

Handbook of Magma Functions

Part VII: Local Fields

Version 2.30-1

Sydney

November 10, 2026

Editors

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

Handbook Contributors

Eran Assaf *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *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*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

49 p-Adic Rings and Their Extensions **1**
- 49.1 Introduction . . . 1
- 49.2 Background . . . 1
- 49.3 Overview of the p-adics in Magma . . . 2
- 49.4 Creation of Local Rings and Fields . . . 3
- 49.5 Elementary Invariants . . . 9
- 49.6 Operations on Structures . . . 14
- 49.7 Element Constructions and Conversions . . . 16
- 49.8 Operations on Elements . . . 20
- 49.9 Linear Algebra . . . 29
- 49.10 Roots of Elements . . . 29
- 49.11 Polynomials . . . 30
- 49.12 Automorphisms of Local Rings and Fields . . . 38
- 49.13 Completions . . . 40
- 49.14 Class Field Theory . . . 41
- 49.15 Extensions . . . 43
- 49.16 Exact p-Adic Rings . . . 45
- 49.17 Bibliography . . . 60

50 General p-Adic Extensions **61**
- 50.1 Introduction . . . 61
- 50.2 Constructions . . . 61
- 50.3 Operations with Fields . . . 62
- 50.4 Maximal Order . . . 64
- 50.5 Homomorphisms . . . 65
- 50.6 Automorphisms and Galois Theory . . . 65
- 50.7 Elements Operations . . . 65
- 50.8 Polynomial Factorization . . . 67

51 Power, Laurent, and Puiseux Series **69**
- 51.1 Introduction . . . 69
- 51.2 Creation Functions . . . 71
- 51.3 Structure Operations . . . 73
- 51.4 Basic Element Operations . . . 75
- 51.5 Transcendental Functions . . . 79
- 51.6 The Hypergeometric Series . . . 81
- 51.7 Polynomials over Series Rings . . . 81
- 51.8 Extensions of Series Rings . . . 83
- 51.9 Bibliography . . . 89

52 Lazy Power Series Rings **90**
- 52.1 Introduction . . . 90
- 52.2 Creation of Lazy Series Rings . . . 90
- 52.3 Functions on Lazy Series Rings . . . 91
- 52.4 Elements . . . 92

53 Algebraic Power Series Rings **100**

53.1 Introduction . . . 100
53.2 Basics . . . 100
53.3 Constructors . . . 101
53.4 Accessors and Expansion . . . 104
53.5 Arithmetic . . . 105
53.6 Predicates . . . 106
53.7 Modifiers . . . 107
53.8 Bibliography . . . 108

54 Valuation Rings **109**
54.1 Introduction . . . 109
54.2 Creation Functions . . . 109
54.3 Structure Operations . . . 110
54.4 Element Operations . . . 110

55 Galois Rings **112**
55.1 Introduction . . . 112
55.2 Creation Functions . . . 112
55.3 Structure Operations . . . 115
55.4 Element Operations . . . 116

56 Newton Polygons **118**
56.1 Introduction . . . 118
56.2 Newton Polygons . . . 119
56.3 Polynomials Associated with Newton Polygons . . . 125
56.4 Finding Valuations of Roots of Polynomials from Newton Polygons . . . 126
56.5 Using Newton Polygons to Find Roots of Polynomials over Series Rings . . . 126
56.6 Bibliography . . . 138

57 Series Rings over p-Adic Rings **139**
57.1 Introduction . . . 139
57.2 Matrices and Modules . . . 145
57.3 Bibliography . . . 151

58 Local Galois Representations **152**
58.1 Overview . . . 152
58.2 Creating Galois Representations . . . 154
58.3 Basic Invariants . . . 163
58.4 Arithmetic . . . 170
58.5 Changing Precision . . . 171
58.6 Changing Fields . . . 172
58.7 Advanced Examples . . . 173
58.8 Bibliography . . . 176

References **177**

Index **178**

Chapter 49

p-Adic Rings and Their Extensions

49.1 Introduction

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

Magma has three different models for working with these locals: fixed precision rings (`RngPadRes` and `RngPadResExt`, with element types `RngPadResElt` and `RngPadResExtElt`), free precision rings (`RngPad` and `FldPad`, with element types `RngPadElt` and `FldPadElt`) and exact p-adic rings (`RngXPad` and `FldXPad` with element types `RngXPadElt` and `FldXPadElt`). The merits of each model are discussed in Section *p-adic Rings*.

