Handbook of Magma Functions

Volume 4

Local Arithmetic Fields

John Cannon Wieb Bosma

Claus Fieker Allan Steel

Editors

Version 2.19

Sydney

December 17, 2012

MAGMA
COMPUTER ● ALGEBRA

Handbook of Magma Functions

Editors:

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors:

Geoff Bailey, Wieb Bosma, Gavin Brown, Nils Bruin, John Cannon, Jon Carlson, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Tim Dokchitser, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Volker Gebhardt, Sergei Haller, Michael Harrison, Florian Hess, Derek Holt, David Howden, Al Kasprzyk, Markus Kirschmer, David Kohel, Axel Kohnert, Dimitri Leemans, Paulette Lieby, Graham Matthews, Scott Murray, Eamonn O'Brien, Dan Roozemond, Ben Smith, Bernd Souvignier, William Stein, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, Alexa van der Waall, Paul van Wamelen, Helena Verrill, John Voight, Mark Watkins, Greg White

Production Editors:

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production:

Claus Fieker *Allan Steel*

VOLUME 4: OVERVIEW

VOLUME 4: CONTENTS

VIII MODULES 1387

PART VII
LOCAL ARITHMETIC FIELDS

45 VALUATION RINGS

Chapter 45

VALUATION RINGS

45.1 Introduction

Magma currently supports basic operations in valuation rings obtained either from the rational field $\mathbf{Q}$ (and a finite prime p), or from a field of rational functions over a field (and an irreducible polynomial, or the infinite prime).

45.2 Creation Functions

45.2.1 Creation of Structures

```
ValuationRing(Q, p)
```

Given the rational field Q and a rational prime number p, create the valuation ring R corresponding to the discrete non-Archimedean valuation v_p, consisting of rational numbers r such that $v_p(r) \geq 0$, that is, $r = \frac{x}{y} \in \mathbf{Q}$ such that $p \not| y$.

```
ValuationRing(F, f)
```

Given the rational function field F as a field of fractions of the univariate polynomial ring $K[x]$ over a field K, as well as a monic irreducible polynomial $f \in K[x]$, create the valuation ring R corresponding to the discrete non-Archimedean valuation v_f. Thus R consists of rational functions $\frac{g}{h} \in F$ with $v_f(g/h) \geq 0$, that is, with $f \not| h$.

```
ValuationRing(F)
```

Given the rational function field F as a field of fractions of the univariate polynomial ring $K[x]$ over a field K, create the valuation ring R corresponding to v_∞, consisting of $\frac{g}{h} \in F$ such that $\deg(h) \geq \deg(g)$.

45.2.2 Creation of Elements

```
V ! r
```

Given a valuation ring V and an element of the field of fractions F of V (from which V was created), coerce the element r into V. This is only possible for elements $r \in F$ for which the valuation on V is non-negative, an error occurs if this is not the case.

45.3 Structure Operations

45.3.1 Related Structures

`Category(V)` `Parent(V)` `PrimeRing(V)` `Center(V)`

`FieldOfFractions(V)`

Return field of fractions of the valuation ring V, which is the rational field or the function field from which V was created.

45.3.2 Numerical Invariants

`Characteristic(V)`

45.4 Element Operations

45.4.1 Arithmetic Operations

`+ v` `- v`

`v + w` `v - w` `v * w` `v ^ k` `v / w`

`v +:= w` `v -:= w` `v *:= w`

`v div w`

The quotient q of the division with remainder $v = qw + r$ of the valuation ring elements v and w, where the remainder will have valuation less than that of w; if the valuation of v is greater than or equal than that of w, this simply returns the quotient v/w, if the valuation of w exceeds that of v it returns 0.

45.4.2 Equality and Membership

`v eq w` `v ne w`

`v in V` `v notin V`

45.4.3 Parent and Category

`Parent(v)` `Category(v)`

45.4.4 Predicates on Ring Elements

`IsZero(n)` `IsOne(n)` `IsMinusOne(n)`

`IsNilpotent(n)` `IsIdempotent(n)`

`IsUnit(n)` `IsZeroDivisor(n)` `IsRegular(n)`

45.4.5 Other Element Functions

`EuclideanNorm(v)`

`Valuation(v)`

Given an element v of a valuation ring V, return the valuation (associated with V) of v.

`Quotrem(v, w)`

Given two elements v, w of a valuation ring V with associated valuation ϕ, return a quotient and remainder q and r in V such that $v = qw + r$ and $0 \leq \phi(r) < \phi(w)$. If $\phi(v) < \phi(w)$ this simply returns $q = 0$ and $r = v$, and if $\phi(v) \geq \phi(w)$ then it returns $q = v/w$ and $r = 0$.

`GreatestCommonDivisor(v, w)`

`Gcd(v, w)`

This function returns a greatest common divisor of two elements v, w in a valuation ring V. This will return u^m, where $m = \min(\phi(v), \phi(w))$ is the minimum of the valuations of v and w $m = \min(\phi(v), \phi(w))$ and u is the uniformizing element of V (with valuation $\phi(u) = 1$).

`ExtendedGreatestCommonDivisor(v, w)`

`Xgcd(v, w)`

`XGCD(v, w)`

This function returns a greatest common divisor $z \in V$ of two elements v, w in a valuation ring V as well as multipliers $x, y \in V$ such that $xv + yw = z$. The principal return value will be $z = u^m$, where $m = \min(\phi(v), \phi(w))$ is the minimum of the valuations of v and w $m = \min(\phi(v), \phi(w))$ and u is the uniformizing element of V (with valuation $\phi(u) = 1$).

46 NEWTON POLYGONS

Chapter 46
NEWTON POLYGONS

46.1 Introduction

This chapter introduces functions which allow the creation and simple study of Newton polygons. It allows data from a number of different contexts to be used to construct the polygons. It also covers different interpretations of Newton polygons, and translation among these interpretations. Recall that a Newton polygon is the intersection of finitely many rational half spaces in the rational plane. An advantage of this definition is that it emphasizes that Newton polygons are often thought of as being noncompact. However, they are not implemented here in this way. Instead polygons are interpreted as the convex hull of finitely many points of the plane (possibly including some points at $+\infty$ along the axes).

Any Newton polygon is contained in some virtual cartesian product of the rational field with itself (virtual since this plane is not a structure that is intended to be accessible to the user or which is characteristic of the polygon). The first and second coordinate functions of this plane are referred to as the x and y coordinates respectively. Points of the plane will always be written $\langle a, b\rangle$ while faces of polygons will be written $\langle a, b, c\rangle$. Geometrically, a face $\langle a, b, c\rangle$ is a one-dimensional boundary intersection of N with the line $ax + by = c$.

The *standard Newton polygon* of a polynomial $f = f(u, v)$ is, by definition, the convex hull of the points $\langle a, b\rangle$, so-called *Newton points*, ranging over monomials $u^a v^b$ having nonzero coefficient in f together with the points $+\infty$ on the two axes. The infinite points, however, are not listed among the vertices of the polygon; they are simply a convenient way of hiding all Newton points other than those on the 'lower lefthand' faces of the hull of those points. A similar definition applies to polynomials $f(y)$ whose coefficients lie in some field of fractional power series, or *Puiseux field*, $k\,\langle\langle x\rangle\rangle$. Newton Polygons can also be created for polynomials over local rings (and fields). The defining points are computed as $\langle i, v(a_i)\rangle$ where a_i is the coefficient of the i-th power of the generator and v denotes the valuation function on the ring. Infinite points arise only as the valuation of zero coefficients.

The main intended application of Newton polygons is to the Newton–Puiseux analysis of singular points of plane curves, or put another way, the factorization of polynomials defined over Puiseux fields.

The examples below should be thought of as running consecutively in a single Magma session.

```
> R<x,y> := PolynomialRing(Rationals(),2);
> f := x^5 + x^2*y + x^2*y^3 + y^4 + x^3*y^5 + y^2*x^7;
> N := NewtonPolygon(f);
> N;
Newton Polygon of x^7*y^2 + x^5 + x^3*y^5 + x^2*y^3 + x^2*y + y^4 over
```

```
Rational Field
> Faces(N);
[ <3, 2, 8>, <1, 3, 5> ]
> Vertices(N);
[ <0, 4>, <2, 1>, <5, 0> ]
```

This is the standard Newton polygon associated with the polynomial f. Only those vertices and faces of the convex hull of points which correspond to the monomials appearing in f which 'face' the origin are considered to be vertices and faces of the polygon. As already mentioned, one interpretation of this is to think of the points $+\infty$ on each axis as being included among the defining points of N, and then N is the convex hull of its defining points.

To consider N as the compact convex hull of only the monomials of f simply use alternative vertex and face functions.

```
> AllVertices(N);
[ <0, 4>, <2, 1>, <5, 0>, <7, 2>, <3, 5> ]
> AllFaces(N);
[ <3, 2, 8>, <1, 3, 5>, <-1, 1, -5>, <-3, -4, -29>, <1, -3, -12> ]
```

However, where there is a choice of interpretation, Magma always interprets N in the way it was defined. So for example, the functions which test for faces and vertices compare a given value with the sequence returned by the functions `Faces(N)` or `Vertices(N)`, which are fixed the first time they are calculated, rather than with any other collections of faces or vertices.

```
> IsFace(N,<3/4,3/6,2>);
true <3, 2, 8>
> IsFace(N,<-3,-4,-29>);
false
```

Notice that faces are reduced to normal integral form and that the correct form is returned as the second return value of the test function.

When a polygon is created using a polynomial, the restriction of the polynomial to faces is important characteristic data.

```
> FaceFunction(Faces(N)[1]);
x^2*y + y^4
```

The face function is simply the sum of those monomial terms of f (they keep their coefficients) whose corresponding Newton point lies on the given face.

46.2 Newton Polygons

All polygons are determined by a finite collection of points in the rational plane. For Magma, these points are the most basic attribute of any Newton polygon. They are always determined and recorded on creation of a polygon. Throughout this chapter, for a Newton polygon N, these points are denoted by P_N. As seen in the introduction, the main class of polygons is that comprising polygons in the first quadrant of the plane and including the points $+\infty$ on the two axes. But there are other useful types, especially when calculating factorizations of univariate polynomials over series rings. The data distinguishing the different flavours of Newton polygon is the collection of lines and points that are considered to be faces and vertices.

46.2.1 Creation of Newton Polygons

These are the functions available for constructing Newton polygons and retrieving the points which describe them.

```
NewtonPolygon(f)
```

`Faces` MonStgElt *Default : "Inner"*

The standard Newton polygon of a polynomial f in two variables. This is the hull of the Newton points of the polynomial together with the points $+\infty$ on each axis. The horizontal and vertical 'end' faces are not listed among the faces of the polygon; the points at infinity are not listed among the vertices of the polygon.

The parameter `Faces` can have the value `"Inner"`, `"Lower"` or `"All"`. This determines which faces are returned by the intrinsic `Faces`.

```
NewtonPolygon(f)
```

`SwapAxes` BoolElt *Default :* `false`

`Faces` MonStgElt *Default :*

The standard Newton polygon of a polynomial in one variable defined over a series ring or a local ring or field. A value of `true` for `SwapAxes` is only valid if the polynomial is over a series ring. If `SwapAxes` is set to `true` then the exponents of the series variable will be plotted on the horizontal axis and the exponents of the polynomial on the vertical axis.

For a polynomial over a series ring, the hull includes the points $+\infty$ on each axis. For a polynomial over a local ring, the infinite points are not included.

The parameter `Faces` can have the value `"All"`, `"Inner"` or `"Lower"`. This determines which faces are returned by the intrinsic `Faces`. The default for series rings is `"Inner"` and for local rings is `"Lower"`.

```
NewtonPolygon(f, p)
```

`Faces` MonStgElt *Default : "Inner"*

The newton polygon of f where p is a prime used for valuations of the coefficients of f. The polynomial f may be over the integers or rationals or a number field or algebraic function field or an order thereof. The prime p may be an integer or a prime ideal. The newton polygon will have points (i, v_i) where i is the exponent of a term of f and v_i is the valuation of the coefficient of the ith term. The points at $+\infty$ on each axis are included.

The parameter `Faces` can have the value `"Inner"`, `"Lower"` or `"All"`. This determines which faces are returned by the intrinsic `Faces`.

```
NewtonPolygon(f, p)
```

`Faces` MonStgElt *Default : "Inner"*

The newton polygon of the polynomial f where the place p of an algebraic function field is the prime used for determining the valuations of the coefficients of f. The points at $+\infty$ on each axis are included.

The parameter `Faces` can have the value `"Inner"`, `"Lower"` or `"All"`. This determines which faces are returned by the intrinsic `Faces`.

```
NewtonPolygon(C)
```

The standard Newton polygon of the defining polynomial of the curve C.

```
NewtonPolygon(V)
```

`Faces` MonStgElt *Default : "All"*

The Newton polygon that is the compact convex hull of the set or sequence V of points of the form $\langle a, b\rangle$ where a, b are integers or rational numbers.

The parameter `Faces` can have the value `"All"`, `"Lower"` or `"Inner"`. This determines which faces are returned by the intrinsic `Faces`.

```
DefiningPoints(N)
```

The points of the rational plane used in the initial creation of N. Applying this function to two polygons allows their defining points to be compared. No explicit function is provided for testing whether defining points of two polygons are equal.

Example H46E1

Some ways of creating Newton Polygons from polynomials are shown below.

```
> P<y> := PuiseuxSeriesRing(Rationals());
> R<x> := PolynomialRing(P);
> f := 3*x^4 + (5*y^3 + 4*y^(1/4))*x^3 + (7*y^2 + 1/2*y^(1/3))*x^2 + 6*x + y^(
> 4/5);
> N := NewtonPolygon(f);
> N;
Newton Polygon of 3*x^4 + (4*y^(1/4) + 5*y^3)*x^3 + (1/2*y^(1/3) + 7*y^2)*x^2 +
6*x + y^(4/5) over Puiseux series field in y over Rational Field
> P<x> := PolynomialRing(Integers());
> L := ext<ext<pAdicRing(5, 100) | 3> | x^2 + 5>;
> R<x> := PolynomialRing(L);
> f := 3*x^4 + 75*x^3 + 78*x^2 + 10*x + 750;
> NR := NewtonPolygon(f);
> NR;
Newton Polygon of 3*x^4 + 75*x^3 + 78*x^2 + 10*x + 750 over L
```

Newton Polygons can also be created by specifying the defining points that the polygon must enclose.

```
> N2 := NewtonPolygon({<2, 0>, <0, 3>, <4, 1>});
> N2;
Newton Polygon with defining points {(0, 3), (2, 0), (4, 1)}
> N6 := NewtonPolygon({<1, 4>, <1, 6>, <2, 4>, <3, 1>, <6, 1>, <5, 2>, <4, 5>,
> <4, 7>, <6, 6>, <7, 7>, <2, 7>, <5, 9>, <8, 4>, <8, 6>, <8, 8>, <7, 9>});
> N6;
Newton Polygon with defining points {(1, 4), (1, 6), (2, 4), (2, 7), (3, 1), (4,
5), (4, 7), (5, 2), (5, 9), (6, 1), (6, 6), (7, 7), (7, 9), (8, 4), (8, 6), (8,
8)}
```

These polygons will be referred to in later examples.

46.2.2 Vertices and Faces of Polygons

Both the vertices $\langle a, b\rangle$ and faces $\langle a, b, c\rangle$ (representing $ax + by = c$) of a given polygon N are computed as needed. As seen above, these will be a particular choice of possible faces and vertices determined by the data used to create the polygon. They can be recovered using the `Faces()` and `Vertices()` intrinsics. A different choice of faces and vertices, those faces and vertices of the compact convex hull of the defining points say, can be made using the other intrinsics below.

Recall that P_N denotes the set of points used in the definition of the Newton polygon N whether they arise as the powers of monomials appearing in a polynomial or have been given explicitly as a sequence of pairs.

`Faces(N)`

The sequence of faces $\langle a, b, c\rangle$ (representing $ax + by = c$) of N listed anticlockwise. How this is interpreted in terms of the points used to create N depends on the creation function used (see Section 46.2.1). The faces are listed anticlockwise starting with the face with its left endpoint being the lowest of the leftmost points.

`InnerFaces(N)`

Those faces of the compact convex hull of P_N starting at the lowest of the leftmost points which have strictly negative gradient.

`LowerFaces(N)`

Those faces of the compact convex hull of P_N which bound it below in the y direction.

`OuterFaces(N)`

The union of lower faces which aren't inner faces and the faces which bound the compact convex hull of P_N above in the y-direction (ignoring infinite points).

`AllFaces(N)`

The faces of the compact convex hull of P_N.

Example H46E2

Using some of the polygons defined before the different types of faces are illustrated.

```
> Faces(N);
[ <4, 5, 4> ]
> InnerFaces(N);
[ <4, 5, 4> ]
> OuterFaces(N);
[ <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> AllFaces(N);
[ <4, 5, 4>, <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> Faces(NR);
[ <4, 1, 6>, <2, 1, 4>, <0, 1, 0> ]
> InnerFaces(NR);
[ <4, 1, 6>, <2, 1, 4> ]
> LowerFaces(NR);
[ <4, 1, 6>, <2, 1, 4>, <0, 1, 0> ]
```

For the polynomial over the Puiseux Field it is no coincidence that `InnerFaces` and `Faces` return the same sequences. Similarly, for the polynomial over the local ring `Faces` is defined to be `LowerFaces`. For both, this is the category of faces that gives the most information for the purposes that the polygon is used. It can also be noted that combining `InnerFaces` and `OuterFaces` will give `AllFaces` with no repetitions (though repetitions will occur if the polygon has only one face and this face is an inner face).

`Vertices(N)`

The sequence of vertices of N. The vertices will be listed anticlockwise from the lowest of the leftmost points.

`InnerVertices(N)`

The sequence of vertices which arise as endpoints of inner faces.

`LowerVertices(N)`

The sequence of vertices which arise as endpoints of lower faces.

`OuterVertices(N)`

The sequence of vertices which arise as endpoints of outer faces.

`AllVertices(N)`

The sequence of vertices of the compact convex hull of P_N.

Example H46E3

This example illustrates the types of vertices that can be calculated. Note that the printing of these polygons, created from their defining points, changes as more information is calculated. This would occur in the same manner if faces were being calculated instead of vertices.

```
> InnerVertices(N2);
[ <0, 3>, <2, 0> ]
> N2;
Newton Polygon with vertices {(0, 3), (2, 0)} and defining points {(0, 3), (2,
0), (4, 1)}
> InnerVertices(N6);
[ <1, 4>, <3, 1> ]
> N6;
Newton Polygon with vertices {(1, 4), (3, 1)} and defining points {(1, 4), (1,
6), (2, 4), (2, 7), (3, 1), (4, 5), (4, 7), (5, 2), (5, 9), (6, 1), (6, 6), (7,
7), (7, 9), (8, 4), (8, 6), (8, 8)}
> Vertices(N2);
[ <0, 3>, <2, 0>, <4, 1> ]
> Vertices(N6);
[ <1, 4>, <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6> ]
> AllVertices(N2);
[ <0, 3>, <2, 0>, <4, 1> ]
> N2;
Newton Polygon with vertices {(0, 3), (2, 0), (4, 1)} and defining points {(0,
3), (2, 0), (4, 1)}
> AllVertices(N6);
[ <1, 4>, <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6> ]
> N6;
Newton Polygon with vertices {(1, 4), (3, 1), (6, 1), (8, 4), (8, 8), (7, 9),
(5, 9), (2, 7), (1, 6)} and defining points {(1, 4), (1, 6), (2, 4), (2, 7), (3,
1), (4, 5), (4, 7), (5, 2), (5, 9), (6, 1), (6, 6), (7, 7), (7, 9), (8, 4), (8,
6), (8, 8)}
```

Here `Vertices` has been defined to be `AllVertices`. All the known vertices of the polygon are printed when the polygon is printed. There is some overlap between the inner and outer vertices as is shown below. Every vertex is either an inner vertex or an outer vertex with some being both. Not all defining points are vertices.

```
> OuterVertices(N6);
[ <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6>, <1, 4> ]
> OuterVertices(N2);
[ <2, 0>, <4, 1>, <0, 3> ]
```

`EndVertices(F)`

A sequence containing the two end vertices of the face $F = \langle a, b, c \rangle$.

`FacesContaining(N,p)`

Those faces of the polygon N returned by `Faces` on which the point $p = \langle a, b\rangle$ lies.

Example H46E4

Using some of the example polygons that have been created above, we illustrate the simple use of `EndVertices` and `FacesContaining`.

```
> AN := AllFaces(N);
> AN;
[ <4, 5, 4>, <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> A6 := AllFaces(N6);
> A6;
[ <3, 2, 11>, <0, 1, 1>, <-3, 2, -16>, <-1, 0, -8>, <-1, -1, -16>, <0, -1, -9>,
<2, -3, -17>, <1, -1, -5>, <1, 0, 1> ]
> AllVertices(N);
[ <0, 4/5>, <1, 0>, <4, 0>, <3, 1/4> ]
> AllVertices(N6);
[ <1, 4>, <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6> ]
> EndVertices(AN[1]);
[ <0, 4/5>, <1, 0> ]
> EndVertices(AN[4]);
[ <0, 4/5>, <3, 1/4> ]
> EndVertices(A6[1]);
[ <1, 4>, <3, 1> ]
> EndVertices(A6[5]);
[ <7, 9>, <8, 8> ]
> EndVertices(A6[9]);
[ <1, 4>, <1, 6> ]
> FacesContaining(N, <1, 0>);
[ <4, 5, 4> ]
> FacesContaining(N6, <1, 0>);
[]
> FacesContaining(N6, <4, 1>);
[ <0, 1, 1> ]
> FacesContaining(N, <4, 1>);
[]
> FacesContaining(N6, <3, 1>);
[ <3, 2, 11>, <0, 1, 1> ]
```

`GradientVector(F)`

The a and b values of the line describing the face F of the form $a * x + b * y = c$ where a, b and c are integers.

`GradientVectors(N)`

A sequence containing the gradient vectors of the faces of the newton polygon N.

`Weight(F)`

The c value of the line describing the face F of the form $a * x + b * y = c$ where a, b and c are integers.

`Slopes(N)`

The slopes of the faces of the newton polygon N.

`InnerSlopes(N)`

`LowerSlopes(N)`

`AllSlopes(N)`

The slopes of the polygon N corresponding of `InnerFaces`, `LowerFaces` and `AllFaces` respectively.

Example H46E5

In this example `GradientVector` and `Weight` can be seen to be access functions on the components of a face of a polygon.

```
> A := AllFaces(N);
> A;
[ <4, 5, 4>, <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> f := A[3];
> GradientVector(f);
<-1, -4>
> Weight(f);
-4
```

The gradient of the face can now be easily computed as shown.

```
> a := GradientVector(f)[1];
> b := GradientVector(f)[2];
> -a/b;
-1/4
```

46.2.3 Tests for Points and Faces

Once more, recall that P_N denotes the finite set of points in the plane used to define the Newton polygon N. Whether or not a point is considered to lie in a polygon depends on what are considered to be its faces. Magma always uses the list of faces returned by `Faces(N)` when testing points. Of course, this is not always the case in applications. One must to perform other tests explicitly when there is doubt.

`IsFace(N, F)`

Return `true` if and only if the tuple $F = \langle a, b, c \rangle$ describes a line coinciding with a face of the polygon N as returned by `Faces`.

```
IsVertex(N, p)
```

Return `true` if and only if the point $p = \langle a, b \rangle$ of the rational plane (given as a tuple) is a vertex of the polygon N as returned by `Vertices`.

```
IsInterior(N,p)
```

Return `true` if and only if the point $p = \langle a, b \rangle$ given as a tuple lies strictly in the interior of the polygon N.

```
IsBoundary(N, p)
```

Return `true` if and only if the point $p = \langle a, b \rangle$ given as a tuple lies on the boundary of the polygon N, that is, the point is contained in a face of N.

```
IsPoint(N,p)
```

Return `true` if and only if the point $p = \langle a, b \rangle$ (given as a tuple) lies on the polygon N.

46.3 Polynomials Associated with Newton Polygons

The polynomial used to define a polygon can be recovered, but more usefully so can those restrictions of that polynomial to parts of the polygon, the so-called face functions in particular.

Note that most of these functions will return an error if N was not defined in terms of a polynomial.

```
HasPolynomial(N)
```

Return `true` if and only if the polygon N was defined as the Newton polygon of some polynomial.

```
Polynomial(N)
```

The polynomial used to define the polygon N.

```
ParentRing(N)
```

The parent ring of the polynomial of the polygon N.

```
IsNewtonPolygonOf(N, f)
```

Return whether the newton polygon N is defined by the polynomial f.

```
FaceFunction(F)
```

If the polygon N is defined by a polynomial in two variables f this returns those monomial terms of f whose corresponding Newton points lie on the face F. On the other hand, if N is determined by a univariate polynomial over a series ring, this returns the univariate polynomial supported on the face F.

`IsDegenerate(F)`

Return `true` if the face function along F is not squarefree.

`IsDegenerate(N)`

Return `true` if a face function on some face of N is degenerate.

46.4 Finding Valuations of Roots of Polynomials from Newton Polygons

Newton polygons can be used to find the valuations of roots of the polynomial from which the polygon was created at the prime used in the creation (given implicitly or explicitly). The following functions use Newton polygons to calculate the valuations of the roots of the polynomial paired with the number of roots with that valuation.

`ValuationsOfRoots(f)`

The valuations of the roots of f, where f is a polynomial over a local ring or a series ring.

`ValuationsOfRoots(f, p)`

The valuations of the roots of f with respect to p where p may be either a prime integer, a prime ideal of a number field or a place of a function field.

46.5 Using Newton Polygons to Find Roots of Polynomials over Series Rings

The operations described in this section are relevant for polynomials over series rings. There are two main algorithms involved.

`SetVerbose("Newton", v)`

Set the verbose printing to level v for `PuiseuxExpansion`, `ExpandToPrecision`, `DuvalPuiseuxExpansion`, `Roots` and `ImplicitFunction`. A level of 1 will mean that any partial solutions that could not be expanded to the precision requested will be printed before an error is returned except for `Roots`. The polynomials used in forming extensions will also be printed before the extension is computed. In `Roots`, the algorithm used to compute the expansions will be printed. When Walker's algorithm is being used the current value of the denominator will be printed. For `ImplicitFunction` a warning about a potentially bad value of d will be printed if the value of d given is not divisible by the exponent denominator of some coefficient of f. A level of 2 will print the last polynomials calculated during the newton polygon part of `PuiseuxExpansion` and `DuvalPuiseuxExpansion` and some evaluated polynomials during `ImplicitFunction`.

46.5.1 Operations not associated with Duval's Algorithm

```
PuiseuxExpansion(f, n)
```

`PreciseRoot`	BoolElt	*Default :* `false`
`TestSquarefree`	BoolElt	*Default :* `true`
`NoExtensions`	BoolElt	*Default :* `false`
`LowerFaces`	BoolElt	*Default :* `true`
`OneRoot`	BoolElt	*Default :* `false`
Verbose	`Newton`	*Maximum :* 2

This function implements the algorithm described in [Wal78].

Return a sequence of partial expansions of the roots of the polynomial f over a series ring as puiseux series. The roots are returned with relative precision at least n/d where d is the least common multiple of exponent denominator for the series expansion and the exponent denominators of the coefficients of the f. An input of $n = 0$ will return the expansions calculated by the newton polygon part of the algorithm and these will be to the precision of what is known. The coefficient ring of the series ring containing the coefficients of f must always be a field and unless extensions are not required it must be able to be extended.

If the coefficient ring of the series ring is a finite field whose characteristic is less than or equal to the degree of the polynomial then the denominators computed in the newton polygon part of the algorithm may not be bounded and the function will return an error. However, it is possible for some polynomials that the denominators will be bounded. This is stated by [Gri95], pg 269 – 272.

Care needs to be taken with polynomials whose coefficients have low precision. The algorithm must extract from f the squarefree part and in doing so lose even more precision. The result is that the algorithm may not have enough precision to calculate the expansions correctly. A solution is to set `TestSquarefree` to `false` if the polynomial is known to have no multiple roots. However this will not solve all precision problems and the answer is only as good as the precision allows it to be.

If `PreciseRoot` is set to `true` then the partial expansions to be returned are checked and if any are exact roots of f they are returned with full precision. If `NoExtensions` is set to `true` then expansions are found within the puiseux series ring only. By default, the coefficient ring of the series ring is extended to find all the partial expansions. If `LowerFaces` is set to `false` then expansions with negative valuations will not be found. If `OneRoot` is set to true then representatives of conjugate roots only will be found instead of each of the roots individually.

Note that it may useful to define the polynomial over an algebraically closed field (via `AlgebraicClosure`), so that all roots may be found.

`ExpandToPrecision(f, c, n)`

`PreciseRoot`	BoolElt	*Default :* `false`
`TestSquarefree`	BoolElt	*Default :* `true`
Verbose	`Newton`	*Maximum :* 2

Given a polynomial f over a Puiseux series ring and a partial root c of that polynomial (found by `PuiseuxExpansion` for example) continue to expand that root until it has relative precision n/d where d is the least common multiple of the exponent denominator of c and the exponent denominators of the coefficients of f. If c is given to greater precision (or length greater than n if its precision is infinite), the relative precision of c is reduced to n/d. An error results if c is not a partial expansion to precision n/d. If `PreciseRoot` is true then the partial expansion to be returned is checked and if it is an exact root of f it is returned with full precision. All input is checked for being an exact root regardless. If `TestSquarefree` is `false` the polynomial will not be made squarefree. This may avoid some loss of precision but may result in some unique partial roots not being recognized as unique partial roots and as such they cannot be expanded.

An error may result if c is a partial root of f but the exponent denominator of the full expansion is greater than that of c. Therefore for c to be expanded it must have the same denominator as the expansion it is part of. This will rarely be an issue for those partial expansions resulting from `PusieuxExpansion` which did not encounter problems with precision in the newton polygon part since the algorithm for this will find at least as much of the expansion as necessary to compute the exponent denominator.

`ImplicitFunction(f, d, n)`

Verbose	`Newton`	*Maximum :* 2

Return a root of the polynomial f over a series ring. The input d is the denominator (or a multiple of) the exponent denominator of the root. The root is given to absolute precision n/d. The evaluation of f at zero (polynomial) evaluated at zero (series) must be zero but that of its derivative must be nonzero.

Example H46E6

This example illustrates the joint use of `PuiseuxExpansion` and `ExpandToPrecision` which can be used together to gain and improve partial roots of a polynomial. The use of `ExpandToPrecision` following `PuiseuxExpansion` avoids the recalculation of information already known.

```
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := y^3 + 2*x^-1*y^2 + 1*x^-2*y + 2*x;
> c := PuiseuxExpansion(f, 0);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> Q<q> := PolynomialRing(A);
> c;
```

```
[
    -2*a^3 + O(a^4),
    -a^-1 + n*a + O(a^2),
    -a^-1 - n*a + O(a^2)
]
> [ExpandToPrecision(f, c[i], 10) : i in [1 .. #c]];
[
    -2*a^3 - 8*a^7 - 56*a^11 + O(a^13),
    -a^-1 + n*a + a^3 + 5/4*n*a^5 + 4*a^7 + O(a^9),
    -a^-1 - n*a + a^3 - 5/4*n*a^5 + 4*a^7 + O(a^9)
]
```

The same results could have been gained using `PuiseuxExpansion` with the required precision in the first place.

```
> c := PuiseuxExpansion(f, 10);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> c;
[
    -2*a^3 - 8*a^7 - 56*a^11 + O(a^13),
    -a^-1 + n*a + a^3 + 5/4*n*a^5 + 4*a^7 + O(a^9),
    -a^-1 - n*a + a^3 - 5/4*n*a^5 + 4*a^7 + O(a^9)
]
```

However, asking for more precision requires time so that if it is not necessary the extra calculation can be avoided and if more precision happens to be required then it can be gained without recalculation. `ExpandToPrecision` is also called on only one root so that if only one expansion is required using `PusieuxExpansion` and then `ExpandToPrecision` will not calculate any unnecessary information.

```
> time c := PuiseuxExpansion(f, 100);
Time: 2.810
> time c := PuiseuxExpansion(f, 10);
Time: 0.060
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> time ExpandToPrecision(f, c[1], 100);
-2*a^3 - 8*a^7 - 56*a^11 - 480*a^15 - 4576*a^19 - 46592*a^23 -
    496128*a^27 - 5457408*a^31 - 61529600*a^35 - 707266560*a^39 -
    8257566720*a^43 - 97654702080*a^47 - 1167349284864*a^51 -
    14082308833280*a^55 - 171221451538432*a^59 -
    2096081963188224*a^63 - 25814314231136256*a^67 -
    319605795242639360*a^71 - 3975750610806374400*a^75 -
    49666299938073477120*a^79 - 622818862289639178240*a^83 -
    7837247078959687925760*a^87 - 98931046460491133091840*a^91 -
    1252424949872174982758400*a^95 - 15897106567806080658702336*a^99
    + O(a^103)
```

```
Time: 0.410
```

`IsPartialRoot(f, c)`

Return `true` if the series c can be expanded to at least one root of the polynomial f.

`IsUniquePartialRoot(f, c)`

`TestSquarefree` BoolElt *Default :* `true`

Return `true` if the series c can be expanded to exactly one distinct root of the polynomial f. By default f will have multiple factors removed to allow partial expansions of multiple roots to be recognized as being unique. If `TestSquarefree` is set to `false` then f will be taken as given which may avoid errors due to lost precision but may not pick partial expansions of multiple roots as being unique and as such is best used when f is squarefree or the expansion is known to be of a single root.

Example H46E7

The above 2 functions can be used to reduce the occurrence of errors from `ExpandToPrecision` by checking that the input can be expanded. Errors resulting from a lack of precision which means that the expansion cannot be calculated to the requested precision are the only errors that cannot be removed. Only unique partial roots can be expanded. If a partial root is not unique then calling `PuiseuxExpansion` will provide several further partial expansions of the partial root that will themselves be unique and so can be used to calculate several expansions of the original.

```
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := (y^2 - x^3)^2 - y*x^6;
> IsPartialRoot(f, x^(3/2));
true
>  ExpandToPrecision(f, x^(3/2), 10);
>> ExpandToPrecision(f, x^(3/2), 10);
                    ^

Runtime error in 'ExpandToPrecision': Element is not a unique partial
root of the polynomial
> IsUniquePartialRoot(f, x^(3/2));
false
> c := PuiseuxExpansion(f, 0);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> Q<q> := PolynomialRing(A);
> c;
[
    a^(3/2) + 1/2*a^(9/4) + O(a^(5/2)),
    a^(3/2) - 1/2*a^(9/4) + O(a^(5/2)),
    -a^(3/2) + 1/2*n*a^(9/4) + O(a^(5/2)),
    -a^(3/2) - 1/2*n*a^(9/4) + O(a^(5/2))
]
> IsUniquePartialRoot(f, x^(3/2) + 1/2*x^(9/4));
true
> ExpandToPrecision(f, x^(3/2) + 1/2*x^(9/4), 10);
x^(3/2) + 1/2*x^(9/4) - 1/64*x^(15/4) + O(x^4)
>  ExpandToPrecision(f, x^(3/2) + x^2, 30);
>> ExpandToPrecision(f, x^(3/2) + x^2, 30);
                     ^
Runtime error in 'ExpandToPrecision': Element is not a partial root of
the polynomial
> IsPartialRoot(f, x^(3/2) + x^2);
false
```

So if `IsPartialRoot` returns `false` then no expansion can be made. If `IsUniquePartialRoot` returns `false` (but `IsPartialRoot` returns `true`) then several expansions can be made after calling `PuiseuxExpansion`.

`PuiseuxExponents(p)`

Given a series expansion return the sequence of exponents $[a/b]$ of the non zero terms of the series p up to and including the first one where b is the global denominator for the series.

`PuiseuxExponentsCommon(p, q)`

Given two series return the sequence of exponents $[a/b]$ of the non zero initial terms of the series p and q which are equal up to but not including the first unequal terms.

Example H46E8

This example illustrates how `PuiseuxExponents` and `PuiseuxExponentsCommon` can be used on output from `PusieuxExpansion`. (Similar can be done with related functions and general series).

```
> P<x> := PuiseuxSeriesRing(FiniteField(5, 3));
> R<y> := PolynomialRing(P);
> f := (1+x)*y^4 - x^(-1/3)*y^2 + y + x^(1/2);
> time c := PuiseuxExpansion(f, 5);
Time: 0.030
> c;
[
    4*x^(1/2) + x^(2/3) + 3*x^(5/6) + x^(7/6) + O(x^(4/3)),
    x^(1/3) + x^(1/2) + 4*x^(2/3) + 2*x^(5/6) + O(x^(7/6)),
    4*x^(-1/6) + 2*x^(1/3) + O(x^(2/3)),
    x^(-1/6) + 2*x^(1/3) + O(x^(2/3))
]
> PuiseuxExponents(c[1]);
[ 1/2, 2/3, 5/6 ]
> PuiseuxExponents(c[3]);
```

```
[ -1/6 ]
> P<x> := PuiseuxSeriesRing(FiniteField(5, 3));
> R<y> := PolynomialRing(P);
> f := ((y^2 - x^3)^2 - y*x^6)^2 - y*x^15;
> c := PuiseuxExpansion(f, 0);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> Q<q> := PolynomialRing(A);
> c;
[
    4*a^(3/2) + 4*a^(9/4) + 4*a^3 + O(a^(13/4)),
    4*a^(3/2) + 4*a^(9/4) + a^3 + O(a^(13/4)),
    4*a^(3/2) + a^(9/4) + 4*a^3 + O(a^(13/4)),
    4*a^(3/2) + a^(9/4) + a^3 + O(a^(13/4)),
    a^(3/2) + 3*a^(9/4) + 4*a^3 + O(a^(13/4)),
    a^(3/2) + 3*a^(9/4) + a^3 + O(a^(13/4)),
    a^(3/2) + 2*a^(9/4) + 4*a^3 + O(a^(13/4)),
    a^(3/2) + 2*a^(9/4) + a^3 + O(a^(13/4))
]
> PuiseuxExponentsCommon(c[1], c[1]);
[ 3/2, 9/4, 3 ]
> PuiseuxExponentsCommon(c[1], c[2]);
[ 3/2, 9/4 ]
> PuiseuxExponentsCommon(c[1], c[3]);
[ 3/2 ]
> PuiseuxExponentsCommon(c[1], c[8]);
[]
```

46.5.2 Operations associated with Duval's algorithm

The following functions have a similar use to those given above but implement a different algorithm, namely that of [Duv89] which is faster and can handle larger degree polynomials. However, it can only be used with polynomials which are essentially over a laurent series ring and the coefficient ring of that laurent series ring has either characteristic zero or characteristic greater than the degrees of the squarefree factors of the polynomial.

`DuvalPuiseuxExpansion(f, n)`

`Version`	MonStgElt	*Default :* "*Rational*"
`TestSquarefree`	BoolElt	*Default :* `true`
`NoExtensions`	BoolElt	*Default :* `false`
`LowerFaces`	BoolElt	*Default :* `true`
`OneRoot`	BoolElt	*Default :* `false`
Verbose	`Newton`	*Maximum :* 2

A sequence of parametrizations of puiseux expansions of roots of f, as puiseux series, where f is a polynomial over a series ring. The expansions will have at least n non zero terms (unless the expansion is finite and has less than n non zero terms), with more than n occurring only if n is less than the number of terms returned by the newton polygon part of the algorithm. The coefficients of f must have exponent denominator 1.

This algorithm is faster than that given by Walker and implemented in `PuiseuxExpansion`, since it doesn't calculate non zero terms explicitly and doesn't make all necessary extensions during the algorithm leaving some to be made when the series is computed from the parametrizations.

If f has coefficients with finite precision then the expansions can only be computed to as many non zero terms as can be known for that expansion. After this limit has been reached, an error results since the next non zero term is not known. If f has roots with finite puiseux expansions then if n is greater than the number of non zero terms in the expansion the expansion is returned with infinite precision.

If `Version` is set to `"Classical"` then the (slower) classical branch of the algorithm will be run which makes all extensions necessary for the computation of the expansions. It is still faster than `PusieuxExpansion` since it does not iterate through and calculate zero terms but will encounter the same problems that `PuiseuxExpansion` does with field extensions over the rationals. The classical version will return as many parametrizations as there are expansions and some of these parametrizations will give the same set of expansions.

If `NoExtensions` is set to `true` then only the expansions which lie in the puiseux series ring corresponding to the coefficient ring of f are calculated. Otherwise, all the expansions of roots of f are calculated regardless of where they lie. `LowerFaces` and `OneRoot` work as for `PuiseuxExpansion`.

This algorithm works with the squarefree part of f only. If any coefficient of f has low precision then this step may make it impossible for any information about the expansions to be gained due to a loss of further precision. A way around this is to set `TestSquarefree` to `false` if the polynomial is known to be squarefree. This may result in some information being returned but such information is only as good as the precision it was allowed.

```
ParametrizationToPuiseux(T)
```

The series that satisfy the parametrization `T`. These are found by evaluating `T[2]` at t where `T[1]` $= \lambda t^e$.

```
PuiseuxToParametrization(S)
```

A parametrization of the series S. It is the simplest one which takes the denominator out of S and makes it the exponent of the first entry in the parametrization.

