 time=  0.578125   R1. ?2
======================================
=    Nontrivial braiding = 2         =
======================================
<1/16>    6 time=  0.734375   R1. ?1
<1/16>    7 time=   0.84375   . ?0.
<1/16>    8 time=  0.859375   ?1R0?1
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[1]:=B(2);
# segment 1/16 Time=0.1sec
```

`FollowMonodromy` computes its results by subdividing the segment into smaller subsegments on which the approximations are controlled. It starts at one end and moves subsegment after subsegment. A new line is displayed at each step.

The first column indicates which segment is studied. In the example above, the function is computing the monodromy along the first segment (out of 16). This gives a rough indication of the time left before completion of the total procedure. The second column is the number of iterations so far (number of subsegments). In our example, `FollowMonodromy` had to cut the segment into 8 subsegments. Each subsegment has its own length. The cumulative length at a given step, as a fraction of the total length of the segment, is displayed after `time=`. This gives a rough indication of the time left before completion of the computation of the monodromy of this segment. The segment is completed when this fraction reaches 1.

The last column has to do with the piecewise-linear approximation of the geometric monodromy braid. It is subdivided into sub-columns for each string. In the example above, there are three strings. At each step, some strings are fixed (they are indicated by `.` in the corresponding column). A symbol like `R5` or `?3` indicates that the string is moving. The exact meaning of the symbol has to do with the complexity of certain sub-computations.

As some strings are moving, it happens that their real projections cross. When such a crossing occurs, it is detected and the corresponding element of B_n is displayed on screen (`Nontrivial braiding =`...) The monodromy braid is the product of these elements of B_n, multiplied in the order in which they occur.

3.8 ApproxFollowMonodromy

`ApproxFollowMonodromy(`r`,`$segno$`,`pr`)`

This function computes an approximation of the monodromy braid of the solution in x of an equation $P(x, y) = 0$ along a segment $[y_0, y_1]$. It is called by `FundamentalGroup`, once for each of the segments. The first argument is a global record, similar to the one produced by `FundamentalGroup` (see the documentation of this function) but only containing intermediate information. The second argument is the position of the segment in `r.segments`. The third argument is a print function, determined by the printlevel set by the user (typically, by calling `FundamentalGroup` with a second argument).

Contrary to `FollowMonodromy`, `ApproxFollowMonodromy` does not control the approximations; it just uses a heuristic for how much to move along the segment between linear braid computations, and this heuristic may possibly fail. However, we have not yet found an example for which the result is actually incorrect, and thus the existence is justified by the fact that for some difficult computations, it is sometimes many times faster than `FollowMonodromy`. We illustrate its typical output when *printlevel* is 2.

```
VKCURVE.monodromyApprox:=true;
 FundamentalGroup((x+3*y)*(x+y-1)*(x-y),2);
```

....