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 *General p-Adic Extensions*.

49.2 Background

The p-adic field $\mathbb{Q}_p$ arises naturally as the completion of $\mathbb{Q}$ 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 $\mathbb{Q}_p$, denoted $\mathbb{Z}_p$, is the set of all elements of non-negative valuation. The ring $\mathbb{Z}_p$ has a unique maximal ideal, generated by the prime p; the *residue class field* K is the quotient $\mathbb{Z}_p/p\mathbb{Z}_p$. Any element x of $\mathbb{Q}_p$ 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, $\mathbb{Q}_p$ is a local field, with its ring of integers $\mathbb{Z}_p$ being a local ring. A uniformizer π of $\mathbb{Q}_p$ is the prime p.

More generally, consider an irreducible polynomial over some local field L_1 (such as $\mathbb{Q}_p$). 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 keep their degree when mapped over the residue class field and 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.

49.3 Overview of the p-adics in Magma

49.3.1 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 *Background*, 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 three 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.

The third model is that of the exact p-adic rings which use a lazy representation and delay any computations to a precision. Such exact objects know how to compute digits to a given precision although they may not know what any of those digits are. These rings are explained further in Section *Exact p-Adic Rings*.

49.3.2 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.

49.3.3 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.

49.3.4 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.

49.4 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 *Background*, 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.

49.4.1 Creation Functions for the p-adics

pAdicRing(p, k): RngIntElt, RngIntElt -> RngPad

pAdicField(p, k): RngIntElt, RngIntElt -> FldPad

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): RngIntElt -> RngPad

pAdicField(p): RngIntElt -> FldPad

```
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): RngIntElt, RngIntElt -> RngPadRes

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

quo<L | x>: RngPad, RngPadElt -> .

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

Example: Creating p-adic Rings (ex-65d82a)

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;
```

(continues on next page)

(continued from previous page)

```
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
```

49.4.2 Creation of Unramified Extensions

UnramifiedExtension(L, n): RngPad, RngIntElt -> RngPad

UnramifiedExtension(L, n): FldPad, RngIntElt -> FldPad

UnramifiedExtension(L, n): RngPadRes, RngIntElt -> RngPadResExt

UnramifiedExtension(L, n): RngPadResExt, RngIntElt -> RngPadResExt

ext<L | n>: RngPad, RngIntElt -> RngPad

ext<L | n>: FldPad, RngIntElt -> FldPad

ext<L | n>: RngPadRes, RngIntElt -> RngPadResExt

ext<L | n>: RngPadResExt, RngIntElt -> RngPadResExt

```
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): FldFin, RngIntElt -> Rng

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): RngPad, RngUPolElt -> RngPad

UnramifiedExtension(L, f): FldPad, RngUPolElt -> FldPad

UnramifiedExtension(L, f): RngPadRes, RngUPolElt -> RngPadResExt

UnramifiedExtension(L, f): RngPadResExt, RngUPolElt -> RngPadResExt

ext<L | f>: RngPad, RngUPolElt -> RngPad

ext<L | f>: FldPad, RngUPolElt -> FldPad

ext<L | f>: RngPadRes, RngUPolElt -> RngPadResExt

ext<L | f>: RngPadResExt, RngUPolElt -> RngPadResExt

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): RngUPolElt -> BoolElt`

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): RngPad, RngIntElt, RngIntElt -> BoolElt`

`HasGNB(L, n, t): FldPad, RngIntElt, RngIntElt -> BoolElt`

`HasGNB(R, n, t): RngPadRes, RngIntElt, RngIntElt -> BoolElt`

`HasGNB(R, n, t): RngPadResExt, RngIntElt, RngIntElt -> BoolElt`

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): FldPad, RngIntElt -> FldPad`

`CyclotomicUnramifiedExtension(R, f): RngPad, RngIntElt -> RngPad`

`CyclotomicUnramifiedExtension(R, f): RngPadRes, RngIntElt -> RngPadRes`

`CyclotomicUnramifiedExtension(R, f): RngPadResExt, RngIntElt -> RngPadResExt`

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: Creating Unramified Extensions `(ex-f8037d)`

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
```