Example H46E9

This example illustrates the use of `DuvalPuiseuxExpansion` and `ParametrizationToPuiseux` to gain the information given by `PuiseuxExpansion` and also compares the performance of the two algorithms. It also highlights some of the anomalies that may be encountered due to precision concerns.

```
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := (y^2 - x^3)^2 - y*x^6;
> time D := DuvalPuiseuxExpansion(f, 0);
Time: 0.000
> D;
[
    <16*x^4, 64*x^6 + 256*x^9 + O(x^10)>
]
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.060
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) + O(a^(5/2)),
    a^(3/2) - 1/2*a^(9/4) + O(a^(5/2)),
    -a^(3/2) + 1/2*n*a^(9/4) + O(a^(5/2)),
    -a^(3/2) - 1/2*n*a^(9/4) + O(a^(5/2))
]
> time c := PuiseuxExpansion(f, 0);
Time: 0.050
```

Here it can be seen that the newton polygon part of the algorithm is substantially faster using Duval's method, though the converting of the parametrization to a series is not as fast. Asking for more terms shows this more substantially.

```
> time D := DuvalPuiseuxExpansion(f, 10);
Time: 0.020
> D;
[
    <16*x^4, 64*x^6 + 256*x^9 - 512*x^15 + 2048*x^18 - 4608*x^21 + 56320*x^27 -
        294912*x^30 + 792064*x^33 - 12082176*x^39 + 68157440*x^42 + O(x^43)>
]
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.129
> time c := PuiseuxExpansion(f, 10);
Time: 0.100
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> c;
[
    a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + O(a^4),
    a^(3/2) - 1/2*a^(9/4) + 1/64*a^(15/4) + O(a^4),
    -a^(3/2) + 1/2*n*a^(9/4) + 1/64*n*a^(15/4) + O(a^4),
    -a^(3/2) - 1/2*n*a^(9/4) - 1/64*n*a^(15/4) + O(a^4)
]
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + 1/128*a^(9/2) - 9/4096*a^(21/4) +
        55/131072*a^(27/4) - 9/32768*a^(15/2) + 1547/16777216*a^(33/4) -
        11799/536870912*a^(39/4) + 65/4194304*a^(21/2) + O(a^(43/4)),
    a^(3/2) - 1/2*a^(9/4) + 1/64*a^(15/4) + 1/128*a^(9/2) + 9/4096*a^(21/4) -
        55/131072*a^(27/4) - 9/32768*a^(15/2) - 1547/16777216*a^(33/4) +
        11799/536870912*a^(39/4) + 65/4194304*a^(21/2) + O(a^(43/4)),
    -a^(3/2) + 1/2*n*a^(9/4) + 1/64*n*a^(15/4) - 1/128*a^(9/2) -
        9/4096*n*a^(21/4) - 55/131072*n*a^(27/4) + 9/32768*a^(15/2) +
        1547/16777216*n*a^(33/4) + 11799/536870912*n*a^(39/4) -
        65/4194304*a^(21/2) + O(a^(43/4)),
    -a^(3/2) - 1/2*n*a^(9/4) - 1/64*n*a^(15/4) - 1/128*a^(9/2) +
        9/4096*n*a^(21/4) + 55/131072*n*a^(27/4) + 9/32768*a^(15/2) -
        1547/16777216*n*a^(33/4) - 11799/536870912*n*a^(39/4) -
        65/4194304*a^(21/2) + O(a^(43/4))
]
```

It can be seen that the computation of the information is a lot faster using Duval's method. It is only the cosmetic of converting this information into series that could make this algorithm seem slow. But also note that there is much greater information given by Duval's algorithm. The equivalent information is given below.

```
> time D := DuvalPuiseuxExpansion(f, 3);
Time: 0.009
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.049
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + O(a^4),
    a^(3/2) - 1/2*a^(9/4) + 1/64*a^(15/4) + O(a^4),
    -a^(3/2) + 1/2*n*a^(9/4) + 1/64*n*a^(15/4) + O(a^4),
    -a^(3/2) - 1/2*n*a^(9/4) - 1/64*n*a^(15/4) + O(a^4)
]
```

One thing that may be taken for granted from `PuiseuxExpansion` is that all the expansions lie in the same puiseux series ring. However, for `DuvalPuiseuxExpansion` this may not be the case. It will always be true that each of the parametrizations will lie in the same puiseux series ring but series resulting from different parametrizations may not. This occurs since some extensions are left to the stage of calculating the series from the parametrization to be made and for different parametrizations these extensions may be different.

```
> f := (-x^3 + x^4) - 2*x^2*y - x*y^2 + 2*x*y^4 + y^5;
> time D := DuvalPuiseuxExpansion(f, 0);
Time: 0.010
```

```
> D;
[
    <x^2, -x^2 - x^3 + O(x^4)>,
    <x^3, x + O(x^2)>
]
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.000
> P;
[
    -x - x^(3/2) + O(x^2),
    -x + x^(3/2) + O(x^2)
]
> Parent(P[1]);
Puiseux series field in x over Rational Field
> time P := ParametrizationToPuiseux(D[2]);
Time: 0.030
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(1/3) + O(a^(2/3)),
    n*a^(1/3) + O(a^(2/3)),
    (-n - 1)*a^(1/3) + O(a^(2/3))
]
> Parent(P[1]);
Puiseux series field in a over N
> N;
Number Field with defining polynomial $.1^2 + $.1 + 1 over the Rational Field
```

`DuvalPuiseuxExpansion` reacts differently to `PuiseuxExpansion` when given input which has finite expansions either due to finite precision or exact roots. These differences are shown below and are due to the fact that `DuvalPuiseuxExpansion` always looks for the next non zero term in an expansion whereas `PuiseuxExpansion` will calculate zero terms.

```
> f := y - x^3 - x^7 - x^76 + O(x^200);
> D := DuvalPuiseuxExpansion(f, 0);
> D;
[
    <x, x^3 + O(x^4)>
]
> D := DuvalPuiseuxExpansion(f, 3);
> D;
[
    <x, x^3 + x^7 + x^76 + O(x^77)>
]
>  D := DuvalPuiseuxExpansion(f, 4);
>> D := DuvalPuiseuxExpansion(f, 4);
                            ^

Runtime error in 'DuvalPuiseuxExpansion': Insufficient precision to calculate to
```

```
requested precision
> c := PuiseuxExpansion(f, 197);
> c;
[
    x^3 + x^7 + x^76 + O(x^200)
]
>  c := PuiseuxExpansion(f, 200);
>> c := PuiseuxExpansion(f, 200);
                        ^
Runtime error in 'PuiseuxExpansion': Insufficient precision to calculate to
requested precision
> f := y - x^3 - x^7 - x^76;
> D := DuvalPuiseuxExpansion(f, 0);
> D;
[
    <x, x^3 + O(x^4)>
]
> D := DuvalPuiseuxExpansion(f, 3);
> D;
[
    <x, x^3 + x^7 + x^76 + O(x^77)>
]
> D := DuvalPuiseuxExpansion(f, 4);
> D;
[
    <x, x^3 + x^7 + x^76>
]
> c := PuiseuxExpansion(f, 10);
> c;
[
    x^3 + x^7 + O(x^13)
]
> c := PuiseuxExpansion(f, 100);
> c;
[
    x^3 + x^7 + x^76 + O(x^103)
]
> c := PuiseuxExpansion(f, 200);
> c;
[
    x^3 + x^7 + x^76 + O(x^203)
]
> c := PuiseuxExpansion(f, 200 : PreciseRoot := true);
> c;
[
    x^3 + x^7 + x^76
]
```

The two methods can be combined. Given a series there is no way that Duval's REGULAR algorithm can be used to further the precision of an expansion. But `ExpandToPrecision` can be used to gain the extra precision. Using the REGULAR algorithm would be preferable since it is faster but this is not possible for the type of input that is available. This is explored in the case of our first example.

```
> f := (y^2 - x^3)^2 - y*x^6;
> time D := DuvalPuiseuxExpansion(f, 0);
Time: 0.009
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.050
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) + O(a^(5/2)),
    a^(3/2) - 1/2*a^(9/4) + O(a^(5/2)),
    -a^(3/2) + 1/2*n*a^(9/4) + O(a^(5/2)),
    -a^(3/2) - 1/2*n*a^(9/4) + O(a^(5/2))
]
> time ExpandToPrecision(f, P[1], 20);
a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + 1/128*a^(9/2) - 9/4096*a^(21/4) +
    O(a^(13/2))
Time: 0.070
> time D := DuvalPuiseuxExpansion(f, 5);
Time: 0.010
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.059
```

It can be seen that using `ExpandToPrecision` is slower even than rerunning Duval's algorithm from the beginning. Even more so when it is remembered that running Duval's algorithm with the extra precision will give parametrizations of all the expansions of the roots of f and not just one expansion. This seems to still be the case when larger examples are considered so that if more terms of an expansion are required it is probably best to start from the beginning asking for these extra terms.

```
> time p1 := ExpandToPrecision(f, P[1], 50);
> A<a> := Parent(p1);
> N<n> := CoefficientRing(A);
> p1;
a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + 1/128*a^(9/2) - 9/4096*a^(21/4) +
    55/131072*a^(27/4) - 9/32768*a^(15/2) + 1547/16777216*a^(33/4) -
    11799/536870912*a^(39/4) + 65/4194304*a^(21/2) - 189805/34359738368*a^(45/4)
    + 1584999/1099511627776*a^(51/4) - 2261/2147483648*a^(27/2) + O(a^14)
Time: 0.439
> time D := DuvalPuiseuxExpansion(f, 13);
Time: 0.009
> time P := ParametrizationToPuiseux(D[1]);
```

```
Time: 0.200
```

46.5.3 Roots of Polynomials

This section describes two similar functions that can be used for finding roots of polynomials over series rings in a similar way to finding roots of polynomials over any other ring for which roots can be computed in Magma.

`Roots(f)`

`Roots(f, n)`

Verbose `Newton` *Maximum :* 2

Find the roots of the polynomial f which lie in the coefficient ring of f. The first form of this function can be used on any polynomial over any ring for which Magma can compute roots. Since precision is an issue in series rings and some roots may have infinite expansions, the second version of this function which is specific to series rings allows a lower bound for the precision to which these roots will be known to be specified. The first computes the roots to at least the default precision of the ring if that ring has infinite precision otherwise it computes them to at least the precision of the ring. The first version will be enough when all the coefficients of the polynomial have infinite precision. The second version may be required when a precision other than that assumed by the first is sought which may be due to the impossibility of computing the roots to such a high precision as the default. The precision is relative to the least common multiple of the exponent denominators of the coefficients of f and the exponent denominator of the root. Roots which are known to be different but are identical to the precision specified will be returned as two distinct roots.

Duval's algorithm as implemented in `DuvalPuiseuxExpansion` will usually be used. Walker's algorithm as implemented in `PuiseuxExpansion` will be used if the polynomial has coefficients involving fractional powers or the characteristic of the coefficient ring of the series ring is less than the degree of a squarefree factor.

If Walker's algorithm is used and the characteristic of the field is less than the degree of the polynomial then the computation may not finish (see remarks under `PuiseuxExpansion`) and control will return to the user when interrupted.

If verbose printing of partial output that doesn't have enough precision is required the functions `PuiseuxExpansion` and `DuvalPuiseuxExpansion` should be used with the appropriate precision. `Roots` also requires that there is enough precision in the roots so that the multiplicities can be calculated correctly and the parts of the roots that are returned are distinct. Therefore an error will be given if there is not enough precision to calculate the part of the root that results from the newton polygon part of the algorithm and the root is not known to be a single root since the multiplicity may not be able to be calculated correctly. Information at such a low precision can be gained correctly by using the `PuiseuxExpansion` functions and determining the multiplicities manually.

```
HasRoot(f)
```

Return `true` if the polynomial f has a root in its coefficient ring and that root can be found to the fixed or default precision of the ring as applicable. A root is also returned in this case. If f is irreducible over its coefficient ring then return `false`.

Example H46E10

Below are some examples of the use of the `Roots` function.

```
> SetVerbose("Newton", 1);
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := y^3 + 2*x^-1*y^2 + 1*x^-2*y + 2*x;
> Roots(f);
DUVAL :
[
    <-2*x^3 - 8*x^7 - 56*x^11 - 480*x^15 - 4576*x^19 + O(x^23), 1>
]
> f := f^2;
> Roots(f);
DUVAL :
[
   <-2*x^3 - 8*x^7 - 56*x^11 - 480*x^15 - 4576*x^19 + O(x^23), 2>
]
> f := y^3 + 2*x^-1*y^2 + 1*x^-2*y + 2*x;
> f +:= O(x^20)*(y^3 + y^2 + y + 1);
> f;
(1 + O(x^20))*y^3 + (2*x^-1 + O(x^20))*y^2 + (x^-2 + O(x^20))*y + 2*x + O(x^20)
>  Roots(f);
DUVAL :
>> Roots(f);
        ^
Runtime error in 'Roots': Roots not calculable to default precision
> Roots(f, 10);
DUVAL :
[
   <-2*x^3 - 8*x^7 - 56*x^11 + O(x^13), 1>
]
> f := f^2;
> f;
(1 + O(x^20))*y^6 + (4*x^-1 + O(x^19))*y^5 + (6*x^-2 + O(x^18))*y^4 + (4*x^-3 +
   4*x + O(x^18))*y^3 + (x^-4 + 8 + O(x^18))*y^2 + (4*x^-1 + O(x^18))*y + 4*x^2
   + O(x^21)
> Roots(f, 10);
DUVAL :
[
   <-2*x^3 - 8*x^7 - 56*x^11 + O(x^13), 2>
]
```

```
> f := (y - x^(1/4))*(y - x^(1/3));
> Roots(f);
WALKER :
[
    <x^(1/3) + O(x^2), 1>,
    <x^(1/4) + O(x^(23/12)), 1>
]
```

46.6 Bibliography

[**Duv89**] Dominique Duval. Rational Puiseux Expansions. *Compositio Mathematica*, 70:119 – 154, 1989.

[**Gri95**] Deryn Griffiths. Series Expansions of Algebraic Functions. In W. Bosma and A. van der Poorten, editors, *Computational Algebra and Number Theory*, pages 267 – 277. Kluwer Academic Publishers, Netherlands, 1995.

[**Wal78**] Robert J. Walker. *Algebraic Curves*, pages 98 – 99. Springer-Verlag, 1978.

47 *p*-ADIC RINGS AND THEIR EXTENSIONS

Chapter 47

p-ADIC RINGS AND THEIR EXTENSIONS

47.1 Introduction

Magma supports finite extensions of the ring $\mathbf{Z}_p$ of p-adic integers or the field $\mathbf{Q}_p$ of p-adic numbers. Within this chapter, we mean these objects if we refer to local rings or fields. Section 47.2 provides more background information on the theory behind the p-adics.

Magma has two different models for working with these locals: fixed precision rings (`RngPadRes` and `RngPadResExt`, with element types `RngPadResElt` and `RngPadResExtElt`) and free precision rings (`RngPad` and `FldPad`, with element types `RngPadElt` and `FldPadElt`). The merits of each model are discussed in Section 47.3.1.

Magma also contains a type of local field where extensions can be made by any irreducible polynomial. For more information on these local fields, see Chapter 51.

47.2 Background

The p-adic field $\mathbf{Q}_p$ arises naturally as the completion of $\mathbf{Q}$ with respect to an absolute value function $|x|_p = p^{-v_p(x)}$, where $v_p(x)$ is the p-adic *valuation* of x (that is, a power of p such that $x = p^{v_p(x)}\frac{a}{b}$ but $p \not| ab$). The *ring of integers* of $\mathbf{Q}_p$, denoted $\mathbf{Z}_p$, is the set of all elements of non-negative valuation. The ring $\mathbf{Z}_p$ has a unique maximal ideal, generated by the prime p; the *residue class field* K is the quotient $\mathbf{Z}_p/p\mathbf{Z}_p$. Any element x of $\mathbf{Q}_p$ can be expressed as a power series in the prime p, so that $x = \sum_{i=v}^{\infty} a_i p^i$, where v is the valuation of x, a_v is non-zero, and each a_i is a lift of an element from the residue class field. In more general terms, $\mathbf{Q}_p$ is a local field, with its ring of integers $\mathbf{Z}_p$ being a local ring. A uniformizer π of $\mathbf{Q}_p$ is the prime p.

More generally, consider an irreducible polynomial over some local field L_1 (such as $\mathbf{Q}_p$). Then the extension given by adjoining a root α of this polynomial to L_1, $L_2 = L_1[\alpha]$, is also a local field. Let π_1 and π_2 be uniformizers of L_1 and L_2, respectively. Then $\pi_2^e = \pi_1 u$, where u is a unit of L_2. The number e is the *ramification degree* of L_2 over L_1, and divides the degree n of the extension. If $e = n$, we say L_2 is *totally ramified* over L_1; if $e = 1$, we say L_2 is *unramified* over L_1. The degree of the residue class field K_2 of L_2 over the residue class field K_1 of L_1 is $f = \frac{n}{e}$. Finite extensions in Magma must be either *unramified* or *totally ramified*; Magma also allows towers of extensions to be built.

It is well known that up to isomorphism there is only one degree n unramified extension of L, which can be obtained by adjoining a $p^n - 1$-th root of unity ζ to L. This extension is Galois, with Galois group isomorphic to the cyclic group of order n. The Galois group is generated by the Frobenius automorphism σ, which takes ζ to ζ^p. For some applications, it is necessary to have a fast Frobenius action, so Magma supports such a representation. However, the defining polynomial in this representation is particularly dense and can be expensive to construct, hence it is ill-suited for general applications. An unramified extension can only be defined by *inertial* polynomials, which are polynomials that are irreducible over the residue class field. Magma allows an unramified extension to be defined by any inertial polynomial.

All totally ramified extensions contain a root of an Eisenstein polynomial of L. An Eisenstein polynomial $f(x) = \sum_{i=0}^{n} a_i x^i$ satisfies $v_\pi(a_n) = 0$, $v_\pi(a_i) \geq 1$ for all $0 < i < n$, and $v_\pi(a_0) = 1$. Magma allows a totally ramified extension to be defined by any Eisenstein polynomial; note that this does not allow arbitrary representations of totally ramified extensions to be constructed.

47.3 Overview of the p-adics in Magma

47.3.1 p-adic Rings

Since p-adic rings are completions (like the real numbers), it is difficult to represent their elements in an exact form. As described in Section 47.2, any element x of a local ring L can be expressed as a power series in the uniformizing element of L. Most problems can be solved by computing in a finite quotient of L, which is equivalent to truncating the infinite expansion of x at some point. More precisely, we work in the quotient rings $L/\pi^k L$ for non-negative k, which is called the *precision* of the ring L.

Unfortunately, working with finite approximations to elements does have problems. Some operations on these approximations will yield results with reduced precision. For example, in a ring constructed with precision k the quotient of two elements with valuations $v_1 \geq v_2$ lies in the local ring, but can only be determined up to $k - v_2$ digits. Also, it is only possible in general to construct an approximation to the GCD of two polynomials f and g over a local ring.

Magma offers two models for computing in a local ring L. The first model (the *fixed precision model*) allows the user to work with the quotient rings defined above. These quotient rings are finite structures, and their elements can easily be represented exactly; hence, many exact algorithms can be applied to them. On the other hand, precision management is left to the user, since operations which may lose precision, such as division, will still return results to the full precision of the ring. In the second model (the *free precision model*), the user works more directly with L, instead of a finite quotient of L. Each element is still represented by a finite approximation, however, these approximations can be of varying precisions, thus freeing the user from precision management. However, these rings are inexact (for instance, there is no zero element of the ring, only approximations to it), and hence many algorithms in Magma which are designed for exact rings may cause significant precision loss. The situation is analogous to the support for real numbers in Magma.

47.3.2 p-adic Fields

A local field in Magma is the field of fractions of a local ring. Representing a local field on a machine causes some trouble, since there is no algebraic structure in which truncations of the elements could be interpreted (as in the case of the finite quotients for the local rings). Moreover, all structural information of a local field is already contained in its ring of integers, so that the main motivation for supporting local fields is to provide some basic arithmetic.

An element x of a local field L is stored internally as $x = \pi^v u + O(\pi^{v+k})$, where v is the (possibly negative) valuation of x, and u is a unit known to precision k in the ring of integers of L. It can happen that operations in a field yield elements with no known digits at all, since the unit parts may not overlap.

47.3.3 Free Precision Rings and Fields

Free precision rings and fields can be created with bounded or unbounded precision. If they are created with some bounded precision k, then no element in the structure can have precision greater than k. Thus, bounded free precision rings are "inexact" versions of the fixed precision rings. Unbounded precision structures can have elements with arbitrarily large, but not infinite, precision. In general, it is recommended that free precision structures are utilized unless speed is absolutely critical.

47.3.4 Precision of Extensions

In Magma, all measurements of precision are made with respect to a valuation function v_π which takes the uniformizer π of the local ring L to unity (throughout this chapter, the subscript will be dropped on v_π where there is no ambiguity). This can have surprising results. For instance, consider a local ring L_1 which extends L_2, such that the ramification degree of L_1 over L_2 is e. Let π_1 and π_2 be the uniformizing elements of L_1 and L_2, respectively. Then, $v_{\pi_1}(\pi_1) = 1$, and $v_{\pi_1}(\pi_2) = e$. Hence, an element of precision k in L_2 will have precision ek in when mapped into L_1. Thus, it is important to always remember in which ring precision is determined.

47.4 Creation of Local Rings and Fields

A local ring in Magma can be constructed in two ways: as either a p-adic ring, or as an extension of another local ring. Magma supports the construction of towers of extensions of local rings; the only restriction is that each extension must be either unramified or totally ramified. As discussed in Section 47.2, Magma requires that the defining polynomial is either inertial or Eisenstein.

Additionally, the user must specify whether to construct a fixed precision or free precision structure, and, if necessary, assign a precision to the structure. For local rings the precision is interpreted as an absolute precision, specifying to what precision the element is known, but for local fields it is interpreted as a relative precision, specifying to what precision the unit part of the element is known.

47.4.1 Creation Functions for the p-adics

```
pAdicRing(p, k)
```

```
pAdicField(p, k)
```

Given a prime integer p and non-negative single-precision integer k, construct the bounded free precision ring (field) of p-adic integers with maximum precision k.

```
pAdicRing(p)
```

```
pAdicField(p)
```

Precision RngIntElt *Default :* 20

Given a prime integer p, construct the unbounded free precision ring (field) of p-adic numbers. The optional parameter `Precision`, which must be a non-negative single precision integer, controls the default precision to which elements are created, e.g., when coercing precise elements such as integers or rationals into the ring.

```
pAdicQuotientRing(p, k)
```

Given a prime integer p and non-negative single precision integer k, construct the fixed precision quotient ring $\mathbf{Z}_p/p^k\mathbf{Z}_p$.

```
quo< L | x >
```

Given a local ring L, construct the quotient ring L/xL, where x is an element of L.

Example H47E1

The creation of p-adic rings using the above functions is illustrated below.

```
> R := pAdicRing(5);
> R;
5-adic ring
> R`DefaultPrecision;
20
> R!1;
1 + O(5^20)
> R := pAdicRing(5 : Precision := 20);
> R!1;
1 + O(5^20)
> Q := quo<R | 5^20>;
> Q;
Quotient of the 5-adic ring modulo the ideal generated by 5^20
> Q!1;
1
> Q eq pAdicQuotientRing(5, 20);
true
```

47.4.2 Creation Functions for Unramified Extensions

`UnramifiedExtension(L, n)`

`ext< L | n >`

`Cyclotomic` BoolElt *Default :* `false`

`GNBType` RngIntElt *Default :* 0

Given a local ring or field L and a positive single precision integer n, construct the default unramified extension of L of degree n. If K is the residue class field of L, then the defining polynomial of the default degree n extension of K is lifted to be an inertial polynomial of L; this polynomial is used as the defining polynomial of the extension. If `Cyclotomic` is `true`, then the lift of the defining polynomial will be such that $p^n - 1$-st root of unity will be adjoined to L (this representation makes computation of the Frobenius automorphism particularly efficient). The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `K<t> := UnramifiedExtension(L, n)`. If `Cyclotomic` is `false` but `GNBType` is $t > 0$ then a Gaussian normal basis of type t is used. This allows extremely fast multiple Frobenius computations but multiplication is slower than the usual or `Cyclotomic` representation if $t > 2$. Because of this, currently only 1 or 2 are legal values for t. Only certain extension degrees will have a Gaussian normal basis of type 1 or 2. To determine if this is true, the `HasGNB` functions described below may be used.

`UnramifiedQuotientRing(K, k)`

Given a finite field K and a non-negative single precision integer k, construct the fixed precision quotient ring which has residue class field K and precision k. The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `L<t> := UnramifiedExtension(K, f)`.

`UnramifiedExtension(L, f)`

`ext< L | f >`

Given a local ring or field L and a polynomial f with coefficients coercible to L, construct the unramified extension of L defined by f. The polynomial f must be an inertial polynomial over L. The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `K<t> := UnramifiedExtension(L, f)`. Free precision rings can only be extended by a polynomial if they are of bounded precision, in which case f must be specified to the maximum precision of the ring.

`IsInertial(f)`

Given a polynomial f with coefficients over a local ring or field L, return `true` if and only if f is an inertial polynomial. A polynomial is inertial over L if it is irreducible over the residue class field of L.

`HasGNB(R, n, t)`

Given a local ring or field, returns `true` iff the unramified extension of degree n can be generated by a Gaussian Normal Basis (GNB) of Type t. A GNB allows particularly fast multiple Frobenius and Norm computations. Multiplication will tend to be slower though, unless $t = 1$.

`CyclotomicUnramifiedExtension(R, f)`

`CyclotomicUnramifiedExtension(R, f)`

`CyclotomicUnramifiedExtension(R, f)`

`CyclotomicUnramifiedExtension(R, f)`

Given a local ring of field R, construct the unramified degree f extension by adjoining a $p^f - 1$-th root of unity to R. Functionally equivalent to calling `UnramifiedExtension(R, f:Cyclotomic := true)`.

Example H47E2

The creation of unramified extensions of local rings using the above functions is illustrated below.

```
> R1 := pAdicRing(2, 20);
> R2 := ext<R1 | 5>;
> R2;
Unramified extension defined by the polynomial x^5 + x^2 + 1
 over 2-adic ring mod 2^20
> DefiningPolynomial(R2);
x^5 + x^2 + 1
> R3 := ext<R1 | 5 : Cyclotomic>;
> R3;
Cyclotomic unramified extension of degree 5 over 2-adic ring mod 2^20
> DefiningPolynomial(R3);
x^5 + 426248*x^4 - 14172*x^3 - 147105*x^2 + 293314*x - 1
> R3.1^(2^5-1);
1
> P1<x> := PolynomialRing(R1);
> f1 := x^3 + 3*x + 1;
> IsInertial(f1);
true
> R4 := ext<R1 | f1>;
> R4;
Unramified extension defined by the polynomial x^3 + 3*x + 1
 over 2-adic ring mod 2^20
> P2<y> := PolynomialRing(R2);
> f2 := y^3 + 3*y + 1;
> IsInertial(f2);
true
> ext<R2 | f2>;
Unramified extension defined by the polynomial x^3 + 3*x + 1
```

```
 over Unramified extension defined by the polynomial x^5 + x^2 + 1
 over 2-adic ring mod 2^20
```

47.4.3 Creation Functions for Totally Ramified Extensions

`TotallyRamifiedExtension(L, f)`

`ext< L | f >`

Given a local ring or field L and a polynomial f with coefficients coercible to L, construct the totally ramified extension of L defined by f. The polynomial f must be an Eisenstein polynomial, that is, the leading coefficient is a unit, the constant coefficient has valuation 1 and all other coefficients have valuation greater than or equal to 1. The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `K<t> := TotallyRamifiedExtension(L, f)`. Free precision rings can only be extended by a polynomial if they are of bounded precision, in which case f must be specified to the maximum precision of the ring.

`IsEisenstein(f)`

Given a polynomial f with coefficients over a local ring or field L, return `true` if and only if f is an Eisenstein polynomial over L. An Eisenstein polynomial satisfies the following properties: the leading coefficient is a unit, the constant coefficient has valuation 1 and all other coefficients have valuation greater than or equal to 1.

Example H47E3

The creation of totally ramified extensions of local rings using the above functions is illustrated below.

```
> L1<a> := ext<pAdicRing(5, 20) | 4>;
> L1;
Unramified extension defined by the polynomial x^4 + 4*x^2 + 4*x + 2
 over 5-adic ring mod 5^20
> L2<b> := ext<L1 | x^4 + 125*x^2 + 5>;
> L2;
Totally ramified extension defined by the polynomial x^4 + 125*x^2 + 5
 over Unramified extension defined by the polynomial x^4 + 4*x^2 + 4*x + 2
 over 5-adic ring mod 5^20
> P<y> := PolynomialRing(L2);
> L3<c> := TotallyRamifiedExtension(L2, y^3 + b^4*a^5*y + b*a^2);
> L3;
Totally ramified extension defined by the polynomial x^3 + ((500*a^3 + 500*a^2 +
    250*a)*b^2 + 20*a^3 + 20*a^2 + 10*a)*x + a^2*b
 over Totally ramified extension defined by the polynomial x^4 + 125*x^2 + 5
 over Unramified extension defined by the polynomial x^4 + 4*x^2 + 4*x + 2
 over 5-adic ring mod 5^20
```

If the precision of the base ring is only 1, then it is not possible to construct a ramified extension, as there is not enough precision to allow the constant coefficient to be non-zero to that precision.

```
> R<x> := PolynomialRing(Integers());
> L<a> := UnramifiedExtension(pAdicRing(5, 1), 3);
>  TotallyRamifiedExtension(L, x^4 + 5);
>> TotallyRamifiedExtension(L, x^4 + 5);
                           ^

Runtime error in 'TotallyRamifiedExtension': Polynomial must be Eisenstein
> L<a> := UnramifiedExtension(pAdicRing(5, 2), x^5 + x^2 + 2);
> TotallyRamifiedExtension(L, x^4 + 5);
Totally ramified extension defined by the polynomial x^4 + 5 over Unramified
extension defined by the polynomial x^5 + x^2 + 2 over 5-adic ring mod 5^2
> ext<L | x^4 + 125*x^2 + 5>;
Totally ramified extension defined by the polynomial x^4 + 5 over Unramified
extension defined by the polynomial x^5 + x^2 + 2 over 5-adic ring mod 5^2
```

47.4.4 Creation Functions for Unbounded Precision Extensions

Suppose we have an unbounded precision local ring or field L, and we wish to create a finite extension of it. If we need the default degree n unramified extension, then we can use the construction functions defined in Section 47.4.2 to construct this extension. However, suppose we wish to define the extension by some polynomial f. As there is no upper bound on the precision of elements of L, it is impossible for us to represent the polynomial f sufficiently precisely, and hence we cannot use the creation functions defined in previous sections for this task. To allow such extensions to be created, Magma allows extensions to be defined by a map $\phi : \mathbf{Z}_{\geq 0} \to R[x]$, where R is a ring whose elements are coercible to the quotient rings $L/\pi^k L$ for all $k \in \mathbf{Z}_{\geq 0}$. The map ϕ, given an input precision k, returns the defining polynomial of the extension to precision k. Internally, whenever Magma needs to represent an element of the extension to some precision, it will use ϕ to compute the defining polynomial up to this precision. Magma may call ϕ on any precision between zero and the precision of the most precise element created by the user.

```
ext< L | m >
```

Given a free precision local ring or field L and a map m with domain $\mathbf{Z}$ and codomain $R[x]$, where elements of R are coercible to the quotient rings $L/\pi^k L$ for all $k \in \mathbf{Z}_{\geq 0}$, construct an extension of L defined by m. Given a non-negative single precision integer k, the map m must return the defining polynomial of the extension to precision k, as a polynomial over R. The map m's behaviour for other input values is undefined. Internally, Magma will coerce the value returned by the map m to be a polynomial over $L/\pi^k L$. Examples of suitable codomains R include the integers, rationals, or L itself.

Example H47E4

The creation of extensions of local rings using maps is illustrated below. We show how it is possible to define an extension of a free precision ring using an "exact" polynomial.

```
> R := pAdicRing(2);
> Z := Integers();
> P<x> := PolynomialRing(Z);
> m := map<Z -> P | k :-> x^3 + x + 1>;
> R2 := ext<R | m>;
> R2;
Unramified extension defined by a map over 2-adic ring
> DefiningPolynomial(R2);
(1 + O(2^20))*$.1^3 + O(2^20)*$.1^2 + (1 + O(2^20))*$.1 + 1 + O(2^20)
> R2`DefaultPrecision := 1000;
> DefiningPolynomial(R2);
(1 + O(2^1000))*$.1^3 + O(2^1000)*$.1^2 + (1 + O(2^1000))*$.1 + 1 + O(2^1000)
```

47.4.5 Miscellaneous Creation Functions

`IntegerRing(F)`

`Integers(F)`

`RingOfIntegers(F)`

Given a local field F, construct the ring of integers R of F. The ring R is the set of elements of F of non-negative valuation.

`RingOfIntegers(R)`

Given a ring R, this function simply returns it, it is provided to support generic functionality for finite extensions of rings and fields.

`FieldOfFractions(R)`

Given a local ring R, construct the field of fractions F of R. The relative precision of F is equal to the precision of R.

`SplittingField(f, R)`

Given a polynomial f over the integers and a p-adic ring R, compute an extension S over R such that f splits into linear factors over S. The algorithms uses the R4-methods as developed by Pauli ([Pau01]).

`AbsoluteTotallyRamifiedExtension(R)`

Given a tower of ramified extensions over some unramified ring S, compute a more efficient representation of R, ie. an extension of S that is totally ramified and defined by a single Eisenstein polynomial. The map returned allows to convert between the new and old representations.

47.4.6 Other Elementary Constructions

`Composite(R, S)`

For two p-adic fields that are normal over $\mathbf{Q}_p$, compute the compositum of R and S, ie. the smallest field containing both R and S.

47.4.7 Attributes of Local Rings and Fields

There is only one attribute for local rings and fields that is accessible to the user.

`` L`DefaultPrecision ``

Used to retrieve or set the default precision of the local ring or field L. This attribute is only relevant if L is an unbounded free precision ring, in which case this will change the precision with which elements are created by default. For bounded precision structures, the default precision of the ring is equal to the upper bound on precision; attempting to set this attribute will result in an error in this case.

47.5 Elementary Invariants

These functions return some simple information partially defining a local ring.

`Prime(L)`

Given a local ring or field L, return the prime p defining the p-adic ring or field underlying L. This is also the characteristic of the residue class field of L.

`InertiaDegree(L)`

Return the inertia degree of the local ring or field L over its coefficient ring.

`InertiaDegree(K, L)`

Return the inertia degree of the local ring or field K relative to its subring L.

`AbsoluteInertiaDegree(L)`

Return the inertia degree of the local ring or field L over the p-adic ring.

`RamificationDegree(L)`

`RamificationIndex(L)`

Return the ramification degree of the local ring or field L over its coefficient ring.

`RamificationDegree(K, L)`

`RamificationIndex(K, L)`

Return the ramification degree of the local ring or field K relative to its subring L.

`AbsoluteRamificationDegree(L)`

`AbsoluteRamificationIndex(L)`

Return the ramification degree of the local ring or field L over the p-adic ring.

`AbsoluteDegree(L)`

The degree of L over $\mathbf{Z}_p$.

`Degree(L)`

Return the degree of the local ring or field L over its coefficient ring.

`Degree(K, L)`

Return the degree of the local ring or field K relative to its subring L.

`DefiningPolynomial(L)`

Return the minimal polynomial of the generator of L over its coefficient ring. If L is p-adic, the polynomial $x - 1$ is returned. For free precision rings and fields, the coefficients of the defining polynomial are given to the default precision of L.

`DefiningMap(L)`

Given a free precision local ring or field L, return the map that was used to define the extension (see Section 47.4.4 for information on defining extension by maps). If a map was not used, then an error is raised.

`HasDefiningMap(L)`

Given a free precision local ring or field L, return `true` if L is defined by a map; if so, the defining map is also returned.

`PrimeRing(L)`

`PrimeField(L)`

`pAdicRing(L)`

`pAdicField(L)`

Given a local ring or field L, return the p-adic ring or field which is a subring of L.

`BaseRing(L)`

`CoefficientRing(L)`

`BaseField(L)`

`CoefficientField(L)`

`BaseRing(L)`

Given a local ring or field L, return the base ring of L.

```
ResidueClassField(L)
```

Given a local ring or field L, return the residue class field K of L, and a map from L to K.

```
ResidueSystem(R)
```

Given a p-adic ring or field R, compute a set of representatives of the residue class field of R as elements of R.

```
UniformizingElement(L)
```

Given a local ring or field L, return the uniformizing element of L.

```
L . 1
```

Given a local ring or field L, return an element α of L such that if K is L's base ring or field, then the powers of α give a basis of L as a vector space over K.

```
Precision(L)
```

Given a local ring or field L, return the precision with which L has been created. If L is a local ring this is the maximum absolute precision to which its elements can be created. If L is a local field this is the maximum relative precision to which its elements can be created. If L is an unbounded free precision ring or field, then infinity is returned.

```
HasPRoot(R)
```

Given a local ring R extending $\mathbf{Z}_p$ for some prime p, decide if R contains a primitive p-th root of unity.

```
HasRootOfUnity(L, n)
```

Given a local ring L and some positive integer n, decide if L contains a primitive nth root of unity.

```
Discriminant(R)
```

Compute the discriminant of the local ring R over its coefficient ring. Since R is defined by either an inertial polynomial or an Eisenstein one, this is equivalent to computing the discriminant of the defining polynomial.

```
Discriminant(K, k)
```

Given p-adic rings K/k, compute the discriminant of K as an extension of k.

```
AdditiveGroup(R)
```

The additive group of the p-adic quotient R as an abelian group and the isomorphism from this group back to R.

Example H47E5

We illustrate the functions in this section for rings. Similar constructions can be used for fields.

```
> Zp := pAdicRing(5, 20);
> I<a> := UnramifiedExtension(Zp, 3);
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^3 + 5*a*x^2 + 5>;
> Prime(L);
5
> InertiaDegree(L);
1
```

The inertia degree of L is returned as 1 because L has been defined as a totally ramified extension of I. However, the inertia degree of L over Z_p is 3, because I itself is an unramified extension of Z_p.

```
> InertiaDegree(L, Zp);
3
> Degree(L);
3
> Degree(L, Zp);
9
> DefiningPolynomial(L);
x^3 + 5*a*x^2 + 5
> P<y> := PolynomialRing(Zp);
> DefiningPolynomial(I);
y^3 + 3*y + 3
> BaseRing(L);
Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic ring mod 5^20
> PrimeRing(L);
5-adic ring mod 5^20
> PrimeRing(I);
5-adic ring mod 5^20
> ResidueClassField(L);
Finite field of size 5^3
Mapping from: RngPad: L to GF(5^3)
> ResidueClassField(I);
Finite field of size 5^3
Mapping from: RngPad: I to GF(5^3)
```

Here, we see that the residue class fields of I and L are identical. This is due to the fact that L is a totally ramified extension of I.

```
> UniformizingElement(L);
b
> Precision(L);
60
> Precision(I);
20
```

```
> R<a> := ext<pAdicRing(2) | 2>;
> DefiningPolynomial(R);
(1 + O(2^20))*$.1^2 + (1 + O(2^20))*$.1 + 1 + O(2^20)
> Precision(R);
Infinity
```

47.6 Operations on Structures

`AssignNames(~L, S)`

`AssignNames(~L, S)`

Assign a name to the generator of L. The sequence must have only one element, which must be a string. This element is assigned to be the name of the generator when L is considered as a linear associative algebra over its base ring.

`Characteristic(L)`

`Characteristic(L)`

`Characteristic(L)`

The characteristic of the local ring or field L.

`#L`

The number of elements in the local ring or field L. The cardinality is finite only if L is a quotient ring or a bounded free precision ring.

Iterating over the elements of a local ring is possible if it is bounded, but it will take time in proportion to the cardinality of L. It is recommended only in the case of “small” local rings (i.e., rings for which the precision is be very small).

`Name(L, k)`

Given a local ring or field L and an integer k, return the generator of L if k is 1; otherwise, raise an error.

`ChangePrecision(L, k)`

`ChangePrecision(~L, k)`

`ChangePrecision(L, k)`

`ChangePrecision(~L, k)`

`ChangePrecision(L, k)`

`ChangePrecision(~L, k)`

`ChangePrecision(L, k)`

`ChangePrecision(~L, k)`

`ChangePrecision(L, k)`

`ChangePrecision(~L, k)`

`ChangePrecision(L, k)`

`ChangePrecision(~L, k)`

Given a local ring or field L and a non-negative single precision integer k, change the maximum precision with which elements can be created to be k. Depending on how L and its subrings have been constructed, there may be an upper bound (possibly infinite for free structures) on the precision to which L can be changed. For instance, the precision to which a defining polynomial has been given places a bound on the precision of the extension — no defining polynomial can be expanded beyond the precision with which it was originally specified.

`L eq K`

Given local rings or fields L and K, return whether or not L and K are the same object.

`L ne K`

Given local rings or fields L and K, return whether or not L and K are different objects.

Example H47E6

```
> Zp := pAdicRing(5, 20);
> I<a> := UnramifiedExtension(Zp, 3);
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^3 + 5*a*x^2 + 5>;
> ChangePrecision(Zp, Infinity());
5-adic ring
> L;
Totally ramified extension defined by the polynomial x^3 + 5*a*x^2 + 5
 over Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic ring mod 5^20
```

```
> ChangePrecision(~L, 50);
> L;
Totally ramified extension defined by the polynomial x^3 + 5*$.1*x^2 + 5
 over Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic ring mod 5^17
> #L;
875811540203010669327330989556197582050 1\
637136728223573481676774311194266786628 7\
59291488677263259887695312 5
> AssignNames(~L, ["t"]);
> L.1;
t
> b;
b
> L eq ChangePrecision(L, 10);
false
```

Note that b is an element of the original ring L with precision 60 which is why it retains its print name.

47.6.1 Ramification Predicates

```
IsRamified(R)
```

```
IsUnramified(R)
```

```
IsTotallyRamified(R)
```

Return whether the local ring or field extension R is ramified, unramified or totally ramified.

```
IsTamelyRamified(R)
```

```
IsWildlyRamified(R)
```

Return whether the local ring or field extension R is tamely ramified (the prime does not divide the ramification degee) or wildly ramified (the prime does divide the ramification degree).

47.7 Element Constructions and Conversions

Local ring elements are implemented using a balanced mod representation. This allows small negative elements x to be represented as x rather than $p^k - x$ where k is a p-adic precision.

47.7.1 Constructions

To simplify the creation of elements in a local ring, various coercions are provided. The most obvious is to regard the integer ring or rational field as embedded in the p-adic ring or field. But there is a range of coercions available, including that of elements from the residue class field.

To create an element of a local field not lying in the local ring, constructors are provided that create an element coercible into the ring, and which increase or decrease this element's valuation in the field.

```
Zero(L)
```

Given a local ring or field L, create the additive identity of L. Note that if L is an unbounded precision structure, this will only be the zero element to the default precision of the ring, and hence only an approximation to the additive identity of L.

```
One(L)
```

Given a local ring or field L, create the multiplicative identity of L. Note that if L is an unbounded precision structure, this will only be the one element to the default precision of the ring, and hence only an approximation to the multiplicative identity of L.

```
Random(L)
```

Given a local ring or field L, return a random element of L, which must be a quotient ring or bounded precision ring. The element will have the default precision of the ring.

```
Representative(L)
```

Return an element of the local ring or field L.

```
elt< L | u >
L ! u
```

Coerce the object u into the local ring or field L. The resulting element will have as much precision as possible. The element u is allowed to be one of the following:

(i) An integer.

(ii) An element of $\mathbf{Z}/p^m\mathbf{Z}$; (where m is a precision).

(iii) An element of the residue class field of L.

(iv) An element of a local ring or field with something in common with L.

(v) A rational number. If L is a ring then u must not have valuation in the denominator.

(vi) An element of a valuation ring over the rationals with the same prime as L.

(vii) A sequence s. In this case, the sequence s is coerced to a sequence t over the base ring or field of L. This sequence is coerced to $\sum_{i=1}^{\#t} t[i] L.1^{i-1}$.

```
elt< L | u, r >
```

Create an element of the local ring or field L by coercing u into L and returning with it precision r.

```
elt< L | v, u, r >
```

Create an element of the local ring or field L by coercing u into L, multiplying by the v-th power of the uniformizing element and returning it with precision r.

```
BigO(x)
O(x)
```

For an element x of a local ring or field L of valuation v, create an element of valuation v and relative precision 0. For rings this is the zero element in the quotient $L/\pi^v L$.

```
UniformizingElement(L)
```

Given a local ring or field L, return the uniformizing element of L to the default precision of L.