```
5.3.6. ***rejected
4.3.6.<15/16>mindist=3 step=1/2 total=0 logdisc=1 ***rejected
3.3.4.<15/16>mindist=3 step=1/4 total=0 logdisc=1 ***rejected
3.3.4.<15/16>mindist=3 step=1/8 total=0 logdisc=1 ***rejected
3.3.3.<15/16>mindist=3 step=1/16 total=0 logdisc=1
3.2.3.<15/16>mindist=2.92 step=1/16 total=1/16 logdisc=1
3.3.3.<15/16>mindist=2.83 step=1/16 total=1/8 logdisc=1
3.2.3.<15/16>mindist=2.75 step=1/16 total=3/16 logdisc=1
3.3.3.<15/16>mindist=2.67 step=1/16 total=1/4 logdisc=1
======================================
=    Nontrivial braiding = 2         =
======================================
3.2.3.<15/16>mindist=2.63 step=1/16 total=5/16 logdisc=1
3.2.3.<15/16>mindist=2.75 step=1/16 total=3/8 logdisc=1
3.3.3.<15/16>mindist=2.88 step=1/16 total=7/16 logdisc=1
3.2.3.<15/16>mindist=3 step=1/16 total=1/2 logdisc=1
3.3.3.<15/16>mindist=3.13 step=1/16 total=9/16 logdisc=1
3.2.3.<15/16>mindist=3.25 step=1/16 total=5/8 logdisc=1
3.3.3.<15/16>mindist=3.38 step=1/16 total=11/16 logdisc=1
3.2.3.<15/16>mindist=3.5 step=1/16 total=3/4 logdisc=1
3.2.3.<15/16>mindist=3.63 step=1/16 total=13/16 logdisc=1
3.2.3.<15/16>mindist=3.75 step=1/16 total=7/8 logdisc=1
3.2.3.<15/16>mindist=3.88 step=1/16 total=15/16 logdisc=1 ***up
# Monodromy error=0
# Minimal distance=2.625
# Minimal step=1/16=-0.05208125+0.01041875I
# Adaptivity=10
monodromy[15]:=B(2);
# segment 15/16 Time=0.2sec
```

Here at each step the following information is displayed: first, how many iterations of the Newton method were necessary to compute each of the 3 roots of the current polynomial $f(x, y_0)$ if we are looking at the point y_0 of the segment. Then, which segment we are dealing with (here the 15th of 16 in all). Then the minimum distance between two roots of $f(x, y_0)$ (used in our heuristic). Then the current step in fractions of the length of the segment we are looking at, and the total fraction of the segment we have done. Finally, the decimal logarithm of the absolute value of the discriminant at the current point (used in the heuristic). Finally, an indication if the heuristic predicts that we should halve the step (`***rejected`) or that we may double it (`***up`).

The function returns an element of the ambient braid group `r.B`.

3.9 LBraidToWord

`LBraidToWord(`*v1*`,`*v2*`,`*B*`)`

This function converts the linear braid given by *v1* and *v2* into an element of the braid group *B*.

```
gap> B:=Braid(CoxeterGroupSymmetricGroup(3));
function ( arg ) ... end
gap> i:=Complex(0,1);
I
gap> LBraidToWord([1+i,2+i,3+i],[2+i,1+2*i,4-6*i],B);
1
```

The list *v1* and *v2* must have the same length, say n. The braid group B should be the braid group on n strings, in its CHEVIE implementation. The elements of *v1* (resp. *v2*) should be n distinct complex rational numbers. We use the Brieskorn basepoint, namely the contractible set $C + iV_{\mathbb{R}}$ where C is a real chamber; therefore the endpoints need not be equal (hence, if the path is indeed a loop, the final endpoint must be given). The linear braid considered is the one with affine strings connecting each point in *v1* to the corresponding point in *v2*. These strings should be non-crossing. When the numbers in *v1* (resp. *v2*) have distinct real parts, the real picture of the braid defines a unique element of B. When some real parts are equal, we apply a lexicographical desingularization, corresponding to a rotation of *v1* and *v2* by an arbitrary small positive angle.

3.10 BnActsOnFn

`BnActsOnFn(`*braid b*`,`*Free group F*`)`

This function implements the Hurwitz action of the braid group on n strings on the free group on n generators, where the standard generator σ_i of B_n fixes the generators $f_1, \ldots, f_n$, except f_i which is mapped to f_{i+1} and f_{i+1} which is mapped to $f_{i+1}^{-1} f_i f_{i+1}$.