49.4.3 Creation of Totally Ramified Extensions

`TotallyRamifiedExtension(L, f): RngPad, RngUPolElt -> RngPad`

`TotallyRamifiedExtension(L, f): FldPad, RngUPolElt -> FldPad`

`TotallyRamifiedExtension(L, f): RngPadRes, RngUPolElt -> RngPadResExt`

`TotallyRamifiedExtension(L, f): RngPadResExt, RngUPolElt -> RngPadResExt`

`ext<L | f>: RngPad, RngUPolElt -> RngPad`

`ext<L | f>: FldPad, RngUPolElt -> FldPad`

`ext<L | f>: RngPadRes, RngUPolElt -> RngPadResExt`

`ext<L | f>: RngPadResExt, RngUPolElt -> RngPadResExt`

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): RngUPolElt -> BoolElt`

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: Creating Totally Ramified Extensions (ex-11c692)

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
```

(continues on next page)

(continued from previous page)

```
> 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
```

49.4.4 Creation of 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 *Creation of Unramified Extensions* 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 $\phi : \mathbb{Z}_{\geq 0} \to R[x]$, where R is a ring whose elements are coercible to the quotient rings $L/\pi^k L$ for all $k \in \mathbb{Z}_{\geq 0}$. 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>: RngPad, Map -> RngPad

ext<L | m>: FldPad, Map -> RngPad

Given a free precision local ring or field L and a map m with domain $\mathbb{Z}$ and codomain $R[x]$, where elements of R are coercible to the quotient rings $L/\pi^k L$ for all $k \in \mathbb{Z}_{\geq 0}$, 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: Unbounded Precision Extensions from Maps (ex-2d52e1)

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)
```

49.4.5 Creation of Related Rings

IntegerRing(F): FldPad -> RngPad

Integers(F): FldPad -> RngPad

`RingOfIntegers(F): FldPad -> RngPad`

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): RngPad -> RngPad`

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): RngPad -> FldPad`

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): RngUPolElt[RngInt], RngPad -> RngPad`

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 ([Pau01b]).

`AbsoluteTotallyRamifiedExtension(R): RngPad -> RngPad, Map`

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.

49.4.6 Other Elementary Constructions

`Composite(R, S): RngPad, RngPad -> RngPad`

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

49.4.7 Attributes of Local Rings and Fields

`L`DefaultPrecision: RngPad -> RngIntElt`

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.

`L`SeriesPrinting: RngPad -> BoolElt`

This attribute controls whether elements of the ring or field L are printed as a series in the uniformizing element of L. This attribute cannot be set for unramified extensions with Gaussian Normal bases of type 2.

`L`SeriesPrinting: FldPad -> BoolElt`

This attribute controls whether elements of the ring or field L are printed as a series in the uniformizing element of L. This attribute cannot be set for unramified extensions with Gaussian Normal bases of type 2.

`L`SeriesPrinting: RngPadRes -> BoolElt`

This attribute controls whether elements of the ring or field L are printed as a series in the uniformizing element of L. This attribute cannot be set for unramified extensions with Gaussian Normal bases of type 2.

`L`SeriesPrinting: RngPadResExt -> BoolElt`

This attribute controls whether elements of the ring or field L are printed as a series in the uniformizing element of L. This attribute cannot be set for unramified extensions with Gaussian Normal bases of type 2.