Example H47E7

Here we illustrate the usage of element constructors for local fields and imprecise zeros.

```
> Zp := pAdicRing(5, 20);
> I<a> := UnramifiedExtension(Zp, 3);
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^3 + 5*a*x^2 + 5>;
> K<pi> := ext<BaseField(FieldOfFractions(L)) | x^2 + 5>;
> K;
Totally ramified extension defined by the polynomial x^2 + 5
 over Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic field mod 5^20
> elt<K | 64>;
64 + O(pi^40)
> P := PrimeField(K);
> elt<K | 2, 3/4, 6>;
pi^2*7 + O(pi^6)
> 4*$1;
pi^2*3 + O(pi^6)
> K!3/5;
pi^-2*3 + O(pi^38)
> O(K!40^200);
O(pi^400)
```

```
> Precision($1);
0
> O(K!0);
O(pi^40)
> O(K!1);
O(1)
> R<x> := PolynomialRing(Integers());
> K<pi> := ext<pAdicField(5, 100) | x^2 + 5>;
> elt<K | 64>;
64 + O(pi^200)
> Precision($1);
200
> K!3/10;
-pi^-2*3944304526105059027058642826413931148366032175545115023851394653320311 +
O(pi^198)
```

Example H47E8

There is a subtle interplay between the default precision of rings and fields and coercion of sequences, which we demonstrate here. We construct the 5-adic field P, and an unramified extension of R. We set the default precision of R to be higher than that of P.

```
> P := pAdicRing(5);
> R := ext<P | 2>;
> P`DefaultPrecision;
20
> R`DefaultPrecision := 40;
> x := Random(R);
> x;
-1579801843431963201369145587*R.1 - 6805757304589750390333394769 + O(5^40)
> s := [-6805757304589750390333394769, -1579801843431963201369145587];
> R!s;
-13585890629962*R.1 + 47482939261481 + O(5^20)
> P`DefaultPrecision := 40;
> R!s;
-1579801843431963201369145587*R.1 - 6805757304589750390333394769 + O(5^40)
```

As can be seen from the above, it is the default precision of the base ring, not the ring itself, which determines the precision of elements created by sequences.

47.7.2 Element Decomposers

`ElementToSequence(x)`

`Eltseq(x)`

`Coefficients(x)`

Given an element x of a degree n extension K of L, these functions return a sequence s of elements of L such that $x = \sum_{i=1}^{n} s[i] K.1^{i-1}$.

`Coefficient(x, i)`

Equivalent to, but more efficient than, `Coefficients(x)[i]`.

Example H47E9

We want to perform a Galois descent for a polynomial, i.e. we interpret the product of the Galois conjugates of a polynomial in a subring.

```
> p := 3;
> L<a> := UnramifiedExtension(pAdicRing(p), 4 : Cyclotomic);
> R<x> := PolynomialRing(L);
> g := x^2 + (a+a^-2)*x + (a^-1+a^3+1);
> g;
(1 + O(3^20))*x^2 + (-151999392*a^3 - 428033534*a^2 + 1509587217*a - 64512399 +
    O(3^20))*x + 2*a^3 + 307453058*a^2 - 1732356354*a - 151999391 + O(3^20)
> a2 := a^(p^2);
> g2 := R ! [ &+[Eltseq(c)[i]*a2^(i-1) : i in [1..4]] : c in Eltseq(g) ];
```

Here `Eltseq` is being used to replace occurrences of a in the element by a_2.

```
> h := g * g2;
> h;
(1 + O(3^20))*x^4 + (774759822*a^3 - 1559939781*a^2 + 644136111*a + 143311364 +
    O(3^20))*x^3 + (73899384*a^3 - 333497478*a^2 + 1655979363*a + 158024680 +
    O(3^20))*x^2 + (989668189*a^3 - 661853000*a^2 - 1685887308*a + 122035547 +
    O(3^20))*x - 1630826887*a^3 - 328410694*a^2 - 175290219*a + 1601599448 +
    O(3^20)
```

The polynomial g_2 is the image of g under the automorphism induced by the square of the Frobenius automorphism. The product $h = gg_2$ has coefficients in the unramified extension of degree 2, which is the fixed field under the square of the Frobenius and is generated by a^{10}. Next, we determine a representation for the polynomial h with coefficients lying in this unramified extension of degree 2.

```
> K<b> := UnramifiedExtension(pAdicRing(p), 2 : Cyclotomic);
> S<y> := PolynomialRing(K);
> M := RMatrixSpace(PrimeRing(L), 2, 4) ! 0;
> V := RSpace(PrimeRing(L), 4);
> M[1] := V ! Eltseq(a^0);
> M[2] := V ! Eltseq(a^10);
> sol := [ Solution(M, V ! Eltseq(c)) : c in Eltseq(h) ];
> h2 := S ! [ K ! Eltseq(s) : s in sol ];
> h2;
(1 + O(3^20))*y^4 + (-1237301755*b + 505889265 + O(3^20))*y^3 + (-542504216*b +
    1258167831 + O(3^20))*y^2 + (-280210015*b - 1205051127 + O(3^20))*y +
    1213139407*b + 1602993886 + O(3^20)
```

47.8 Operations on Elements

47.8.1 Arithmetic

For local ring elements the usual operations are available. The quotient of two elements can be computed when the result lies in the ring (i.e. the valuation of the dividend is not smaller than that of the divisor). The precision of the result is reduced by the valuation of the divisor. The result of an operation with elements of reduced precision will have as much precision as possible. For addition and subtraction this is the minimum of the precisions of the two elements. For multiplication it is the minimum of $v_1 + k_2$ and $v_2 + k_1$, where v_i and k_i are the valuations and precisions of the two elements, respectively.

For local field elements the operations are performed with the maximum precision possible. For multiplication and division this is the minimum of the (relative) precisions of the two elements. For addition and subtraction of elements x and y with valuations v_x and v_y and precisions k_x and k_y the precision of $x \pm y$ is $\min(v_x + k_x, v_y + k_y) - v_p(x \pm y)$, which may even be 0.

```
-x
```

The negative of the element x.

```
x + y
```

The sum of the elements x and y.

```
x - y
```

The difference of the elements x and y.

```
x * y
```

The product of the elements x and y.

```
x ^ k
```

The k-th power of the element x. If k has valuation (when coerced) and x has precision less than that of its parent ring, the power x^k will have more precision than x.

```
x div y
```

The quotient of the elements x and y. For elements of a local ring, this results in an error if the valuation of x is smaller than that of y.

```
x div:= y
```

Mutation operation: replace the element x by its quotient upon division by y. For elements of a local ring, this results in an error if the valuation of x is smaller than that of y.

```
x / y
```

The quotient of the elements x and y. For elements of a local ring, the result will be returned in its field of fractions.

```
IsExactlyDivisible(x, y)
```

Return `true` if x can be exactly divided by y; in this case also return the quotient.

Example H47E10

Division of non-units in the ring will yield a result in the field of fractions. In that case the result has reduced precision. To ensure that the result is returned as a ring element, the operator `div` should be used.

```
> R := pAdicRing(2);
> pi := UniformizingElement(R);
> pi;
2 + O(2^20)
> 1 / pi;
2^-1 + O(2^18)
> Parent($1);
2-adic field
>  1 div pi;
>> 1 div pi;
     ^
Runtime error in 'div': Division is not exact
> IsExactlyDivisible(1, pi);
false
> IsExactlyDivisible(pi^2, pi);
true 2 + O(2^20)
```

47.8.2 Equality and Membership

It is possible to test whether two local ring or field elements are equal only in quotient rings. Equality testing in free precision rings is disabled, as there are several reasonable definitions of equality in an imprecise ring.

```
x eq y
```

Given local ring or field elements x and y, return `true` if and only if x and y are identical in value. This operator is only defined in fixed precision rings.

`x ne y`

Given local ring or field elements x and y, return `true` if and only if x and y are not identical in value. This operator is only defined in fixed precision rings.

`x in L`

Return `true` if and only if x lies in the local ring or field L.

`x notin L`

Return `true` if and only if x does not lie in the local ring or field L.

Example H47E11

We demonstrate how an unramified extension can be constructed from a given degree. The idea is to interpret the minimal polynomial of a primitive element in the residue class field as a polynomial over $\mathbf{Z}_p$. The quotient of $\mathbf{Z}_p[x]$ by the ideal generated by this polynomial is isomorphic to the unramified extension and $a := \overline{x}$ is a first approximation for the $p^f - 1$-th root of unity. This is improved by iterating $a \mapsto a^{p^f}$ until this remains fixed.

```
> p := 2;
> f := 5;
> Zp := pAdicRing(p, 25);
> R<x> := PolynomialRing(Zp);
> g := R ! MinimalPolynomial(GF(p,f).1);
> Q<r> := quo<R | g>;
> a := [ r, r^(p^f) ];
> while a[#a] ne a[#a-1] do
>     print a[#a];
>     Append(~a, a[#a]^(p^f));
> end while;
34*r^4 - 44*r^3 + 58*r^2 - 23*r + 36
12522914*r^4 + 12522004*r^3 - 12174790*r^2 - 8200343*r - 10407260
8242594*r^4 + 12409364*r^3 + 5143098*r^2 + 15781737*r - 3636572
5490082*r^4 + 8804884*r^3 - 11109830*r^2 + 11456361*r + 11698852
-15481438*r^4 - 5875180*r^3 + 5667386*r^2 + 7262057*r - 884060
> [ Minimum([ Valuation(c) : c in Eltseq(a[i] - a[i-1]) ]) : i in [2..#a-1] ];
[ 1, 6, 11, 16, 21 ]
```

The last statement demonstrates the convergence of the process. The polynomial defining the unramified extension could now easily be obtained as the minimal polynomial of the fixed element.

```
> U := ext<Zp | f>;
> MinimalPolynomial(U ! Eltseq(a[#a]));
x^5 - 13205240*x^4 - 3159900*x^3 - 13778593*x^2 + 9730498*x - 1
```

47.8.3 Properties

Local ring and field elements can be tested for certain properties. However, note that in free precision rings and fields, exact zero, one, and minus one elements do not exist, only approximations to such elements — in these cases, the predicates will always return `false`.

`IsZero(x)`

Given an element x of a local ring or field, return `true` if and only if x is the zero element of its parent ring or field.

`IsOne(x)`

Given an element x of a local ring or field, return `true` if and only if x is the one element of its parent ring or field.

`IsMinusOne(x)`

Given an element x of a local ring or field, return `true` if and only if x is the minus one element of its parent ring or field.

`IsUnit(x)`

Given an element x of a local ring or field, return `true` if and only if x is a unit, that is, x has valuation 0.

`IsIntegral(x)`

Given an element x of a local ring or field, return `true` if and only if x has non-negative valuation.

47.8.4 Precision and Valuation

The parent of a local element can be retrieved, its precision accessed and changed, its valuation computed and its denominator returned.

`Parent(x)`

Return the parent local ring or field of the element x.

`Precision(x)`

For a local ring element x, returns the precision to which it is known; for a local field element x, returns the precision to which its unit part is known.

`AbsolutePrecision(x)`

Returns the precision of a local ring or field element x.

`RelativePrecision(x)`

Returns the difference between the absolute precision and the valuation of the local ring or field element x.

```
ChangePrecision(x, k)
```

```
ChangePrecision(~x, k)
```

```
ChangePrecision(x, k)
```

```
ChangePrecision(~x, k)
```

Given a local ring or field element or polynomial over a local ring or field x and a non-negative single precision integer k, change the precision of x to k. If k is smaller than the precision of x, x is truncated; if k is larger, then an element y is returned such that $v(x - y) \geq k$. Note that the precision of an element cannot be changed to be greater than its parent. Any attempt to do so will result in the element gaining the precision of its parent.

```
Expand(x)
```

Change the precision of the local ring or field element x to be the maximum precision allowed by its parent. This results in an error if x is an element of an unbounded free precision ring.

```
Valuation(x)
```

Return the valuation of the local ring or field element x. This is always bounded by the absolute precision of the element.

Example H47E12

The uses and properties of relative and absolute precision for field elements are illustrated here, as well as the results of `Expand`.

```
> R<x> := PolynomialRing(Integers());
> K<d> := ext<ext<pAdicField(5, 100) | 2> |  x^2 + 5>;
> x := d + d^7;
> x;
-d*124 + O(d^200)
> AbsolutePrecision(x);
200
> RelativePrecision(x);
199
> RelativePrecision(x + O(d^10));
9
> AbsolutePrecision(x + O(d^10));
10
> RelativePrecision(ChangePrecision(x, 19));
19
> RelativePrecision(ChangePrecision(x, 10));
10
> AbsolutePrecision(ChangePrecision(x, 10));
11
> Expand(ChangePrecision(x, 10));
```

```
d*3001 + O(d^201)
> Valuation(x);
1
> ChangePrecision(x, 20) - (x + O(d^21));
O(d^21)
```

The last two lines show that changing the precision of an element is equivalent to adding imprecision to the element effectively cancelling off all terms beyond that of relative valuation 20.

47.8.5 Logarithms and Exponentials

Logarithms and exponentials can also be calculated for certain elements.

`Log(x)`

The logarithm of the local ring or field element x, returned to the precision of x. Note that the power series of the logarithm function only converges if the valuation of $x-1$ is positive. For ring elements x, the answer lies in the ring (and not its field of fractions) only if the valuation of $x-1$ is greater than or equal to the ramification degree of the ring divided by the prime.

The rate of convergence of Log is dependent on the valuation of $x-1$. The greater the valuation the faster the convergence, as is illustrated in the example below.

`Exp(x)`

The exponential of the local ring or field element x, returned to the precision of x. Note that the power series of the exponential function only converges if the valuation of x is strictly larger than $e/(p-1)$, where e is the ramification degree of the parent ring of x.

Example H47E13

This example illustrates the relative timings of the Log function and the Exp function for rings and fields and for various valuations of $x-1$ or x as appropriate.

```
> K := ext<pAdicField(3, 20) | x^3 + x^2 + x + 2>;
> K<d> := ext<K | x^3 + 3*x^2 + 3*x + 3>;
> L<b> := IntegerRing(K);
> x := 1 + b;
> time Log(x);
Time: 0.070
> x := 1 + b^5;
> time Log(x);
874050819*b^2 + 1624571442*b - 914550768
Time: 0.010
> Valuation(Exp(Log(x)) - x);
60
> x := b^2;
> time Exp(x);
1418565628*b^2 + 1334033745*b + 905945461
Time: 0.170
> Valuation(Log(Exp(x)) - x);
60
> x := b^6;
> time Exp(x);
1700312013*b^2 - 72781965*b + 1129064707
Time: 0.020
> Valuation(Log(Exp(x)) - x);
60
```

47.8.6 Norm and Trace Functions

The norm, trace and minimal polynomial of an element can be calculated. A function returning the image of an element under a power of the Frobenius automorphism is also provided.

`Norm(x)`

Given a local ring or field element x, return the norm of x over the base ring of its parent. The norm is the product of all conjugates of x.

`Norm(x, R)`

Given a local ring or field element x, return the norm of x over the local ring or field R. The ring R must be a subring of the parent of x. The norm is the product of all conjugates of x.

`Trace(x)`

Given a local ring or field element x, return the trace of x over the base ring of its parent. The trace is the product of all conjugates of x.

`Trace(x, R)`

Given a local ring or field element x, return the trace of x over the local ring or field R. The ring R must be a subring of the parent of x. The trace is the sum of all conjugates of x.

`MinimalPolynomial(x)`

Given a local ring or field element x, return the minimal polynomial of x over the base ring of its parent.

`MinimalPolynomial(x, R)`

Given a local ring or field element x, return the minimal polynomial of x over the local ring or field R. The ring R must be a subring of the parent of x.

`CharacteristicPolynomial(x)`

Given a local ring element x, return the characteristic polynomial of x over the base ring of its parent.

`CharacteristicPolynomial(x, R)`

Given a local ring or field element x, return the characteristic polynomial of x over the local ring or field R. The ring R must be a subring of the parent of x.

`GaloisImage(x, i)`

Given a local ring or field element x, return the image of x under the Frobenius automorphism composed with itself i times, that is, $a \mapsto a^{(p^i)}$ where a is a $p^f - 1$st root of unity of the parent of x.

Example H47E14

One important application of the p-adics is to counting points on elliptic curves over finite fields. This example demonstrates how a simple version of the Arithmetic-Geometric Mean (AGM) algorithm could be implemented in Magma.

```
> d := 50;
> FF := GF(2^d);
> E := EllipticCurve([FF | 1, 0, 0, 0, Random(FF)]);
> assert Degree(sub<BaseRing(E) | jInvariant(E)>) gt 2;
>
> n := (d + 1) div 2 + 3;
> R := ext<pAdicRing(2) | d>;
> R`DefaultPrecision := n;
>
> a6 := elt<R | jInvariant(E)^-1, 1>;
> lambda := 1 + 8 * a6;
>
> for k in [4..n] do
>     ChangePrecision(~lambda, k + 2);
>     lambda := (1 + lambda) * InverseSqrt(lambda) div 2;
> end for;
> lambda := 2 * lambda div (1 + lambda);
> Exp(Trace(Log(lambda)));
32196193 + O(2^26)
> Trace(E) mod 2^26;
32196193
```

47.8.7 Teichmüller Lifts

Another important operation is the determination of the canonical *Teichmüller lift* of an element u in the residue field of a p-adic ring. By definition this is the unique root of unity of order prime to p which reduces to u modulo the maximal ideal.

```
TeichmuellerLift(u, R)
```

For a fixed precision quotient ring R of the p-adics or an unramified extension and a finite field element u, the function attempts to coerce u into the residue class field of R and then computes and returns its Teichmüller lift to R. This uses a fast iterative lifting method of Harley described in Chapter 12 of [C$^+$05]. For unramified extensions this works most efficiently with `Cyclotomic` or `Gaussian Normal` bases since it uses the Frobenius operation.

47.9 Linear Algebra

All linear algebra over local rings and fields in Magma is implemented in terms of the fixed precision model. Internally, linear algebra over free precision models is performed over an appropriate quotient ring. The advantage of this method is that exact algorithms can be used to solve linear systems; the disadvantage is that the answer may be returned to less precision than is theoretically possible.

47.10 Roots of Elements

Roots of local ring and field elements can be found to some precision.

```
SquareRoot(x)
```

```
Sqrt(x)
```

Given a local ring or field element x, return a square root of x. An error results if x is not a square. The result may have less precision than x.

```
IsSquare(x)
```

Return whether the local ring or field element x is the square of an element in its parent and if it is, the square root is returned. The result may have less precision than x.

```
InverseSquareRoot(x)
```

```
InverseSqrt(x)
```

Given a local ring or field element x, return an inverse square root of x. The element x must be a unit. An error results if x is not a square. The result may have less precision than x.

```
InverseSquareRoot(x, y)
```

```
InverseSqrt(x, y)
```

Given local ring or field elements x and y, return an inverse square root of x lifted from an initial approximation y. The element x must be a unit. An error results if x is not a square, or if y is not a valid initial approximation to an inverse square root of x. The result may have less precision than x.

```
Root(x, n)
```

Return an n-th root of x if one exists. An error results if x is not an n-th power. The result may have less precision than x.

```
IsPower(x, n)
```

Return whether x is an n-th power of some element belonging to its parent and if it is return an n-th root. The result may have less precision than x.

```
InverseRoot(x, n)
```

Given a local ring or field element x, return an inverse n-th root of x. The element x must be a unit. An error results if x is not an n-th power. The result may have less precision than x.

```
InverseRoot(x, y, n)
```

Given local ring or field elements x and y, return an inverse n-th root of x lifted from an initial approximation y. The element x must be a unit. An error results if x is not an n-th power, or if y is not a valid initial approximation to an inverse n-th root of x. The result may have less precision than x.

47.11 Polynomials

Various simple operations for polynomials as well as root finding and factorization functions have been implemented for polynomials over local rings and fields.

47.11.1 Operations for Polynomials

A number of functions for polynomials in general are applicable for polynomials over local rings and fields. Arithmetic functions, including `div` and `mod` can be used with such polynomials (though there may be some precision loss), as well as all the elementary functions to access coefficients and so forth. Derivatives can be taken and polynomials over local rings and fields can be evaluated at elements coercible into the coefficient ring. Along with GCD for these polynomials, the LCM of two polynomials can also be found.

Although the ring of polynomials over a local ring is not a principal ideal domain, it is useful to have a GCD function available. For example, for polynomials which are coprime over the local field, the ideal generated by the two polynomials contains some power of the uniformizing element of the local ring. This power determines whether an approximate factorization can be lifted to a proper factorization.

`GreatestCommonDivisor(f, g)`

`Gcd(f, g)`

`GCD(f, g)`

Determine the greatest common divisor of polynomials f and g where f and g are over a free precision local ring or field. The GCD returned is such that the cofactors of the polynomials will have coefficients in the ring (if the polynomial is not over a field). The process of computing the GCD of two polynomials may result in some inaccuracy. The GCD is computed by echelonizing the Sylvester matrix of f and g.

`f div g` `f mod g` `LeastCommonMultiple(f, g)`

`Coefficient(f, i)` `LeadingCoefficient(f)`

`Derivative(f)` `Evaluate(f, x)`

Example H47E15

This example illustrates the usage and results of the GCD functions. Note the precision loss in the answer.

```
> L := pAdicRing(5, 20);
> R<x> := PolynomialRing(L);
> elts := [Random(L) : i in [1..3]];
> f := (x - elts[1])^3 * (x - elts[2])^2 * (x - elts[3]);
> f;
x^6 + 31722977012336*x^5 - 34568128687249*x^4 +
    4655751767246*x^3 + 11683626356181*x^2 -
    29833674388290*x + 32360011367900
> GCD(f, Derivative(f));
(1 + O(5^18))*x^3 - (934087632277 + O(5^18))*x^2 -
    (89130566204 + O(5^18))*x + 1178670674955 + O(5^18)
> f mod $1;
O(5^18)*x^5 + O(5^18)*x^4 + O(5^18)*x^3 + O(5^18)*x^2 +
    O(5^18)*x + O(5^19)
> (x - elts[1])^2 * (x - elts[2]);
x^3 + 14324701430223*x^2 + 26613750293171*x +
    31696248799955
> ChangePrecision($1, 18);
(1 + O(5^18))*x^3 - (934087632277 + O(5^18))*x^2 -
    (89130566204 + O(5^18))*x + 1178670674955 + O(5^18)
```

`ShiftValuation(f, n)`

Shifts the valuation of each coefficient of f by n, ie. scales the polynomial by π^n.

47.11.2 Roots of Polynomials

Roots of polynomials can be found to some precision by applying the Hensel lift to an approximation or by factorizing the polynomial and looking for linear factors from which the roots can be read off.

47.11.2.1 Hensel Lifting of Roots

Roots of a polynomial f defined over a local ring or field can be found by first determining their valuation (using the Newton polygon), finding a first approximation over the finite field and finally improving this approximation until the Hensel condition $v(f(a)) > 2v(f'(a))$ is satisfied.

```
NewtonPolygon(f)
```

The Newton polygon of a polynomial $f = \sum_{i=0}^{n} c_i x^i$ (over a local ring or field) is the lower convex hull of the points $(i, v(c_i))$. The slopes of the Newton polygon determine the valuations of the roots of f in a splitting ring and the number of roots with that valuation. The faces of the Newton polygon can be determined using the function `Faces` which returns the faces expressed as the line $mx + ny = c$ which coincides with the face. The function `GradientVector` will return the m and n values from the line so that the valuation (gradient) can be calculated. The function `EndVertices` will return the end points of the face, the x coordinates of which will give the number of roots with valuation equal to the gradient of the face.

Newton polygons are discussed in greater detail in Chapter 46 and are illustrated below.

```
ValuationsOfRoots(f)
```

Return a sequence containing pairs which are valuations of roots of f and the number of roots of f which have that valuation.

Example H47E16

For a polynomial of the form $g := \prod(x - r_i)$ we demonstrate that the Newton polygon determines the valuations of the roots of g.

```
> Z3 := pAdicRing(3, 30);
> R<y> := PolynomialRing(Z3);
> pi := UniformizingElement(Z3);
> roots := [ pi^Random([0..3]) * Random(Z3) : i in [1..10] ];
> [ Valuation(r) : r in roots ];
[ 3, 1, 6, 3, 0, 3, 2, 3, 3, 2 ]
> g := &* [ y - r : r in roots ];
> N := NewtonPolygon(g);
> N;
Newton Polygon of y^10 + 44594997030169*y^9 - 85346683389318*y^8 +
    76213593390537*y^7 + 74689026811236*y^6 - 48671968754502*y^5 -
    58608670426020*y^4 - 63609139981179*y^3 + 77334553491246*y^2 +
    39962036019861*y - 94049035648173 over pAdicRing(3, 30)
```

```
> F := Faces(N);
> F;
[ <6, 1, 26>, <3, 1, 23>, <2, 1, 17>, <1, 1, 9>, <0, 1, 0> ]
> [GradientVector(F[i]) : i in [1 .. #F]];
[ <6, 1>, <3, 1>, <2, 1>, <1, 1>, <0, 1> ]
> [$1[i][1]/$1[i][2] : i in [1 ..#$1]];
[ 6, 3, 2, 1, 0 ]
> [EndVertices(F[i]) : i in [1 .. #F]];
[
    [ <0, 26>, <1, 20> ],
    [ <1, 20>, <6, 5> ],
    [ <6, 5>, <8, 1> ],
    [ <8, 1>, <9, 0> ],
    [ <9, 0>, <10, 0> ]
]
> [$1[i][2][1] - $1[i][1][1] : i in [1 .. #$1]];
[ 1, 5, 2, 1, 1 ]
```

So there is one root of valuation 6, five of valuation 3, two of valuation 2, one of valuation 1 and one root with valuation zero. This information could also have been gained using `ValuationsOfRoots`.

```
> ValuationsOfRoots(g);
[ <6, 1>, <3, 5>, <2, 2>, <1, 1>, <0, 1> ]
```

`HenselLift(f, x)`

`HenselLift(f, x, k)`

Return a root of the polynomial f over a local ring or field by lifting the approximate root x to a root with precision k (or the default precision of the structure if not specified). This results in an error, if the Hensel condition $v(f(x)) > 2v(f'(x))$ is not satisfied.

Example H47E17

This examples illustrates how Hensel lifting is used to compute square roots.

```
> Zx<x> := PolynomialRing(Integers());
> L1<a> := ext<pAdicRing(3, 20) | 2>;
> L2<b> := ext<L1 | x^2 + 3*x + 3>;
> R<y> := PolynomialRing(L2);
> c := (a+b)^42;
> r := L2 ! Sqrt(ResidueClassField(L2) ! c);
> r;
a
> rr := HenselLift(y^2-c, r);
> rr;
(-1513703643*a - 1674232545)*b - 1219509587*a + 760894776
> Valuation(rr^2 - c);
```

```
40
> ChangePrecision(rr, 1);
a + O(b)
```

For $p = 2$ the situation is a bit more difficult, since the derivative of $y^2 - c$ is not a unit at this point.

```
> Zx<x> := PolynomialRing(Integers());
> L1<a> := ext<pAdicRing(2, 20) | 2>;
> L2<b> := ext<L1 | x^2 + 2*x + 2>;
> R<y> := PolynomialRing(L2);
> c := (a+b)^42;
> r := L2 ! Sqrt(ResidueClassField(L2) ! c);
> r;
1
>  HenselLift(y^2-c, r);
>> HenselLift(y^2-c, r);
             ^
Runtime error in 'HenselLift': Hensel lift condition is not satisfied
```

We have to find a better approximation for the square root first.

```
> for d in GF(2,2) do
>     if Valuation((r + b*L2!d)^2 - c) gt 4 then
>         print L2!d;
>     end if;
> end for;
a + 1
> r +:= b * (a+1);
> HenselLift( y^2-c, r );
(-199021*a + 100463)*b + 204032*a - 31859 + O(b^38)
> ChangePrecision($1, 1);
1 + O(b)
> ChangePrecision($2, 2);
(a + 1)*b + 1 + O(b^2)
```

The square root is an element of reduced precision, since the proper root is only guaranteed to coincide with the approximation up to valuation 18.

47.11.2.2 Functions returning Roots

These functions determine the roots of a polynomial from the factorization of the polynomial.

`Roots(f)`

`Roots(f, R)`

`IsSquarefree` BoolElt *Default :* `false`

Return the roots of the polynomial f over a local ring or field R as a sequence of tuples of elements in R and multiplicities. If R is not specified it is taken to be the coefficient ring of f. If the polynomial is known to be squarefree, the root-finding algorithm may run considerably faster.

```
HasRoot(f)
```

Try to find a root of the polynomial f over a local ring or field. If a root is found, this function returns `true` and a root as a second value; otherwise it returns `false`.

Example H47E18

We generate the ramified extensions of $\mathbf{Z}_2$ of degree 2 by looping over some Eisenstein polynomials with small coefficients and checking whether a new polynomial has a root in one of the already known rings.

```
> Zx<x> := PolynomialRing(Integers());
> RamExt := [];
> for c0 in [2, -2, 6, -6] do
> for c1 in [0, 2, -2, 4, -4, 6, -6] do
>     g := x^2 + c1*x + c0;
>     new := true;
>     for L in RamExt do
>         if HasRoot(PolynomialRing(L)!g) then
>             new := false;
>             break L;
>         end if;
>     end for;
>     if new then
>         print "new field with polynomial", g;
>         Append(~RamExt, ext<pAdicRing(2, 20) | g>);
>     end if;
> end for;
> end for;
new field with polynomial x^2 + 2
new field with polynomial x^2 + 2*x + 2
new field with polynomial x^2 + 4*x + 2
new field with polynomial x^2 + 2*x - 2
new field with polynomial x^2 + 4*x - 2
new field with polynomial x^2 + 6
```

These are all such extensions, since the extensions of $\mathbf{Q}_2$ of degree 2 are in bijection to the 7 non-trivial classes of $\mathbf{Q}_2^*/(\mathbf{Q}_2^*)^2$ and one of these classes yields the unramified extension.

47.11.3 Factorization

It is possible to factorize (to some precision) polynomials over a local ring or field. Approximate factorizations can also be lifted to a factorization to greater precision.

`HenselLift(f, s)`

Given a sequence s of polynomials with coefficients that can be coerced into the coefficient ring of f such that $f \equiv \prod_{i=1}^{\#s} s[i]$ modulo π, $s[i]$ and $[j]$ are co–prime modulo π for all i and j, and $s[i]$ is monic for all i, find a more accurate factorization $t[1], t[2], \ldots t[\#s]$ such that $f \equiv \prod_{i=1}^{\#s} t[i]$ modulo π^k, where k is the minimum precision of the coefficients of f.

Example H47E19

If the reduction of a polynomial over the residue class field is not a power of an irreducible polynomial, the factorization into powers of different irreducibles can be lifted to a factorization over the local ring.

```
> Z2 := pAdicRing(2, 25);
> R<x> := PolynomialRing(Z2);
> f := &* [ x - i : i in [1..8] ];
> F2 := ResidueClassField(Z2);
> Factorization( PolynomialRing(F2)!f );
[
    <$.1, 4>,
    <$.1 + 1, 4>
]
> h1 := x^4;
> h2 := (x+1)^4;
> h := HenselLift(f, [h1, h2]);
> h[1], h[2], f - h[1]*h[2];
x^4 - 20*x^3 + 140*x^2 - 400*x + 384
x^4 - 16*x^3 + 86*x^2 - 176*x + 105
0
```

`IsIrreducible(f)`

Given a polynomial f with coefficients lying in a local ring or field L, return `true` if and only if f is irreducible over L. Currently, this only works over p-adic rings, unramified extensions of p-adic rings, totally ramified extensions of p-adic rings, and totally ramified extension of unramified extensions of p-adic rings.

`SquareFreeFactorization(f)`

Return a sequence of tuples of polynomials and multiplicities where the polynomials are not divisible by any square of a polynomial. The product of the polynomials to the corresponding multiplicities is the polynomial f (to some precision). Currently, this only works over p-adic rings, unramified extensions of p-adic rings, totally ramified extensions of p-adic rings, and totally ramified extension of unramified extensions of p-adic rings.

`Factorization(f)`

`LocalFactorization(f)`

`Certificates`	BoolElt	*Default :* `false`
`IsSquarefree`	BoolElt	*Default :* `false`
`Ideals`	BoolElt	*Default :* `false`
`Extensions`	BoolElt	*Default :* `false`

Return the factorization of the polynomial f over a local ring or field into irreducible factors as a sequence of pairs, the first entry giving the irreducible factor and the second its multiplicity.

Precision is important since for polynomials over rings of relatively small precision a correct factorization may not be possible and an error will result. A lower bound on the precision needed for the factorization to succeed is given by `SuggestedPrecision`; this precision may still be insufficient, however.

The precision the factorization is returned to is reduced for multiple factors.

The optional parameter `IsSquarefree` can be set to `true`, if the polynomial is known to be square-free. The `Certificates` parameter can be set to `true` to compute certificates for the irreducibility of the individual factors returned. This information can be used to compute the p-maximal order of the equation order defined by the factor. If the `Extensions` parameter is set to `true` then certificates will be returned which will include an extension given by each factor.

`SuggestedPrecision(f)`

For a polynomial f over a local ring or field, return a precision at which the factorization of f as given by `Factorization(f)` will be Hensel liftable to the correct factorization.

The precision returned is not guaranteed to be enough to obtain a factorization of the polynomial. It may be that a correct factorization cannot be found at that precision but may be possible with a little more precision.

Currently, this only works over p-adic rings, unramified extensions of p-adic rings, totally ramified extensions of p-adic rings, and totally ramified extension of unramified extensions of p-adic rings.

`IsIsomorphic(f, g)`

Given two irreiducible polynomials over the same p-adic field, test if the extensions defined by them are isomorphic.

`Distance(f, g)`

Given two Eisenstein polynomials of the same degree and discriminant compute their distance, i.e. the minimal weighted valuation of the difference of the coefficients.

Example H47E20

The use of `SuggestedPrecision` along with `Factorization` is illustrated below for a few different cases.

```
> L<b> := ext<ext<pAdicRing(5, 20) | 2> |  y^2 + 5*y + 5>;
> R<x> := PolynomialRing(L);
> f := (x - 1)^3*(x - b)^2*(x - b^2 + b - 1);
> SuggestedPrecision(f);
80
> Factorization(f);
[
    <x - b + O(b^40), 2>,
    <x - 1 + O(b^40), 3>,
    <x + 6*b + 4 + O(b^40), 1>
]
1 + O(b^40)
> f := (x + 2)^3*(x + b)*(x + 3)^4;
> f;
x^8 + (b + 18 + O(b^40))*x^7 + (18*b + 138 + O(b^40))*x^6 + (138*b + 584 +
    O(b^40))*x^5 + (584*b + 1473 + O(b^40))*x^4 + (1473*b + 2214 + O(b^40))*x^3
    + (2214*b + 1836 + O(b^40))*x^2 + (1836*b + 648 + O(b^40))*x + 648*b +
    O(b^40)
> SuggestedPrecision(f);
80
> Precision(L);
80
> P<y> := PolynomialRing(Integers());
> R<b> := ext<ext<pAdicRing(3, 20) | 2> |  y^2 + 3*y + 3>;
> P<x> := PolynomialRing(R);
> f := x^12 + 100*x^11 + 4050*x^10 + 83700*x^9 + 888975*x^8 + 3645000*x^7 -
> 10570500*x^6 - 107163000*x^5 + 100875375*x^4 + 1131772500*x^3 -
> 329614375*x^2 + 1328602500*x + 332150625;
> SuggestedPrecision(f);
48
> Factorization(f);
[
    <x + 153560934 + O(b^40), 1>,
    <x - 1595360367 + O(b^40), 1>,
    <x + 1273329451*$.1 - 1037496066 + O(b^40), 1>,
    <x + -1273329451*$.1 - 97370567 + O(b^40), 1>,
    <x^4 + (-1598252738*$.1 - 309919655 + O(b^40))*x^3 + (-280421715*$.1 -
        102998055 + O(b^40))*x^2 + (1263867503*$.1 + 923047935 + O(b^40))*x +
        -1252549104*$.1 - 786365260 + O(b^40), 1>,
    <x^4 + (1598252738*$.1 - 600198580 + O(b^40))*x^3 + (280421715*$.1 +
        457845375 + O(b^40))*x^2 + (-1263867503*$.1 - 1604687071 + O(b^40))*x +
        1252549104*$.1 + 1718732948 + O(b^40), 1>
]
1 + O(b^40)
> R<b> := ext<ext<pAdicRing(3, 25) | 2> |  y^2 + 3*y + 3>;
> P<x> := PolynomialRing(R);
> f := x^12 + 100*x^11 + 4050*x^10 + 83700*x^9 + 888975*x^8 + 3645000*x^7 -
> 10570500*x^6 - 107163000*x^5 + 100875375*x^4 + 1131772500*x^3 -
> 329614375*x^2 + 1328602500*x + 332150625;
> Factorization(f);
[
    <x + 143905694229 + O(b^50), 1>,
    <x - 399882754935 + O(b^50), 1>,
    <x + 46601526664*$.1 + 229090274400 + O(b^50), 1>,
    <x + -46601526664*$.1 + 135887221072 + O(b^50), 1>,
    <x^4 + (242476655332*$.1 + 187976437999 + O(b^50))*x^3 + (372805509192*$.1 -
        38457626466 + O(b^50))*x^2 + (126788105939*$.1 + 60198382752 +
        O(b^50))*x + 347425890996*$.1 + 215394267602 + O(b^50), 1>,
    <x^4 + (-242476655332*$.1 - 296976872665 + O(b^50))*x^3 + (-372805509192*$.1
        + 63219964593 + O(b^50))*x^2 + (-126788105939*$.1 - 193377829126 +
        O(b^50))*x + -347425890996*$.1 + 367831095053 + O(b^50), 1>
]
1 + O(b^50)
```

Note that the polynomial itself must have coefficients with precision at least that given by `SuggestedPrecision` (and not just the coefficient ring) for `Factorization` to succeed. Sometimes this will not be possible if the coefficients of the polynomial are not known to sufficient precision.

Example H47E21

In this example we demonstrate how factorizations of a rational polynomial over some local rings can give information on the Galois group.

```
> Zx<x> := PolynomialRing(Integers());
> g := x^5 - x + 1;
> Factorization(Discriminant(g));
[ <19, 1>, <151, 1> ]
> g2 := Factorization( PolynomialRing(pAdicRing(2, 10)) ! g );
> g2;
[
    <$.1^2 + 367*$.1 - 93, 1>,
    <$.1^3 - 367*$.1^2 - 386*$.1 + 11, 1>
]
```

```
> g3 := Factorization( PolynomialRing(pAdicRing(3, 10)) ! g );
> g3;
[
    <$.1^5 - $.1 + 1, 1>
]
> g7 := Factorization( PolynomialRing(pAdicRing(7, 10)) ! g );
> g7;
[
    <$.1^2 + 138071312*$.1 + 2963768, 1>,
    <$.1^3 - 138071312*$.1^2 + 132202817*$.1 - 84067650, 1>
]
```

This shows that the Galois group of g contains elements of orders 2, 3, 5 and 6, and therefore is isomorphic to S_5.

47.12 Automorphisms of Local Rings and Fields

The automorphisms of a local ring or field are determined by their images on the generators of the ring. All computations necessary to determine the automorphism group can be performed in the local ring.

`Automorphisms(L)`

Given a local ring or field L, returns the automorphisms of L over its p-adic sub-field $\mathbf{Q}_p$ as a sequence of maps of L into L.

`Automorphisms(K, k)`

Given a local ring or field K over k, returns the k-automorphisms of K as a sequence of maps of K into K.

`AutomorphismGroup(L)`

Return the automorphism group acting on L over its p-adic sub-field $\mathbf{Q}_p$ as a permutation group (representing the regular action). Also return the map from the permutation group to the group of automorphisms represented explicitly (i.e. like returned from the function above).

`AutomorphismGroup(K, k)`

Return the automorphism group acting on K over its p-adic k as a permutation group (representing the regular action). Also return the map from the permutation group to the group of automorphisms represented explicitly (i.e. like returned from the function above).

`IsNormal(K)`

Given a p-adic ring or field K, test if K is normal over it's prime field $\mathbf{Q}_p$, ie. if K admits exactly n automorphisms where n is the degree of K.

```
IsNormal(K, k)
```

Given a p-adic ring or field K, test if K is normal over the subfield k.

```
IsAbelian(K, k)
```

Given p-adic fields K/k, test if the automorphism group of K over k is abelian.

```
Continuations(m, L)
```

For an automorphism m of the p-adic ring L, compute all possible extensions of m to L.

```
IsIsomorphic(E, K)
```

For two p-adic rings or fields, test if they are isomorphic over $\mathbf{Q}_p$.

Example H47E22

We define an extension of $\mathbf{Z}_2$ with ramification degree 2 and inertia degree 2 and compute automorphisms.

```
> I<a> := ext<pAdicRing(2, 10) | 2>;
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^2 + 2*a*x + 2*a^2>;
> L;
Totally ramified extension defined by the polynomial
x^2 + (2*a)*x + -2*a - 2 over Unramified extension
defined by the polynomial x^2 + x + 1 over 2-adic ring
mod 2^10
> A := Automorphisms(L);
> [<A[i](a), A[i](b)> : i in [1 .. #A]];
[ <a, b + O(b^18)>, <a, -b + -2*a + O(b^18)>, <-a - 1,
a*b + O(b^18)>, <-a - 1, -a*b + 2*a + 2 + O(b^18)> ]
> AutomorphismGroup(L);
Permutation group acting on a set of cardinality 4
    Id($)
    (1, 2)(3, 4)
    (1, 3)(2, 4)
Mapping from: GrpPerm: $, Degree 4 to Power Structure
of Map given by a rule
```

47.13 Completions

Local Rings can be obtained by completing an order at a prime ideal (see Chapter 34 and `Completion` on page 3-889).

```
Completion(O, P)
```

```
Completion(K, P)
```

`Precision` RngIntElt *Default :* 20

The completion (as an unbounded precision local ring or field with default precision given by `Precision`) of the order or number field at the prime ideal P, and the embedding of the order or number field into the resulting local ring.

```
LocalRing(P, k)
```

The completion (as a local ring) of the order of the prime ideal P at P with precision k and the embedding of the order into the resulting local ring.

Example H47E23

Here we demonstrate the use of `Completion`.

```
> K := NumberField(x^6 - 5*x^5 + 31*x^4 - 85*x^3 + 207*x^2 - 155*x + 123);
> lp := Decomposition(K, 7);
> C, mC := Completion(K, lp[2][1]);
> C;
Totally ramified extension defined by a map over Unramified extension defined by
a map over 7-adic field
> mC;
Mapping from: FldNum: K to FldPad: C given by a rule
> mC(K.1);
(46564489*$.1 - 47959419)*C.1 - 116434149*$.1 - 61099304 + O(C.1^20)
> mC(K.1)@@mC - K.1;
833782149340252135048 8*K.1^5 - 6907350696005689646443 2*K.1^4 +
    18984741644344433087772 6*K.1^3 - 45336153029197695133787 6*K.1^2 +
    33697964781411679927609 9*K.1 - 26752086971419700257907 1
> Valuation($1, lp[2][1]);
18
> C`DefaultPrecision := 30;
> mC(K.1);
(1090965976127*$.1 - 1208477074641)*C.1 - 589359803563*$.1 + 288063654676 +
O(C.1^30)
> mC(K.1)@@mC - K.1;
-619800242441603716728687734334907 83*K.1^5 +
    1189796803064803092593291088768968754*K.1^4 -
    3202353946933190588864309180653868957*K.1^3 +
    7537386928046164580731145031872017049*K.1^2 -
    5511297002936682579964210586013308810*K.1 +
    443809944480643131358253343594109872 2
```

```
> Valuation($1, lp[2][1]);
28
> C`DefaultPrecision := 10;
> mC(K.1);
(-7708*$.1 + 7759)*C.1 + 4747*$.1 - 5859 + O(C.1^10)
> mC(K.1)@@mC - K.1;
1908210240*K.1^5 - 7326424608*K.1^4 + 16701662320*K.1^3 - 35965440540*K.1^2 +
    41324075079*K.1 - 30476856505
> Valuation($1, lp[2][1]);
8
```

47.14 Class Field Theory

The class field theory of local fields classifies abelian extensions of local field in a way similar to the way global class field theory deals with extensions of number fields and global function fields.