```
gap> B:=Braid(CoxeterGroupSymmetricGroup(3));
function ( arg ) ... end
gap> b:=B(1);
1
gap> BnActsOnFn(b,FreeGroup(3));
GroupHomomorphismByImages( Group( f.1, f.2, f.3 ), Group( f.1, f.2, f.3 ),
[ f.1, f.2, f.3 ], [ f.2, f.2^-1*f.1*f.2, f.3 ] )
gap> BnActsOnFn(b^2,FreeGroup(3));
GroupHomomorphismByImages( Group( f.1, f.2, f.3 ), Group( f.1, f.2, f.3 ),
[ f.1, f.2, f.3 ], [ f.2^-1*f.1*f.2, f.2^-1*f.1^-1*f.2*f.1*f.2, f.3 ] )
```

The second input is the free group on n generators. The first input is an element of the braid group on n strings, in its CHEVIE implementation.

3.11 VKQuotient

`VKQuotient(`*braids*`,[`*bad*`])`

The input *braid* is a list of braids $b_1, \ldots, b_d$, living in the braid group on n strings. Each b_i defines by Hurwitz action an automorphism ϕ_i of the free group F_n. The function return the group defined by the abstract presentation:

$$< f_1, \ldots, f_n \mid \forall i, j, \phi_i(f_j) = f_j >$$

The optional second argument *bad* is another list of braids $c_1, \ldots, c_e$ (representing the monodromy around bad roots of the discriminant). For each c_k, we denote by ψ_k the corresponding Hurwitz automorphism of F_n. When a second argument is supplied, the function returns the group defined by the abstract presentation:

$$< f_1, \ldots, f_n, g_1, \ldots, g_k \mid \forall i, j, k, \phi_i(f_j) = f_j, \psi_k(f_j)g_k = g_k f_j >$$

```
gap> B:=Braid(CoxeterGroupSymmetricGroup(3));
function ( arg ) ... end
gap> b1:=B(1)^3; b2:=B(2);
1.1.1
2
gap> g:=VKQuotient([b1,b2]);
Group( f.1, f.2, f.3 )
gap>  last.relators;
[ f.2^-1*f.1^-1*f.2*f.1*f.2*f.1^-1, IdWord,
  f.2^-1*f.1^-1*f.2^-1*f.1*f.2*f.1, f.3*f.2^-1, IdWord, f.3^-1*f.2 ]
gap> p:=PresentationFpGroup(g);Display(p);
<< presentation with 3 gens and 4 rels of total length 16 >>
1: c=b
2: b=c
3: bab=aba
4: aba=bab
gap> SimplifyPresentation(p);Display(p);
# I there are 2 generators and 1 relator of total length 6
1: bab=aba
```

3.12 DisplayPresentation

`DisplayPresentation(`*p*`)`

Displays the presentation p in a compact form, using the letters `abc...` for the generators and `ABC...` for their inverses. In addition the program tries to show relations in "positive"form, i.e. as equalities between words involving no inverses.

```
gap> F:=FreeGroup(2);;
gap> p:=PresentationFpGroup(F/[F.2*F.1*F.2*F.1^-1*F.2^-1*F.1^-1]);
<< presentation with 2 gens and 1 rels of total length 6 >>
gap> Display(p);
1: bab=aba
```

```
gap> PrintRec(p);
rec(
  isTietze           := true,
  operations         := PresentationOps,
  generators         := [ f.1, f.2 ],
  tietze             := [ 2, 1, 6, [ f.1, f.2 ], [ 2, 1, 0, -1, -2 ],
  [ [ -2, -1, 2, 1, 2, -1 ] ], [ 6 ], [ 0 ], 0, false, 0, 0, 0, 0,
  [ 2, 1, 6 ], 0, 0, 0, 0, 0 ],
  components         := [ 1, 2 ],
  1                  := f.1,
  2                  := f.2,
  nextFree           := 3,
  identity           := IdWord,
  eliminationsLimit  := 100,
  expandLimit        := 150,
  generatorsLimit    := 0,
  lengthLimit        := infinity,
  loopLimit          := infinity,
  printLevel         := 1,
  saveLimit          := 10,
  searchSimultaneous := 20,
  protected          := 0 )
```

3.13 ShrinkPresentation

`ShrinkPresentation(`*p* `[,`*tries*`])`

This is our own program to simplify group presentations. We have found heuristics which make it somewhat more efficient than GAP3's programs `SimplifiedFpGroup` and `TzGoGo`, but the algorithm depends on random numbers so is not reproducible. The main idea is to rotate relators between calls to GAP3 functions. By default 1000 such rotations are tried (unless the presentation is so small that less rotations exhaust all possible ones), but the actual number tried can be controlled by giving a second parameter *tries* to the function. Another useful tool to deal with presentations is `TryConjugatePresentation` described in the utility functions.