Example: Printing Elements as Series `(ex-29e3f4)`

```
> Zp := pAdicRing(5, 10);
> Zp`SeriesPrinting := true;
> Zp!9348752038967;
2 + 3*5 + 3*5^2 + 5^3 + 2*5^4 + 2*5^5 + 5^7 + 5^9 + O(5^10)
> U := UnramifiedExtension(Zp, 4);
> U`SeriesPrinting := true;
> U![3294875629, 324, 1254234, 13454];
4*U.1^3 + 4*U.1^2 + 4*U.1 + 4 + (U.1^2 + 4*U.1)*5 + (3*U.1^3 + 4*U.1^2 +
    2*U.1)*5^2 + (2*U.1^3 + 3*U.1^2 + 2*U.1)*5^3 + (U.1^3 + U.1^2 + 1)*5^4 +
    (4*U.1^3 + U.1^2)*5^5 + 2*5^6 + (U.1^2 + 4)*5^7 + (3*U.1^2 + 4)*5^8 + 5^9 +
O(5^10)
> R := TotallyRamifiedExtension(U, Polynomial([10, 25, 125, 1]));
> R![[345, 21, 351345], [234, 2354,214], [2134, 2153254]];
(4*U.1 + 4 + 5 + (U.1 + 2)*5^3 + 3*5^4 + 4*U.1*5^5 + 2*U.1*5^6 + 2*U.1*5^7 +
    U.1*5^9)*R.1^2 + (4*U.1^2 + 4*U.1 + 4 + (2*U.1^2 + 1)*5 + (3*U.1^2 + 4*U.1 +
    4)*5^2 + (U.1^2 + 3*U.1 + 1)*5^3 + 3*U.1*5^4)*R.1 + U.1 + (4*U.1^2 + 4*U.1 +
    4)*5 + (3*U.1^2 + 3)*5^2 + 2*5^3 + 2*U.1^2*5^4 + 2*U.1^2*5^5 + 2*U.1^2*5^6 +
    4*U.1^2*5^7 + O(R.1^30)
> R`SeriesPrinting := true;
> R![[345, 21, 351345], [234, 2354,214], [2134, 2153254]];
U.1 + (4*U.1^2 + 4*U.1 + 4)*R.1 + (4*U.1 + 4)*R.1^2 + (3*U.1^2 + 3*U.1 +
    3)*R.1^3 + (4*U.1^2 + 2)*R.1^4 + 2*R.1^5 + 3*U.1*R.1^6 + (2*U.1^2 + 3*U.1 +
    2)*R.1^7 + (U.1^2 + 2)*R.1^8 + (3*U.1^2 + 4*U.1 + 2)*R.1^9 + (3*U.1^2 +
    3)*R.1^10 + (U.1 + 3)*R.1^11 + (2*U.1^2 + U.1)*R.1^12 + (U.1^2 + 4*U.1 +
    3)*R.1^13 + 3*R.1^14 + (4*U.1^2 + 3*U.1 + 2)*R.1^15 + (2*U.1^2 + 2)*R.1^16 +
    3*U.1^2*R.1^17 + (2*U.1^2 + 3*U.1 + 3)*R.1^18 + (4*U.1^2 + 3*U.1 + 4)*R.1^19␣
↪+
    (3*U.1^2 + 4)*R.1^20 + (U.1^2 + U.1 + 1)*R.1^21 + (3*U.1^2 + 4)*R.1^22 +
    (4*U.1^2 + 2*U.1)*R.1^23 + (2*U.1^2 + 4)*R.1^24 + (4*U.1 + 1)*R.1^25 +
    (4*U.1^2 + 2*U.1 + 1)*R.1^26 + (4*U.1^2 + 2)*R.1^27 + (U.1^2 + 2*U.1 +
    1)*R.1^28 + (U.1^2 + U.1 + 2)*R.1^29 + O(R.1^30)
```