While the origins of local class field theory are, via completions and localisations, in the global case, today it is a theory in its own. Although local class field theory can be used to obtain global results, it has very powerful generalisations that the global case (currently) does not allow.

Local class fields are classified in terms of the norm group, ie. the multiplicative group of norms of elements, rather than some ideal or divisor class group as in the global case. Since the multiplicative group of a local field is far better understood than the ideal group of a global field, the theory is much more explicit and easier in the local case.

47.14.1 Unit Group

In contrast to the case of global fields, the multiplicative group of both p-adic rings and fields has a well understood structure which can be computed by algorithms developed and implemented by S. Pauli [Pau06]. It should be noted that all the unit group related functions operate on fixed-precision rings only.

`PrincipalUnitGroupGenerators(R)`

The principal units of a p-adic ring or field R are elements of the form $1 + \pi\mathbf{Z}_R$ where π is a uniformizing element of R and $\mathbf{Z}_R$ is the ring of integers. This function returns a sequence of generators for this group.

`PrincipalUnitGroup(R)`

The principal units of a p-adic ring or field R are elements of the form $1 + \pi\mathbf{Z}_R$ where π is a uniformizing element of R and $\mathbf{Z}_R$ is the ring of integers. This function returns an abstract abelian group isomorphic to the group of principal units and an explicit isomorphism, ie. a map between the abstract group and the p-adic ring or field.

```
UnitGroup(R)
```

Given a p-adic ring R of fixed precision, this function computes an abstract abelian group isomorphic to the unit group as well as an explicit map between the abstract group and R.

```
UnitGroup(F)
```

Given a p-adic field F of fixed precision, this function computes an abstract abelian gorup isomorphic to the multiplicative group of F as well as an explicit map between the abstract group and F.

```
UnitGroupGenerators(R)
```

Given a p-adic ring with fixed precision, this function computes generators for its unit group.

```
UnitGroupGenerators(F)
```

Given a p-adic field with fixed precision, this function computes generators for its multiplicative group.

```
pSelmerGroup(p,F)
```

Given a l-adic field F, return the p-Selmer group, i.e., the group F^*/F^{*p}, as an abstract group, as well as the map from F^* to the abstract group.

47.14.2 Norm Group

Given two p-adic field F/k the norm group of F in k, ie. the image of the norm map from F to k is the central object of local class field theory. Since the norm map will always operate on some multiplicative group, all functions in this section will take the map returned by `UnitGroup` as an argument as this then allows the convenient way of describing the norm group as a subgroup of some explicit finitely generated abelian group.

```
NormGroup(R, m)
```

Given a p-adic ring or field R extending S and a description of the unit group of S encoded by a map m from some abstract abelian group to S as computed by `UnitGroup`, compute the image of the norm map as a subgroup. The map returned is the embedding map returned form the subgroup constructor.

```
NormEquation(R, m, b)
```

Given a p-adic ring R defined over S, the unit group of S encoded by the map m as computed by `UnitGroup(S)` and some element $b \in S$, try to compute an element $a \in R$ such that the norm of a equals b. In case such an element exists, it is returned as a second value.

```
NormEquation(m1, m2, G)
```

Given two p-adic rings R and S and their unit groups U_R and U_S as parameterized by the maps $m_1 : U_R \to R$ and $U_S \to S$ as well as a subgroup $G < U_S$, compute the preimage of G under the norm map operating on the unit groups.

```
Norm(m1, m2, G)
```

Given two p-adic rings R and S and their unit groups U_R and U_S as parameterized by the maps $m_1 : U_R \to R$ and $U_S \to S$ as well as a subgroup $G < U_R$, compute the image of G under the norm map operating on the unit groups.

```
NormKernel(m1, m2)
```

Given two p-adic rings R and S and their unit groups U_R and U_S as parameterized by the maps $m_1 : U_R \to R$ and $U_S \to S$ compute the kernel of the norm map from U_R to U_S as a subgroup of U_R.

47.14.3 Class Fields

Class fields, that is abelian extensions are parameterized by their norm groups. Pauli, in [Pau06] gave explicit algorithms to solve the reverse problem of class field theory: given a suitable subgroup of some (abstractly given) multiplicatively group of some p-adic field, compute explicit defining equations for the class field.

```
ClassField(m, G)
```

Given a p-adic field S and its multiplicative group U_S specified by the map $m : U_S \to S$ and a suitable subgroup $G < U_S$, this function computes for each cyclic factor of U_S/G an explicit defining equation for the class field corresponding to this factor.

```
NormGroupDiscriminant(m, G)
```

Given a p-adic field S and its multiplicative group U_S specified by the map $m : U_S \to S$ and a suitable subgroup $G < U_S$, this function computes the valuation of the discriminant of the extension parameterized by G without computing explicit equations for it.

47.15 Extensions

It is a well known classical theorem that p-adic fields admit only finitely many different extensions of bounded degree (in contrast to number fields which have an infinite number of extensions of any degree). In his thesis, Pauli [Pau01a] developed explicit methods to enumerate those extensions.

```
AllExtensions(R, n)
```

`E`	RngIntElt	*Default* : 0
`F`	RngIntElt	*Default* : 0
`Galois`	BoolElt	*Default* : `false`
`D`	RngIntElt	*Default* : 0
`j`	RngIntElt	*Default* : -1

Given a p-adic ring or field R and some positive integer n, compute defining equations for all extensions of R of degree n. The optional parameters can be used to limit the extensions in various ways:

`E`	specifies the ramification index. 0 implies no restriction.
`F`	specifies the inertia degree, 0 implies no restriction.
`D`	specified the valuation of the discriminant, 0 implies no restriction.
`j`	specifies the valuation of the discriminant via the formula $D := d+j-1$ where d is the degree of R.
`Galois`	when set to `true`, limits the extensions to only list normal extensions.

```
NumberOfExtensions(R, n)
```

`E`	RngIntElt	*Default* : 0
`F`	RngIntElt	*Default* : 0
`Galois`	BoolElt	*Default* : `false`
`D`	RngIntElt	*Default* : 0
`j`	RngIntElt	*Default* : -1

Given a p-adic ring or field R and some positive integer n, compute the number of extensions of R of degree n. Similarly to the above function, the optional parameters can be used to impose restrictions on the fields returned.

```
OreConditions(R, n, j)
```

Given a p-adic ring or field R and positive integers n and j, test if there exists extensions of R of degree n with valuation of the discriminant being $n + j - 1$.

47.16 Bibliography

[**C$^+$05**] H. Cohen et al. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Discrete Mathematics and Its Applications. CRC Pr Llc, 2005.

[**Pau01a**] Sebastian Pauli. *Efficient Enumeration of Extensions of Local Fields with Bounded Discriminant.* PhD thesis, Concordia University, 2001.

[**Pau01b**] Sebastian Pauli. Factoring polynomials over local fields. *Journal of Symbolic Computation*, 32(5):533–547, 2001.

[**Pau06**] Sebastian Pauli. Constructing Class Fields over Local Fields. *J. Théorie des Nombres de Bordeaux*, 18(3):627–652, 2006.

48 GALOIS RINGS

Chapter 48
GALOIS RINGS

48.1 Introduction

Magma provides facilities for computing with Galois rings. The features are currently very basic, but advanced features will be available in the near future, including support for the creation of subrings and appropriate embeddings, allowing lattices of compatible embeddings, just as for finite fields.

A Galois ring R in Magma is considered as a finite algebraic extension of $\mathbf{Z}_{p^a}$ (where p is prime) by a monic polynomial $D \in \mathbf{Z}[x]$ which is irreducible modulo p. Thus R is presented as the polynomial quotient ring

$$\mathbf{Z}_{p^a}[x]/\langle D\rangle$$

and is usually written as $\mathbf{GR}(p^a, d)$, where d is the degree of D. The cardinality of R is easily seen to be p^{ad}.

R has a unique maximal ideal generated by p, and the quotient ring $R/\langle p\rangle$ is a finite field isomorphic to $\mathbf{Z}_p[x]/\langle D\rangle$, where D is here considered as a polynomial in $\mathbf{Z}_p[x]$ (the coefficients are reduced modulo p). This finite field is called the *residue field* of R. In the following, we will also call the integer residue ring $\mathbf{Z}_{p^a}$ the *base ring* of R, because this is the subring of R generated by 1 and we can think of R as an extension of $\mathbf{Z}_{p^a}$.

For an non-zero element x of R, the valuation of x is defined to be the largest power of p which divides the coefficients of x, where x is considered as a polynomial in $\mathbf{Z}_p[x]/\langle D\rangle$. x is a unit if and only if the valuation of x is zero.

Because of the valuation defined on them, Galois rings are Euclidean rings, so they may be used in Magma in any place where general Euclidean rings are valid. This includes many matrix and module functions, and the computation of Gröbner bases. Linear codes over Galois rings will be supported in the near future.

48.2 Creation Functions

48.2.1 Creation of Structures

```
GaloisRing(q, d)
```

```
GR(q, d)
```

Given integers q, $d \geq 1$, where $q = p^a$ for prime p and $a \geq 1$, create the default Galois ring $\mathbf{GR}(p^a, d)$. The defining polynomial used to construct the ring will be that used for $\mathbf{F}_{p^d}$, lifted to $\mathbf{Z}_{p^a}$. If p is very large, it is advised to use the next function instead, because Magma must first factor q completely.

The angle bracket notation can be used to assign names to the generator; e.g.: `R<w> := GaloisRing(2, 3)`.

```
GaloisRing(p, a, d)
GR(p, a, d)
```

Check BoolElt *Default :* `true`

Given a prime p and integers $a, d \geq 1$, create the default Galois ring $\mathbf{GR}(p^a, d)$. The defining polynomial used to construct the ring will be that used for $\mathbf{F}_{p^d}$, lifted to $\mathbf{Z}_{p^a}$.

By default p is checked to be a strong pseudoprime for 20 random bases b with $1 < b < p$; if the parameter `Check` is false, then no check is done on p at all (this is useful when p is very large and one does not wish to perform an expensive primality test on p).

```
GaloisRing(q, D)
GR(q, D)
```

Given an integer q, where $q = p^a$ for prime p and $a \geq 1$, and a monic polynomial D over $\mathbf{Z}$ such that D is irreducible mod p, create the Galois ring $R = \mathbf{GR}(p^a, D)$. The coefficients of D are reduced modulo p^a, and R is constructed to behave like the polynomial quotient ring $\mathbf{Z}_{p^a}[x]/\langle D\rangle$. If p is very large, it is advised to use the next function instead, because Magma must first factor q completely.

```
GaloisRing(p, a, D)
GR(p, a, D)
```

Check BoolElt *Default :* `true`

Given a prime p, an integer $a \geq 1$, and a monic polynomial D over $\mathbf{Z}$ such that D is irreducible mod p, create the Galois ring $R = \mathbf{GR}(p^a, D)$. The coefficients of D are reduced modulo p^a, and R is constructed to behave like the polynomial quotient ring $\mathbf{Z}_{p^a}[x]/\langle D\rangle$. The parameter `Check` is as above.

48.2.2 Names

```
AssignNames(~R, [f])
```

Procedure to change the name of the generating element in the Galois ring R to the contents of the string f. When R is created, the name will be `R.1`.

This procedure only changes the name used in printing the elements of R. It does *not* assign to an identifier called f the value of the generator in R; to do this, use an assignment statement, or use angle brackets when creating the ring.

Note that since this is a procedure that modifies R, it is necessary to have a reference `~R` to R in the call to this function.

```
Name(R, 1)
```

Given a Galois ring R, return the element which has the name attached to it, that is, return the element `R.1` of R.

48.2.3 Creation of Elements

`R . 1`

`Generator(R)`

The generator for R as an algebra over its base ring $\mathbf{Z}_{p^a}$. Thus, if R is viewed as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$, then `R.1` corresponds to x in this presentation.

`R ! a`

Given a Galois ring R create the element specified by a; here a is allowed to be an element coercible into R, which means that a may be

(i) An element of R;

(ii) An integer, to be identified with a modulo the characteristic p^a of R;

(iii) An element of the base ring $\mathbf{Z}_{p^a}$ of R, to be identified with the corresponding element of R.

(iv) A sequence of elements of the base ring $\mathbf{Z}_{p^a}$ of R. In this case the element $a_0 + a_1 w + \cdots + a_{n-1}w^{n-1}$ is created, where $a = [a_0, \dots a_{n-1}]$ and w is the generator `R.1` of R over $\mathbf{Z}_{p^a}$.

`One(R)` `Identity(R)`

`Zero(R)` `Representative(R)`

These generic functions create 1, 1, 0, and 0 respectively, in any Galois ring.

`Random(R)`

Create a pseudo-random element of Galois ring R.

48.2.4 Sequence Conversions

`ElementToSequence(a)`

`Eltseq(a)`

Given an element a of the Galois ring R, return the sequence of coefficients $[a_0, \dots, a_{n-1}]$ in the base ring $\mathbf{Z}_{p^a}$ of R (where R is $\mathbf{Z}_{p^a}[x]/\langle D\rangle$), such that $a = a_0 + a_1 w + \cdots + a_{n-1}w^{d-1}$, with w the generator of R, and d the degree of D.

Example H48E1

We can define the Galois ring $\mathbf{GR}(2^3, 2)$ using the default function:

```
> R<w> := GaloisRing(2^3, 2);
> R;
GaloisRing(2, 3, 2)
```

We note that R has characteristic 8 and that $w^2 + w + 1 = 0$ in R.

```
> R!8;
```

```
0
> R!9;
1
> w;
w
> 4*w;
4*w
> 4*w + 4*w;
0
> w^2;
7*w + 7
> w^2 + w + 1;
0
```

We can list all the elements of R by simply looping over R:

```
> [x: x in R];
[ 0, 1, 2, 3, 4, 5, 6, 7, w, w + 1, w + 2, w + 3, w + 4, w + 5, w + 6,
    w + 7, 2*w, 2*w + 1, 2*w + 2, 2*w + 3, 2*w + 4, 2*w + 5, 2*w + 6, 2*w
    + 7, 3*w, 3*w + 1, 3*w + 2, 3*w + 3, 3*w + 4, 3*w + 5, 3*w + 6,
    3*w + 7, 4*w, 4*w + 1, 4*w + 2, 4*w + 3, 4*w + 4, 4*w + 5, 4*w +
    6, 4*w + 7, 5*w, 5*w + 1, 5*w + 2, 5*w + 3, 5*w + 4, 5*w + 5, 5*w
    + 6, 5*w + 7, 6*w, 6*w + 1, 6*w + 2, 6*w + 3, 6*w + 4, 6*w + 5,
    6*w + 6, 6*w + 7, 7*w, 7*w + 1, 7*w + 2, 7*w + 3, 7*w + 4, 7*w +
    5, 7*w + 6, 7*w + 7 ]
```

We see that the elements of R can be considered as polynomials of degree at most 1, with coefficients in the range $\{0 \ldots 7\}$.
We can easily create elements of R also using the `!` operator, and use the `Eltseq` function to recover the corresponding sequence of coefficients.

```
> R ! [1, 2];
2*w + 1
> Eltseq(2*w + 1);
[ 1, 2 ]
> Eltseq(w);
[ 0, 1 ]
```

48.3 Structure Operations

48.3.1 Related Structures

`Category(R)` `Parent(R)` `Centre(R)`

`PrimeRing(R)` `PrimeField(R)`

`FieldOfFractions(R)`

48.3.2 Numerical Invariants

```
Characteristic(R)
```

The characteristic of R, which is p^a where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$.

```
#R
```

The cardinality of R, which is p^{ad} where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$ and d is the degree of D.

```
Degree(R)
```

The degree of R, which is the degree of D, where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$.

```
ResidueField(R)
```

The residue field of R, which is the finite field $\mathbf{Z}_p[x]/\langle D\rangle$, where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$.

48.3.3 Ring Predicates and Booleans

The following functions are described for rings in general in Section 17.4.3.

```
IsCommutative(R)    IsUnitary(R)
IsFinite(R)    IsOrdered(R)
IsField(R)    IsEuclideanDomain(R)
IsPID(R)    IsUFD(R)
IsDivisionRing(R)    IsEuclideanRing(R)
IsPrincipalIdealRing(R)    IsDomain(R)
R eq G    R ne G
```

48.4 Element Operations

See also Section 17.5.

48.4.1 Arithmetic Operators

```
+ a    - a
a + b    a - b    a * b    a ^ k
a +:= b    a -:= b    a *:= b
```

48.4.2 Euclidean Operations

`a div b` `a mod b` `Quotrem(a, b)`

`GCD(a, b)` `LCM(a, b)` `XGCD(a, b)`

48.4.3 Equality and Membership

`a eq b` `a ne b`

`a in R` `a notin R`

48.4.4 Parent and Category

`Parent(a)` `Category(a)`

48.4.5 Predicates on Ring Elements

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)`

Example H48E2

This simple example shows how one can compute Gröbner bases over Galois rings. We create an ideal in 3 variables over a Galois ring and compute its Gröbner basis.

```
> R<w> := GaloisRing(3, 3, 2);
> #R;
729
> P<x,y,z> := PolynomialRing(R, 3);
> I := ideal<P | [x^2 - w*y, 3*y^3 - 3*w*x*z, 9*z^5 - 9*w]>;
> GroebnerBasis(I);
[
    x^2 + 26*w*y,
    3*x*y^3 + (6*w + 6)*y*z,
    3*x*z + (15*w + 3)*y^3,
    9*x + (9*w + 9)*y^3*z^4,
    3*y^6 + (21*w + 15)*y*z^2,
    9*z^5 + 18*w
]
```

Notice that the leading coefficients include 3 and 9, which are not units in the ring. See Chapter 105 for much more information on such Gröbner bases.

49 POWER, LAURENT AND PUISEUX SERIES

Chapter 49

POWER, LAURENT AND PUISEUX SERIES

49.1 Introduction

This chapter describes the operations on formal power, Laurent and Puiseux series available in Magma.

49.1.1 Kinds of Series

Internally Magma has only one kind of formal series, where general fractional exponents are allowed, but externally there are three kinds of series: power, Laurent and Puiseux. Power series must have a non-negative integral valuation and integral exponents, Laurent series must have an integral valuation and integral exponents (possibly negative), while Puiseux series may have a general rational valuation and general rational exponents. The main reason for the three kinds is simply to supply error checking for operations illegal to a specific kind, so the user will not move into the next kind of exponents inadvertently. Apart from these error checks, there is no difference at all between the different kinds of series, so one can simply use Puiseux series always if the restrictions on exponents are not desired, and this will cause no loss of efficiency or functionality at all even if all exponents remain integral or integral and non-negative.

In Magma, the set of all power series over a given base ring R is called a power series ring (denoted by $R[[x]]$) and the category names are `RngSerPow` for the ring and `RngSerPowElt` for the elements. The set of all Laurent series over a given base ring R is called a Laurent series ring (denoted by $R((x))$) and the category names are `RngSerLaur` for the ring and `RngSerLaurElt` for the elements. Finally, the set of all Puiseux series over a given base ring R is called a Puiseux series ring (denoted by $R\langle\langle x\rangle\rangle$) and the category names are `RngSerPuis` for the ring and `RngSerPuisElt` for the elements. For the rest of this chapter, the term "series ring" refers to any of the above rings and the corresponding virtual category name is `RngSer`; the term "series" refers to any of the above series kinds and the corresponding virtual category name is `RngSerElt`.

49.1.2 Puiseux Series

Puiseux series in a variable x are often mathematically defined to be Laurent series in another variable y say, where $y = x^{1/d}$, for a fixed positive integer d; this d is usually fixed for all the series under consideration. Magma is more general, in that although each series is internally represented in this way (i.e., its valuation, exponents and precision are thought to be divided by a single denominator associated with it), different Puiseux series may have different exponent denominators and may be freely mixed (for example, in addition, where the exponent denominator of the result will be derived from that of the arguments). Thus there is no restriction whatsoever to a fixed exponent denominator for a given Puiseux series ring.

49.1.3 Representation of Series

Formal series are stored in truncated form, unless only finitely many terms are non-zero. If a series has precision p, that means that its coefficients are unknown from exponent p onwards. This truncated form is similar to the floating point representation of real numbers except for one significant difference: because for series terms do not "carry" in arithmetic operations like they do in integer and real arithmetic, error propagation common in floating point methods does not occur. Consequently, given a result of any sequence of arithmetic operations with formal series, the coefficients of the known terms (the ones up to the given precision) will always be *exactly correct*, with no error (assuming the coefficient ring has exact arithmetic for its elements of course).

Elements

$$c_v x^v + c_{v+1} x^{v+1} + \ldots$$

(with $v \in \mathbf{Q}$ and $r_v \neq 0$) of series ring over a commutative ring R are stored in the form of approximations

$$c_v x^v + c_{v+1} x^{v+1} + \ldots + O(x^p)$$

to a certain *precision* $p \geq v$. The $O(x^p)$ notation is used to refer to terms of degree at least the precision p in x. For Laurent series v must be an integer and for power series v must be a non-negative integer. Note that for Puiseux series the above element is actually internally stored in the form

$$c_w x^{w/d} + c_{w+1} x^{(w+1)/d} + \ldots + O(x^{q/d})$$

where $v = w/d$ and $p = q/d$ and d (the exponent denominator) is minimal.

49.1.4 Precision

Associated with any series there are two types of precision: the *absolute precision* and the *relative precision*. For the element

$$c_v x^v + c_{v+1} x^{v+1} + \cdots + O(x^p)$$

(with $v \in \mathbf{Q}$ and $r_v \neq 0$), the absolute precision is p and the relative precision is $p - v$.

The absolute precision indicates the largest known term, and the relative precision indicates the size of the range over which terms are known. The two types of precision and the *valuation* of the series, which equals v in the above, are therefore always related via $p = v + r$.

49.1.5 Free and Fixed Precision

For each type of series ring, there are two sub-kinds: a *free precision* ring (the usual case), where elements of the ring have arbitrary precision, and a *fixed precision* ring, where all elements of the ring have a fixed precision. In the latter case, for power series rings the fixed precision is absolute, while for Laurent and Puiseux series ring the fixed precision is relative.

The free precision rings most closely resemble the mathematical objects $R[[x]]$ and $R((x))$; elements in these free rings and fields carry their own precision with them. Operations usually return results to a precision that is maximal given the input (and the nature of the operation). Operations which are given infinite precision series but which must return finite precision series (e.g., division) return series whose precision is the *default* precision for the series ring (this is different from the fixed precision of fixed precision rings); the default precision is 20 by default but may be set to another value at creation by a parameter (see below). Elements of free structures that have finite series expansion (i.e., polynomials) can be created and stored exactly, with infinite (absolute and relative) precision. Also note that the relative precision will be 0 for approximations to 0.

The structures with fixed precision, which we will sometimes refer to as *quotient* structures, behave differently. All elements in a *power* series ring of fixed precision have the same fixed *absolute* precision p, and the relative precision may be anything from 0 to p. This means that the ring with fixed precision p behaves like a quotient of a polynomial ring, $R[x]/x^p$. All elements in a *Laurent* or *Puiseux* series ring of fixed precision have the same fixed *relative* precision; the only exception to this rule is that 0 in the ring is stored as zero with infinite absolute precision. The absolute precision of elements in a free Laurent or Puiseux series ring can be anything.

49.1.6 Equality

Since a given series ring will contain series truncated at arbitrary precision, care has to be taken as to the meaning of equality of two elements. Two series are considered equal iff they are identical (the equality operator `eq` follows this convention). But we call two series A and B in the same ring *weakly equal* if and only if their coefficients are the same whenever both are defined, that is, if and only if for every $n \leq p$ the coefficients A_n and B_n are equal, where p is the minimum of the precisions of A and B. Thus, for example, $A = 3+x+O(x^2)$ and $B = 3+x+17x^2+O(x^4)$ are the same, but $C = 3+x+17x^2+x^3+O(x^4)$ is different from B. Note that A and C are equal under this definition, and hence weak equality is not transitive!

49.1.7 Polynomials over Series Rings

For a discussion of operations for polynomials over series rings see Chapter 46 and Section 49.7.

49.2 Creation Functions

49.2.1 Creation of Structures

```
PowerSeriesRing(R)
```

```
PowerSeriesRing(R, p)
```

`Global`	BoolElt	*Default :* `true`
`Precision`	RngIntElt	*Default :* 20

Given a commutative ring R, create the ring $R[[x]]$ of formal power series over R. If a second integer argument p is given, the resulting ring is a fixed precision series ring with fixed precision p; otherwise the resulting ring is a free precision series ring and the optional argument `Precision` may be used to set the default precision for elements of the power series ring (it will be 20 otherwise; see the section above on free and fixed precision). By default, a global series ring will be returned; if the parameter `Global` is set to `false`, a non-global series ring will be returned (to which a separate name for the indeterminate can be assigned). The angle bracket notation can be used to assign a name to the indeterminate: `S<x> := PowerSeriesRing(R)`.

```
LaurentSeriesRing(R)
LaurentSeriesRing(R, p)
```

`Global`	BoolElt	*Default :* `true`
`Precision`	RngIntElt	*Default :* 20

Given a commutative ring R, create the ring $R((x))$ of formal Laurent series over R. If a second integer argument p is given, the resulting ring is a fixed precision series ring with fixed precision p; otherwise the resulting ring is a free precision series ring and the optional argument `Precision` may be used to set the default precision for elements of the power series ring (it will be 20 otherwise; see the section above on free and fixed precision). By default, a global series ring will be returned; if the parameter `Global` is set to `false`, a non-global series ring will be returned (to which a separate name for the indeterminate can be assigned). The angle bracket notation can be used to assign a name to the indeterminate: `S<x> := LaurentSeriesRing(R)`.

```
PuiseuxSeriesRing(R)
PuiseuxSeriesRing(R, p)
```

`Global`	BoolElt	*Default :* `true`
`Precision`	RngIntElt	*Default :* 20

Given a commutative ring R, create the ring $R\langle\langle x\rangle\rangle$ of formal Puiseux series over R. If a second integer argument p is given, the resulting ring is a fixed precision series ring with fixed precision p; otherwise the resulting ring is a free precision series ring and the optional argument `Precision` may be used to set the default precision for elements of the power series ring (it will be 20 otherwise; see the section above on free and fixed precision). The optional argument `Precision` may be used to set the default precision for elements of the power series ring. By default, a global series ring will be returned; if the parameter `Global` is set to `false`, a non-global series ring will be returned (to which a separate name for the indeterminate can be assigned). The angle bracket notation can be used to assign a name to the indeterminate: `S<x> := PuiseuxSeriesRing(R)`.

Example H49E1

We demonstrate the difference between global and non-global rings. We first create the global power series ring over **Q** twice.

```
> Q := RationalField();
> P<x> := PowerSeriesRing(Q);
> PP := PowerSeriesRing(Q);
> P;
Power series ring in x over Rational Field
> PP;
Power series ring in x over Rational Field
> PP.1;
x
```

PP is identical to P. We now create non-global series rings (which are also different to the global series ring). Note that elements of all the rings are mathematically equal by automatic coercion.

```
> Pa<a> := PowerSeriesRing(Q: Global := false);
> Pb<b> := PowerSeriesRing(Q: Global := false);
> Pa;
Power series ring in a over Rational Field
> Pb;
Power series ring in b over Rational Field
> a;
a
> b;
b
> P;
Power series ring in x over Rational Field
> x;
x
> x eq a; // Automatic coercion
true
> x + a;
2*x
```

49.2.2 Special Options

```
AssertAttribute(S,"DefaultPrecision", n)
```

Procedure to change the default precision on a free series ring series S; the default precision will be set to n, which must be a non-negative integer.

```
HasAttribute(S, "DefaultPrecision")
```

Function that returns a Boolean indicating whether a default precision has been set on the free series ring S (which will always be `true`), as well as its (non-negative) integer value (which is 20 by default).

```
AssignNames(~S, ["x"])
```

Procedure to change the name of the 'indeterminate' transcendental element generating the series ring or field S; the name (used in printing elements of S) is changed to the string x. Note that no assignment to the identifier x is made (so x cannot be used for the specification of elements of S without further assignment).

```
Name(S, 1)
```

```
S . 1
```

Return the element of the series ring or field with a name attached to it, that is, return the 'indeterminate' transcendental element generating S over its coefficient ring.

49.2.3 Creation of Elements

The easiest way to create power and Laurent series in a given ring is to use the angle bracket construction to attach names to the indeterminate, and to use these names to express the series (see the examples). Below we list other options.

```
R . 1
```

```
UniformizingElement(R)
```

Return the generator (indeterminate) for the series ring R.

```
elt< R | v, [ a1, ..., ad], p >
```

Given a series ring R, integers v and p (where $p > 0$ or $p = \infty$), and a sequence $a = [a_1, \ldots, a_d]$ of elements of R, create the element in R with valuation v, known coefficients given by a and relative precision p. That is, this function returns the series $a_1x^v + \cdots + a_dx^{v+d-1} + O(x^{v+p})$, or, if $p = -1$, the exact series $a_1x^v + \cdots + a_dx^{v+d-1}$. If R is a power series ring, then v must be non-negative.

The integer v or the integer p or both may be omitted. If v is omitted, it will be set to zero by default; if p is omitted it will be taken to be $v + d$, where d is the length of the sequence a.

```
R ! s
```

Coerce s into the series ring R. Here s is allowed to be a sequence of elements from (or coercible into) the coefficient ring of R, or just an element from (or coercible into) R. A sequence $[a_1, \ldots, a_d]$ is converted into the series $a_1+a_2x^1+\cdots+a_dx^{d-1}+O(x^d)$.

`BigO(f)`

`O(f)`

Create the series $O(x^v)$ where x is the generator of the parent of f and v is the valuation of f. The most typical usage of this function is the expression $O(x^n)$ where x is the generator of a series ring, but a general series f is actually allowed.

`One(Q)` `Identity(Q)`

`Zero(Q)` `Representative(Q)`

49.3 Structure Operations

49.3.1 Related Structures

`Parent(R)` `Category(R)`

`BaseRing(R)`

`CoefficientRing(R)`

Return the coefficient ring of the series ring R.

`IntegerRing(R)`

`Integers(R)`

`RingOfIntegers(R)`

Return the power series ring which is the integer ring of the laurent series ring R.

`FieldOfFractions(R)`

Return the laurent series ring which is the field of fractions of the series ring R.

`ChangePrecision(R, r)`

`ChangePrecision(~R, r)`

Return a series ring identical to the series ring R but having precision r.

`ChangeRing(R, C)`

Return the series ring identical to the series ring R but having coefficient ring C.

`ResidueClassField(R)`

Return the residue class field of the series ring R (which will be the same as the coefficient ring of R) and the map from R into the residue class field.

49.3.2 Invariants

`Characteristic(R)`

`Precision(R)`
`GetPrecision(R)`

Return the precision of the fixed precision series ring R. If R is a fixed precision power series ring, then this is the fixed absolute precision for all elements of the ring. If R is a fixed precision Laurent series ring, then this is the maximum relative precision for all elements of the ring.

49.3.3 Ring Predicates and Booleans

`IsCommutative(Q)` `IsUnitary(Q)`
`IsFinite(Q)` `IsOrdered(Q)`
`IsField(Q)` `IsEuclideanDomain(Q)`
`IsPID(Q)` `IsUFD(Q)`
`IsDivisionRing(Q)` `IsEuclideanRing(Q)`
`IsPrincipalIdealRing(Q)` `IsDomain(Q)`
`R eq S` `R ne S`

49.4 Basic Element Operations

49.4.1 Parent and Category

`Parent(r)` `Category(r)`

49.4.2 Arithmetic Operators

`+ b` `- b`
`a + b` `a - b` `a * b` `a ^ k`
`a div b` `a / b`

49.4.3 Equality and Membership

```
a eq b        a ne b
a in R        a notin R
```

49.4.4 Predicates on Ring Elements

Note the definition of equality in the introduction to this Chapter. This not only affects the result of the application of `eq` and `ne`, but also that of `IsOne, IsZero` and `IsMinusOne`.

```
IsZero(a)        IsOne(a)        IsMinusOne(a)
IsNilpotent(x)        IsIdempotent(x)
IsUnit(a)        IsZeroDivisor(x)        IsRegular(x)
IsIrreducible(x)        IsPrime(x)
```

```
IsWeaklyZero(f)
```

Given a series f, return whether f is weakly zero, which is whether f is exactly zero or of the form $O(x^p)$ for some p.

```
IsWeaklyEqual(f, g)
```

Given series f and g, return whether f is weakly equal to g, which is whether $(f-g)$ is weakly zero (see `IsWeaklyZero`).

```
IsIdentical(f, g)
```

Given series f and g, return whether f is identical to g, which is whether f and g have exactly the same valuation, precision, and coefficients.

49.4.5 Precision

```
AbsolutePrecision(f)
```

Given a series f, this returns the absolute precision that is stored with f. If f is a series in x, the absolute precision of f is the exponent p such that x^p is the first term of f of which the coefficient is not known, that is, it is the least p such that $f \in O(x^p)$. If f is known exactly (in a free ring), the absolute precision is infinite and an error occurs. Note that the absolute precision may be a non-integral rational number if f is a Puiseux series.

```
RelativePrecision(f)
```

Given a series f, this returns the relative precision that is stored with f. The relative precision counts the number of coefficients of f that is known, starting at the first non-zero term. Hence the relative precision is the difference between the absolute precision and the valuation of f, and is therefore always non-negative; however, if f is exact, the relative precision is infinite and the value ∞ is returned. Note that the relative precision may be a non-integral rational number if f is a Puiseux series.

```
ChangePrecision(f, r)
ChangePrecision(~f, r)
```

The (non puiseux) series f with absolute precision r (which can be positive infinity).

49.4.6 Coefficients and Degree

```
Coefficients(f)
ElementToSequence(f)
Eltseq(f)
```

Let f be a series with coefficients in a ring R and with indeterminate x. This function returns the sequence Q of coefficients of f, the unscaled valuation v and the exponent denominator d of f (v is the true valuation of f multiplied by d). The i-th entry $Q[i]$ of Q equals the coefficient of

$$x^{\frac{v+i-1}{d}}$$

in f. Thus the first entry of Q is the 'first' (lowest order) non-zero coefficient of f, i.e., the coefficient of x^w where w is the true valuation of f.

```
Coefficient(f, i)
```

Given a series f with coefficients in a ring R, and a rational or integer i, return the coefficient of the i-th power of the indeterminate x of f as an element of R. If f is a Puiseux series i may be a (non-integral) rational; otherwise i must be an integer (and also must be non-negative if f is a power series). Also, i must be less than p, the precision of f.

```
LeadingCoefficient(f)
```

Given a series f with coefficients in a ring R, return the leading coefficient of f as an element of R, which is the first non-zero coefficient of f (i.e., the coefficient x^v in f, where x is the indeterminate of f and v is the valuation of f).

```
LeadingTerm(f)
```

Given a series f with coefficients in a ring R, return the leading term of f, which is the first non-zero term of f (i.e., the term of f whose monomial is x^v, where x is the indeterminate of f and v is the valuation of f).

```
Truncate(f)
```

Given a series f, return the exact series obtained by truncating f after the last known non-zero coefficient.

```
ExponentDenominator(f)
```

Given a series f, return the exponent denominator of f, i.e., the lowest common denominator of all the exponents of the non-zero terms of f (always an integer). For power series and Laurent series, this will always be 1 of course.

```
Degree(f)
```

Given a series f, return the degree of the truncation of f, that is, the exponent of the last known non-zero term. Note that this may be a non-integral rational number if f is a Puiseux series.

```
Valuation(f)
```

Given a series f, return the smallest integer v (possibly negative for Laurent series) such that the coefficient of x^v in f is not known to be zero. For the exact 0 element (in a free ring), the valuation is ∞. Note that the valuation may be a non-integral rational number if f is a Puiseux series.

```
ExponentDenominator(f)
```

The exponent denominator of the series f. This is the lowest common denominator of the exponents of the non-zero terms of f.

49.4.7 Evaluation and Derivative

```
Derivative(f)
```

Given a series $f \in R$, return the derivative of f with respect to its indeterminate, as an element of R. Note that the precision decreases by 1 (unless f has infinite precision).

```
Derivative(f, n)
```

Given a series $f \in R$ and an integer $n > 0$, return the n-th derivative of f with respect to its indeterminate, as an element of R. Note that the precision decreases by n (unless f has infinite precision).

```
Integral(f)
```

Given a series $f \in R$, return an anti-derivative F of f with respect to its indeterminate, which is an element of R which has derivative f. The coefficient of x^{-1} in f must be zero. Note that the precision of F will be exceeding that of f by 1 (unless f has infinite precision).

```
Evaluate(f, s)
```

Given an element f of a series ring over the coefficient ring R, and an element s of the ring S, return the value of $f(s)$ when the indeterminate x is evaluated at s. The result will be an element of the common overstructure over R and S.

```
Laplace(f)
```

The Laplace transform of the series f; if f has expansion $\sum_{i\geq 0} a_i x^i$, its Laplace transform has expansion $\sum_{i\geq 0}(i! a_i)x^i$. The valuation of f must be integral and non-negative.

49.4.8 Square Root

```
SquareRoot(f)
Sqrt(f)
```

Return the square root of the series f, f must have even valuation if it is a power or Laurent series.

49.4.9 Composition and Reversion

```
Composition(f, g)
```

Given elements f and g from the same series ring P, return their composition, defined by

$$f \circ g = \sum_{i<p} f_i (g^i),$$

where $f = \sum_{i<p} f_i x^i$.

```
Reversion(f)
Reverse(f)
```

Given a series f (in x, say), this returns the inverse of f under composition, that is, an element g of the same power series ring such that its composition with f equals x to the best possible precision. If f is a power or Laurent series, the valuation of f must be 1. If f is a Puiseux series, the valuation of f must be positive (but need not equal 1), and if the valuation of f is not 1, the leading coefficient of f must be 1.

```
Convolution(f, g)
```

Given elements f and g from the same series ring P, return their convolution $f * g$, defined by

$$f * g = \sum_{i<\min(p,q)} f_i g_i x^i,$$

where $f = \sum_{i<p} f_i x^i + O(x^p)$ and $g = \sum_{i<q} g_i x^i + O(x^q)$.

Example H49E2

We demonstrate the functions `Composition` and `Reversion`. First we check that `Arcsin` is the reversion of `Sin`.

```
> S<x> := PowerSeriesRing(RationalField());
> f := Sin(x);
> g := Arcsin(x);
> f;
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 -
    1/39916800*x^11 + 1/6227020800*x^13 - 1/1307674368000*x^15 +
    1/355687428096000*x^17 - 1/121645100408832000*x^19 + O(x^21)
> g;
x + 1/6*x^3 + 3/40*x^5 + 5/112*x^7 + 35/1152*x^9 + 63/2816*x^11 +
    231/13312*x^13 + 143/10240*x^15 + 6435/557056*x^17 +
    12155/1245184*x^19 + O(x^21)
> Composition(f, g);
x + O(x^21)
> Composition(g, f);
x + O(x^21)
> Reversion(f) - g;
O(x^21)
> Reversion(g) - f;
O(x^21)
```

Next we compute the reversion of a series whose valuation is not 1.

```
> S<x> := PuiseuxSeriesRing(RationalField());
> f := x^3 - x^5 + 2*x^8;
> r := Reversion(f);
> f;
x^3 - x^5 + 2*x^8
> r;
x^(1/3) + 1/3*x + 4/9*x^(5/3) - 2/3*x^2 + 65/81*x^(7/3) -
    22/9*x^(8/3) + 5/3*x^3 - 208/27*x^(10/3) + 5005/729*x^(11/3)
    - 70/3*x^4 + 206264/6561*x^(13/3) - 50830/729*x^(14/3) +
    134*x^5 - 498674/2187*x^(16/3) + 31389020/59049*x^(17/3) +
    O(x^6)
> Composition(r, f);
x + O(x^18)
> Composition(f, r);
x + O(x^(20/3))
```

Finally we compute the reversion of a proper Puiseux series.

```
> f := x^(2/5) - x^(2/3) + x^(3/2) + O(x^2);
> r := Reversion(f);
> r;
x^(5/2) + 5/2*x^(19/6) + 145/24*x^(23/6) + 715/48*x^(9/2) +
    389795/10368*x^(31/6) - 5/2*x^(21/4) + O(x^(11/2))
> Composition(f, r);
```

```
x + O(x^4)
> Composition(r, f);
x + O(x^(11/5))
```

49.5 Transcendental Functions

In each of the functions below, the precision of the result will be approximately equal to the precision of the argument if that is finite; otherwise it will be approximately equal to the default precision of the parent of the argument. An error will result if the coefficient ring of the series is not a field. If the first argument has a non-zero constant term, an error will result unless the coefficient ring of the parent is a real or complex domain (so that the transcendental function can be evaluated in the constant term).

See also the chapter on real and complex fields for elliptic and modular functions which are also defined for formal series.

49.5.1 Exponential and Logarithmic Functions

`Exp(f)`

Given a series f defined over a field, return the exponential of f.

`Log(f)`

Given a series f defined over a field of characteristic zero, return the logarithm of f. The valuation of f must be zero.

Example H49E3

In this example we show how one can compute the Bernoulli number B_n for given n using generating functions. The function `BernoulliNumber` now actually uses this method (since V2.4). Using this method, the Bernoulli number B_{10000} has been computed, taking about 14 hours on a 250MHz Sun Ultrasparc (the computation depends on the new asymptotically fast methods for series division).

The exponential generating function for the Bernoulli numbers is:

$$E(x) = \frac{x}{e^x - 1}.$$

This means that the n-th Bernoulli number B_n is $n!$ times the coefficient of x^n in $E(x)$. The Bernoulli numbers $B_0, \ldots, B_n$ for any n can thus be calculated by computing the above power series and scaling the coefficients.

In this example we will compute B_{500}. We first set the final coefficient index n we desire to be 500. We then create the denominator $D = e^x - 1$ of the exponential generating function to precision $n+2$ (we need $n+2$ since we lose precision when we divide by the denominator and the valuation changes). For each series we create, we print the sum of it and $\mathrm{O}(x^{20})$, which will only print the terms up to x^{19}.

```
> n := 500;
> S<x> := LaurentSeriesRing(RationalField(), n + 2);
> time D := Exp(x) - 1;
Time: 0.040
> D + O(x^20);
x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6 + 1/5040*x^7 +
    1/40320*x^8 + 1/362880*x^9 + 1/3628800*x^10 + 1/39916800*x^11 +
    1/479001600*x^12 + 1/6227020800*x^13 + 1/87178291200*x^14 +
    1/1307674368000*x^15 + 1/20922789888000*x^16 + 1/355687428096000*x^17 +
    1/6402373705728000*x^18 + 1/121645100408832000*x^19 + O(x^20)
```

We then form the quotient $E = x/D$ which gives the exponential generating function.

```
> time E := x / D;
Time: 5.330
> E + O(x^20);
1 - 1/2*x + 1/12*x^2 - 1/720*x^4 + 1/30240*x^6 - 1/1209600*x^8 +
    1/47900160*x^10 - 691/1307674368000*x^12 + 1/74724249600*x^14 -
    3617/10670622842880000*x^16 + 43867/5109094217170944000*x^18 + O(x^20)
```

We finally compute the Laplace transform of E (which multiplies the coefficient of x^i by $i!$) to yield the generating function of the Bernoulli numbers up to the x^n term. Thus the coefficient of x^n here is B_n.

```
> time B := Laplace(E);
Time: 0.289
> B + O(x^20);
1 - 1/2*x + 1/6*x^2 - 1/30*x^4 + 1/42*x^6 - 1/30*x^8 + 5/66*x^10 -
    691/2730*x^12 + 7/6*x^14 - 3617/510*x^16 + 43867/798*x^18 + O(x^20)
> Coefficient(B, n);
-16596380640568557229852123088077134206658664302806671892352650993155331641220\
960084014956088135770921465025323942809207851857992860213463783252745409096420\
932509953165466735675485979034817619983727209844291081908145597829674980159889\
976244240633746601120703300698329029710482600069717866917229113749797632930033\
559794717838407415772796504419464932337498642714226081743688706971990010734262\
076881238322867559275748219588404488023034528296023051638858467185173202483888\
794342720837413737644410765563213220043477396887812891242952336301344808165757\
942109887803692579439427973561487863524556256869403384306433922049078300720480\
361757680714198044230522015775475287075315668886299978958150756677417180004362\
981454396613646612327019784141740499835461/8365830
> n;
500
```

49.5.2 Trigonometric Functions and their Inverses

```
Sin(f)
```

Given a series f defined over a field of characteristic zero, return the sine of f. The valuation of f must be zero.

```
Cos(f)
```

Given a series f defined over a field of characteristic zero, return the cosine of f. The valuation of f must be zero.

```
Sincos(f)
```

Given a series f defined over a field of characteristic zero, return both the sine and cosine of f. The valuation of f must be zero.

```
Tan(f)
```

Return the tangent of the series f defined over a field.

```
Arcsin(f)
```

Given a series f defined over a field of characteristic zero, return the inverse sine of f.

```
Arccos(f)
```

Given a series f defined over the real or complex field, return the inverse cosine of f.

```
Arctan(f)
```

Given a series f defined over a field of characteristic zero, return the inverse tangent of f.