```
gap> DisplayPresentation(p);
1: ab=ba
2: dbd=bdb
3: bcb=cbc
4: cac=aca
5: adca=cadc
6: dcdc=cdcd
7: adad=dada
8: Dbdcbd=cDbdcb
9: adcDad=dcDadc
10: dcdadc=adcdad
11: dcabdcbda=adbcbadcb
12: caCbdcbad=bdcbadBcb
```

```
13: cbDadcbad=bDadcbadc
14: cdAbCadBc=bdcAbCdBa
15: cdCbdcabdc=bdcbadcdaD
16: DDBcccbdcAb=cAbCdcBCddc
17: CdaBdbAdcbCad=abdcAbDadBCbb
18: bdbcabdcAADAdBDa=cbadcbDadcBDABDb
19: CbdbadcDbbdCbDDadcBCDAdBCDbdaDCDbdcbadcBCDAdBCDBBdacDbdccb
    =abdbcabdcAdcbCDDBCDABDABDbbdcbDadcbCDAdBCabDACbdBadcaDbAdd
gap> ShrinkPresentation(p);
# I there are 4 generators and 19 relators of total length 332
# I there are 4 generators and 17 relators of total length 300
# I there are 4 generators and 17 relators of total length 282
# I there are 4 generators and 17 relators of total length 278
# I there are 4 generators and 16 relators of total length 254
# I there are 4 generators and 15 relators of total length 250
# I there are 4 generators and 15 relators of total length 248
# I there are 4 generators and 15 relators of total length 246
# I there are 4 generators and 14 relators of total length 216
# I there are 4 generators and 13 relators of total length 210
# I there are 4 generators and 13 relators of total length 202
# I there are 4 generators and 13 relators of total length 194
# I there are 4 generators and 12 relators of total length 174
# I there are 4 generators and 12 relators of total length 170
# I there are 4 generators and 12 relators of total length 164
# I there are 4 generators and 12 relators of total length 162
# I there are 4 generators and 12 relators of total length 148
# I there are 4 generators and 12 relators of total length 134
# I there are 4 generators and 12 relators of total length 130
# I there are 4 generators and 12 relators of total length 126
# I there are 4 generators and 12 relators of total length 124
# I there are 4 generators and 12 relators of total length 118
# I there are 4 generators and 12 relators of total length 116
# I there are 4 generators and 11 relators of total length 100
gap> DisplayPresentation(p);
1: ba=ab
2: dbd=bdb
3: cac=aca
4: bcb=cbc
5: dAca=Acad
6: dcdc=cdcd
7: adad=dada
8: dcDbdc=bdcbdB
9: dcdadc=adcdad
10: adcDad=dcDadc
11: BcccbdcAb=dcbACdddc
```

Chapter 4

Some VKCURVE utility functions

We document here various utility functions defined by VKCURVE package and which may be useful also in other contexts.

4.1 BigNorm

`BigNorm(`c`)`

Given a `complex` number c with real part r and imaginary part j, returns a "cheap substitute"to the norm of c given by $\mathtt{r} + \mathtt{j}$.

```
gap> BigNorm(Complex(-1,-1));
2
```

4.2 DecimalLog

`DecimalLog(`r`)`

Given a rational number r, returns an integer k such that $10^k < \mathtt{r} \leq 10^{k+1}$.