49.5.3 Hyperbolic Functions and their Inverses

```
Sinh(f)
```

Given a series f defined over a field, return the hyperbolic sine of f.

```
Cosh(f)
```

Given a series f defined over a field, return the hyperbolic cosine of f.

```
Tanh(f)
```

Given a series f defined over a field, return the hyperbolic tangent of f.

```
Argsinh(f)
```

Given a series f defined over a field of characteristic zero, return the inverse hyperbolic sine of f.

```
Argcosh(f)
```

Given a series f defined over the real or complex field, return the inverse hyperbolic cosine of f.

```
Argtanh(f)
```

Given a series f defined over a field of characteristic zero, return the inverse hyperbolic tangent of f.

49.6 The Hypergeometric Series

For more information on the Hypergeometric Series, see [Hus87], page 176.

```
HypergeometricSeries(a,b,c, z)
```

Return the hypergeometric series $F(a, b, c; z)$ defined by

$$F(a,b,c;z) = \sum_{0 \leq n} \frac{(a)_n (b)_n}{n!(c)_n z^n}$$

where $(a)_n = a(a+1) \cdots (a+n-1)$.

49.7 Polynomials over Series Rings

Factorization is available for polynomials over series rings defined over finite fields. We recommend constructing polynomials from sequences rather than by addition of terms, especially over fields, to avoid some precision loss. For example, $x^4 + t$ will not have full precision in the constant coefficient as there will be a O term in the constant coefficient of x^4 which will reduce its precision as a field element. The equivalent polynomial `Polynomial([t, 0, 0, 0, 1])` will have more precision than that constructed as $x^4 + t$.

```
HenselLift(f, L)
```

Given a polynomial f over a series ring over a finite field or an extension of a series ring and a factorization L of f to precision 1 return a factorization of f known to the full precision of the coefficient ring of f.

```
Factorization(f)
```

`Certificates`	BoolElt	*Default :* `false`
`Ideals`	BoolElt	*Default :* `false`
`Extensions`	BoolElt	*Default :* `false`

The factorization of the polynomial f over a power series ring, laurent series field over a finite field or an extension of either into irreducibles.

If `Certificates` is set to `true`, a two-element certificate for each factor, proving its irreducibility, is returned.

If `Ideals` is set to `true`, two generators of some ideal for each factor are returned within the certificates.

If `Extensions` is set to `true`, an extension for each factor is returned within the certificates.

Example H49E4

We illustrate factorizations over series rings and the extensions which can be gained through them.

```
> P<t> := PowerSeriesRing(GF(101));
> R<x> := PolynomialRing(P);
> Factorization(x^5 + t*x^4 - t^2*x^3 + (1 + t^20)*x^2 + t*x + t^6);
[
    <(1 + O(t^20))*x + t^5 + O(t^8), 1>,
    <(1 + O(t^20))*x + t + 100*t^4 + 100*t^5 + t^7 + O(t^8), 1>,
    <(1 + O(t^20))*x + 1 + 34*t^2 + 34*t^3 + 34*t^4 + 90*t^5 + 29*t^6 + 16*t^8 +
        32*t^9 + 6*t^10 + 66*t^11 + 41*t^12 + 93*t^13 + t^14 + 69*t^15 + 8*t^16
        + 61*t^17 + 86*t^18 + 19*t^19 + O(t^20), 1>,
    <(1 + O(t^20))*x^2 + (100 + 67*t^2 + 67*t^3 + 68*t^4 + 11*t^5 + 72*t^6 +
        100*t^7 + 85*t^8 + 69*t^9 + 94*t^10 + 31*t^11 + 59*t^12 + 16*t^13 +
        14*t^14 + 35*t^15 + 77*t^16 + 28*t^17 + 33*t^18 + 72*t^19 + O(t^20))*x +
        1 + 67*t^2 + 68*t^3 + 11*t^4 + 67*t^5 + 57*t^6 + 16*t^7 + 57*t^8 +
        21*t^9 + 68*t^10 + 68*t^11 + 61*t^12 + 98*t^13 + 4*t^14 + 21*t^15 +
        7*t^16 + 95*t^17 + 23*t^18 + 76*t^19 + O(t^20), 1>
]
1 + O(t^20)
> P<t> := PowerSeriesRing(GF(101), 50);
> R<x> := PolynomialRing(P);
> Factorization(x^5 + t*x^4 - t^2*x^3 + (1 + t^20)*x^2 + t*x + t^6 :
> Extensions);
[
    <x + t^5 + t^9 + 2*t^13 + t^16 + 5*t^17 + 6*t^20 + 14*t^21 + 26*t^24 +
        42*t^25 + 3*t^27 + 4*t^28 + 32*t^29 + 35*t^31 + 10*t^32 + 29*t^33 +
        46*t^35 + t^36 + 31*t^37 + O(t^38), 1>,
    <x + t + 100*t^4 + 100*t^5 + t^7 + 100*t^9 + t^10 + 4*t^11 + t^12 + 91*t^13
        + 86*t^14 + 98*t^15 + 15*t^16 + 7*t^17 + 83*t^18 + 10*t^19 + 23*t^20 +
        54*t^21 + 47*t^22 + 72*t^23 + 87*t^24 + 55*t^25 + 4*t^26 + 15*t^27 +
        90*t^28 + 82*t^29 + 97*t^30 + 15*t^31 + 23*t^32 + 86*t^33 + 57*t^34 +
        10*t^35 + 79*t^36 + 52*t^37 + O(t^38), 1>,
    <x + 1 + 34*t^2 + 34*t^3 + 34*t^4 + 90*t^5 + 29*t^6 + 16*t^8 + 32*t^9 +
        6*t^10 + 66*t^11 + 41*t^12 + 93*t^13 + t^14 + 69*t^15 + 8*t^16 + 61*t^17
        + 86*t^18 + 19*t^19 + 47*t^20 + 11*t^21 + 42*t^22 + 38*t^23 + 46*t^24 +
        90*t^25 + 14*t^26 + 7*t^27 + 89*t^28 + 85*t^29 + 70*t^30 + 24*t^31 +
        28*t^32 + 71*t^33 + 53*t^34 + 55*t^35 + 90*t^36 + 26*t^37 + 4*t^38 +
        56*t^39 + 44*t^40 + 8*t^41 + 25*t^42 + 94*t^43 + 14*t^44 + 92*t^45 +
        56*t^46 + 83*t^47 + 26*t^48 + 41*t^49 + O(t^50), 1>,
    <x^2 + (100 + 67*t^2 + 67*t^3 + 68*t^4 + 11*t^5 + 72*t^6 + 100*t^7 + 85*t^8
        + 69*t^9 + 94*t^10 + 31*t^11 + 59*t^12 + 16*t^13 + 14*t^14 + 35*t^15 +
        77*t^16 + 28*t^17 + 33*t^18 + 72*t^19 + 25*t^20 + 22*t^21 + 12*t^22 +
        92*t^23 + 43*t^24 + 15*t^25 + 83*t^26 + 76*t^27 + 19*t^28 + 3*t^29 +
        35*t^30 + 27*t^31 + 40*t^32 + 16*t^33 + 92*t^34 + 91*t^35 + 32*t^36 +
        93*t^37 + 84*t^38 + 98*t^39 + 85*t^40 + 54*t^41 + 25*t^42 + 88*t^43 +
        35*t^44 + 17*t^45 + t^46 + 39*t^47 + 89*t^48 + 67*t^49 + O(t^50))*x + 1
        + 67*t^2 + 68*t^3 + 11*t^4 + 67*t^5 + 57*t^6 + 16*t^7 + 57*t^8 + 21*t^9
        + 68*t^10 + 68*t^11 + 61*t^12 + 98*t^13 + 4*t^14 + 21*t^15 + 7*t^16 +
        95*t^17 + 23*t^18 + 76*t^19 + 62*t^20 + 59*t^21 + 66*t^22 + 35*t^23 +
        41*t^24 + 45*t^25 + 32*t^26 + 56*t^27 + 35*t^28 + 19*t^29 + 21*t^30 +
        59*t^31 + 50*t^32 + 72*t^33 + 58*t^34 + 75*t^35 + 59*t^36 + 76*t^37 +
        83*t^38 + 66*t^39 + 6*t^40 + 8*t^41 + 94*t^42 + 77*t^43 + 100*t^44 +
        30*t^45 + 72*t^46 + 26*t^47 + 54*t^48 + 21*t^49 + O(t^50), 1>
]
1 + O(t^50)
[
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 1,
        Rho := 1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Power series ring in t over GF(101) with fixed absolute
            precision 50>,
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 1,
        Rho := 1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Power series ring in t over GF(101) with fixed absolute
            precision 50>,
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 1,
        Rho := 1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Power series ring in t over GF(101) with fixed absolute
            precision 50>,
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 2,
        Rho := (1 + O($.1^50))*$.1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Extension of Power series ring in t over GF(101) with fixed
            absolute precision 50 by x^2 + (100 + O(t^50))*x + 1 + O(t^50)>
]
```

49.8 Extensions of Series Rings

Extensions of series rings are either unramified or totally ramified. Only series rings defined over finite fields can be extended. We recommend constructing polynomials from sequences rather than by addition of terms, especially over fields, to avoid some precision loss. For example, $x^4 + t$ will not have full precision in the constant coefficient as there will be a O term in the constant coefficient of x^4 which will reduce its precision as a field element. Extensions require full precision polynomials and some polynomials such as $x^4 + t$ may not have enough precision to be used to construct an extension whereas the equivalent `Polynomial([t, 0, 0, 0, 1])` will.

49.8.1 Constructions of Extensions

`UnramifiedExtension(R, f)`

Construct the unramified extension of R, a series ring or an extension thereof, defined by the inertial polynomial f, that is, adjoin a root of f to R.

`TotallyRamifiedExtension(R, f)`

`MaxPrecision` RngIntElt *Default :*

Construct a totally ramified extension of R, a series ring or an extension thereof, defined by the eisenstein polynomial f, that is, adjoin a root of f to R.

The parameter `MaxPrecision` defaults to the precision of R. It can be set to the maximum precision the coefficients of f are known to, which must not be less than the precision of R. This allows the precision of the result to be increased to the degree of f multiplied by this maximum precision.

The polynomial f may be given over a series ring or an extension of a series ring having a higher precision than R. This allows the precision of the result to be increased up to the precision the polynomial is known to (or `MaxPrecision` if set) without losing any of the polynomial known past the precision of R.

The precision of a ramified extension cannot be increased unless the defining polynomial is known to more precision than the coefficient ring, indicated either by providing the polynomial to greater precision or by setting `MaxPrecision`. This should be taken into account when constructing ramified extensions, especially if polynomials are to be factored over the extension.

`ChangePrecision(E, r)`

`ChangePrecision(~E, r)`

The extension of a series ring E with precision r. It is not possible to increase the precision of a ramified extension unless the parameter `MaxPrecision` was set on construction of the extension and r is less than or equal to this value multiplied by the ramification degree or the polynomial used in creating the extension was given to more precision than the coefficient ring of E.

`FieldOfFractions(E)`

The field of fractions of the extension of a series ring E.

Example H49E5

We show a simple creation of a two-step extension and change its precision.

```
> P<t> := PowerSeriesRing(GF(101), 50);
> PP<tt> := PowerSeriesRing(GF(101));
> R<x> := PolynomialRing(PP);
> U := UnramifiedExtension(P, x^2 + 2);
> T := TotallyRamifiedExtension(U, x^2 + tt*x + tt); T;
Extension of Extension of Power series ring in t over GF(101) with fixed
absolute precision 50 by x^2 + 2 + O(t^50) by x^2 + (t + O(t^50))*x + t +
    O(t^50)
> Precision($1);
100
> ChangePrecision($2, 200);
Extension of Extension of Power series ring in $.1 over GF(101) with fixed
absolute precision 100 by x^2 + 2 + O($.1^100) by x^2 + ($.1 + O($.1^100))*x +
    $.1 + O($.1^100)
> ChangePrecision($1, 1000);
Extension of Extension of Power series ring in $.1 over GF(101) with fixed
absolute precision 500 by x^2 + 2 + O($.1^500) by x^2 + ($.1 + O($.1^500))*x +
    $.1 + O($.1^500)
> ChangePrecision($1, 20);
Extension of Extension of Power series ring in $.1 over GF(101) with fixed
absolute precision 10 by x^2 + 2 + O($.1^10) by x^2 + ($.1 + O($.1^10))*x + $.1
    + O($.1^10)
```

Both U and T have a field of fractions.

```
> FieldOfFractions(U);
Extension of Laurent series field in $.1 over GF(101) with fixed relative
precision 50 by (1 + O($.1^50))*x^2 + O($.1^50)*x + 2 + O($.1^50)
> FieldOfFractions(T);
Extension of Extension of Laurent series field in $.1 over GF(101) with fixed
relative precision 50 by (1 + O($.1^50))*x^2 + O($.1^50)*x + 2 + O($.1^50) by (1
    + O($.1^50))*x^2 + ($.1 + O($.1^50))*x + $.1 + O($.1^50)
```

49.8.2 Operations on Extensions

`Precision(E)`

`GetPrecision(E)`

The maximum precision elements of the extension of a series ring E may have.

`CoefficientRing(E)`

`BaseRing(E)`

The ring the extension of a series ring E was defined over.

```
DefiningPolynomial(E)
```

The polynomial used to define the extension E.

```
InertiaDegree(E)
```

The degree of the extension E if E is an unramified extension, 1 otherwise.

```
RamificationIndex(E)
RamificationDegree(E)
```

The degree of the extension E if E is a totally ramified extension, 1 otherwise.

```
ResidueClassField(E)
```

The residue class field of the extension E, that is, $E/\pi * E$.

```
UniformizingElement(E)
```

A uniformizing element for the extension E, that is, an element of E of valuation 1.

```
IntegerRing(E)
Integers(E)
RingOfIntegers(E)
```

The ring of integers of the extension E of a series ring if E is a field (extension of a laurent series ring).

```
E1 eq E2
```

Whether extensions $E1$ and $E2$ are considered to be equal.

```
E . i
```

The primitive element of the extension E, that is, the root of the defining polynomial of E adjoined to the coefficient ring of E to construct E.

```
AssignNames(~E, S)
```

Assign the string in the sequence S to be the name of the primitive element of E, that is, the root of the defining polynomial adjoined to the coefficient ring of E to construct E.

Example H49E6

A number of the operations above are applied to a two-step extension

```
> L<t> := LaurentSeriesRing(GF(53), 30);
> P<x> := PolynomialRing(L);
> U := UnramifiedExtension(L, x^3 + 3*x^2 + x + 4);
> P<y> := PolynomialRing(U);
> T := TotallyRamifiedExtension(U, Polynomial([t, 0, 0, 0, 1]));
> Precision(U);
30
> Precision(T);
120
> CoefficientRing(U);
Laurent series field in t over GF(53) with fixed relative precision 30
> CoefficientRing(T);
Extension of Laurent series field in t over GF(53) with fixed relative precision
30 by (1 + O(t^30))*x^3 + (3 + O(t^30))*x^2 + (1 + O(t^30))*x + 4 + O(t^30)
> DefiningPolynomial(U);
(1 + O(t^30))*x^3 + (3 + O(t^30))*x^2 + (1 + O(t^30))*x + 4 + O(t^30)
> DefiningPolynomial(T);
(1 + O(t^30))*y^4 + O(t^30)*y^3 + O(t^30)*y^2 + O(t^30)*y + t + O(t^30)
> InertiaDegree(U);
3
> InertiaDegree(T);
1
> RamificationDegree(U);
1
> RamificationDegree(T);
4
> ResidueClassField(U);
Finite field of size 53^3
Mapping from: RngSerExt: U to GF(53^3)
> ResidueClassField(T);
Finite field of size 53^3
Mapping from: RngSerExt: T to GF(53^3)
> UniformizingElement(U);
t + O(t^31)
> UniformizingElement(T);
(1 + O(t^30))*$.1 + O(t^30)
> Integers(T);
Extension of Extension of Power series ring in $.1 over GF(53) with fixed
absolute precision 30 by x^3 + (3 + O($.1^30))*x^2 + x + 4 + O($.1^30) by x^4 +
    $.1 + O($.1^30)
> U.1;
(1 + O(t^30))*$.1 + O(t^30)
> T.1;
```

```
(1 + O(t^30))*$.1 + O(t^30)
```

49.8.3 Elements of Extensions

`x * y` `x + y` `x - y` `- x` `x ^ n` `x div y` `x / y`

`x eq y` `IsZero(e)` `IsOne(e)` `IsMinusOne(e)` `IsUnit(e)`

`Valuation(e)`

The valuation of the element e of an extension of a series ring. This is the index of the largest power of π which divides e.

`RelativePrecision(e)`

The relative precision of the element e of an extension of a series ring. This is the number of digits of e (in π) which are known.

`AbsolutePrecision(e)`

The absolute precision of the element e of an extension of a series ring. This is the same as the sum of the relative precision of e and the valuation of e.

`Coefficients(e)`

`Eltseq(e)`

`ElementToSequence(e)`

Given an element e of an extension of a series ring, return the coefficients of e with respect to the powers of the uniformizing element of the extension.

Example H49E7

We show some simple arithmetic with some elements of some extensions.

```
> P<t> := PowerSeriesRing(GF(101), 50);
> R<x> := PolynomialRing(P);
> U<u> := UnramifiedExtension(P, x^2 + 2*x + 3);
> UF := FieldOfFractions(U);
> R<y> := PolynomialRing(U);
> T<tt> := TotallyRamifiedExtension(U, y^2 + t*y + t);
> TF<tf> := FieldOfFractions(T);
> UF<uf> := FieldOfFractions(U);
> u + t;
u + t + O(t^50)
> uf * t;
($.1 + O($.1^50))*uf + O($.1^51)
> tf eq tt;
false
> tf - tt;
```

```
O($.1^50)*tf + O($.1^50)
> IsZero($1);
false
> Valuation($2);
100
> Valuation(tt);
1
> Valuation(U!t);
1
> Valuation(T!t);
2
> RelativePrecision(u);
50
> AbsolutePrecision(u);
50
> AbsolutePrecision(uf);
50
> RelativePrecision(uf);
50
> RelativePrecision(u - uf);
0
> AbsolutePrecision(u - uf);
50
> u^7;
(13 + O(t^50))*u + 71 + O(t^50)
> Coefficients($1);
[
    71 + O(t^50),
    13 + O(t^50)
]
> tt^8;
(4*t^4 + 91*t^5 + 6*t^6 + 100*t^7 + O(t^50))*tt + t^4 + 95*t^5 + 5*t^6 + 100*t^7
    + O(t^50)
> Coefficients($1);
[ t^4 + 95*t^5 + 5*t^6 + 100*t^7 + O(t^50), 4*t^4 + 91*t^5 + 6*t^6 + 100*t^7 +
    O(t^50) ]
```

49.8.4 Optimized Representation

```
OptimizedRepresentation(E)
```

```
OptimisedRepresentation(E)
```

An optimized representation for the unramified extension E of a series ring. The defining polynomial of E must be coercible into the residue class field. The result is a series ring over the residue class field of E and a map from E to this series ring.

Example H49E8

We give a simple example of the use of `OptimizedRepresentation`.

```
> P<t> := PowerSeriesRing(GF(101), 50);
> R<x> := PolynomialRing(P);
> U<u> := UnramifiedExtension(P, x^2 + 2*x + 3);
> U;
Extension of Power series ring in t over GF(101) with fixed absolute precision
50 by x^2 + (2 + O(t^50))*x + 3 + O(t^50)
> OptimizedRepresentation(U);
Power series ring in $.1 over GF(101^2) with fixed absolute precision 50
Mapping from: RngSerExt: U to Power series ring in s over GF(101^2) with fixed
absolute precision 50 given by a rule
```

49.9 Bibliography

[**Hus87**] Dale Husemöller. *Elliptic Curves*, volume 111 of *Graduate Texts in Mathematics*. Springer, New York, 1987.

50 LAZY POWER SERIES RINGS

Chapter 50

LAZY POWER SERIES RINGS

50.1 Introduction

The lazy series rings RngLaz and their elements RngLazElt allow the creation of infinite precision series by providing series for which all coefficients can be calculated by some given formula. Only finitely many coefficients of such a series can be known at any one time but all infinitely many of the coefficients are knowable. Any coefficient of a lazy series can be generated and once a coefficient is computed it will be stored in the series for quick retrieval.

The simplest implementation of this idea is creating a series by providing a map. This map is a formula for computing coefficients of a series. Given the exponents of the variables of a term it will give you the coefficient of that term. Series with finitely many non–zero terms can be created in special ways and several usual arithmetic operations can be applied to lazy series. Such constructions yield lazy series with more complicated formulas for their coefficients.

Consider the series $\sum_{i=0}^{n} i * x^i$. A formula for the coefficients is given by $i \mapsto i$. This series can be created as a lazy series as follows:

```
> L<x> := LazyPowerSeriesRing(Integers(), 1);
> m := map<Integers() -> Integers() | i :-> i>;
> s := elt<L | m>;
> s;
Lazy power series
```

Currently s does not know what any of its coefficients are yet it is possible to calculate any coefficient of s.

```
> Coefficient(s, 0);
0
> Coefficient(s, 100);
100
> Coefficient(s, 100000000000000000000000);
100000000000000000000000
```

50.2 Creation of Lazy Series Rings

Both univariate and multivariate lazy series rings can be created.

`LazyPowerSeriesRing(C, n)`

The lazy power series ring with coefficient ring C and n variables. Any ring is valid input for C and n can be any positive integer.

`ChangeRing(L, C)`

Given a lazy series ring L defined over a ring R and some ring C, return the lazy series ring with coefficient ring C but the same number of variables as L. A map from L to the new lazy series ring is also returned which takes an series s in L to a series whose coefficients are those of s coerced into C.

Example H50E1

Here we illustrate the creation and printing of lazy power series rings.

```
> L := LazyPowerSeriesRing(Rationals(), 5);
> L;
Lazy power series ring in 5 variables over Rational Field
> ChangeRing(L, MaximalOrder(CyclotomicField(7)));
Lazy power series ring in 5 variables over Maximal Equation Order with defining
polynomial x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 over Z
```

50.3 Functions on Lazy Series Rings

Lazy series rings have variables, names for their variables and a coefficient ring.

`R . i`

The ith variable of the lazy power series ring R, where i is between 1 and the rank of R.

`AssignNames(~R, S)`

Given a lazy series ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

`BaseRing(R)`

`CoefficientRing(R)`

The coefficient ring of the lazy power series ring R. The coefficients of all the series in R will lie in this ring.

`Rank(R)`

The number of variables associated with the lazy power series ring R.

`R1 eq R2`

Return `true` if the lazy series rings $R1$ and $R2$ are the same ring, that is, they have the same coefficient ring and rank.

Example H50E2

The functions on lazy power series rings are illustrated here.

```
> L := LazyPowerSeriesRing(FiniteField(73), 7);
> L.4;
Lazy power series
> AssignNames(~L, ["a", "b", "c", "d", "fifth", "sixth", "seventh"]);
> L.4;
Lazy power series
```

The names for the variables of L are not used in default series printing. However they are used when printing an element to a given precision using PrintToPrecision.

```
> CoefficientRing(L);
Finite field of size 73
> Rank(L);
7
> L eq LazyPowerSeriesRing(CoefficientRing(L), Rank(L));
true
```

50.4 Elements

Lazy series may be created in a number of ways. Arithmetic for ring elements is available as well as a few predicates. Coefficients of the monomials in the series can be calculated.

50.4.1 Creation of Finite Lazy Series

`R ! c`

Return the series in the lazy series ring R with constant term c and every other coefficient 0 where c is any ring element coercible into the coefficient ring of R.

`R ! s`

Return the lazy series in the lazy series ring R whose coefficients are those of the lazy series s coerced into the coefficient ring of R.

`R ! S`

Return the series in the lazy series ring R whose coefficients are the elements of S where S is a sequence of elements each coercible into the coefficient ring of R. The coefficients are taken to be given in the order which Coefficients will return them and PrintToPrecision will print the terms of the series. Any coefficient not given is assumed to be zero so that all coefficients are calculable. The resulting series can only have finitely many non zero terms and all such non zero coefficients must be given in S.

```
LazySeries(R, f)
```

Create the lazy series in the lazy series ring R with finitely many non zero terms which are the terms of the polynomial f. The number of variables of the parent ring of f must be the same as the number of variables of R. The coefficients of f must be coercible into the coefficient ring of R.

It is also possible for f to be a rational function, p/q. The series created from f will be `LazySeries(R, p)*LazySeries(R, q)`$^{-1}$.

Example H50E3

Creation of series using the above functions is shown here.

```
> L := LazyPowerSeriesRing(AlgebraicClosure(), 3);
> LR := LazyPowerSeriesRing(Rationals(), 3);
> s := L!1;
> s;
Lazy power series
> LR!s;
Lazy power series
> P<x, y, z> := RationalFunctionField(Rationals(), 3);
> LazySeries(L, (x + y + 8*z)^7/(1 + 5*x*y*z + x^8)^3);
Lazy power series
```

50.4.1.1 Creation of Lazy Series from Maps

Creating a lazy series from a map simulates the existence of lazy series as series for which coefficients are not known on creation of the series but calculated by some rule given when creating the series.

This rule can be coded in a map where the input to the map is the exponents of the variables in a term and the output is the coefficient of the term described by the input. So for a lazy series ring with variables x, y and z, a general term of a series will look like $c_{ijk} * x_i^r * y_j^s * z_k^t$. The coefficient of this term, c_{ijk}, in a series defined by a map m is the result of $m(\langle i, j, k\rangle)$.

```
elt< R | m >
```

Creates a series in the lazy series ring R from the map m. The map m must take as input either an integer (when R is univariate only) or a tuple of integers (of length the number of variables of R) and return an element of the coefficient ring of R. This element will be taken as the coefficient of the term of the series whose variables have the exponents given in the input tuple, as described above.

Example H50E4

We first illustrate the univariate case.

```
> L<x> := LazyPowerSeriesRing(MaximalOrder(QuadraticField(5)), 1);
> Z := Integers();
> m := map<Z -> CoefficientRing(L) | t :-> 2*t>;
> s := elt<L | m>;
> PrintToPrecision(s, 10);
2*x + 4*x^2 + 6*x^3 + 8*x^4 + 10*x^5 + 12*x^6 + 14*x^7 + 16*x^8 + 18*x^9 +
    20*x^10
> Coefficient(s, 34);
68
> m(34);
68
> Coefficient(s, 2^30 + 10);
2147483668
> m(2^30 + 10);
2147483668
```

Example H50E5

And now for the multivariate case.

```
> L<x, y, z> := LazyPowerSeriesRing(AlgebraicClosure(), 3);
> Z := Integers();
> m := map<car<Z, Z, Z> -> CoefficientRing(L) | t :-> t[1]*t[2]*t[3]>;
> s := elt<L | m>;
> PrintToPrecision(s, 5);
x*y*z + 2*x^2*y*z + 2*x*y^2*z + 2*x*y*z^2 + 3*x^3*y*z + 4*x^2*y^2*z +
    4*x^2*y*z^2 + 3*x*y^3*z + 4*x*y^2*z^2 + 3*x*y*z^3
> Coefficient(s, [1, 1, 1]);
1
> m(<1, 1, 1>);
1
> Coefficient(s, [3, 1, 2]);
6
> m(<3, 1, 2>);
6
```

50.4.2 Arithmetic with Lazy Series

All the usual arithmetic operations are possible for lazy series.

```
s + t
```

The sum of the two lazy series s and t.

```
-s
```

The negation of the lazy series s.

```
s - t
```

The difference between lazy series s and t.

```
s * t
```

The product of the lazy series s and t.

```
s + r
r + s
```

The sum of the lazy series s and the element r of the coefficient ring of the parent ring of s.

```
c * s
s * c
```

The product of the lazy series s and the element c of the coefficient ring of the parent ring of s.

```
s * n
```

The product of the lazy series s and the monomial x^n, where x^n is $x_1^{n_1} \times \ldots \times x_r^{n_r}$ where r is the number of variables of the parent ring of s, n is the sequence $[n_1, \ldots, n_r]$ and $x = x_1, \ldots, x_r$ are the series variables of the parent ring of s.

```
s ^ n
```

Given a lazy series s and an integer n, return the nth power of s. It is allowed for n to be negative and inverses will be taken where possible.

Example H50E6

Here we demonstrate the above arithmetic operations.

```
> L<a, b, c, d> := LazyPowerSeriesRing(Rationals(), 4);
> (a + 4*b + (-c)*[8, 9, 2^30 + 10, 2] - d*b + 5)^-8;
Lazy power series
```

50.4.3 Finding Coefficients of Lazy Series

In theory, all the coefficients of most series can be calculated. In practice it is not possible to compute infinitely many. Some problems arise however when a series has been created using multiplication, inversion, evaluation or by taking square roots. For such series the jth coefficient for all $j < i$ must be known for the ith coefficient to be calculated. In such cases the exponents specified for each variable in the monomial whose coefficient is required must be small integers ($< 2^{30}$).

The default ordering of coefficients is by total degree of the corresponding monomials. This is the same order on multivariate polynomials by default. When drawn on paper or imagined in 3 or more dimensions this looks like a spiral if the coordinates representing the monomial exponents are joined. This is the same ordering which is used to compute the valuation of a series.

Once computed, coefficients are stored with the series so any subsequent call to these functions will be faster in non–trivial cases than the first call. It is possible to interrupt any of these functions and return to the prompt.

```
Coefficient(s, i)
```

Returns the coefficient in the univariate lazy series s of x^i where x is the series variable of the parent of s and i is a non negative integer.

```
Coefficient(s, T)
```

Returns the coefficient in the multivariate lazy series s of the monomial $x_1^{T_1} * \ldots * x_r^{T_r}$ where T is the sequence $[T_1, \ldots, T_r]$, $x_1, \ldots, x_r$ are the variables of the parent ring of s and r is the rank of the parent of s.

```
Coefficients(s, n)
```

```
Coefficients(s, l, n)
```

The coefficients of the lazy series s whose monomials have total degree at least l (0 if not given) and at most n where the monomials are ordered using the default "spiral" degree order. The bounds l and n must be non negative integers.

```
Valuation(s)
```

The valuation of the lazy series s. This is the exponent of the monomial with the first non–zero coefficient as returned by `Coefficients` above. The return value will be either `Infty`, an integer (univariate case) or a sequence (multivariate case).

```
PrintToPrecision(s, n)
```

Print the sum of all terms of the lazy series s whose degree is no more than n where n is a non negative integer. The series is printed using the "spiral" ordering.

```
PrintTermsOfDegree(s, l, n)
```

Print the sum of the terms of the lazy series s whose degree is at least l and at most n. The terms are printed using the spiral ordering. The bounds l and n must be non negative integers.

```
LeadingCoefficient(s)
```

The coefficient in the lazy series s whose monomial exponent is the valuation of s. That is, the first non–zero coefficient of s where the ordering which determines "first" is the "spiral" ordering used by `Coefficients` and `Valuation`.

```
LeadingTerm(s)
```

The term in the lazy series s whose monomial exponent is the valuation of s. That is, the first non–zero term of s where the ordering which determines "first" is the "spiral" ordering used by `Coefficients` and `Valuation`.

Example H50E7

In this example we look at some coefficients of an infinite series.

```
> L<a, b, c, d> := LazyPowerSeriesRing(Rationals(), 4);
> s := (1 + 2*a + 3*b + 4*d)^-5;
```

Find the coefficient of $a * b * c * d$.

```
> Coefficient(s, [1, 1, 1, 1]);
0
```

Find the coefficients of all monomials with total degree at most 6.

```
> time Coefficients(s, 6);
[ 1, -10, -15, 0, -20, 60, 180, 0, 240, 135, 0, 360, 0, 0, 240, -280, -1260, 0,
-1680, -1890, 0, -5040, 0, 0, -3360, -945, 0, -3780, 0, 0, -5040, 0, 0, 0,
-2240, 1120, 6720, 0, 8960, 15120, 0, 40320, 0, 0, 26880, 15120, 0, 60480, 0, 0,
80640, 0, 0, 0, 35840, 5670, 0, 30240, 0, 0, 60480, 0, 0, 0, 53760, 0, 0, 0, 0,
17920, -4032, -30240, 0, -40320, -90720, 0, -241920, 0, 0, -161280, -136080, 0,
-544320, 0, 0, -725760, 0, 0, 0, -322560, -102060, 0, -544320, 0, 0, -1088640,
0, 0, 0, -967680, 0, 0, 0, 0, -322560, -30618, 0, -204120, 0, 0, -544320, 0, 0,
0, -725760, 0, 0, 0, 0, -483840, 0, 0, 0, 0, 0, -129024, 13440, 120960, 0,
161280, 453600, 0, 1209600, 0, 0, 806400, 907200, 0, 3628800, 0, 0, 4838400, 0,
0, 0, 2150400, 1020600, 0, 5443200, 0, 0, 10886400, 0, 0, 0, 9676800, 0, 0, 0,
0, 3225600, 612360, 0, 4082400, 0, 0, 10886400, 0, 0, 0, 14515200, 0, 0, 0, 0,
9676800, 0, 0, 0, 0, 0, 2580480, 153090, 0, 1224720, 0, 0, 4082400, 0, 0, 0,
7257600, 0, 0, 0, 0, 7257600, 0, 0, 0, 0, 0, 3870720, 0, 0, 0, 0, 0, 0, 860160 ]
Time: 0.140
> #$1;
210
```

`PrintToPrecision` will display the monomials to which these coefficients correspond.

```
> time PrintToPrecision(s, 6);
1 - 10*a - 15*b - 20*d + 60*a^2 + 180*a*b + 240*a*d + 135*b^2 + 360*b*d +
    240*d^2 - 280*a^3 - 1260*a^2*b - 1680*a^2*d - 1890*a*b^2 - 5040*a*b*d -
    3360*a*d^2 - 945*b^3 - 3780*b^2*d - 5040*b*d^2 - 2240*d^3 + 1120*a^4 +
    6720*a^3*b + 8960*a^3*d + 15120*a^2*b^2 + 40320*a^2*b*d + 26880*a^2*d^2 +
    15120*a*b^3 + 60480*a*b^2*d + 80640*a*b*d^2 + 35840*a*d^3 + 5670*b^4 +
    30240*b^3*d + 60480*b^2*d^2 + 53760*b*d^3 + 17920*d^4 - 4032*a^5 -
    30240*a^4*b - 40320*a^4*d - 90720*a^3*b^2 - 241920*a^3*b*d - 161280*a^3*d^2
    - 136080*a^2*b^3 - 544320*a^2*b^2*d - 725760*a^2*b*d^2 - 322560*a^2*d^3 -
    102060*a*b^4 - 544320*a*b^3*d - 1088640*a*b^2*d^2 - 967680*a*b*d^3 -
    322560*a*d^4 - 30618*b^5 - 204120*b^4*d - 544320*b^3*d^2 - 725760*b^2*d^3 -
    483840*b*d^4 - 129024*d^5 + 13440*a^6 + 120960*a^5*b + 161280*a^5*d +
    453600*a^4*b^2 + 1209600*a^4*b*d + 806400*a^4*d^2 + 907200*a^3*b^3 +
    3628800*a^3*b^2*d + 4838400*a^3*b*d^2 + 2150400*a^3*d^3 + 1020600*a^2*b^4 +
    5443200*a^2*b^3*d + 10886400*a^2*b^2*d^2 + 9676800*a^2*b*d^3 +
    3225600*a^2*d^4 + 612360*a*b^5 + 4082400*a*b^4*d + 10886400*a*b^3*d^2 +
    14515200*a*b^2*d^3 + 9676800*a*b*d^4 + 2580480*a*d^5 + 153090*b^6 +
    1224720*b^5*d + 4082400*b^4*d^2 + 7257600*b^3*d^3 + 7257600*b^2*d^4 +
    3870720*b*d^5 + 860160*d^6
Time: 0.010
```

The valuation of s can be obtained as follows. The valuation of zero is a special case.

```
> Valuation(s);
[ 0, 0, 0, 0 ]
> Valuation(s*0);
Infinity
```

`CoefficientsNonSpiral(s, n)`

Returns the coefficients of the monomials in the lazy series s whose exponents are given by $[i_1, ..., i_r]$ where each $i_j \leq n_j$. The argument n may either be a non negative integer (univariate case) or a sequence of non negative integers of length r where r is the rank of the parent ring of s. The index of the $[i_1, ..., i_r]$-th coefficient in the return sequence is given by

$$\sum_{j=1}^{r} i_j * (\prod_{k=j+1}^{r} (n_k + 1)).$$

`Index(s, i, n)`

Return the index in the return value of `CoefficientsNonSpiral(s, n)` of the monomial in the lazy series s whose exponents are given by the (trivial in the univariate case) sequence i.

Example H50E8

We find the coefficients of the series used in the last example using the alternative algorithm.

```
> L<a, b, c, d> := LazyPowerSeriesRing(Rationals(), 4);
> s := (1 + 2*a + 3*b + 4*d)^-5;
> time CoefficientsNonSpiral(s, [3, 3, 3, 2]);
[ 1, -20, 240, 0, 0, 0, 0, 0, 0, 0, 0, 0, -15, 360, -5040, 0, 0, 0, 0, 0, 0, 0,
0, 0, 135, -3780, 60480, 0, 0, 0, 0, 0, 0, 0, 0, 0, -945, 30240, -544320, 0, 0,
0, 0, 0, 0, 0, 0, 0, -10, 240, -3360, 0, 0, 0, 0, 0, 0, 0, 0, 0, 180, -5040,
80640, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1890, 60480, -1088640, 0, 0, 0, 0, 0, 0, 0,
0, 0, 15120, -544320, 10886400, 0, 0, 0, 0, 0, 0, 0, 0, 0, 60, -1680, 26880, 0,
0, 0, 0, 0, 0, 0, 0, 0, -1260, 40320, -725760, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15120,
-544320, 10886400, 0, 0, 0, 0, 0, 0, 0, 0, 0, -136080, 5443200, -119750400, 0,
0, 0, 0, 0, 0, 0, 0, 0, -280, 8960, -161280, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6720,
-241920, 4838400, 0, 0, 0, 0, 0, 0, 0, 0, 0, -90720, 3628800, -79833600, 0, 0,
0, 0, 0, 0, 0, 0, 0, 907200, -39916800, 958003200, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
Time: 0.370
> #$1;
192
```

It appears that the "spiral" algorithm may be faster since it computed more coefficients in less time. This may be because of the operations (inversion) involved in calculating s and that `CoefficientsNonSpiral` computed coefficients of larger degree monomials than `Coefficients`. In calculating these coefficients for larger degrees, coefficients which were not asked for may have been calculated. `Coefficients` will only need to calculate the coefficients which have been asked for — all intermediate calculations would have been asked for.

50.4.4 Predicates on Lazy Series

`s eq t`

Return `true` if the lazy series s and t are exactly the same series.

`IsZero(s)`

Return `true` if the lazy series s was created as the zero series.

`IsOne(s)`

Return `true` if the lazy series s was created as the one series.

`IsMinusOne(s)`

Return `true` if the lazy series s was created as the minus one series.

`IsUnit(s)`

Return `true` if the lazy series s is a unit.

`IsWeaklyZero(s, n)`

Return `true` if all the terms of the lazy series s with degree at most n are zero. Calling this function without the second argument returns `IsZero(s)`.

`IsWeaklyEqual(s, t, n)`

Return `true` if the terms of the lazy series s and t with degree at most n are the same. Calling this function without the third argument returns `s eq t`.

50.4.5 Other Functions on Lazy Series

`Derivative(s)`

`Derivative(s, v)`

`Derivative(s, v, n)`

Return the (nth) derivative of the lazy series s with respect to the vth variable of the parent ring of s. If v is not given, the parent ring of s must be univariate and the derivative with respect to the unique variable is returned. It is only allowed to give values of v from 1 to the rank of the parent of s and for n to be a positive integer.

`Integral(s)`

`Integral(s, v)`

Return the integral of the lazy series s with respect to the vth variable of the parent ring of s. If v is not given, the parent ring of s must be univariate and the integral with respect to the unique variable is returned. It is only allowed to give values of v from 1 to the rank of the parent of s.

`Evaluate(s, t)`

`Evaluate(s, T)`

Return the lazy series s evaluated at the lazy series t or the sequence T of lazy series. The series t or the series in T must have zero constant term so that every coefficient of the result can be finitely calculated.

Example H50E9

Some usage of `Evaluate` is shown below.

```
> R := LazyPowerSeriesRing(Rationals(), 2);
> AssignNames(~R, ["x","y"]);
> m := map<car<Integers(), Integers()> -> Rationals() | t :-> 1>;
> s := elt<R | m>;
> PrintToPrecision(s, 3);
1 + x + y + x^2 + x*y + y^2 + x^3 + x^2*y + x*y^2 + y^3
> R1 := LazyPowerSeriesRing(Rationals(), 1);
> AssignNames(~R1, ["z"]);
> m1 := map<car<Integers()> -> Rationals() | t :-> t[1]>;
> s1 := elt<R1 | m1>;
> PrintToPrecision(s1, 3);
z + 2*z^2 + 3*z^3
> e := Evaluate(s, [s1,s1]);
> PrintToPrecision(e, 10);
1 + 2*z + 7*z^2 + 22*z^3 + 67*z^4 + 200*z^5 + 588*z^6 + 1708*z^7 + 4913*z^8 +
    14018*z^9 + 39725*z^10
> Parent(e);
Lazy power series ring in 1 variable over Rational Field
> f := Evaluate(s1, s - 1);
> PrintToPrecision(f, 10);
x + y + 3*x^2 + 5*x*y + 3*y^2 + 8*x^3 + 18*x^2*y + 18*x*y^2 + 8*y^3 + 20*x^4 +
    56*x^3*y + 75*x^2*y^2 + 56*x*y^3 + 20*y^4 + 48*x^5 + 160*x^4*y + 264*x^3*y^2
    + 264*x^2*y^3 + 160*x*y^4 + 48*y^5 + 112*x^6 + 432*x^5*y + 840*x^4*y^2 +
    1032*x^3*y^3 + 840*x^2*y^4 + 432*x*y^5 + 112*y^6 + 256*x^7 + 1120*x^6*y +
    2496*x^5*y^2 + 3600*x^4*y^3 + 3600*x^3*y^4 + 2496*x^2*y^5 + 1120*x*y^6 +
    256*y^7 + 576*x^8 + 2816*x^7*y + 7056*x^6*y^2 + 11616*x^5*y^3 +
    13620*x^4*y^4 + 11616*x^3*y^5 + 7056*x^2*y^6 + 2816*x*y^7 + 576*y^8 +
    1280*x^9 + 6912*x^8*y + 19200*x^7*y^2 + 35392*x^6*y^3 + 47280*x^5*y^4 +
    47280*x^4*y^5 + 35392*x^3*y^6 + 19200*x^2*y^7 + 6912*x*y^8 + 1280*y^9 +
    2816*x^10 + 16640*x^9*y + 50688*x^8*y^2 + 103168*x^7*y^3 + 154000*x^6*y^4 +
    175344*x^5*y^5 + 154000*x^4*y^6 + 103168*x^3*y^7 + 50688*x^2*y^8 +
    16640*x*y^9 + 2816*y^10
> f := Evaluate(s1 + 1, s - 1);
> PrintToPrecision(f, 10);
1 + x + y + 3*x^2 + 5*x*y + 3*y^2 + 8*x^3 + 18*x^2*y + 18*x*y^2 + 8*y^3 + 20*x^4
    + 56*x^3*y + 75*x^2*y^2 + 56*x*y^3 + 20*y^4 + 48*x^5 + 160*x^4*y +
    264*x^3*y^2 + 264*x^2*y^3 + 160*x*y^4 + 48*y^5 + 112*x^6 + 432*x^5*y +
    840*x^4*y^2 + 1032*x^3*y^3 + 840*x^2*y^4 + 432*x*y^5 + 112*y^6 + 256*x^7 +
    1120*x^6*y + 2496*x^5*y^2 + 3600*x^4*y^3 + 3600*x^3*y^4 + 2496*x^2*y^5 +
    1120*x*y^6 + 256*y^7 + 576*x^8 + 2816*x^7*y + 7056*x^6*y^2 + 11616*x^5*y^3 +
    13620*x^4*y^4 + 11616*x^3*y^5 + 7056*x^2*y^6 + 2816*x*y^7 + 576*y^8 +
    1280*x^9 + 6912*x^8*y + 19200*x^7*y^2 + 35392*x^6*y^3 + 47280*x^5*y^4 +
    47280*x^4*y^5 + 35392*x^3*y^6 + 19200*x^2*y^7 + 6912*x*y^8 + 1280*y^9 +
    2816*x^10 + 16640*x^9*y + 50688*x^8*y^2 + 103168*x^7*y^3 + 154000*x^6*y^4 +
    175344*x^5*y^5 + 154000*x^4*y^6 + 103168*x^3*y^7 + 50688*x^2*y^8 +
    16640*x*y^9 + 2816*y^10
```

`SquareRoot(s)`

`Sqrt(s)`

The square root of the lazy series s.

```
IsSquare(s)
```

Return `true` if the lazy series s is a square and the square root if so.

```
PolynomialCoefficient(s, i)
```

Given a series s in a lazy series ring whose coefficient ring is a polynomial ring (either univariate or multivariate), consider the polynomial which would be formed if this series was written as a polynomial with series as the coefficients. For i a non negative integer and the coefficient ring of the series ring a univariate polynomial ring this function returns the series which is the ith coefficient of the polynomial resulting from the rewriting of the series.

When the coefficient ring of the parent of s is a multivariate polynomial ring i should be a sequence of non negative integers of length the rank of the coefficient ring. The series returned will be the series which is the coefficient of $x_1^{i_1} * \ldots * x_r^{i_r}$ in the rewritten series where r is the rank of the coefficient ring and x_j are the indeterminates of the coefficient ring.

51 GENERAL LOCAL FIELDS

Chapter 51
GENERAL LOCAL FIELDS

51.1 Introduction

The local fields described in this chapter are extensions of any local field in Magma by any irreducible polynomial over that field. They are not constrained by requirements that the polynomial defining the extension be inertial or eisenstein. These local fields allow for ramified and inertial extensions to be made in one step rather than forcing such an extension to be split into two – being a ramified extension and an unramified extension. They are typed `RngLocA` with elements of type `RngLocAElt`. To compare these fields to the local fields where extensions are either totally ramified or unramified, see Chapter 47.

The fields are represented as a polynomial quotient ring. A map into an isomorphic `FldPad` can be constructed and the isomorphic field used for various calculations.

51.2 Constructions

Local fields can be constructed as extensions of other local fields and as subfields of other local fields.

`LocalField(L, f)`

Construct a local field F as an extension of the local field L by the polynomial f over L.

Example H51E1

We can use the 2 step local fields to help us find an irreducible polynomial over a p-adic field which can be used to extend that p-adic field in 1 step.

```
> P<x> := PolynomialRing(Integers());
> Zp := pAdicRing(7, 50);
> U := UnramifiedExtension(Zp, x^2 + 6*x + 3);
> R := TotallyRamifiedExtension(U, x^3 + 7*x^2 + 7*x + 7);
> L<a> := LocalField(pAdicField(7, 50), MinimalPolynomial(R.1 + U.1, Zp));
> L;
Extension of 7-adic field mod 7^50 by x^6 + (32 + O(7^50))*x^5 + (390 +
    O(7^50))*x^4 + (2284 + O(7^50))*x^3 + (6588 + O(7^50))*x^2 + (8744 +
    O(7^50))*x + 5452 + O(7^50)
```

We can also use any irreducible polynomial we can find to define a 1 step extension.

```
> LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
Extension of 7-adic field mod 7^50 by x^6 + O(7^50)*x^5 + O(7^50)*x^4 +
    O(7^50)*x^3 - (7^2 + O(7^52))*x^2 + O(7^50)*x + 2*7^3 + O(7^53)
```

sub< L | $a_1, ..., a_n$ >

sub< L | S >

Construct the local field F as a subfield of the local field L containing the elements a_i of L or the elements of the sequence S.

Example H51E2

```
> Qp := pAdicField(5, 20);
> P<x> := PolynomialRing(Qp);
> L<a> := LocalField(Qp, x^4 + 4*x^2 + 2);
> P<x> := PolynomialRing(L);
> LL<aa> := LocalField(L, x^4 + 4*L.1);
> r := (10236563738184*a^3 - 331496727861*a^2 + 10714284669258*a +
> 8590525712453*5)*aa^2
> -12574685904653*a^3 + 19786544763736*a^2 + 4956446023134*a + 37611818678747;
> S, m := sub< LL | r >;
> S;
Extension of Extension of 5-adic field mod 5^20 by x^4 + O(5^20)*x^3 + (4 +
    O(5^20))*x^2 + (4 + O(5^20))*x + 2 + O(5^20) by (O(5^20)*a^3 + O(5^20)*a^2 +
    O(5^20)*a + (1 + O(5^20)))*x^2 + (O(5^20)*a^3 + O(5^20)*a^2 + O(5^20)*a +
    O(5^20))*x + O(5^20)*a^3 + O(5^20)*a^2 + (4 + O(5^20))*a + O(5^20)
> m(S.1);
(O(5^20)*a^3 + O(5^20)*a^2 + O(5^20)*a + O(5^20))*aa^3 + (O(5^20)*a^3 +
    O(5^20)*a^2 + O(5^20)*a + (1 + O(5^20)))*aa^2 + (O(5^20)*a^3 + O(5^20)*a^2 +
    O(5^20)*a + O(5^20))*aa + O(5^20)*a^3 + O(5^20)*a^2 + O(5^20)*a + O(5^20)
```

51.3 Operations with Fields

BaseRing(L)

CoefficientRing(L)

Return the coefficient field of the local field L. This is the field which was extended to construct L.

DefiningPolynomial(L)

Return the polynomial used to define the local field L as an extension of its coefficient field.

Degree(L)

Return the degree of the local field L, that is, the degree of its defining polynomial.

Degree(L, R)

Return the degree of L as an extension of R where R is some coefficient ring of L.

`InertiaDegree(L)`

`RamificationDegree(L)`

`RamificationIndex(L)`

Return the degree of the inertial or totally ramified subfield of the local field L as an extension of the coefficient field of L.

`Precision(L)`

Return the precision of the local field L. This is the maximum number of digits which can occur in an element of L, the difference between the valuation of an element of L and the valuation of the term of highest valuation occurring in that element.

`Prime(L)`

Return the prime of the local field L. This is the same as the prime of the coefficient field of L.

Example H51E3

Continuing from the first example we have :

```
> CoefficientRing(L);
7-adic field mod 7^50
> DefiningPolynomial(L);
$.1^6 + (32 + O(7^50))*$.1^5 + (390 + O(7^50))*$.1^4 + (2284 + O(7^50))*$.1^3 +
    (6588 + O(7^50))*$.1^2 + (8744 + O(7^50))*$.1 + 5452 + O(7^50)
> Precision(L);
150
> Prime(L);
7
> Degree(L); RamificationDegree(L); InertiaDegree(L);
6
3
2
```

`QuotientRepresentation(L)`

Return the polynomial quotient ring which is isomorphic to the local field L and is used to represent L.

`RamifiedRepresentation(L)`

Return the local field isomorphic to the local field L constructed as an unramified then a ramified extension and the map from L into the isomorphic field.

Example H51E4

We create a 1 step local field and compute its representation as a 2 step local field.

```
> P<x> := PolynomialRing(Integers());
> L<a> := LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
> L;
Extension of 7-adic field mod 7^50 by x^6 + O(7^50)*x^5 + O(7^50)*x^4 +
    O(7^50)*x^3 - (7^2 + O(7^52))*x^2 + O(7^50)*x + 2*7^3 + O(7^53)
> QuotientRepresentation(L);
Univariate Quotient Polynomial Algebra in $.1 over 7-adic field mod 7^50
with modulus $.1^6 + O(7^50)*$.1^5 + O(7^50)*$.1^4 + O(7^50)*$.1^3 - (7^2 +
    O(7^52))*$.1^2 + O(7^50)*$.1 + 2*7^3 + O(7^53)
> RR, m := RamifiedRepresentation(L);
> RR;
Totally ramified extension defined by the polynomial x^2 +
    417092732355694537113348201703437033788663*$.1^2 +
    586194602218356762336379252895075698537137*$.1 -
    1300941132246339981668875337559012140965 97
 over Unramified extension defined by the polynomial x^3 + 6*x + 2
 over 7-adic field mod 7^50
> m(L.1);
RR.1 + O(RR.1^92)
> RR.1 @@ m;
O(7^48)*$.1^5 + O(7^48)*$.1^4 + O(7^49)*$.1^3 + O(7^49)*$.1^2 + $.1 + O(7^50)
> CoefficientRing(RR).1 @@ m;
O(7^34)*$.1^5 - (954564700580430506024960512238*7^-1 + O(7^35))*$.1^4 +
    O(7^36)*$.1^3 + (1031213687115590174398504554631*7^-1 + O(7^35))*$.1^2 +
    O(7^37)*$.1 + 1312848773660672951063501735568*7 + O(7^36)
```

AssignNames(~L, S)

Assign the name in the sequence S to the generator of the extension defining the local field L.

Name(L, i)

Return the generator of the local field L which has assigned to it the name in the sequence S which was input to AssignNames. The only valid input for i is 1.

Discriminant(L)

Return the discriminant of the local field L.

ResidueClassField(L)

Return the residue class field of the maximal order of the local field L and the map between L and its residue class field.

RelativeField(L, m)

Return L as an extension of the domain of the map m which should be a map from a subfield of L (having the same coefficient ring as L) into L.

51.3.1 Predicates on Fields

```
IsRamified(L)
```

Return whether the local field L has an non trivial ramified subfield, that is, the ramification degree of L is greater than 1.

```
IsTamelyRamified(L)
IsWildlyRamified(L)
```

Return whether the local field L is tamely or wildly ramified.

```
IsTotallyRamified(L)
```

Return whether the local field L is a totally ramified extension, that is, L has a trivial inertial subfield.

```
IsUnramified(L)
```

Return whether the local field L is equal to its inertial subfield.

51.4 Maximal Order

```
IntegralBasis(L)
```

Return a basis for the maximal order of the local field L.

```
IsIntegral(a)
```

Return whether the local field element a lies in the maximal order of its parent L and a sequence giving the coordinates of a with respect to the integral basis of L if so.

Example H51E5

We construct a local field and compute an integral basis for it.

```
> P<x> := PolynomialRing(Integers());
> L := LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
> IntegralBasis(L);
[ 1 + O(7^50), (7^-1 + O(7^49))*$.1^2 + O(7^49)*$.1 + O(7^49), (7^-2 +
    O(7^48))*$.1^4 + O(7^48)*$.1^3 + O(7^48)*$.1^2 + O(7^50)*$.1 + O(7^50), $.1
    + O(7^50), (7^-1 + O(7^49))*$.1^3 + O(7^49)*$.1^2 + O(7^49)*$.1 + O(7^99),
    (7^-2 + O(7^48))*$.1^5 + O(7^48)*$.1^4 + O(7^48)*$.1^3 + O(7^50)*$.1^2 +
    O(7^50)*$.1 + O(7^50) ]
```

51.5 Homomorphisms from Fields

```
hom< L → R | a >
```

```
hom< L → R | cfm, a >
```

Return the homomorphism from the local field L into the ring R whose image of the generator of L is a and whose action on the coefficient field of L is given by cfm if given.

51.6 Automorphisms and Galois Theory

```
FrobeniusAutomorphism(L)
```

Return the automorphism of the unramified extension L which is the lift of the frobenius automorphism on the residue class field of L.

```
AutomorphismGroup(L)
```

Return the automorphism group of the local field L and a map from the group to the parent of automorphisms of L.

```
DecompositionGroup(L)
```

```
InertiaGroup(L)
```

```
RamificationGroup(L, i)
```

Return the subgroup of the automorphism group of the local field L whose elements are the automorphisms (represented as group elements) σ such that $v(\sigma(z) - z) \geq i+1$. The decomposition group is the -1th ramification group and the inertia group is the 0th ramification group.

```
FixedField(L, G)
```

Return the subfield of the local field L which is fixed by the automorphisms (represented as group elements) in the subgroup G of the automorphism group of L.

Example H51E6

The automorphism and inertia groups of a local field are computed and their fixed fields examined.

```
> P<x> := PolynomialRing(Integers());
> L := LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
> A, am := AutomorphismGroup(L);
> am(Random(A));
Mapping from: RngLocA: L to RngLocA: L
> $1(L.1);
-(279674609046925265141076018485*7^-2 + O(7^34))*$.1^5 + O(7^35)*$.1^4 +
    (103590525174898812945888146412 3*7^-1 + O(7^35))*$.1^3 + O(7^36)*$.1^2 -
    (1009443907710864908501983735501 + O(7^36))*$.1 + O(7^37)
> FixedField(L, A);
Extension of 7-adic field mod 7^50 by (1 + O(7^33))*x + O(7^33)
```

```
> InertiaGroup(L);
Permutation group acting on a set of cardinality 6
    Id($)
    (1, 2)(3, 5)(4, 6)
> FixedField(L, InertiaGroup(L));
Extension of 7-adic field mod 7^50 by (1 + O(7^37))*x^3 - (2*7^2 + O(7^37))*x^2
    + (7^4 + O(7^37))*x - 4*7^6 + O(7^37)
```

51.7 Local Field Elements

`L ! r`

Return the element of the local field L described by r where r may be anything which is coercible into the quotient representation of L.

`L . i`

Return the generator of the local field L. The only valid input for i is 1.

`InertialElement(L)`

Return a generator for the inertial subfield of the local field L.

`UniformizingElement(L)`

Return an element of the local field L of valuation 1.

51.7.1 Arithmetic

`a * b`   `a + b`   `a - b`   `- a`   `a ^ n`   `a / b`

51.7.2 Predicates on Elements

`a eq b`

Return whether the local field elements a and b are considered equal.

`IsOne(a)`

`IsMinusOne(a)`

Return whether the local field element a is known to be 1 or -1 to the precision of the field.

`IsWeaklyZero(a)`

Return whether the local field element a is not known to be non zero.

`IsZero(a)`

Return whether the local field element a is known to be zero.

51.7.3 Other Operations on Elements

```
Valuation(a)
```

The valuation of the element a in a local field.

```
RelativePrecision(a)
```

The relative precision of the element a in a local field.

```
Eltseq(a)
```

Return the coefficients of powers of the generator of the parent of a in a.

```
RepresentationMatrix(a)
```

The representation matrix of the element a of a local field.

Example H51E7

Continuing from the first example we have :

```
> UniformizingElement(L);
a^2 + (6 + O(7^50))*a + 3 + O(7^50)
> InertialElement(L);
a + O(7^50)
> Valuation(UniformizingElement(L));
1
> Valuation(InertialElement(L));
0
> Eltseq(UniformizingElement(L));
[ 3 + O(7^50), 6 + O(7^50), 1 + O(7^50) ]
```

51.8 Polynomials over General Local Fields

Polynomials over local fields can be factored and their roots computed.

```
Factorization(f)
```

`Certificates` BoolElt *Default :* `false`

The factorization of the polynomial f over a local field defined by an arbitrary polynomial. The factorization is returned as a sequence of tuples of prime polynomials and exponents along with a scalar factor. If the parameter `Certificates` is `true` then certificates proving the primality of each prime are also returned.

```
SuggestedPrecision(f)
```

For a polynomial f over a general local field, return a precision at which the factorization of f as given by `Factorization(f)` will be Hensel liftable to the correct factorization.

The precision returned is not guaranteed to be enough to obtain a factorization of the polynomial. It may be that a correct factorization cannot be found at that precision but may be possible with a little more precision.

```
Roots(f)
Roots(f, R)
```

The roots of the polynomial f over the general local field R where R is taken to be the coefficient ring of f if it is not given.

Example H51E8

```
> Q3:=pAdicField(3,40);
> Q3X<x>:=PolynomialRing(Q3);
> L<a>:=LocalField(Q3,x^6-6*x^4+9*x^2-27);
> Factorization(Polynomial(L,x^6-6*x^4+9*x^2-27));
[
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + (5*3^35 + O(3^38))*a^5 + O(3^38)*a^4 - (10*3^36 +
        O(3^39))*a^3 + O(3^39)*a^2 + (2701703435345984179 + O(3^40))*a +
        O(3^40), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + -(29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 +
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 - (116512378274*3 +
        O(3^25))*a + O(3^26), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + -(29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 +
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 - (349537134821 +
        O(3^25))*a + O(3^26), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + -(5*3^35 + O(3^38))*a^5 + O(3^38)*a^4 + (10*3^36 +
        O(3^39))*a^3 + O(3^39)*a^2 - (2701703435345984179 + O(3^40))*a +
        O(3^40), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + (29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 -
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 + (116512378274*3 +
        O(3^25))*a + O(3^26), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + (29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 -
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 + (349537134821 +
        O(3^25))*a + O(3^26), 1>
]
O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + 1 + O(3^40)
```

52 ALGEBRAIC POWER SERIES RINGS

Chapter 52
ALGEBRAIC POWER SERIES RINGS

52.1 Introduction

Algebraic Power Series are a lazy representation of multivariate power series with fractional exponents, which are roots of univariate polynomials with coefficients in multivariate polynomial rings. The functionality allows the "lazy" computation of the power series expansion to any finite degree, this being well-determined by the defining algebraic equation.

The package was designed with the computation of formal resolutions of singularities of surfaces in mind but should provide a useful general tool for users. As well as allowing definition directly from a polynomial equation, the user can compose algebraic power series and recursively define series which are roots of polynomials whose coefficients are polynomial functions in other algebraic power series. There are also functions for basic arithmetic operations and tests for exact equality and the like.

The defined series must be expandable in fractional (positive) powers of the base variables. This is true for all roots of a *quasi-ordinary* polynomial possibly after a finite extension of the base field.

The package was designed and implemented by Tobias Beck at the RICAM institute in Linz, Austria. Some low-level adaptations for added efficiency and integration were carried out by the Magma group. The algorithms are described in [Bec07, Sec. 4].

52.2 Basics

In Magma, algebraic power series are represented in a hybrid lazy-exact way. Eventually every power series is given by a defining polynomial and a sufficiently large initial segment. Intermediate operations are represented in a lazy way. This makes it possible to compute both quickly and to high precision if necessary.

Note, however, that decision procedures may be very time intensive. In the sequel we have indicated in each function whether it is fast or whether it has to be used with care.

52.2.1 Data Structures

Algebraic power series are of type `RngPowAlgElt`.

There are two types of algebraic series: `Atomic` and `Substitution` which we sometimes refer to briefly as type A or type B. There is no difference in the functionality available but they are structurally different. The user can determine the type of a series, if (s)he desires, from the attribute `type` which is 0 for type A and 1 for type B.

`Atomic` series are the basic type that are given directly as the root of a univariate polynomial $p(z)$ with given initial expansion. $p(z)$ comes from a polynomial $f(z)$ with coefficients in a multivariate polynomial ring. Either $p = f$ or p is determined from f by evaluating the coefficients of f at a given array, *subs*, of algebraic power series. This allows the construction of algebraic power series as roots of polynomials over finitely-generated fields of already-constructed series. Most of the constructors return a series of this type.

`Substitution` series allow the composition of algebraic power series. The principal defining data is an algebraic power series s in n variables and an array of n algebraic series that are substituted into this. In fact, the substitution is not necessarily direct, but through n given elements in the dual lattice of the exponent lattice of s as explained in the constructor `EvaluationPowerSeries`.

Both types of series have an associated exponent lattice specified by two components: Γ, a sublattice of a standard integral lattice, and e, a positive integer. The expansion of the series will in general have fractional exponents and e is the LCM of the denominators of these (e may be 1). The finite expansions that are returned are always integral-exponent multivariate polynomials. The *actual* mathematical expansion is derived from this return value by dividing all exponents by e. With this scaling up of fractional exponents by e to get integral exponents, all exponent vectors for monomials occurring in the expansion (up to any degree) will lie in the lattice Γ. So the actual exponent lattice for the series is $(1/e)\Gamma$.

The user doesn't have to worry too much about the lattice Γ. It is automatically computed by most of the constructors for algebraic power series and the default of the standard integral lattice can always be used (assuming e is correct!). It's utility is that, in computing the exponent lattice for composite constructions on series, factors of e may be cancelled out from the resulting lattice. So an algebraic construction involving series with non-integral exponents may produce a result with only integral exponents. Such lattice computations are carried out automatically.

Important note: To speed up some of the basic internal polynomial operations, it is assumed that the domain of an algebraic power series is a multivariate polynomial ring with a *degree* ordering (`glex` or `grevlex`). Attempts to create series using polynomial rings with a non-degree ordering will result in a user error.

52.2.2 Verbose Output

A verbose flag `AlgSeries` exists which can take values `true`, `false`, 0 or 1. Setting to `true` (or 1) will output information on the progress of some of the potentially more time-consuming intrinsics.

52.3 Constructors

Using constructors one can construct power series starting from polynomial data or using other power series recursively.

`PolyToSeries(s)`

Given a multivariate polynomial s, returns the series representation of s.

```
AlgebraicPowerSeries(dp, ip, L, e)
```

subs SeqEnum *Default :* []

Define a power series root of a polynomial p using an initial expansion *ip* and its exponent lattice `1/e L`. The defining polynomial p is either *dp* when *subs* is empty or obtained by substituting the elements of *subs* into the variables of *dp*. The initial expansion has to be sufficiently long in order to uniquely identify a root, see [Bec07, Cond. 4.3]. In this initial finite expansion and in subsequent ones, the variables occurring actually represent e-th roots, *ie* $x_1x_2^2$ is really $x_1^{1/e}x_2^{2/e}$. All exponents of monomials occurring in these expansions and the coefficients of p should lie in L (although this is not checked in many places), monomials of p giving true values rather than e-th roots.

There are simpler constructors where L is omitted (when it is assumed to be the standard integral lattice) or e is (when it is taken as 1).

As noted earlier, no strong checks are performed at construction time on the correctness of L or e or whether *ip* is indeed the initial expansion of a unique root in this "raw data" constructor. Incorrect initial data will only be revealed when failure occurs in further expansion of the series. The preferred methods of series creation are `ImplicitFunction`, `EvaluationPowerSeries` and `RationalPuiseux` because in these cases the sanity checks are more easily verified.

```
EvaluationPowerSeries(s, nu, v)
```

Given a series s, a sequence *nu* of vectors in the dual of its exponent lattice of s and a sequence v (of the same length) of power series in some other common domain (with compatible coefficient field). Returns the series obtained by substituting $\underline{x}^\mu \mapsto \prod_i \texttt{v[i]}^{\langle \texttt{nu[i]}, \mu\rangle}$. This requires that *nu* and v fulfill a certain condition on the orders to guarantee convergence of the resulting series, see [Bec07, Cond. 4.6].

```
ImplicitFunction(dp)
```

subs SeqEnum *Default :* []

The unique series with zero constant term defined by a polynomial $p \in k[x_1, .., x_n][z]$ or $k[[x_1, \ldots, x_n]][z]$, fulfilling the conditions of the implicit function theorem, *i.e.*, $p(0, \ldots, 0) = 0$ and $\partial p/\partial z(0, \ldots, 0) \neq 0$. The polynomial p is equal to *dp* possibly substituted with the series in *subs* as in `AlgebraicPowerSeries`. *dp* should have coefficients in a multivariate polynomial ring.

52.3.1 Rational Puiseux Expansions

Let $p \in k[[x_1, \ldots, x_n]][z]$ be a quasi-ordinary polynomial over a field k of characteristic zero. This means that p is non-zero, squarefree and monic (i.e., its leading coefficient in z is a unit in the power series ring) and if $d \in k[[x_1, \ldots, x_n]]$ denotes its discriminant then $d = x_1^{e_1} \cdots x_n^{e_n} u(x_1, \ldots, x_n)$ where u is a unit in the power series ring.

In this case the Theorem of Jung-Abhyankar states that p has deg(p) distinct Puiseux series roots, i.e., power series roots with fractionary exponents and coefficients in the algebraic closure of k.

These roots are computed by a generalization of the so called Newton-Puiseux algorithm. Also Duval's extension for computing rational parametrization has been implemented.

`RationalPuiseux(p)`

`Gamma`	Lattice	*Default :* `StandardLattice`
`subs`	SeqEnum	*Default :* []
`Duval`	BoolElt	*Default :* `false`
`OnlySingular`	BoolElt	*Default :* `false`
`ExtName`	MonStgElt	*Default :* "*gamma*"
`ExtCount`	RngIntElt	*Default :* 0

We first specify the behavior of this function in the case that no special value of *subs* has been given. This function assumes that p is a univariate polynomial over a multivariate polynomial ring $S = k[x_1, ..., x_r]$ and that p is quasi-ordinary. In this case it will compute a set of rational parametrizations of p. Note that for reasons of efficiency the user has to make sure that p is actually quasi-ordinary! (Otherwise, further processing of the output may result in runtime errors.)

The first return value will be the exponent lattice of the input polynomial in the usual format $< \Gamma_0, e_0 >$. If the parameter `Gamma` has been specified, then $\Gamma_0 =$ `Gamma` and $e_0 = 1$. In this case `Gamma` has to be an integral 'r'-dimensional lattice of full rank containing all the exponents of p. Otherwise Γ_0 will be set to the r-dimensional standard lattice and again $e_0 = 1$.

As a second value a complete list of rational parametrizations in the format $< \lambda, s, N, E >$ is returned. Here λ is a sequence of r field elements and s is a fractionary algebraic power series of type `RngPowAlgElt`. Let p_1 denote the image of p under the transformation $x^{\mu_i} \mapsto \lambda_i x^{\mu_i}$ where $(\mu_i)_i$ is the basis of the exponent lattice $e_0^{-1}\Gamma_0$ then s is a solution of p_1, i.e., we have $p_1(s) = 0$. Note that if neither `Gamma` nor `subs` have been supplied this just means that x_i is substituted by $\lambda_i x_i$. Finally N is the index of $e_0^{-1}\Gamma_0$ in the exponent lattice of s and E is the degree of the extension of the coefficient field needed for defining s.

The behavior described above corresponds to the Newton-Puiseux algorithm with Duval's trick. The field extensions that are used for expressing the series fulfill a certain minimality condition. If `Duval` is set to `false` then the function returns a complete set of representatives (up to conjugacy) of Puiseux series roots of the original polynomial p, in other words, the λ-vectors will always be vectors of ones.

If `OnlySingular` is set to `true` then only those parametrizations that correspond to singular branches are returned.

If the ground field has to be extended, the algebraic elements will be assigned the name `ExtName_`*i* where i starts from `ExtCount`. The last return value is the value of `ExtCount` plus the number of field extensions that have been introduced during the computation.

Finally, if the parameter `subs` is passed, then it has to be a sequence of r power series in a common domain and internally the variables in p will be substituted by the corresponding series. Again the resulting polynomial has to be quasi-ordinary. In this case Γ_0 and e_0 are determined by building the sum of the exponent lattices of all series in subs. The parameter Gamma then has no effect.

For further details on the algorithm and other references see [Bec07, Sec. 4.3]

Example H52E1

We illustrate the constructors by examples. For displaying results we already use the command Expand that will be explained later.

```
> Q := Rationals(); Qs<s> := FunctionField(Q);
> Qxy<x,y> := PolynomialRing(Q, 2, "glex");
> Qxyz<z> := PolynomialRing(Qxy);
> Qst<t> := PolynomialRing(Qs, 1, "glex");
> Qstu<u> := PolynomialRing(Qst);
```

One can consider polynomials as series.

```
> s0 := PolyToSeries(1 - 3*x + x^2*y + y^20);
> Expand(s0, 10);
true x^2*y - 3*x + 1
```

One can define series by the implicit function theorem at the origin.

```
> s1 := ImplicitFunction(z*(1 - x - y) - x - y);
> Expand(s1, 4);
true x^3 + 3*x^2*y + 3*x*y^2 + y^3 + x^2 + 2*x*y + y^2 + x + y
```

One can define a power series if an initial expansion is known. Note that the following power series has exponent lattice $\mathbf{Z}(\frac{1}{5}, -\frac{2}{5}) + \mathbf{Z}(\frac{2}{5}, \frac{1}{5})$ but its “expansions” are polynomials supported on $\mathbf{Z}(1, -2) + \mathbf{Z}(2, 1)$.

```
> defpol := (1+5*y+10*y^3+10*y^2+5*y^4+y^5)*z^5+(-20*y^3*x-
>   30*y^2*x-5*y^4*x-5*x-20*y*x)*z^4+(10*x^2+30*y^2*x^2+10*y^3*x^2+
>   30*x^2*y)*z^3+(-20*y*x^3-10*x^3-10*y^2*x^3)*z^2+
>   (5*y*x^4+5*x^4)*z-x^5-x^2*y;
> Gamma := Lattice(RMatrixSpace(Integers(), 2, 2) ! [1,-2, 2,1]);
> init := x^2*y;
> s2 := AlgebraicPowerSeries(defpol, init, Gamma, 5);
> Expand(s2, 20);
true
-x^2*y^16 + x^5*y^10 + x^2*y^11 - x^5*y^5 - x^2*y^6 + x^5 + x^2*y
```

We can “substitute” series into each other.

```
> X := AlgebraicPowerSeries(u^3-t+s*t^2, t, StandardLattice(1), 3);
> Y := PolyToSeries(t);
> duals := [RSpace(Integers(), 2) | [1, 3], [2, 1]];
> s3 := EvaluationPowerSeries(s2, duals, [X, Y]);
> Expand(s3, 13);
```

```
true (-1/9*s^2 - 1/3*s - 1)*t^10 + (-1/3*s + 1)*t^7 + t^4
```

We can compute all the Puiseux series roots of a quasi-ordinary polynomial up to conjugacy over **Q**.

```
> qopol := z^6 + 3*x*y^2*z^4 + x*y*z^3 + 3*x^2*y^4*z^2 + x^3*y^6;
> _, prms := RationalPuiseux(qopol : Duval := false); prms;
[*
    <[ 1, 1 ], Algebraic power series -x*y, 3, 1>,
    <[ 1, 1 ], Algebraic power series gamma_0*x*y, 3, 2>,
    <[ 1, 1 ], Algebraic power series -x^2*y^5, 3, 1>,
    <[ 1, 1 ], Algebraic power series gamma_1*x^2*y^5, 3, 2>
*]
> Domain(prms[2][2]); ExponentLattice(prms[2][2]);
Polynomial ring of rank 2 over Number Field with defining
polynomial $.1^2 - $.1 + 1 over the Rational Field
Graded Lexicographical Order
Variables: x, y
<
    Lattice of rank 2 and degree 2
    Basis:
    ( 1  1)
    ( 1 -2),
    3
>
> Expand(prms[2][2], 15);
true x^3*y^9 + (gamma_0 - 1)*x^2*y^5 + gamma_0*x*y
```

We find that the sum over all field extensions $1 + 2 + 1 + 2 = 6$ is equal to the degree of the defining polynomial `qopol`. The third parametrization involves a field extension of **Q** by `gamma_0` s.t. $\mathtt{gamma\_0}^2 - \mathtt{gamma\_0} + 1 = 0$ and an extension of the exponent lattice to $\mathbf{Z}(\frac{1}{3}, \frac{1}{3}) + \mathbf{Z}(\frac{1}{3}, -\frac{2}{3})$. It turns out that the field extension is not necessary if we are only interested in parametrizations.

```
> _, prms := RationalPuiseux(qopol : Duval := true); prms;
[*
    <[ -1, -1 ], Algebraic power series -x*y, 3, 1>,
    <[ -1, 1 ], Algebraic power series -x^2*y^5, 3, 1>
*]
```

No field extensions have been introduced, but this required the application of automorphisms $\mathbf{Q}[[x, y]] \to \mathbf{Q}[[x, y]]$ in advance (more precisely $x \mapsto -x, y \mapsto -y$ resp. $x \mapsto -x, y \mapsto y$). This time we can sum up the overall extension degrees (*i.e.*, for fields and lattices) $3 \cdot 1 + 3 \cdot 1 = 6$ to the degree of `qopol`.

52.4 Accessors and Expansion

The following functions provide an interface to conveniently extract information from a power series defined as above.

`Domain(s)`

Return the multivariate polynomial ring that is used for approximating the series s by its truncations.

`ExponentLattice(s)`

Return the exponent lattice $(1/e)\Gamma$ of the series as tuple (Γ, e) where Γ is an integral lattice and e is an integer.

`DefiningPolynomial(s)`

Return a defining polynomial of the series which is a squarefree univariate polynomial over the multivariate polynomial domain `Domain(s)`. In the case of series defined with substitutions, the computation may be expensive and can involve recursive resultant computations.

`Order(s)`

`TestZero` BoolElt *Default :* `false`

Given a series s, return the integral order (total degree of smallest non-zero term occurring) of its expansion as returned by `Expand`, *i.e.*, its fractionary order times the exponent denominator. If s is zero, this function will **not terminate**. Set `TestZero` to `true` to get a return value -1 in this case, but note that this involves the computationally complex call `IsZero`.

`Expand(s,ord)`

Given the power series β which is represented by s, let α be the result of substituting variables $x_i \mapsto x_i^e$ where e is taken from the output of `ExponentLattice(s)` (*i.e.*, α is β without exponent denominators). Returns `true` and the truncation of α modulo terms of order greater or equal `ord`. A return of `false` indicates that the representation is inconsistent (which should only happen when `RationalPuiseux` is called with non quasi-ordinary input or `AlgebraicPowerSeries` is used inconsistently).

Example H52E2

We can study the series `s3`.

```
> Domain(s3);
Polynomial ring of rank 1 over Univariate rational function
field over Rational Field
Graded Lexicographical Order
Variables: t
> ExponentLattice(s3);
<
    Standard Lattice of rank 1 and degree 1,
    3
>
> DefiningPolynomial(s3);
(s^45*t^45 - ... - 15*s^30*t^2 - s^30)*u^15 + ... +
(5*s^37*t^41 - ... + 120*s^31*t^19 - 30*s^30*t^18)*u^3 -
s^35*t^40 + ... - 5*s^31*t^21 + s^30*t^20
> Order(s3);
4
```

These commands reveal the following about s3: It is a power series in $\mathbf{Q}(s)[[t^{1/3}]]$, because it is approximated in $\mathbf{Q}(s)[t]$ and has exponent lattice $\frac{1}{3}\mathbf{Z}$. A defining polynomial in $\mathbf{Q}(s)[t][u]$ was also computed. (Recall that s3 has been defined recursively.) The order is $\frac{4}{3}$ which we know already from a previous expansion.

52.5 Arithmetic

There are functions to perform basic arithmetic operations (addition, subtraction etc.) on power series.

`AlgComb(c,ss)`

Given a polynomial c in r variables and a sequence ss of r power series (in a common domain with compatible coefficient field) return the series obtained by substituting the elements of ss for the variables of c. This allows the construction of completely arbitrary algebraic combinations.

`s + t`

`s - t`

`s * t`

Add, subtract or multiply two power series.

Example H52E3

One can easily substitute power series into polynomials.

```
> // construct the series s0^2+s1^2
> h0 := AlgComb(x^2 + y^2, [s0,s1]);
> Expand(h0, 3);
true 10*x^2 + 2*x*y + y^2 - 6*x + 1
```

This includes of course the ring operations.

```
> h1 := Add(s1, PolyToSeries(One(Qxy)));
> Expand(h1, 4);
true
x^3 + 3*x^2*y + 3*x*y^2 + y^3 + x^2 + 2*x*y + y^2 + x + y + 1
> h2 := Mult(h1, PolyToSeries(1 - x - y));
> Expand(h2, 4);
true 1
```

```
> h3 := Add(h2, PolyToSeries(-One(Qxy)));
> Expand(h3, 4);
true 0
```

52.6 Predicates

The following functions provide decision algorithms for algebraic power series. They may involve **recursive resultant computations**, hence, have a high complexity and should be used with care.

`IsZero(s)`

Decides if the series is zero.

`s eq t`

Decides if two series are equal.

`IsPolynomial(s)`

Decides whether the series is actually a polynomial (with integral exponents) in the multivariate polynomial domain as returned by `Domain(s)`. In the positive case also returns that polynomial. This function relies on `SimplifyRep`.

Example H52E4

The previous computations suggest that `h2` is 1, in particular it is polynomial (in contrast to `h1`). In this case `h3` would be zero.

```
> IsPolynomial(h1);
false
> IsPolynomial(h2);
true 1
> IsEqual(h2, PolyToSeries(One(Qxy)));
true
> IsZero(h3);
true
```

52.7 Modifiers

The following functions modify the representation of the power series or apply a simple automorphism.

```
ScaleGenerators(s,ls)
```

Let $\{\gamma_i\}_i$ be the basis (determined from the representation chosen by MAGMA) of the exponent lattice of the series s, and let $\sigma : \underline{x}^{\gamma_i} \mapsto ls[i]\underline{x}^{\gamma_i}$. Return the series $\sigma(s)$.

```
ChangeRing(s,R)
```

If R is a multivariate polynomial domain compatible with the approximation domain Domain(s), return the same power series with new approximation domain R. This is sort of a “coercion between power series rings”.

```
SimplifyRep(s)
```

Factorizing BoolElt *Default :* true

“Simplifies” the internal representation of a series. The result will be a series of atomic type without recursive (substitution) dependencies on other power series. The defining polynomial of the simplified series will be irreducible and therefore a minimal polynomial over Domain(s) (unless Factorizing is false when it will only be guaranteed to be squarefree). After the simplification, DefiningPolynomial returns this polynomial, which can be useful (*e.g.*, for IsPolynomial). However, experience shows that the resulting representation is in general neither simple nor more efficient for subsequent computations.

There is a **dangerous pitfall:**

Assume we have a series represented by a tree with nodes of type A and B. Assume further that the leaves have been constructed by RationalPuiseux with parameter Gamma set to some value. Then the intention was probably to work over the subring of a polynomial ring with restricted support. If now SimplifyRep, with Factorizing as true, is called, then a minimal polynomial over the whole polynomial ground ring is computed which is maybe not what one wants.

Example H52E5

We can modify s2 by mapping generators (of Laurent polynomials) $x^{1/5}y^{-2/5} \mapsto 3x^{1/5}y^{-2/5}$ and $x^{2/5}y^{1/5} \mapsto 4x^{2/5}y^{1/5}$.

```
> Expand(ScaleGenerators(s2, [3,4]), 15);
true
64/81*x^2*y^11 - 64/3*x^5*y^5 - 16/9*x^2*y^6 + 48*x^5 + 4*x^2*y
```

One can naturally view h1 as a series in $\mathbf{Q}(i)[[u, v]]$.

```
> Qi<i> := NumberField(R.1^2 + 1) where R is PolynomialRing(Q);
> Qiuv<u,v> := PolynomialRing(Qi, 2, "glex");
> h4 := ChangeRing(s1, Qiuv);
> Expand(h4, 4); Domain(h4);
true u^3 + 3*u^2*v + 3*u*v^2 + v^3 + u^2 + 2*u*v + v^2 + u + v
```

```
Polynomial ring of rank 2 over Qi
Graded Lexicographical Order
Variables: u, v
```

We have seen that the power series `h3` is zero, but its representation does not show this immediately. We can “explicitize” its representation.

```
> SimplifyRep(h3 : Factorizing := true);
Algebraic power series
0
> DefiningPolynomial($1);
z
```

52.8 Bibliography

[**Bec07**] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007.
URL:http://www.ricam.oeaw.ac.at/publications/reports/.

PART VIII
MODULES

53 INTRODUCTION TO MODULES

Chapter 53
INTRODUCTION TO MODULES

53.1 Overview

This section of the Handbook describes the Magma facilities for linear algebra and module theory. Since this topic is absolutely fundamental for much of algebra, it is important that the reader understand how linear algebra is presented in Magma. The structures covered under this heading include:

(i) Vector spaces;

(ii) Inner product spaces;

(iii) Modules defined over any ring or algebra

(iv) $R[G]$-modules, where R is a ring and G is a group;

(v) Linear transformations and R-module homomorphisms.

Although vector spaces are, of course, subsumed under general modules, we present a separate treatment of them, firstly because of their importance and secondly because their theory is somewhat cleaner than that of a general module. Magma users who are unfamiliar with the language of module theory will find a self-contained treatment of the vector space machinery in Chapter 28.

In the Magma universe, rectangular matrices are regarded as forming a module (actually a bimodule). We shall regard a rectangular matrix as the concrete realization of a linear transformation or R-module homomorphism. Thus, an $m \times n$ matrix over a ring R is considered to be an element of the module $\mathrm{Hom}_R(M, N)$. Reflecting the dual nature of matrices, the $\mathrm{Hom}_R(M, N)$ operations include the standard module-theoretic operations as well as operations that interpret an element of $\mathrm{Hom}_R(M, N)$ as a homomorphism.

53.2 General Modules

A module M is always regarded as a submodule or quotient module of the free module $S^{(n)}$, for some ring or algebra S. The types of module that are definable in the system fall into three classes:

(a) **Abstract Modules**: Given a ring R, a set M and a mapping $\phi : R \times M \to M$, the pair (M, ϕ) will be referred to as an *abstract* R-module. Because of the very general nature of this construction, only the basic arithmetic operations may be applied to modules of this type.

(b) **Modules with Scalar Action**: Given a general ring R, an R-module with *scalar action* is a submodule or quotient module of the free R-module $R^{(n)}$, where the action is that of ring multiplication in R.