```
gap> List([1,1/10,1/2,2,10],DecimalLog);
[ -1, -2, -1, 0, 1 ]
```

4.3 ComplexRational

`ComplexRational(`c`)`

c is a cyclotomic or a `Complex` number with `Decimal` or real cyclotomic real and imaginary parts. This function returns the corresponding rational complex number.

```
gap> evalf(E(3)/3);
-0.1666666667+0.2886751346I
gap> ComplexRational(last);
-16666666667/100000000000+28867513459/100000000000I
gap> ComplexRational(E(3)/3);
-1/6+28867513457/100000000000I
```

4.4 Dispersal

`Dispersal(v)`

v is a list of `complex` numbers representing points in the real plane. The result is a pair whose first element is the minimum distance between two elements of *v*, and the second is a pair of indices `[i,j]` such that `v[i]`, `v[j]` achieves this minimum distance.

```
gap> Dispersal([Complex(1,1),Complex(0),Complex(1)]);
[ 1, [ 1, 3 ] ]
```

4.5 ConjugatePresentation

`ConjugatePresentation(`*p* `[,`*conjugation*`])`

This program modifies a presentation by conjugating a generator by another. The conjugaction to apply is described by a length-3 string of the same style as the result of `DisplayPresentation`, that is `"abA"` means replace the second generator by its conjugate by the first, and `"Aba"` means replace it by its conjugate by the inverse of the first.

```
gap> F:=FreeGroup(4);;
gap> p:=PresentationFpGroup(F/[F.4*F.1*F.2*F.3*F.4*F.1^-1*F.4^-1*
> F.3^-1*F.2^-1*F.1^-1,F.4*F.1*F.2*F.3*F.4*F.2*F.1^-1*F.4^-1*F.3^-1*
> F.2^-1*F.1^-1*F.3^-1,F.2*F.3*F.4*F.1*F.2*F.3*F.4*F.3^-1*F.2^-1*
> F.4^-1*F.3^-1*F.2^-1*F.1^-1*F.4^-1]);
gap> DisplayPresentation(p);
1: dabcd=abcda
2: dabcdb=cabcda
3: bcdabcd=dabcdbc
gap> DisplayPresentation(ConjugatePresentation(p,"cdC"));
# I there are 4 generators and 3 relators of total length 36
1: cabdca=dcabdc
2: dcabdc=bdcabd
3: cabdca=abdcab
```

4.6 TryConjugatePresentation

`TryConjugatePresentation(`*p* `[,`*goal* `[,`*printlevel*`]])`

This program tries to simplify group presentations by applying conjugations to the generators. The algorithm depends on random numbers, and on tree-searching, so is not reproducible. By default the program stops as soon as a shorter presentation is found. Sometimes this does not give the desired presentation. One can give a second argument *goal*, then the program will only stop when a presentation of length less than *goal* is found. Finally, a third argument can be given and then all presentations the programs runs over which are of length less than or equal to this argument are displayed. Due to the non-deterministic nature of the program, it may be useful to run it several times on the same input. Upon failure (to improve the presentation), the program returns *p*.