(c) **Modules with Matrix Action**: Let R be a PIR and suppose S is a R-algebra. Thus there exists a ring homomorphism $\phi : R \to S$, and so S is a (left) R-module with the

R-action defined by $r * s = \phi(r) * s$. Indeed, any S-module M is a (left) S-module with action defined by $r * m = \phi(r) * m$. Furthermore, if $\phi(R)$ lies in the centre of S, then S acts on M as a ring of R-module endomorphisms. Consequently, M is an S-module. We take M to be the free R-module $R^{(n)}$, and so the action of S on M is given by the action of a subring of $M_n(R)$ on M. Thus, given an R-algebra S, an S-module of the form $M = R^{(n)}$ may be specified by giving M together with a homomorphism of S into $M_n(R)$.

53.3 The Presentation of Submodules

Let N be a submodule of the module $M = R^{(m)}$. For simplicity, assume that N is free, and has dimension n where $n < m$. There are two ways in which N may be viewed:

(a) As a submodule embedded in M. Thus, though N is a module of dimension n, its elements are regarded as elements of M.

(b) As the module $R^{(n)}$ represented on a reduced basis, together with a morphism ϕ defining the inclusion of N into M.

The presentation (a) is the usual way submodules are regarded in elementary linear algebra. However, this presentation is inconvenient for more advanced applications. For example, many of the major functions available for studying an $R[G]$-module N expect that N is given relative to a reduced basis. We shall refer to a submodule presentation (a) as the *embedded presentation*, and (b) as the *reduced presentation*.

To provide the user with the maximum flexibility, Magma supports both forms of submodule presentation for the important classes of modules. Usually, a module calculation commences with the definition of one or two modules from which further modules are created by the operations of forming submodules, quotient modules and extensions. Let us call these latter modules *descendants* of the original module. Magma provides parallel creation functions which allow the user to choose the form of submodule presentation. Once that choice has been made, all descendants of the initial module(s) will follow the same presentation convention.

The module creation functions that select the embedded form are usually of the form *Qualifier*`Space`, while those that adopt the standard form are usually of the form *Qualifier*`Module`. Thus in the case of vector spaces the function `KSpace(K, n)` constructs the n-dimensional vector space over the field K, where submodules are to be presented in embedded form. On the other hand, `RModule(K, n)` constructs the n-dimensional vector space over the field K, where submodules are to be presented in reduced form.

54 FREE MODULES

Chapter 54
FREE MODULES

54.1 Introduction

54.1.1 Free Modules

This chapter describes the facilities provided for modules in the following representations:

Tuple Modules: Modules whose elements are n-tuples over a fixed ring R, i.e., modules $R^{(n)}$;

Matrix Modules: Modules whose elements are homomorphisms of modules, i.e., $Hom_R(M, N)$. The elements of these modules are $m \times n$ matrices over the ring R;

The ring R acts on the right of the module element by scalar multiplication. If R is not an Euclidean Domain then, currently, only arithmetic with vectors is supported. In particular, the ability to work with submodules and quotient modules is restricted to situations where R is either a field or Euclidean Domain.

In the first part of the chapter we describe the operations that apply to modules generally, while in the second half we describe the creation of modules $Hom_R(M, N)$ together with the operations that are specific to them. Insofar as elementary module-theoretic operations are concerned, there is no real difference between tuple modules and matrix modules except for the input and display of elements. Many special operations provided for matrices are described in the chapter on matrices.

The reader is referred to the chapter on vector spaces for descriptions of the extensive functionality provided for modules over fields.

54.1.2 Module Categories

The family of all finitely generated modules over a given ring R forms a category, while the set of all finitely generated modules forms a family of categories indexed by the ring R. In this family of categories, objects are modules and the morphisms are module homomorphisms. The category name for modules is `ModRng`. We distinguish the following subcategories of `ModRng`:

`ModTupFld` - the category of modules of n-tuples over a field;

`ModMatFld` - the category of modules of $m \times n$ matrices over a field;

`ModTupEd` - the category of modules of n-tuples over an euclidean domain.

`ModTupRng` - the category of modules of n-tuples over a ring;

`ModMatRng` - the category of modules of $m \times n$ matrices over a ring;

54.1.3 Presentation of Submodules

Let N be a free submodule of the R-module M. We have two alternative ways of presenting N. Firstly, we can present it on a set of generators that are elements of M; we call such a presentation an *embedded presentation.* Alternatively, given that N has rank r, we can present it as the module $S^{(r)}$, with appropriate action induced from the action of R on M. We call this presentation of N a *reduced presentation.*

The user can control the method of submodule presentation at the time of creation of an initial module through selection of the appropriate creation function. Thus, the function `RModule` will create a module with the convention that it and all its submodules and quotient modules will have their submodules presented in reduced form. The use of `RSpace`, on the other hand, signifies that submodules are to be presented in embedded form.

54.1.4 Notation

Throughout this chapter, R will denote a ring (possibly a field) while K will denote a field. The letters M and N will denote modules, while U and V will denote vector spaces.

54.2 Definition of a Module

54.2.1 Construction of Modules of n-tuples

```
RSpace(R, n)
```

```
RModule(R, n)
```

Given a ring R and a non-negative integer n, create the free right R-module $R^{(n)}$, consisting of all n-tuples over R. The module is created with the standard basis, $e_1, \ldots, e_n$, where e_i $(i = 1, \ldots, n)$ is the vector containing a 1 in the i-th position and zeros elsewhere.

The function `RModule` creates a module in reduced mode while `RSpace` creates a module in embedded mode.

```
RSpace(R, n, F)
```

Given a ring R, a non-negative integer n and a square $n \times n$ symmetric matrix F, create the free right R-module $R^{(n)}$ (in embedded form), with inner product matrix F. This is the same as `RSpace(R, n)`, except that the functions `Norm` and `InnerProduct` (see below) will be with respect to the inner product matrix F.

Example H54E1

We construct the module consisting of 6-tuples over the integers.

```
> Z := IntegerRing();
> M := RModule(Z, 6);
> M;
RModule M of dimension 6 with base ring Integer Ring
```

54.2.2 Construction of Modules of $m \times n$ Matrices

```
RMatrixSpace(R, m, n)
```

The module comprising all $m \times n$ matrices over the ring R.

54.2.3 Construction of a Module with Specified Basis

```
RModuleWithBasis(Q)
RSpaceWithBasis(Q)
RSpaceWithBasis(a)
```

Given a sequence Q (or matrix a) of k independent vectors each lying in a module M, construct the submodule of M of dimension k whose basis is Q (or the rows of a). The basis is echelonized internally but all functions which depend on the basis of the space (e.g. `Coordinates`) will use the given basis.

```
RMatrixSpaceWithBasis(Q)
```

The module of $m \times n$ matrices whose basis is given by the linearly independent matrices of the sequence Q.

54.3 Accessing Module Information

```
M . i
```

Given an R-module M and a positive integer i, return the i-th generator of M. The integer i must lie in the range $[1, r]$, where r is the number of generators for M.

```
CoefficientRing(M)
BaseRing(M)
CoefficientRing(M)
BaseRing(M)
CoefficientField(M)
BaseField(M)
```

Given an R-module M which is defined as a submodule of $S^{(n)}$, return the ring S.

```
Generators(M)
```

The generators for the R-module M, returned as a set.

```
OverDimension(M)
```

Given an R-module M which is an embedded submodule of the module $S^{(n)}$, return n.

`OverDimension(u)`

Given an element u of an embedded submodule of the module $S^{(n)}$, return n.

`Moduli(M)`

The column moduli of the module M over a euclidean domain.

`Parent(u)`

Given an element u belonging to the R-module M, return M.

`Generic(M)`

Given an R-module M which is a submodule of the module $R^{(n)}$, return the module $R^{(n)}$ as an R-module.

54.4 Standard Constructions

Given one or more existing modules, various standard constructions are available to construct new modules.

54.4.1 Changing the Coefficient Ring

`ChangeRing(M, S)`

Given a module M with base ring R, together with a ring S, construct the module N with base ring S obtained by coercing the components of elements of M into N, together with the homomorphism from M to N.

`ChangeRing(M, S, f)`

Given a module M with base ring R, together with a ring S, and a homomorphism $f : R \to S$, construct the module N with base ring S obtained by mapping the components of elements of M into N by f, together with the homomorphism from M to N.

`ChangeUniverse(~x, R)`

Change the coefficient ring of x to be R.

54.4.2 Direct Sums

`DirectSum(M, N)`

Given R-modules M and N, construct the direct sum D of M and N as an R-module. The embedding maps from M into D and from N into D respectively, and the projection maps from D onto M and from D onto N respectively are also returned.

`DirectSum(Q)`

Given a sequence Q of R-modules, construct the direct sum D of these modules. The embedding maps from each of the elements of Q into D and the projection maps from D onto each of the elements of Q are also returned.

54.5 Elements

54.6 Construction of Elements

```
elt< M | a1, ..., an >
```

Given a module M with base module $S^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to S, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

```
M ! Q
```

Given the module M with base module $S^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to S, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

```
CharacteristicVector(M, S)
```

Given a submodule M of the module $R^{(n)}$ together with a set S of integers lying in the interval $[1, n]$, return the characteristic number of S as a vector of R.

```
Zero(M)
M ! 0
```

The zero element for the R-module M.

```
Random(M)
```

Given a module M defined over a finite ring or field, return a random vector.

Example H54E2

We create the module of 4-tuples over the polynomial ring $\mathbf{Z}[x]$ and define various elements.

```
> P<x> := PolynomialRing(IntegerRing());
> M := RModule(P, 4);
> a := elt< M | 1+x, -x, 2+x, 0 >;
> a;
(x + 1    -x x + 2     0)
> b := M ! [ 1+x+x^2, 0, 1-x^7, 2*x ];
> b;
(x^2 + x + 1           0    -x^7 + 1          2*x)
> zero := M ! 0;
> zero;
(0 0 0 0)
```

54.6.1 Deconstruction of Elements

```
ElementToSequence(u)
```

```
Eltseq(u)
```

Given an element u belonging to the R-module M, return u in the form of a sequence Q of elements of R. Thus, if u is an element of $R^{(n)}$, then $Q[i] = u[i], 1 \leq i \leq n$, while if u is an element of $R^{(m\times n)}$, then $Q[(i-1)n+j] = u[i,j], 1 \leq i \leq m, 1 \leq j \leq n$.

54.6.2 Operations on Module Elements

54.6.2.1 Arithmetic

```
u + v
```

Sum of the elements u and v, where u and v lie in the same R-module M.

```
-u
```

Additive inverse of the element u.

```
u - v
```

Difference of the elements u and v, where u and v lie in the same R-module M.

```
x * u
```

Given an element x belonging to a ring R, and an element u belonging to the left R-module M, return the (left) scalar product $x * u$ as an element of M.

```
u * x
```

Given an element x belonging to a ring R, and an element u belonging to the right R-module M, return the (right) scalar product $u * x$ as an element of M.

```
u / x
```

Given a non-zero element x belonging to a field K, and an element u belonging to the right K-module M, return the scalar product $u * (1/x)$ as an element of M.

54.6.2.2 Indexing

```
u[i]
```

Given an element u belonging to a submodule M of the R-module $R^{(n)}$ and a positive integer i, $1 \leq i \leq n$, return the i-th component of u (as an element of the ring R).

```
u[i] := x
```

Given an element u belonging to a submodule M of the R-module $T = R^{(n)}$, a positive integer i, $1 \leq i \leq n$, and an element x of the ring R, redefine the i-th component of u to be x. The parent of u is changed to T (since the modified element u need not lie in M).

54.6.2.3 Normalization

```
Normalize(u)
```

```
Normalise(u)
```

The element u must belong to an R-module, where R is either a field, the ring of integers or a univariate polynomial ring over a field. Assume that the vector u is non-zero. If R is a field then `Normalize` returns $\frac{1}{a} * u$, where a is the first non-zero component of u. If R is the ring of integers, `Normalize` returns $\epsilon * u$, where ϵ is $+1$ if the first non-zero component of u is positive, and -1 otherwise. If R is the polynomial ring $K[x]$, K a field, then `Normalize` returns $\frac{1}{a} * u$, where a is the leading coefficient of the first non-zero (polynomial) component of u. If u is the zero vector, it is returned as the value of this function.

```
Rotate(u, k)
```

Given a vector u, return the vector obtained from u by rotating by k coordinate positions.

```
Rotate(~u, k)
```

Given a vector u, destructively rotate u by k coordinate positions.

Example H54E3

We illustrate the use of the arithmetic operators for module elements by applying them to elements of the module of 4-tuples over the polynomial ring $\mathbf{Z}[x]$.

```
> P<x> := PolynomialRing(IntegerRing());
> M := RModule(P, 4);
> a :=  M ! [ 1+x, -x, 2+x, 0 ];
> b := M ! [ 1+x+x^2, 0, 1-x^7, 2*x ];
> a + b;
(x^2 + 2*x + 2    -x    -x^7 + x + 3     2*x)
> -a;
(-x - 1      x -x - 2      0)
> a - b;
(       -x^2          -x x^7 + x + 1        -2*x)
> (1-x + x^2)*a;
(x^3 + 1    -x^3 + x^2 - x    x^3 + x^2 - x + 2    0)
> a*(1-x);
(    -x^2 + 1      x^2 - x -x^2 - x + 2            0)
> a[3];
x + 2
> a[3] := x - 2;
> a;
(x + 1    -x x - 2     0)
> ElementToSequence(a - b);
[
    -x^2,
    -x,
    x^7 + x - 3,
    -2*x
]
> Support(a);
{ 1, 2, 3 }
```

54.6.3 Properties of Vectors

`IsZero(u)`

Returns `true` if the element u of the R-module M is the zero element.

`Depth(v)`

The index of the first non-zero entry of the vector v (0 if none such).

`Support(u)`

A set of integers giving the positions of the non-zero components of the vector u.

`Weight(u)`

The number of non-zero components of the vector u.

54.6.4 Inner Products

`(u, v)`

`InnerProduct(u, v)`

Return the inner product of the vectors u and v with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot v^{tr}$. Otherwise, this is simply $u \cdot v^{tr}$.

`Norm(u)`

Return the norm product of the vector u with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot u^{tr}$. Otherwise, this is simply $u \cdot u^{tr}$.

54.7 Bases

The application of the functions in this section is restricted either to vector spaces or to torsion-free modules over a Euclidean Domain.

For a full description of the basis functions for a module defined over a field, the reader is referred to the chapter on vector spaces.

```
Basis(M)
```

The current basis for the free R-module M, R an ED, returned as a sequence of module elements.

```
Rank(M)
```

The rank of the free R-module M.

```
Coordinates(M, u)
```

Given a vector u belonging to the rank r free R-module M, R an Euclidean Domain, with basis $u_1, \ldots, u_r$, return a sequence $[a_1, \ldots, a_r]$ giving the coordinates of u relative to the M-basis: $u = a_1 * u_1 + \cdots + a_r * u_r$.

54.8 Submodules

54.8.1 Construction of Submodules

Submodules may be defined for any type of module. However, functions that depend upon membership testing are only implemented for modules over Euclidean Domains (EDs). The conventions defining the presentations of submodules are as follows:

If M has been created using the function `RSpace`, then every submodule of M is given in terms of a generating set consisting of elements of M, i.e. by means of an embedded generating set.

If M has been created using the function `RModule`, then every submodule of M is given in terms of a reduced basis.

```
sub< M | L >
```

Given an R-module M, construct the submodule N generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators stored for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i. Repetitions of an element and occurrences of the zero element are removed (unless N is trivial).

The constructor returns the submodule N and the inclusion homomorphism $f : N \to M$.

Example H54E4

We construct a submodule of the 4-dimensional vector space over the field of rational function $F[x]$, where F is $\mathbf{F}_5$.

```
> P := PolynomialRing(GF(5));
> R<x> := FieldOfFractions(P);
> M := RSpace(R, 4);
> N := sub< M | [1, x, 1-x, 0], [1+2*x-x^2, 2*x, 0, 1-x^4 ] >;
> N;
Vector space of degree 4, dimension 2 over Field of Fractions in x over
Univariate Polynomial Algebra over GF(5)
Generators:
(1    x    4*x + 1    0)
(4*x^2 + 2*x + 1    2*x    0    4*x^4 + 1)
Echelonized basis:
(1  0  3/(x + 4)  (x^3 + x^2 + x + 1) / (x + 4))
(0  1  (4*x^2 + 2*x + 1) / (x^2 + 4*x)  (4*x^3 + 4*x^2 + 4*x + 4) / (x^2 + 4*x))
```

54.8.2 Operations on Submodules

54.8.3 Membership and Equality

The following operations are only available for submodules of $R^{(n)}$, $\mathrm{Hom}_R(M, N)$ and $R[G]$, where R is a Euclidean Domain. If the modules involved are $R[G]$-modules, the operators refer to the underlying R-module.

```
u in M
```

Returns `true` if the element u lies in the R-module M, where u and M belong to the same R-module.

```
u notin M
```

Returns `true` if the element u does not lie in the R-module M, where u and M belong to the same R-module.

```
N subset M
```

Returns `true` if the R-module N is contained in the R-module M, where M and N belong to a common R-module.

```
N notsubset M
```

Returns `true` if the R-module N is not contained in the R-module M, where M and N belong to a common R-module.

```
M eq N
```

Returns `true` if the R-modules N and M are equal, where N and M belong to a common R-module.

```
M ne N
```

Returns true if the R-modules N and M are not equal, where N and M belong to a common R-module.

54.8.4 Operations on Submodules

The following operations are only available for submodules of $R^{(n)}$, $\mathrm{Hom}_R(M,N)$ and $R[G]$, where R is a Euclidean Domain. If the modules involved are $R[G]$-modules, the operators refer to the underlying R-module.

```
M + N
```

Sum of the submodules M and N, where M and N belong to a a common R-module.

```
M meet N
```

Intersection of the submodules M and N, where M and N belong to a common R-module.

54.9 Quotient Modules

54.9.1 Construction of Quotient Modules

```
quo< M | L >
```

Given an R-module M, construct the quotient module $P = M/N$, where N is the submodule generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators constructed for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i.

The constructor returns the quotient module P and the natural homomorphism $f : M \to P$.

54.10 Homomorphisms

Throughout this part of the chapter, when discussing the set of all R-homomorphisms from the R-module M into the R-module N, it will be assumed that R is a **commutative** ring. We further assume that M and N are free R-modules and that bases for these modules are present. The module $\mathrm{Hom}_R(M, N)$ will be identified with the module of $m \times n$ matrices over R. Thus, an element of $\mathrm{Hom}_R(M, N)$ is represented as a matrix relative to the bases of the generic modules corresponding to M and N. For this reason, we will refer to these modules as *matrix modules*.

We remind the reader that submodules of $\mathrm{Hom}_R(M, N)$ are always presented in embedded form. If the user wishes to have submodules presented in reduced form then he/she should use the natural isomorphism between $R^{(m\times n)}$ and $R^{(mn)}$.

It should be noted that essentially operation defined for tuple modules, their elements and submodules applies to matrix modules. Thus, all of the operations discussed earlier in this chapter apply to matrix modules.

The modules M and N may themselves be matrix modules. In this case, the resulting matrix module has either a right or left action and an element belonging to it transforms a (homomorphism) element of M into a (homomorphism) element of N.

The function `Reduce` may be used to construct for a matrix module H the matrix module H' equivalent to H whose elements are with respect to the actual bases of the domain and codomain of elements of H (not the generic bases of the domain and codomain).

54.10.1 $\mathrm{Hom}_R(M, N)$ for R-modules

```
Hom(M, N)
```

If M is the tuple module $R^{(m)}$ and N is the tuple module $R^{(n)}$, create the module $\mathrm{Hom}_R(M, N)$ as the (R, R)-bimodule $R^{(m\times n)}$, represented as the set of all $m \times n$ matrices over R. The module is created with the standard basis, $\{E_{ij} \mid i = 1\ldots, m, j = 1\ldots, n\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

```
RMatrixSpace(R, m, n)
```

Given a ring R and positive integers m and n, construct $H = \mathrm{Hom}(M, N)$, where $M = R^{(m)}$ and $N = R^{(n)}$, as the free (R, R)-bimodule $R^{(m\times n)}$, consisting of all $m \times n$ matrices over R. The module is created with the standard basis, $\{E_{ij} \mid i = 1\ldots, m, j = 1\ldots, m\}$. Note that the modules M and N are created by this function and may be accessed as `Domain(H)` and `Codomain(H)`, respectively.

Example H54E5

We construct the vector spaces V and W of dimensions 3 and 4, respectively, over the field of two elements and then define M to be the module of homomorphisms from V into W.

```
> F2 := GaloisField(2);
> V  := VectorSpace(F2, 3);
> W  := VectorSpace(F2, 4);
> M  := Hom(V, W);
> M;
Full KMatrixSpace of 3 by 4 matrices over GF(2)
```

54.10.2 $\mathrm{Hom}_R(M, N)$ for Matrix Modules

`Hom(M, N, "right")`

Suppose M is a matrix module over the coefficient ring R whose elements are a by b matrices and have domain D and codomain C. Suppose also that N is a matrix module over the coefficient ring R whose elements are a by c matrices and have domain D and codomain C'. Then the homomorphism module $H = \mathrm{Hom}(M, N)$ with right multiplication action exists and consists of all b by c matrices over R which multiply an element of M on the right to yield an element of N. This function constructs H explicitly. The domain of elements of H is then M and the codomain of elements of H is N and the elements are b by c matrices over R which multiply an element of M on the right to yield an element of N. Note that if M and N are proper submodules of their respective generic modules, then H may be a proper submodule of its generic module, and the correct basis of H will be explicitly constructed.

`Hom(M, N, "left")`

Suppose M is a matrix module over the coefficient ring R whose elements are a by c matrices and have domain D and codomain C. Suppose also that N is a matrix module over the coefficient ring R whose elements are b by c and have domain D' and codomain C. Then the homomorphism module $H = \mathrm{Hom}(M, N)$ with left multiplication action exists and consists of all b by a matrices over R which multiply an element of M on the left to yield an element of N. This function constructs H explicitly. The domain of elements of H is then M and the codomain of elements of H is N and the elements are b by a matrices over R which multiply an element of M on the right to yield an element of N. Note that if M and N are proper submodules of their respective generic modules, then H may be a proper submodule of its generic module, and the correct basis of H will be explicitly constructed.

Example H54E6

We construct two homomorphism modules H_1 and H_2 over $\mathbf{Q}$ and then the homomorphism module $H = \mathrm{Hom}(H_1, H_2)$ with right matrix action.

```
> Q := RationalField();
> H1 := sub<RMatrixSpace(Q, 2, 3) | [1,2,3, 4,5,6], [0,0,1, 1,3,3]>;
> H2 := sub<RMatrixSpace(Q, 2, 4) | [6,5,7,1, 15,14,16,4], [0,0,0,0, 1,2,3,4]>;
> H := Hom(H1, H2, "right");
> H: Maximal;
KMatrixSpace of 3 by 4 matrices and dimension 1 over Rational Field
```

```
Echelonized basis:

[   1    2    3    4]
[-1/2   -1 -3/2   -2]
[   0    0    0    0]

> H1.1 * H.1;
[  0   0   0   0]
[3/2   3 9/2   6]
> H1.1 * H.1 in H2;
true
> Image(H.1): Maximal;
KMatrixSpace of 2 by 4 matrices and dimension 1 over Rational Field
Echelonized basis:

[0 0 0 0]
[1 2 3 4]

> Kernel(H.1): Maximal;
KMatrixSpace of 2 by 3 matrices and dimension 1 over Rational Field
Echelonized basis:

[ 1  2  6]
[ 7 14 15]

> H1 := sub<RMatrixSpace(Q,2,3) | [1,2,3, 4,5,6]>;
> H2 := sub<RMatrixSpace(Q,3,3) | [1,2,3, 5,7,9, 4,5,6]>;
> H := Hom(H1, H2, "left");
> H: Maximal;
KMatrixSpace of 3 by 2 matrices and dimension 1 over Rational Field
Echelonized basis:

[1 0]
[1 1]
[0 1]
> Image(H.1);
KMatrixSpace of 3 by 3 matrices and dimension 1 over Rational Field
> Kernel(H.1);
KMatrixSpace of 2 by 3 matrices and dimension 0 over Rational Field
```

54.10.3 Modules $\mathrm{Hom}_R(M, N)$ with Given Basis

```
RMatrixSpaceWithBasis(Q)
```

Given a sequence Q of k independent matrices each lying in a matrix space $H = \mathrm{Hom}(M, N)$, where $M = R^{(m)}$ and $N = R^{(n)}$, construct the subspace of H of dimension k whose basis is Q. The basis is echelonized internally but all functions which depend on the basis of the matrix space (e.g. `Coordinates`) will use the given basis Q.

```
KMatrixSpaceWithBasis(Q)
```

Given a sequence Q of k independent matrices each lying in a matrix space $H = \mathrm{Hom}(M, N)$, where $M = K^{(m)}$ and $N = K^{(n)}$, with K a field, construct the subspace of H of dimension k whose basis is Q. The basis is echelonized internally but all functions which depend on the basis of the matrix space (e.g. `Coordinates`) will use the given basis Q.

54.10.4 The Endomorphsim Ring

```
EndomorphismAlgebra(M)
```

If M is the free R-module $R^{(m)}$, create the matrix algebra $\mathrm{Mat}_m(R)$. The algebra is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, m\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

Example H54E7

We construct the endomorphism ring of the 4-dimensional vector space over the rational field.

```
> Q  := RationalField();
> R4 := RModule(Q, 4);
> M  := EndomorphismAlgebra(R4);
> M;
Full Matrix Algebra of degree 4 over Rational Field
```

54.10.5 The Reduced Form of a Matrix Module

```
Reduce(H)
```

Suppose H is a matrix module whose elements have domain A and codomain B. Suppose first that A and B are tuple modules (R-spaces, R-modules, or RG-modules) and that A has degree a and dimension d while B has degree b and dimension e. (For the reduced cases (R-modules or RG-modules), a equals d and b equals e). The elements of H have the natural representation with respect to the standard embedded basis of the generic modules of A and B. Thus H has degree a by b. So one can multiply a 1 by a vector of A directly by an element h of H (in the natural matrix way) to get a 1 by b vector of B. Now suppose A and B are in the embedded form (R-space) and h is in H. Then h is an a by b matrix but there is a corresponding d by e matrix h' which gives the same transformation of h from A to B but is *with respect to the bases of* A *and* B. We call h' the *reduced* form of h. Also, there is the *reduced module* H' corresponding to H. This function constructs the reduced module H' corresponding to H, together with the epimorphism f from H onto H'. Note that if A and B are in reduced form, then H' is the same as H.

Suppose secondly that A and B are matrix modules themselves. Suppose $A = \mathrm{Hom}(D_1, C_1)$, $B = \mathrm{Hom}(D_1, C_2)$, and $H = \mathrm{Hom}(A, B)$ with the right multiplication action. Suppose also that A has degree r by s and dimension d while B has degree r by t, and dimension e. Then H would have degree s by t so an element h of H would be s by t and would multiply a r by s element of A on the right to yield an r by t element of B. Then the reduced matrix h' corresponding to a matrix h of H would be a d by e matrix corresponding to the bases of A and B. This function similarly constructs the reduced module H' corresponding to H, together with the epimorphism f from H onto H'. Note also that in this case the domain and codomains of H' are the generic R-spaces (tuple modules) corresponding to A (of dimension d) and B (of dimension e). Similarly, for the left multiplication action there is the corresponding reduced module constructed in the obvious way.

Note also that the kernel of the epimorphism f is the submodule of H which consists of all matrices which transform all elements of A to the zero element of B.

Example H54E8

We demonstrate the function `Reduce` for a homomorphism module from one vector space to another.

```
> V1 := sub<VectorSpace(GF(3), 3) | [1,0,1], [0,1,2]>;
> V2 := sub<VectorSpace(GF(3), 4) | [1,1,0,2], [0,0,1,2]>;
> H := Hom(V1, V2);
> H;
KMatrixSpace of 3 by 4 matrices and dimension 8 over GF(3)
> R, f := Reduce(H);
> R;
Full KMatrixSpace of 2 by 2 matrices over GF(3)
> H.1;
```

```
[1 0 0 0]
[0 1 0 2]
[0 1 0 2]
> f(H.1);
[1 0]
[0 0]
> V1.1;
(1 0 1)
> V1.1 * H.1;
(1 1 0 2)
> Coordinates(V2, V1.1 * H.1);
[ 1, 0 ]
> Coordinates(V2, V1.2 * H.1);
[ 0, 0 ]
> Kernel(f): Maximal;
KMatrixSpace of 3 by 4 matrices and dimension 4 over GF(3)
Echelonized basis:

[1 0 0 0]
[2 0 0 0]
[2 0 0 0]

[0 1 0 0]
[0 2 0 0]
[0 2 0 0]

[0 0 1 0]
[0 0 2 0]
[0 0 2 0]

[0 0 0 1]
[0 0 0 2]
[0 0 0 2]
> R.1@@f;
[1 0 0 0]
[0 1 0 2]
[0 1 0 2]
```

Example H54E9

We demonstrate the function `Reduce` for a homomorphism module from one homomorphism module to another. Note that the reduced module has the same dimension as the original module but larger degrees!

```
> V1 := VectorSpace(GF(3), 2);
> V2 := VectorSpace(GF(3), 3);
> V3 := VectorSpace(GF(3), 4);
> H1 := Hom(V1, V2);
> H2 := Hom(V1, V3);
> H := Hom(H1, H2, "right");
> H1;
Full KMatrixSpace of 2 by 3 matrices over GF(3)
> H2;
Full KMatrixSpace of 2 by 4 matrices over GF(3)
> H;
Full KMatrixSpace of 3 by 4 matrices over GF(3)
> R,f := Reduce(H);
> R;
KMatrixSpace of 6 by 8 matrices and dimension 12 over GF(3)
> X := H.1;
> X;
[1 0 0 0]
[0 0 0 0]
[0 0 0 0]
> f(X);
[1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
> Domain(X);
Full KMatrixSpace of 2 by 3 matrices over GF(3)
> Domain(f(X));
Full Vector space of degree 6 over GF(3)
> Image(X): Maximal;
KMatrixSpace of 2 by 4 matrices and dimension 2 over GF(3)
Echelonized basis:

[1 0 0 0]
[0 0 0 0]

[0 0 0 0]
[1 0 0 0]

> Image(f(X));
Vector space of degree 8, dimension 2 over GF(3)
Echelonized basis:
(1 0 0 0 0 0 0 0)
(0 0 0 0 1 0 0 0)
> Kernel(X): Maximal;
KMatrixSpace of 2 by 3 matrices and dimension 4 over GF(3)
Echelonized basis:

[0 1 0]
[0 0 0]
```

```
[0 0 1]
[0 0 0]

[0 0 0]
[0 1 0]

[0 0 0]
[0 0 1]

> Kernel(f(X)): Maximal;
Vector space of degree 6, dimension 4 over GF(3)
Echelonized basis:
(0 1 0 0 0 0)
(0 0 1 0 0 0)
(0 0 0 0 1 0)
(0 0 0 0 0 1)
```

54.10.6 Construction of a Matrix

```
M ! Q
```

Given the matrix bimodule M over the ring R, and the sequence $Q = [a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{m1}, \ldots, a_{mn}]$ whose terms are elements of the ring R, construct the $m \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$$

as an element of M. In the context of the `sub` or `quo` constructors the coercion clause `M !` may be omitted.

Example H54E10

We create the 4×4 Hilbert matrix $h4$ as an element of the endomorphism ring of the 4-dimensional vector space over the rational field.

```
> Q  := RationalField();
> R4 := RModule(Q, 4);
> M  := EndomorphismAlgebra(R4);
> h4 := M ! [ 1/i :  i in [1 .. 16 ] ];
> h4;
[   1  1/2  1/3  1/4]
[ 1/5  1/6  1/7  1/8]
[ 1/9 1/10 1/11 1/12]
[1/13 1/14 1/15 1/16]
```

54.10.7 Element Operations

All operations that apply to elements of tuple modules also apply to elements of matrix modules. Here, we confine our discussion to those operations which are special to matrix modules.

Throughout this section, M is a submodule of $R^{(m)}$, N is a submodule of $R^{(n)}$ and a is a homomorphism belonging to $\mathrm{Hom}_R(M, N)$, where R is a Euclidean Domain.

See also the chapter on general matrices for many other functions applicable to matrices (e.g., `EchelonForm`).

```
u * a
```

```
a(u)
```

Given an element u belonging to the module M, return the image of u under the homomorphism a as an element of the module N.

```
a * b
```

Given a homomorphism a belonging to a submodule of $\mathrm{Hom}(M, N)$, and a homomorphism b belonging to a submodule of $\mathrm{Hom}(N, P)$, return the composition of the homomorphisms a and b as an element of $\mathrm{Hom}(M, P)$. Note that if $\mathrm{Hom}(M, P)$ does not already exist, it will be created.

```
a ^ -1
```

Given a homomorphism a belonging to a submodule of $\mathrm{Hom}(M, N)$ with M and N having the same dimension, return the inverse of a as an element of $\mathrm{Hom}(N, M)$.

```
Codomain(S)
```

Given a submodule S of the module $\mathrm{Hom}(M, N)$, return the module N.

```
Codomain(a)
```

The codomain N of the homomorphism a belonging to $\mathrm{Hom}(M, N)$.

```
Cokernel(a)
```

The cokernel for the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$.

```
Domain(S)
```

The domain M of the submodule S belonging to the module $\mathrm{Hom}(M, N)$.

```
Domain(a)
```

The domain M of the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$.

```
Image(a)
```

The image of the homomorphism a belonging to the module $H = \mathrm{Hom}(M, N)$, returned as a submodule of N. Note that if the domain and codomain of a are matrix modules themselves, the image will be with respect to the appropriate action (right or left).

```
Kernel(a)
```

```
NullSpace(a)
```

The kernel of the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$, returned as a submodule of M. Note that if the domain and codomain of a are matrix modules themselves, the kernel will be with respect to the appropriate action (right or left).

```
Morphism(M, N)
```

Assuming the R-module M was created as a submodule of the module N, return the matrix defining the inclusion homomorphism $\phi : M \to N$ as an element of $\mathrm{Hom}_R(M, N)$. Thus ϕ gives the correspondence between elements of M (represented with respect to the standard basis of M) and elements for N.

```
Rank(a)
```

The dimension of the image of the homomorphism a, i.e. the rank of a.

```
IsBijective(a)
```

Returns `true` if the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$ is a bijective mapping.

```
IsInjective(a)
```

Returns `true` if the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$ is an injective mapping.

```
IsSurjective(a)
```

Returns `true` if the homomorphism a belonging to the module $\mathrm{Hom}_R(M, N)$ is a surjective mapping.

Example H54E11

We illustrate some of these operations in the context of the module $\mathrm{Hom}_R(M, N)$, where M and N are, respectively, the 4-dimensional and 3-dimensional vector spaces over $GF(8)$.

```
> K<w> := GaloisField(8);
> V3 := VectorSpace(K, 3);
> V4 := VectorSpace(K, 4);
> M := Hom(V4, V3);
> A := M ! [1, w, w^5, 0,  w^3, w^4, w, 1,  w^6, w^3, 1, w^4 ];
> A;
[  1   w w^5]
[  0 w^3 w^4]
[  w   1 w^6]
[w^3   1 w^4]
> Rank(A);
3
> Image(A);
```

```
Full Vector space of degree 3 over GF(2^3)
> Kernel(A);
Vector space of degree 4, dimension 1 over GF(2^3)
Echelonized basis:
(  1 w^5   1   1)
> Cokernel(A);
Vector space of degree 3, dimension 0 over GF(2^3)
```

55 MODULES OVER DEDEKIND DOMAINS

Chapter 55

MODULES OVER DEDEKIND DOMAINS

55.1 Introduction

Since the structure theory for modules over arbitrary orders (which are in general not Dedekind domains) is very unsatisfactory, modules over orders in Magma are always modules over some maximal order of a number field or function field, they form a magma of type `ModDed`.

Let k be a number field or function field and $\mathcal{O}_k$ its ring of integers. Since $\mathcal{O}_k$ is a Dedekind domain, every finitely generated torsion free module M over $\mathcal{O}_k$ has a representation as a direct sum

$$M = \sum_{i=1}^{m} \mathcal{A}_i \alpha_i = \{\sum_{i=1}^{m} a_i \alpha_i \mid a_i \in \mathcal{A}_i\}$$

with (fractional) ideals $\mathcal{A}_i$ and elements $\alpha_i \in kM \cong k^r$.

A (not necessarily direct) sum $\sum_{i=1}^{m} \mathcal{A}_i \alpha_i$ will be represented as a pseudo–matrix $(\mathcal{A}|A)$ where $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_m)^t$ is a column vector of ideals and $A = (\alpha_1, \ldots, \alpha_m)^t \in k^{m \times r}$ is a matrix. The ideals $\mathcal{A}_i$ are called coefficient ideals.

This pseudo–matrix is called a pseudo–basis iff the sum is direct. A pseudo–matrix $(\mathcal{A}|A)$ is in Hermite normal form iff there are $s \le m$, $1 \le i_1 < i_2 < \cdots < i_s$ such that $A_{j,l} = 0$ $(1 \le l < i_j)$, $A_{j,i_j} = 1$ and $A_{j,l}$ is reduced modulo $\mathcal{A}_j {\mathcal{A}_l}^{-1}$. For $j > s$ we have $A_{j,l} = 0$.

This normal form is unique if a suitable reduction is used.

As a consequence of this normalisation, usually $\alpha_i \notin M$. To be precise: $\alpha_i \in M$ iff $1 \in \mathcal{A}_i$.

All modules are in Hermite normal form, i.e. every module is represented by a pseudo–basis in Hermite normal form.

General (non torsion free) modules are represented as quotients of a torsion free module M and a submodule S. Elements of $Q := M/S$ are represented as elements of M, arithmetic in Q is reduced to arithmetic in M followed by a reduction modulo the pseudo–basis of S.

55.2 Creation of Modules

Modules over dedekind domains can be created from orders of number fields and function fields and combinations of ideals and vector space elements. Submodules and quotient modules by submodules can also be created.

`Module(O, n)`

Create the free module O^n where O is a dedekind domain.

`Module(O)`

Create the relative order O as a module over its coefficient ring. Also returns the map from the resulting module into O.

`Module(I)`

Create the ideal I of a relative order O as a module over the coefficient ring of the order. Also returns the map from the module into O.

`Module(S)`

Create a module from the sequence of tuples of ideals of a dedekind domain and `ModElts` with entries in the dedekind domain or its field of fractions. The elements of the resulting module will be the sum of products of an element of an ideal and the corresponding `ModElt`. Also returns the map from the vector space into the module.

`Module(S)`

Create the module which is equal to the direct sum of the ideals in the sequence.

`Module(S)`

Create the module which is freely generated by the elements of the sequence S. The elements of the sequence must be `ModElts` with entries in a dedekind domain or field of fractions of a dedekind domain. Also returns the map from the vector space into the module.

Example H55E1

The creation of some simple modules is shown.

```
> x := ext<Integers()|>.1;
> M := MaximalOrder(x^2 + 5);
> Module(M, 5);
Module over Maximal Equation Order with defining polynomial x^2 + 5 over Z
 generated by: (in echelon form)
Principal Ideal of M
Generator:
    M.1 * ( M.1 0 0 0 0 )
Principal Ideal of M
Generator:
    M.1 * ( 0 M.1 0 0 0 )
Principal Ideal of M
```

```
Generator:
    M.1 * ( 0 0 M.1 0 0 )
Principal Ideal of M
Generator:
    M.1 * ( 0 0 0 M.1 0 )
Principal Ideal of M
Generator:
    M.1 * ( 0 0 0 0 M.1 )
> I := 1/5*M;
> Module([I, I^3, I^8]);
Module over Maximal Equation Order with defining polynomial x^2 + 5 over Z
 generated by: (in echelon form)
Fractional Principal Ideal of M
Generator:
    1/5*M.1 * ( M.1 0 0 )
Fractional Principal Ideal of M
Generator:
    1/125*M.1 * ( 0 M.1 0 )
Fractional Principal Ideal of M
Generator:
    1/390625*M.1 * ( 0 0 M.1 )
> V := RModule(M, 3);
> Module([<I, V![0, 1, 0]>, <I^4, V![2, 3, 5]>]);
Module over Maximal Equation Order with defining polynomial x^2 + 5 over Z
Fractional Principal Ideal of M
Generator:
    1/5*M.1 car Fractional Principal Ideal of M
Generator:
    1/125*M.1
```

The same can be done using orders of function fields.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(x^2 - y);
> M := MaximalOrderFinite(F);
> Module(M, 5);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x over
Rational Field
 generated by: (in echelon form)
Ideal of M
Generator:
1 * ( 1 0 0 0 0 )
Ideal of M
Generator:
1 * ( 0 1 0 0 0 )
Ideal of M
Generator:
1 * ( 0 0 1 0 0 )
```

```
Ideal of M
Generator:
1 * ( 0 0 0 1 0 )
Ideal of M
Generator:
1 * ( 0 0 0 0 1 )
> I := 1/5*M;
> Module([I, I^3, I^8]);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x over
Rational Field
 generated by: (in echelon form)
Ideal of M
Generator:
1/5 * ( 1 0 0 )
Ideal of M
Generator:
1/125 * ( 0 1 0 )
Ideal of M
Generator:
1/390625 * ( 0 0 1 )
> V := RModule(M, 3);
> Module([<I, V![0, 1, 0]>, <I^4, V![2, 3, 5]>]);
Integral Module over Maximal Equation Order of F over Univariate Polynomial Ring
in x over Rational
Field
Ideal of M
Generator:
1/5 car Ideal of M
Generator:
1/125
```

---

sub< M | m >

sub< M | m1, .., mn >

Construct the submodule of the module M generated by the elements in the sequence or list of elements m. Also returns the inclusion map of the submodule into M.

quo< M | S >

quo< M | m >

quo< M | m1, .., mn >

Construct the quotient of the module M by the submodule S or the submodule generated by the elements of the sequence or list of elements m. Also returns the inclusion map of the quotient module into M.

Example H55E2

Use of the sub and quo constructors is illustrated below. Let M and V be as above when they were referring to number fields.

```
> Mod := Module([V|[0,1,0], [4,4,0]]);
> S1 := sub<Mod | >;
> S1;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
(0)
> Q1 := quo<Mod | Mod>;
> Q1;
Quotient of Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
by Integral Module over Maximal Equation Order with defining polynomial
x^2 + 5 over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
> S2 := sub<Mod | Mod.2>;
> S2;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    M.1
> Q2 := quo<Mod | Mod.2>;
> Q2;
Quotient of Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
by Integral Module over Maximal Equation Order with defining polynomial
x^2 + 5 over Z
Principal Ideal of M
Generator:
    M.1
> S3 := sub<Mod | 4*Mod.1, Mod.2>;
> S3;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
> Q3 := quo<Mod | >;
> Q3;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
> Q4 := quo<Mod | S1>;
> Q4;
Quotient of Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
by Integral Module over Maximal Equation Order with defining polynomial
x^2 + 5 over Z
(0)
```

55.3 Elementary Functions

Various simple properties of a module can be retrieved using the following functions.

`BaseRing(M)`

`CoefficientRing(M)`

The dedekind domain which M is a module over.

`Degree(M)`

The dimension of the vector space the module M embeds into.

`Ngens(M)`

`NumberOfGenerators(M)`

The minimum number of vectors and ideals which generate the module M.

`M . i`

The vector of the *ith* vector and ideal pair generating the module M.

`Determinant(M)`

The determinant of the module M.

`Dimension(M)`

The dimension of the vector space spanned by the module M over its coefficient ring. This is the same as the number of generators of a pseudo basis of M.

`Contents(M)`

`UseBasis` BoolElt *Default :* `false`

The contents of the module M, ie. the gcd of the ideals obtained by multiplying the coefficient ideals by the ideal generated by the coefficients in the corresponding generators. The parameter `UseBasis` decides whether a pseudo basis or pseudo generators are used.

`Simplify(M)`

`UseBasis` BoolElt *Default :* `false`

Computes a module of contents 1 by scaling each coefficient ideal by the inverse of the contents of the module M. The parameter `UseBasis` determines if the operations are performed on the pseudo generators or the pseudo basis of M.

`EmbeddingSpace(M)`

The canonical vector space containing the module M, ie.. M tensored with the field of fractions of the coefficient ring.

Example H55E3

The use of some elementary functions on a module is shown below.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(x^2 - y);
> M := MaximalOrderFinite(F);
> Vs := RModule(M, 2);
> s := [Vs | [1, 3], [2, 3]];
> Mods := Module(s);
> CoefficientRing(Mods);
Maximal Equation Order of F over Univariate Polynomial Ring in x over Rational
Field
> Mods.1;
(1 0)
> Determinant(Mods);
Ideal of M
Generator:
```

```
-3
> Vs := RSpace(M, 2);
> s := [Vs | [1, 3], [2, 3]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!Vs![1, 3]>;
> qMods := quo<Mods | sMods>;
> Degree(Mods);
2
> Ngens(Mods);
2
> Ngens(sMods);
1
> Degree(sMods);
2
> Degree(qMods);
2
> Ngens(qMods);
2
> Determinant(Mods);
Ideal of M
Basis:
[1]
>  Determinant(sMods);
>>   Determinant(sMods);
               ^
Runtime error in 'Determinant': Module must be square
> Determinant(qMods);
Ideal of M
Basis:
[1]
```

55.4 Predicates on Modules

Modules and potential elements can be tested against each other for a few properties.

`M eq N`

Return `true` if M and N are equal as modules.

`x in M`

Return `true` if x can be coerced into the module M.

`M subset N`

Return `true` if M is a submodule of N. An embedding map of M in N can be returned by `IsSubmodule(M, N)`.

55.5 Arithmetic with Modules

Some arithmetic operations can be carried out involving modules and their elements and compatible ideals to gain more modules.

```
I * M
```

```
M * I
```

The module generated by the products of the ideals of the module M with I.

```
M1 + M2
```

The union of the modules $M1$ and $M2$.

```
u * I
```

```
I * u
```

The module containing elements which are products of the dedekind module element u and an element lying in the ideal I.

Example H55E4

Some module predicates and arithmetic are shown below.

```
> P<x> := PolynomialRing(Integers());
> K :=  NumberField([x^5 + 3, x^2 + 2]);
> M := MaximalOrder(K);
> Vs := RModule(M, 2);
> s := [Vs | [1, 3], [2, 3]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!Vs![1, 3]>;
> Mods eq sMods;
false
> [1, 0] in Mods;
true
> [1, 0] in sMods;
false
> sMods subset Mods;
true
> Vs := RSpace(M, 2);
> s := [Vs | [Random(M, 3), 3], [2, Random(M, 2)]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!s[1]>;
> (7*M + 11*K.1*M)*sMods;
Module over Maximal Equation Order with defining polynomial x^5 + [3, 0] over
its ground order
 generated by:
Ideal of M
Two element generators:
    7/1*$.1*M.1
    11/1*$.1*M.2 * ( M.1 + (-$.1 - 2/1*$.2)*M.2 + $.2*M.3 + (2/1*$.1 +
2/1*$.2)*M.5 3/1*$.1*M.1 )
```

```
 in echelon form:
Ideal of M
Two element generators:
    21/1*$.1*M.1
    33/1*$.1*M.2 * ( 1/3*$.1*M.1 + (-1/3*$.1 - 2/3*$.2)*M.2 + 1/3*$.2*M.3 +
(2/3*$.1 + 2/3*$.2)*M.5 M.1 )
> Mods + sMods;
Module over Maximal Equation Order with defining polynomial x^5 + [3, 0] over
its ground order
 generated by: (in echelon form)
Ideal of M
Two element generators:
    148919257164/1*$.1*M.1
    (148919257048/1*$.1 + 26/1*$.2)*M.1 + (32/1*$.1 + 148919257015/1*$.2)*M.2 +
    (196/1*$.1 + 148919257117/1*$.2)*M.3 + (148919257163/1*$.1 + 76/1*$.2)*M.4 +
    (148919257077/1*$.1 + 148919257116/1*$.2)*M.5 * ( M.1 0 )
Ideal of M
Two element generators:
    3/1*$.1*M.1
    ($.1 - $.2)*M.1 + -M.3 + M.4 + ($.1 + $.2)*M.5 * ( (-19866460521/1*$.1 -
33/1*$.2)*M.1 + (1/3*$.1 + 1/3*$.2)*M.4 + (1/3*$.1 + 1/3*$.2)*M.5 M.1 )
> 4*sMods;
Module over Maximal Equation Order with defining polynomial x^5 + [3, 0] over
its ground order
 generated by:
Principal Ideal of M
Generator:
    4/1*$.1*M.1 * ( (2/1*$.1 - 2/1*$.2)*M.1 + ($.1 + 2/1*$.2)*M.2 + (2/1*$.1 +
2/1*$.2)*M.3 + (2/1*$.1 - $.2)*M.5 3/1*$.1*M.1 )
 in echelon form:
Principal Ideal of M
Generator:
    12/1*$.1*M.1 * ( (2/3*$.1 - 2/3*$.2)*M.1 + (1/3*$.1 + 2/3*$.2)*M.2 +
(2/3*$.1 + 2/3*$.2)*M.3 + (2/3*$.1 - 1/3*$.2)*M.5 M.1 )
```

55.6 Basis of a Module

The basis of a module is given by vectors. However, more complete information can be supplied which includes the ideals.

`Basis(M)`

A sequence of vectors which correspond to a pseudo basis of the module M.

`PseudoBasis(M)`

A sequence of tuples containing ideals and vectors which generate the module M. The vectors are guaranteed to be linearly independent.

`PSeudoGenerators(M)`

A sequence of tuples containing ideals and vectors which generate the module M. This will return the data used to define the module, so that in contrast to `PseudoBasis` the vectors will in general not be independent.

55.7 Other Functions on Modules

Intersections of modules can taken. Several other functions are also available.

`M1 meet M2`

Return the intersection of the modules $M1$ and $M2$.

`Dual(M)`

The module dual to M.

`ElementaryDivisors(M, N)`

The elementary divisors (ideals) of the torsion part of the quotient R-module M/N: For $N \subseteq M$ we get

$$T(M/N) \cong \oplus_{i=1}^{n} R/\mathcal{A}_i$$

The $\mathcal{A}_i$ are unique if we require $R \subseteq \mathcal{A}_1 \subseteq \cdots \subseteq \mathcal{A}_n$. The $\mathcal{A}_i$ are called the elementary divisors (or elementary ideals) of M/N. This corresponds to the Smith normal form for integral matrices.

`SteinitzClass(M)`

The Steinitz class of the module M.

`SteinitzForm(M)`

The Steinitz (almost–free) form of the module M.

Example H55E5

Some bases and other functions are demonstrated below.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^3 - x^3*y^2 + y - x^7);
> M := MaximalOrderFinite(F);
> Vs := RSpace(M, 2);
> s := [Vs | [1, Random(M, 3)], [Random(M, 3), 3]];
> Mods := Module(s);
> qMods := quo<Mods | Mods!s[2]>;
> Basis(Mods);
[
    ([ 1, 0, 0 ] [ 2*x^2 - 3*x + 1/2, -2/3*x^2 - x + 1/3, 2/3*x^2 - 2*x + 1/2
        ]),
    ([ -x^2 + x + 2/3, -1/3*x^2 - 1, -3/2*x^2 - 2/3*x + 1 ] [ 3, 0, 0 ])
]
> Basis(qMods);
[
    ([ 1, 0, 0 ] [ 2*x^2 - 3*x + 1/2, -2/3*x^2 - x + 1/3, 2/3*x^2 - 2*x + 1/2
        ]),
    ([ -x^2 + x + 2/3, -1/3*x^2 - 1, -3/2*x^2 - 2/3*x + 1 ] [ 3, 0, 0 ])
]
> PseudoBasis(Mods) eq PseudoBasis(qMods);
> Vs := RModule(M, 2);
> s := [Vs | [Random(M, 3), Random(M, 3)], [2, 3]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!s[1]>;
> Mods meet sMods;
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
(x + 1)*c^2 + (-3/2*x^2 - 3/2*x)*c + 1/2*x^2 + 2/3*x + 1/2
> ElementaryDivisors(Mods, sMods);
[ Ideal of M
Basis:
[1 0 0]
[0 1 0]
[0 0 1], Ideal of M
Generator:
0 ]
> Dual(Mods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Fractional ideal of M
Generator:
(-3*x^4 + 21*x^3 + 20*x^2 + 80*x - 387)/(x^17 - 24*x^16 + 192*x^15 - 510*x^14 -
    94/3*x^13 + 304/3*x^12 + 902/3*x^11 - 3109/3*x^10 - 389/9*x^9 + 664/9*x^8 +
    1094/3*x^7 - 1540/3*x^6 - 101/9*x^5 + 128/3*x^4 + 2000/27*x^3 - 4361/9*x^2 -
    427*x + 2056)*c^2 + (-3*x^9 + 48*x^8 - 189*x^7 - 24*x^6 + 3*x^5 - 48*x^4 +
    195*x^3 - 3*x^2 - x + 24)/(x^17 - 24*x^16 + 192*x^15 - 510*x^14 - 94/3*x^13
    + 304/3*x^12 + 902/3*x^11 - 3109/3*x^10 - 389/9*x^9 + 664/9*x^8 + 1094/3*x^7
    - 1540/3*x^6 - 101/9*x^5 + 128/3*x^4 + 2000/27*x^3 - 4361/9*x^2 - 427*x +
    2056)*c + (3*x^12 - 48*x^11 + 192*x^10 + 3*x^9 - 20*x^8 - 56*x^7 + 192*x^6 +
    3*x^5 - 5*x^4 - 8*x^3 + 272/3*x^2 + 128*x - 771)/(x^17 - 24*x^16 + 192*x^15
    - 510*x^14 - 94/3*x^13 + 304/3*x^12 + 902/3*x^11 - 3109/3*x^10 - 389/9*x^9 +
    664/9*x^8 + 1094/3*x^7 - 1540/3*x^6 - 101/9*x^5 + 128/3*x^4 + 2000/27*x^3 -
    4361/9*x^2 - 427*x + 2056) car Ideal of M
Generator:
1/3
```

```
> Dual(sMods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Fractional ideal of M
Generator:
(3/2*x^6 + 3*x^5 - 3/4*x^4 - 4*x^3 - 25/12*x^2 - 5/6*x - 1/2)/(x^17 + 3*x^16 +
    21/4*x^15 + 27/4*x^14 + 1/24*x^13 - 247/24*x^12 - 173/24*x^11 + 61/24*x^10 +
    305/72*x^9 + 17/72*x^8 - 4*x^7 - 53/12*x^6 - 1/8*x^5 + 23/6*x^4 +
    377/108*x^3 + 23/18*x^2 + 1/3*x + 1/8)*c^2 + (-5/2*x^9 - 5*x^8 - 1/4*x^7 +
    9/2*x^6 + 13/4*x^5 + 5/4*x^4 - 3/4*x^3 - 7/4*x^2 - 3/4*x)/(x^17 + 3*x^16 +
    21/4*x^15 + 27/4*x^14 + 1/24*x^13 - 247/24*x^12 - 173/24*x^11 + 61/24*x^10 +
    305/72*x^9 + 17/72*x^8 - 4*x^7 - 53/12*x^6 - 1/8*x^5 + 23/6*x^4 +
    377/108*x^3 + 23/18*x^2 + 1/3*x + 1/8)*c + (x^12 + 2*x^11 - 1/2*x^10 -
    7/2*x^9 - 8/3*x^8 - 5/12*x^7 + 11/4*x^6 + 19/4*x^5 - 1/4*x^4 - 25/6*x^3 -
    67/36*x^2 - 1/3*x - 1/4)/(x^17 + 3*x^16 + 21/4*x^15 + 27/4*x^14 + 1/24*x^13
    - 247/24*x^12 - 173/24*x^11 + 61/24*x^10 + 305/72*x^9 + 17/72*x^8 - 4*x^7 -
    53/12*x^6 - 1/8*x^5 + 23/6*x^4 + 377/108*x^3 + 23/18*x^2 + 1/3*x + 1/8)
> SteinitzClass(Mods) eq SteinitzClass(sMods);
false
> SteinitzForm(Mods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
3 car Ideal of M
Generator:
1
> SteinitzForm(sMods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
1
```

55.8 Homomorphisms between Modules

It is possible to create a homomorphism between two modules, take the image and kernel of such and verify that these are submodules of the codomain and domain respectively. The Hom–module can also be created as a module of a dedekind domain.

`hom< M -> N | T >`

ModuleBasis BoolElt *Default :* true

Return a homomorphism from the module M into the module N as specified by T from which the images of the generators can be inferred. T may be a map between the vector spaces of same degree as M and N, a matrix over the field of fractions or a sequence of vectors. If ModuleBasis is true then the matrix will be taken to be a transformation between the modules and as such will be expected to have size Dimension(M)*Dimension(N) otherwise it will be interpreted as a transformation between the corresponding vector spaces and will be expected to have size Degree(M)*Degree(N).

`Hom(M, N)`

The module of homomorphisms between the module M and the module N and the map from the hom–module to the collection of maps from M to N, (such that given an element of the hom–module a homomorphism from M to N is returned). The module is over the same dedekind domain as M and N.

`IsSubmodule(M, N)`

Return true if M is a submodule of N and the map embedding M into N.

`Morphism(M, N)`

The map giving the morphism from the module M to the module N. Either M is a submodule of N, in which case the embedding of M into N is returned, or N is a quotient module of M, in which case the natural epimorphism from M onto N is returned.

Example H55E6

This example demonstrates the use of homomorphisms between modules over dedekind domains. Let M and V be as above referring to function fields.

```
> S := [V|[0,1,0], [4,4,0]];
> Mod := Module(S);
> W := KModule(FieldOfFractions(M), 4);
> S := [W|[3, 2, 1, 0]];
> N := Module(S);
>  h := hom<Mod -> N | >;
>> h := hom<Mod -> N | >;
          ^
Runtime error in map< ... >: No images given
>  h := hom<Mod -> N | V.1, V.2, V.3>;
>> h := hom<Mod -> N | V.1, V.2, V.3>;
          ^
Runtime error in map< ... >: An image for each generator is required
> h := hom<Mod -> N | W![3, 2, 1, 0], W![3*(M!F.1 + 1), 2*(M!F.1 + 1),
> M!F.1 + 1, 0] >;
> h(Mod!(4*V.1));
( 4 )
> h(Mod!V![0, 1, 0]);
( x^2 + 1 )
> I := Image(h);
> I;
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
1
> K := Kernel(h);
> K;
Integral Module over Maximal Equation Order of F over Univariate Polynomial Ring
in x over Rational Field
Ideal of M
Generator:
1
> IsSubmodule(K, Mod);
true Mapping from: ModDed: K to ModDed: Mod
> H, m := Hom(Mod, N);
> H; m;
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x over
Rational Field
 generated by: (in echelon form)
Ideal of M
Generator:
1/4 * ( 1 0 )
Ideal of M
Generator:
1 * ( 0 1 )
Mapping from: ModDed: H to Power Structure of Map given by a rule [no inverse]
> m(H![5, 20]);
Mapping from: ModDed: Mod to ModDed: N
using
[5]
[20]
```

55.9 Elements of Modules

55.9.1 Creation of Elements

M ! v

Coerce v into an element of M. v can be a sequence of length dimension of M, a module element or vector or an element of another module over a dedekind domain which is compatible with M.

Example H55E7

Let Mod and its submodules and quotient modules be as in the sub and quotient module example above.

```
> m := 4*Mod.1;
> m;
(4/1*M.1 0)
> Q1!m;
( 4/1*M.1 0 )
> Q2!m;
( 4/1*M.1 0 )
> m := Mod!m;
> Q3!m;
( 4/1*M.1 0 )
> Q4!m;
( 4/1*M.1 0 )
>  S1!m;
>> S1!m;
     ^
Runtime error in '!': Illegal coercion
LHS: ModDed
RHS: ModDedElt
> S1!Mod!V!0;
(  )
> S2!Mod!Mod.2;
( M.1 )
> S3!Mod!(4*Mod.1);
( 4/1*M.1 0 )
```

55.9.2 Arithmetic with Elements

Basic arithmetic can be performed with elements of a module over a dedekind domain.

```
x + y
```

The sum of the module elements.

```
x - y
```

The difference of the module elements.

```
u * c
c * u
```

The product of the module element u and the ring element c.

```
u / c
```

The product of u and $1/c$ if it lies in the parent module of u.

```
I * u
u * I
```

The module containing elements which are products of u and an element lying in I.

55.9.3 Other Functions on Elements

Elements of modules over a dedekind domain can be tested for equality and represented as a sequence.

```
x eq y
```

Return `true` if x and y are the same element of a module.

```
ElementToSequence(a)
Eltseq(a)
```

The module element a expressed as a sequence.

55.10 Pseudo Matrices

A pseudo matrix, ie. an object of type `PMat`, is a sequence of ideals together with a matrix. Pseudo matrices arise naturally in the (computational) theory of finitely generated torsion free modules over Dedekind domains, they are a natural extension of ordinary matrices which should be thought of as pseudo matrices where all the ideals are generated by 1. Pseudo matrices are generally used to represent the module that is generated by the rows of the matrix scaled by the elements of the corresponding ideal. Thus, if the matrix is regular, a linear combination of the rows lies in the module if and only if the coefficient of the ith row is a member of the ith ideal. The ideals are therefore called coefficient ideals.

55.10.1 Construction of a Pseudo Matrix

`PseudoMatrix(I, m)`

Construct the pseudo matrix with coefficient ideals the elements of the sequence I and matrix m.

`PseudoMatrix(m)`

Construct the pseudo matrix with trivial coefficient ideals and the matrix m.

`PseudoMatrix(M)`

`Generators` BoolElt *Default :* `false`

Construct the pseudo matrix described by the pseudo basis of the module M. If `Generators` is `true` the pseudo matrix described by the pseudo generators of M is returned.

55.10.2 Elementary Functions

`CoefficientIdeals(P)`

Return the coeffficient ideals of the pseudo matrix P.

`Matrix(P)`

Return the matrix of the pseudo matrix P.

`Order(pm)`

Return the order the pseudo matrix pm is over.

`Dimension(pm)`

The dimension of the pseudo matrix pm. This is the numer of columns of the matrix.

`Length(pm)`

The length or dimension of the pseudo matrix pm. This is the number of coefficient ideals of pm.

55.10.3 Basis of a Pseudo Matrix

`Basis(P)`

Return a list of sequences of ring elements corresponding to the entries of each row of the matrix of the pseudo matrix P.

55.10.4 Predicates

`p1 eq p2`

Return whether the pseudo matrices $p1$ and $p2$ are equal, that is, whether they have the same matrix and the same sequence of coefficient ideals.

55.10.5 Operations with Pseudo Matrices

`Transpose(P)`

Return the pseudo matrix whose coefficient ideals are the same as those of P but whose matrix is the transpose of the matrix of P. This function requires the matrix to be square.

`HermiteForm(X)`

Return the Hermite normal form H of the pseudo matrix X together with a regular transformation matrix such that the module generated by X is the same as the one generated by H and such that (for the matrix parts) $H = TX$ holds.

`VerticalJoin(X, Y)`

Return the pseudo matrix whose matrix is the vertical join of the matrices of the pseudo matrices X and Y with coefficient ideals the concatenation of those of X and Y.

`X meet Y`

Return the intersection of the pseudo matrices X and Y.

`Module(X)`

Return the module $\sum C_i * m_i$ where C_i are the coefficient ideals of the pseudo matrix X and m_i are the rows of the matrix of X.

`I * X`

`X * I`

The pseudo matrix whose coefficient ideals are those of the pseudo matrix X multiplied by I and whose matrix is the matrix of X.

56 CHAIN COMPLEXES

Chapter 56
CHAIN COMPLEXES

56.1 Complexes of Modules

Complexes of modules are a fundamental object in homological algebra. Magma supports the type `ModCpx` representing a complex of modules. Conceptually, a complex is an infinite sequence of modules, indexed by integers, with maps between successive modules such that the composition of any two maps is zero. Complexes are often written

$$\ldots \xrightarrow{f_{n+1}} M_n \xrightarrow{f_n} M_{n-1} \xrightarrow{f_{n-1}} M_{n-2} \xrightarrow{f_{n-2}} \ldots$$

where the map f_n has domain M_n and codomain M_{n-1}. The indices on the modules and maps decrease to the right. In practice, Magma requires all but a finite number of the modules and maps to be zero.

The homomorphism from M_n to M_{n-1} in the complex is the n^{th} boundary map of the complex. The homology of the complex in degree n is the quotient of the kernel of the n^{th} boundary map by the cokernel of the boundary map of degree $n-1$. Magma computes the homology only if both boundary maps are defined. A complex is said to be *exact* if the image of each map is equal to the kernel of the next.

Currently, there are two types of modules over which complexes are supported:

(a) Modules over a basic algebra A (see Chapter 85);

(b) Modules over a multivariate polynomial ring over a field (see Chapter 109 and in particular the function `FreeResolution`).

Most of the functions in this chapter work for either type of module but exceptions are noted.

56.1.1 Creation

`Complex(L, d)`

Given a list L of maps between successive A-modules create the corresponding complex. The last term of the complex has degree d. This function returns an error if the maps don't actually form a complex.

`Complex(f, d)`

Given a map f between A-modules M and N, form the complex consisting of the two term complex whose only map is f. The term N is in degree d.

`ZeroComplex(A, m, n)`

Given a basic algebra A and integers m and n such that $m > n$, create a complex of modules over the basic algebra A, starting with a term of degree m and ending with a term of degree n, where all the modules and maps are zero.

```
Dual(C)
```

```
Dual(C, n)
```

The dual of the complex C over a basic algebra A as a complex over the opposite algebra of A. If an integer n is supplied then the last term of the dual complex is in degree n. Otherwise, the last term of the dual complex is in degree 0.

56.1.2 Subcomplexes and Quotient Complexes

Let C be a complex of A-modules, $M_m, \ldots, M_n$. A subcomplex S of C is a complex whose terms are submodules of the terms of C and whose maps are the restrictions of the maps of C to the terms of S.

```
sub< C | Q >
```

```
sub< C | L >
```

Given a complex C and a sequence Q of submodules of the terms of C, returns the smallest subcomplex whose terms contain the modules in Q. The input can also be a list L of sequences of elements of the successive terms of C. In this case the submodules generated by the elements is computed. The function also returns the chain map giving the inclusion of the subcomplex in C.

```
RandomSubcomplex(C, Q)
```

Given a chain complex C over a basic algebra in degrees a to $a-t+1$ and a sequence $Q = [q_1, \ldots, q_t]$ of natural numbers, the function creates the minimal chain complex whose term in degree $a-i+1$ is a submodule generated by q_i random elements of the term in degree $a-i+1$ of C. The function also returns the chain map that is the inclusion of the subcomplex into C.

```
quo< C | D >
```

```
quo< C | S >
```

```
quo< C | L >
```

Given a complex C and a subcomplex D of C, returns the quotient complex C/D together with the natural quotient map. If given as a sequence S of submodules of the terms of C or a list L of sequences of elements of the terms of C, then the submodule generated by S or L is created and the quotient computed. The function also returns the chain map of C on the quotient.

56.1.3 Access Functions

```
Degrees(C)
```

Returns the first and last degrees of the defined terms of the complex C.

```
Algebra(C)
```

Given a complex C over a basic algebra A, this function returns the algebra A.

`BoundaryMap(C, n)`

Returns the boundary map of the complex C from the term of degree n to the term of degree $n-1$.

`BoundaryMaps(C)`

The list of boundary maps of the complex C.

`DimensionsOfHomology(C)`

Returns the list of the dimensions of the homology groups of the complex C.

`DimensionOfHomology(C, n)`

Returns the dimension of the homology group of the complex C in degree n.

`DimensionsOfTerms(C)`

Returns the list of the dimensions of the terms of the complex C.

`Term(C, n)`

The module in the complex C in degree n.

`Terms(C)`

The sequence of terms of the complex C.

56.1.4 Elementary Operations

`DirectSum(C, D)`

Returns the direct sum of the complex C and the complex D.

`Homology(C)`

`HomologyOfChainComplex(C)`

The sequence of homology groups of the complex C as a sequence of modules.

`Homology(C, n)`

Returns the homology group in degree n of the complex C, as an A-module.

`Prune(C)`

Returns the complex that consists of the terms of C with the right end term (term of lowest degree) removed.

`Prune(C,n)`

Returns the complex that consists of the terms of C with n terms removed from the right end.

```
Preprune(C)
```

Returns the complex that consists of the terms of C with the left end term (term of highest degree) removed.

```
Preprune(C,n)
```

Returns the complex that consists of the terms of C with n terms removed from the left end.

```
Shift(C, n)
```

Given the complex C in degrees r to s, returns the complex in shifted degrees $r+n$ to $s+n$. The integer n may be either positive or negative. The maps in the complex are all multiplied by the scalar $(-1)^n$.

```
ShiftToDegreeZero(C)
```

Given the complex C, returns the shift of C so that the last term, the term of lowest degree, is in degree 0.

```
Splice(C, D)
```

Given two complexes C and D over the same basic algebra, such that the end term of C coincides with the initial term of D, form the complex that corresponds to the concatenation of C and D (as sequences of maps) This function checks that the resulting sequence of maps forms a complex. The degrees of complex D remain unchanged while the degrees of the terms in complex C are changed to fit.

```
Splice(C, D, f)
```

The splice of the complex C with the complex D along the map f from the last term of C to the first term of D. The degree of the last term of the splice is the same as the degree of the last term of the complex D.

56.1.5 Extensions

The following are elementary operations related to extending complexes

```
LeftExactExtension(C)
```

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the inclusion map from the kernel of the boundary map in highest degree to the term of highest degree in C.

```
RightExactExtension(C)
```

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the quotient map to the cokernel of the boundary map of lowest degree from the term of lowest degree in C.

```
ExactExtension(C)
```

Returns the left and right exact extensions of the complex C.

```
LeftZeroExtension(C)
LeftZeroExtension(C, n)
```

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the zero map from the zero module to the term of highest degree in the complex. If a natural number n is included in the input then the operation is performed n times.

```
RightZeroExtension(C)
RightZeroExtension(C, n)
```

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the zero map to the zero module from the term of lowest degree in the complex. If a natural number n is included in the input then the operation is performed n times.

```
ZeroExtension(C)
```

Returns the left and right zero extensions of the complex C.

```
EqualizeDegrees(C, D)
EqualizeDegrees(Q)
```

Given complexes C and D over the same algebra, the function returns the complexes obtained by taking zero extensions of C and D, if necessary, so that both complexes have the same degrees. The input can also be given as a sequence Q of complexes, in which case the function returns the sequence of complexes obtained by taking zero extensions of the elements of Q, if necessary, until all of the elements of the sequence have the same degrees.

```
EqualizeDegrees(C, D, n)
```

The two complexes, C and D, with zero extension sufficient that the first and the shift by degree n of the second have the same degrees.

56.1.6 Predicates

The following functions return a Boolean value.

```
IsExact(C)
```

Returns `true` if and only if the complex C is an exact sequence, i.e. the image of each map in C is equal to the kernel of the succeeding map (with the obvious exceptions of the first and last maps of the complex). If the complex has only two terms then this is true trivially.

```
IsShortExactSequence(C)
```

Returns `true` if the complex C consists of a short exact sequence together with other terms that are zero. The function returns also the degrees of the complex of nonzero terms.

```
IsExact(C, n)
```

Returns `true` if and only if the complex C is an exact sequence in degree n, i.e. the image of the map in C into the term of degree n is equal to the kernel of the succeeding map.

```
IsZeroComplex(C)
```

Returns `true` if and only if the complex C is composed entirely of zero modules and maps.

```
IsZeroMap(C, n)
```

Returns `true` if and only if the boundary map in degree n of the complex C is the zero map.

```
IsZeroTerm(C, n)
```

Returns `true` if and only if the term in degree n of the complex C is the zero object.

Example H56E1

We construct the quiver algebra of a quiver with three nodes and three arrows going from node 1 to node 2, from 2 to 1 and 2 to 3. The relation is that $(ab)^3a = 0$ where a is the first arrow and b is the second. Then we construct projective and injective resolutions of the first simple module.

```
> ff := GF(8);
> FA<e1,e2,e3,a,b,c> := FreeAlgebra(ff,6);
> rrr := [a*b*a*b*a*b*a];
> D := BasicAlgebra(FA,rrr,3,[<1,2>,<2,1>,<2,3>]);
> D;
Basic algebra of dimension 23 over GF(2^3)
Number of projective modules: 3
Number of generators: 6
> DimensionsOfProjectiveModules(D);
[ 10, 12, 1 ]
> DimensionsOfInjectiveModules(D);
[ 8, 7, 8 ]
```

Here is the opposite algebra:

```
> OD := OppositeAlgebra(D);
reverse trees
> OD;
Basic algebra of dimension 23 over GF(2^3)
Number of projective modules: 3
Number of generators: 6
```

```
> s1 := SimpleModule(D,1);
> P,mu := ProjectiveResolution(s1,7);
> P;
Basic algebra complex with terms of degree 7 down to 0
Dimensions of terms: 12 12 12 12 12 12 12 10
> Q,nu := InjectiveResolution(s1,7);
> Q;
Basic algebra complex with terms of degree 0 down to -7
Dimensions of terms: 8 7 0 0 0 0 0 0
```

Note that the projective and injective resolutions are complexes with the appropriate augmentation and coaugmentation maps. First we form the two term complex whose boundary map is the composition of the augmentation and coaugmentation maps.

```
> theta := MapToMatrix(hom<Term(P,0)-> Term(Q,0)|mu*nu>);
> E := Complex(theta,0);
```

Then we splice all of this together.

```
> R := Splice(P,E);
> S := Splice(R,Q);
> S;
Basic algebra complex with terms of degree 8 down to -7
Dimensions of terms: 12 12 12 12 12 12 12 10 8 7 0 0 0 0 0 0
```

56.2 Chain Maps

A chain map from complex C to complex D is a sequence of homomorphisms between the terms of C and the terms of D such that the maps commute with the boundary homomorphisms on the two complexes. The chain map has degree zero if the map on C_n has it's image in D_n. Otherwise the degree of the chain map expresses the extent to which the degrees of the terms are raised or lowered by the chain map.

Chain maps do not have to be defined in every degree for which either its domain or codomain is defined. On the other hand it must be defined wherever possible, i.e. for any degree i for which the term of the domain is defined and for which the term of the codomain is defined in degree $i + n$ where n is the degree of the chain map.

The freedom of definition can cause problems with constructions such as the kernel or cokernel. That is, the kernel of a chain complex may not be a complex. For this reason we allow the Kernel and Cokernel functions to work only for proper chain maps where "proper" is defined as follows. Suppose C is a chain complex in degrees a to b and D is a complex in degrees u to v. A chain map in degree n from C to D is proper provided $u \geq a + n$ and $v \geq b + n$.

56.2.1 Creation

```
ChainMap(Q, C, D, n)
```

Returns the chain map given by the sequence of maps in Q. Each element in Q is a map from a term of the complex C to a corresponding term of the complex D. The integer n is the degree of the chain map. This means that for any i the term in degree i of C is mapped to the term of degree $n+i$ of D. There should be a map defined for any degree i in the range of degrees of C such that there is a corresponding term in degree $n+i$ for the complex D.

```
ZeroChainMap(C, D)
```

Given chain complexes C and D, construct the chain map from C to D all of whose terms are zero.

56.2.2 Access Functions

```
Degree(f)
```

Given a chain map f, returns the degree of f. Note that $f : C \to D$ has degree n if it takes the term in degree i of the complex C to the term in degree $i+n$ of the complex D.

```
ModuleMap(f, n)
```

Given a chain map $f : C \to D$ and an integer n, this function returns the map from the term in degree n of the complex C to the term in degree $n + Degree(f)$ of the complex D.

```
Kernel(f)
```

Returns the kernel complex of f and the inclusion of the kernel in the domain of f. The function is only defined for proper chain maps. If the chain map is not defined for a certain term of the domain then that term is equal to the term of the kernel complex. That is, if f is not defined on a term then the functions acts as if f were the zero map on that term.

```
Cokernel(f)
```

Returns the cokernel complex of f and the projection of the cokernel onto the codomain of f. The function is only defined for proper chain maps. If the chain map is not defined for a certain term of the codomain then that term is equal to the term of the cokernel complex. That is, if f is not defined on a term then the function acts as if f were the zero map on that term.

```
Image(f)
```

Returns the image complex of f and the inclusion of the image in the codomain of f and the projection of the domain of f on to the image. The function is only defined for proper chain maps.

56.2.3 Elementary Operations

```
f + g
```

The sum of two chain maps with the same domain and codomain.

```
a * g
```

The product of the chain map g by the scalar a in the base ring of the algebra.

```
f * g
```

The composition of the two chain maps f and g.

56.2.4 Predicates

The following functions return a Boolean value.

```
IsSurjective(f)
```

Returns `true` if the chain map f is a surjection in every degree, `false` otherwise.

```
IsInjective(f)
```

Returns `true` if the chain map f is an injection in every degree, `false` otherwise.

```
IsZero(f)
```

Returns `true` if the chain map f is zero in every degree, `false` otherwise.

```
IsIsomorphism(f)
```

Returns `true` if the chain map f is an isomorphism of chain complexes, `false` otherwise.

```
IsShortExactSequence(f, g)
```

Returns `true` if the sequence of chain complexes, $0 \to \mathit{Domain}(f) \to \mathit{Domain}(g) \to \mathit{Codomain}(g) \to 0$, where the internal maps are f and g, is exact.

```
IsChainMap(L, C, D, n)
```

Returns `true` if the list of maps L from the terms of complex C to the terms of the complex D is a chain map of degree n, i. e. it has the right length and the diagram commutes.

```
IsChainMap(f)
```

Returns `true` if the supposed chain map f really is a chain map, i. e. the diagrams commute.

```
IsProperChainMap(f)
```

Returns `true` if the chain map f is a proper chain map, a necessary condition for taking kernel and cokernel.

```
HasDefinedModuleMap(C,n)
```

Returns `true` if the module map in degree n of the complex C is defined.

Example H56E2

We from the basic algebra of the direct product of a cyclic group of order 3 with symmetric group on three letters over the field with three elements.

```
> FA<e1,e2,a,b> := FreeAlgebra(GF(3),4);
> MM:= [e1 +e2 - FA!1, a*b*a, b*a*b];
> BS3 := BasicAlgebra(FA, MM, 2, [<1,2>,<2,1>]);
> gg := CyclicGroup(3);
> BC3 := BasicAlgebra(gg,GF(3));
> A := TensorProduct(BS3,BC3);
> A;
Basic algebra of dimension 18 over GF(3)
Number of projective modules: 2
Number of generators: 6
```

Now we want the projective resolution of the second simple module as a complex. This we will manipulate and take a somewhat random complex.

```
> PR := ProjectiveResolution(SimpleModule(A,2),12);
> PR;
Basic algebra complex with terms of degree 12 down to 0
Dimensions of terms: 117 108 99 90 81 72 63 54 45 36 27 18 9
> PR := Prune(PR);
> PR := Prune(PR);
> PR := Prune(PR);
> PR;
Basic algebra complex with terms of degree 12 down to 3
Dimensions of terms: 117 108 99 90 81 72 63 54 45 36
> PR := Prune(PR);
> PR := Prune(PR);
> PR;
Basic algebra complex with terms of degree 12 down to 5
Dimensions of terms: 117 108 99 90 81 72 63 54
> PR := ZeroExtension(PR);
> PR;
Basic algebra complex with terms of degree 13 down to 4
Dimensions of terms: 0 117 108 99 90 81 72 63 54 0
> PR := Shift(PR,-4);
> PR;
Basic algebra complex with terms of degree 9 down to 0
Dimensions of terms: 0 117 108 99 90 81 72 63 54 0
> S := [* *];
> for i := 1 to 10 do
>    S[i] := [Random(Term(PR,10-i)),Random(Term(PR,10-i))];
> end for;
> C,mu := Subcomplex(PR,S);
> C;
Basic algebra complex with terms of degree 9 down to 0
Dimensions of terms: 0 36 67 64 62 63 58 54 51 0
```

```
> Homology(C);
[
    AModule of dimension 4 over GF(3),
    AModule of dimension 6 over GF(3),
    AModule of dimension 5 over GF(3),
    AModule of dimension 3 over GF(3),
    AModule of dimension 2 over GF(3),
    AModule of dimension 1 over GF(3),
    AModule of dimension 4 over GF(3),
    AModule of dimension 26 over GF(3)
]
[
    Mapping from: AModule of dimension 4 over GF(3) to AModule of dimension 4
    over GF(3),
    Mapping from: AModule of dimension 38 over GF(3) to AModule of dimension 6
    over GF(3),
    Mapping from: AModule of dimension 34 over GF(3) to AModule of dimension 5
    over GF(3),
    Mapping from: AModule of dimension 33 over GF(3) to AModule of dimension 3
    over GF(3),
    Mapping from: AModule of dimension 31 over GF(3) to AModule of dimension 2
    over GF(3),
    Mapping from: AModule of dimension 33 over GF(3) to AModule of dimension 1
    over GF(3),
    Mapping from: AModule of dimension 29 over GF(3) to AModule of dimension 4
    over GF(3),
    Mapping from: AModule of dimension 51 over GF(3) to AModule of dimension 26
    over GF(3)
]
>  D := Cokernel(mu);
>  D;
Basic algebra complex with terms of degree 9 down to 0
Dimensions of terms: 0 81 41 35 28 18 14 9 3 0
>  Homology(D);
[
    AModule of dimension 64 over GF(3)
    AModule of dimension 5 over GF(3),
    AModule of dimension 3 over GF(3),
    AModule of dimension 2 over GF(3),
    AModule of dimension 1 over GF(3),
    AModule of dimension 4 over GF(3),
    AModule of dimension 3 over GF(3),
    AModule of dimension 3 over GF(3)
]
[
    Mapping from: AModule of dimension 64 over GF(3) to AModule of dimension 64
    over GF(3),
    Mapping from: AModule of dimension 22 over GF(3) to AModule of dimension 5
    over GF(3),
    Mapping from: AModule of dimension 22 over GF(3) to AModule of dimension 3
    over GF(3),
    Mapping from: AModule of dimension 15 over GF(3) to AModule of dimension 2
    over GF(3),
    Mapping from: AModule of dimension 14 over GF(3) to AModule of dimension 1
    over GF(3),
    Mapping from: AModule of dimension 8 over GF(3) to AModule of dimension 4
    over GF(3),
    Mapping from: AModule of dimension 9 over GF(3) to AModule of dimension 3
    over GF(3),
    Mapping from: AModule of dimension 3 over GF(3) to AModule of dimension 3
    over GF(3)
]
```

56.2.5 Maps on Homology

`InducedMapOnHomology(f, n)`

The homomorphism induced on homology by the chain map f in degree n.

`ConnectingHomomorphism(f, g, n)`

The connecting homomorphism in degree n of the short exact sequence of chain complexes given by the chain maps f and g.

`LongExactSequenceOnHomology(f, g)`

The long exact sequence on homology for the exact sequence of complexes given by the chain maps f and g as a chain complex with the homology group in degree i for the Cokernel of the complex C appearing in degree $3i$.

Example H56E3

We create a basic algebra with two simple modules and four nonidempotent generators. The relations are given in the sequence `rrr`.

```
> ff := GF(3);
> p := Characteristic(ff);
> FA<e1, e2, y, x, a, b> := FreeAlgebra(ff,6);
> rrr := [y^p,x^p,x*y+y*x,x*a*b-a*b*x,y*a*b-a*b*y,(b*a)^2,(b*a)^2];
> A := BasicAlgebra(FA,rrr,2,[<1,1>,<1,1>,<1,2>,<2,1>]);
> A;
Basic algebra of dimension 81 over GF(3)
Number of projective modules: 2
Number of generators: 6
> DimensionsOfProjectiveModules(A);
```

```
[ 45, 36 ]
```

Now we generate the simple modules and their projective covers. We want a module PP that is the direct sum of two copies of each of the projective indecomposable module.

```
> S1 := SimpleModule(A,1);
> PP := ProjectiveModule(A,[2,2]);
```

Now we are going to create a complex all of whose terms are isomorphic to PP. To make it interesting we will insist that the first homomorphism have its image in the second radical layer of PP.

```
> J1 := JacobsonRadical(PP);
> theta1 := Morphism(J1,PP);
> J2 := JacobsonRadical(J1);
> theta2 := Morphism(J2,J1);
> theta := theta2*theta1;
> HomPJ := AHom(PP, J2);
> HomPJ;
KMatrixSpace of 162 by 150 matrices and dimension 300 over GF(3)
> gamma := Random(HomPJ)*theta;
> LL := [* gamma *];
```

So we now have the first boundary map of the complex. The other boundary maps are generated randomly.

```
> for i := 1 to 15 do
>     K, phi := Kernel(gamma);
>     HomPK := AHom(PP,K);
>     gamma := Random(HomPK)*MapToMatrix(phi);
>     LL := [* gamma *] cat LL;
> end for;
```

Now we make the list into a chain complex.

```
> C := Complex(LL,0);
> C;
Basic algebra complex with terms of degree 16 down to 0
Dimensions of terms: 162 162 162 162 162 162 162 162 162 162 162 162 162 162
162 162 162
> DimensionsOfHomology(C);
[ 30, 23, 34, 32, 35, 32, 35, 42, 30, 34, 32, 33, 29, 42, 21 ]
Now we will get a random subcomplex.
> a,b := Degrees(C);
> S, mu := RandomSubcomplex(C,[2: i in [1 .. a-b+1]]);
> S;
Basic algebra complex with terms of degree 16 down to 0
Dimensions of terms: 162 155 162 162 162 154 142 162 162 148 154 162 162 152
155 152 162
> DimensionsOfHomology(S);
```

```
[ 27, 27, 34, 32, 35, 33, 48, 42, 28, 44, 38, 33, 29, 48, 24 ]
```

Now take the quotient.

```
> Q,nu := quo<C|S>;
> Q;
Basic algebra complex with terms of degree 16 down to 0
Dimensions of terms: 0 7 0 0 0 8 20 0 0 14 8 0 0 10 7 10 0
> DimensionsOfHomology(Q);
[ 7, 0, 0, 0, 6, 18, 0, 0, 12, 6, 0, 0, 8, 5, 10 ]
```

Now we check to see if this is a short exact sequence of chain maps.

```
> IsShortExactSequence(mu,nu);
true
> lll := LongExactSequenceOnHomology(mu,nu);
Basic algebra complex with terms of degree 47 down to 3
Dimensions of terms: 27 30 7 27 23 0 34 34 0 32 32 0 35 35 6 33 32 18 48 35 0
42 42 0 28 30 12 44 34 6 38 32 0 33 33 0 29 29 8 48 42 5 24 21 10
> IsExact(lll);
true
```