```
gap> Display(p);
1: ba=ab
2: dbd=bdb
```

```
3: cac=aca
4: bcb=cbc
5: dAca=Acad
6: dcdc=cdcd
7: adad=dada
8: dcDbdc=bdcbdB
9: dcdadc=adcdad
10: adcDad=dcDadc
11: BcccbdcAb=dcbACdddc
gap> p:=TryConjugatePresentation(p);
# I there are 4 generators and 11 relators of total length 100
# I there are 4 generators and 11 relators of total length 120
# I there are 4 generators and 10 relators of total length 100
# I there are 4 generators and 11 relators of total length 132
# I there are 4 generators and 11 relators of total length 114
# I there are 4 generators and 11 relators of total length 110
# I there are 4 generators and 11 relators of total length 104
# I there are 4 generators and 11 relators of total length 114
# I there are 4 generators and 11 relators of total length 110
# I there are 4 generators and 11 relators of total length 104
# I there are 4 generators and 8 relators of total length 76
# I there are 4 generators and 8 relators of total length 74
# I there are 4 generators and 8 relators of total length 72
# I there are 4 generators and 8 relators of total length 70
# I there are 4 generators and 7 relators of total length 52
#  d->adA gives length 52
<< presentation with 4 gens and 7 rels of total length 52 >>
gap> Display(p);
1: ba=ab
2: dc=cd
3: aca=cac
4: dbd=bdb
5: bcb=cbc
6: adad=dada
7: aBcADbdac=dBCacbdaB
gap> TryConjugatePresentation(p,48);
# I there are 4 generators and 7 relators of total length 54
# I there are 4 generators and 7 relators of total length 54
# I there are 4 generators and 7 relators of total length 60
# I there are 4 generators and 7 relators of total length 60
# I there are 4 generators and 7 relators of total length 48
#  d->bdB gives length 48
<< presentation with 4 gens and 7 rels of total length 48 >>
gap> Display(last);
1: ba=ab
2: bcb=cbc
3: cac=aca
4: dbd=bdb
```

```
5: cdc=dcd
6: adad=dada
7: dAbcBa=bAcBad
```

4.7 FindRoots

`FindRoots(`*p*`,` *approx*`)`

p should be a univariate `Mvp` with cyclotomic or `Complex` rational or decimal coefficients or a list of cyclotomics or `Complex` rationals or decimals which represents the coefficients of a complex polynomial. The function returns `Complex` rational approximations to the roots of *p* which are better than *approx* (a positive rational). Contrary to the functions `SeparateRoots`, etc... described in the previous chapter, this function handles quite well polynomials with multiple roots. We rely on the algorithms explained in detail in [HSS01].

```
gap> FindRoots((x-1)^5,1/100000000000);
[ 6249999999993/6250000000000+29/12500000000000I,
  12499999999993/12500000000000-39/12500000000000I,
  12500000000023/12500000000000+11/6250000000000I,
  12500000000023/12500000000000+11/6250000000000I,
  312499999999/312500000000-3/6250000000000I ]
gap> evalf(last);
[ 1, 1, 1, 1, 1 ]
gap> FindRoots(x^3-1,1/10);
[ -1/2-108253175473/125000000000I, 1, -1/2+108253175473/125000000000I ]
gap> evalf(last);
[ -0.5-0.8660254038I, 1, -0.5+0.8660254038I ]
gap> List(last,x->x^3);
[ 1, 1, 1 ]
```

4.8 Cut

`Cut(`*string s* `[, opt])`

The first argument is a string, and the second one a record of options, if not given taken equal to `rec()`. This function prints its string argument *s* on several lines not exceeding *opt.width*; if not given *opt.width* is taken to be equal `SizeScreen[1]-2`. This is similar to how GAP3 prints strings, excepted no continuation line characters are printed. The user can specify after which characters to cut the string by specifying a field `opt.places`; the defaut is `opt.places=","`, but some other characters can be used — for instance a good choice for printing big polynomials could be `"+-"`. If a field `opt.file` is given, the result is appended to that file instead of written to standard output; this may be quite useful for dumping some GAP3 values to a file for later re-reading.

```
gap> Cut("an, example, with, plenty, of, commas\n",rec(width:=10));
an,
example,
with,
plenty,
of,
commas
gap>
```

Bibliography

[Bes01] D. Bessis. Zariski theorems and diagrams for braid groups. *Invent. Math.*, 145:487–507, 2001.

[Che73] D. Cheniot. Une démonstration du théorème de Zariski sur les sections hyperplanes d'une hypersurface projective et du théorème de Van Kampen sur le groupe fondamental du complémentaire d'une courbe projective plane. *Compositio Math.*, 27:141–158, 1973.

[HSS01] J. Hubbard, D. Schleicher, and S. Sutherland. How to find all roots of complex polynomials by Newton's method. *Invent. Math.*, 146:1–33, 2001.

Index

Index