49.5 Elementary Invariants

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

Prime(L): RngPad -> RngIntElt

Prime(L): FldPad -> RngIntElt

Prime(L): RngPadRes -> RngIntElt

Prime(L): RngPadResExt -> RngIntElt

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): RngPad -> RngIntElt

InertiaDegree(L): FldPad -> RngIntElt

InertiaDegree(L): RngPadRes -> RngIntElt

InertiaDegree(L): RngPadResExt -> RngIntElt

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

InertiaDegree(K, L): RngPad, RngPad -> RngIntElt

InertiaDegree(K, L): FldPad, FldPad -> RngIntElt

InertiaDegree(K, L): RngPadRes, RngPadRes -> RngIntElt

InertiaDegree(K, L): RngPadResExt, RngPadResExt -> RngIntElt

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

AbsoluteInertiaDegree(L): RngPad -> RngIntElt

`AbsoluteInertiaDegree(L): FldPad -> RngIntElt`
`AbsoluteInertiaDegree(L): RngPadRes -> RngIntElt`
`AbsoluteInertiaDegree(L): RngPadResExt -> RngIntElt`
`AbsoluteInertiaIndex(L): RngPad -> RngIntElt`
`AbsoluteInertiaIndex(L): FldPad -> RngIntElt`
`AbsoluteInertiaIndex(L): RngPadRes -> RngIntElt`
`AbsoluteInertiaIndex(L): RngPadResExt -> RngIntElt`

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

`RamificationDegree(L): RngPad -> RngIntElt`
`RamificationDegree(L): FldPad -> RngIntElt`
`RamificationDegree(L): RngPadRes -> RngIntElt`
`RamificationDegree(L): RngPadResExt -> RngIntElt`
`RamificationIndex(L): RngPad -> RngIntElt`
`RamificationIndex(L): FldPad -> RngIntElt`
`RamificationIndex(L): RngPadRes -> RngIntElt`
`RamificationIndex(L): RngPadResExt -> RngIntElt`

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

`RamificationDegree(K, L): RngPad, RngPad -> RngIntElt`
`RamificationDegree(K, L): FldPad, FldPad -> RngIntElt`
`RamificationDegree(K, L): RngPadRes, RngPadRes -> RngIntElt`
`RamificationDegree(K, L): RngPadResExt, RngPadResExt -> RngIntElt`
`RamificationIndex(K, L): RngPad, RngPad -> RngIntElt`
`RamificationIndex(K, L): FldPad, FldPad -> RngIntElt`
`RamificationIndex(K, L): RngPadRes, RngPadRes -> RngIntElt`
`RamificationIndex(K, L): RngPadResExt, RngPadResExt -> RngIntElt`

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

`AbsoluteRamificationDegree(L): RngPad -> RngIntElt`
`AbsoluteRamificationDegree(L): FldPad -> RngIntElt`
`AbsoluteRamificationDegree(L): RngPadRes -> RngIntElt`
`AbsoluteRamificationDegree(L): RngPadResExt -> RngIntElt`
`AbsoluteRamificationIndex(L): RngPad -> RngIntElt`
`AbsoluteRamificationIndex(L): FldPad -> RngIntElt`
`AbsoluteRamificationIndex(L): RngPadRes -> RngIntElt`
`AbsoluteRamificationIndex(L): RngPadResExt -> RngIntElt`

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

`AbsoluteDegree(L): RngPad -> RngIntElt`

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

`Degree(L): RngPad -> RngIntElt`
`Degree(L): FldPad -> RngIntElt`
`Degree(L): RngPadRes -> RngIntElt`
`Degree(L): RngPadResExt -> RngIntElt`

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

`Degree(K, L): RngPad, RngPad -> RngIntElt`
`Degree(K, L): FldPad, FldPad -> RngIntElt`
`Degree(K, L): RngPadRes, RngPadRes -> RngIntElt`

`Degree(K, L): RngPadResExt, RngPadResExt -> RngIntElt`

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

`DefiningPolynomial(L): RngPad -> RngUPolElt`

`DefiningPolynomial(L): FldPad -> RngUPolElt`

`DefiningPolynomial(L): RngPadRes -> RngUPolElt`

`DefiningPolynomial(L): RngPadResExt -> RngUPolElt`

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.

`DefiningPolynomial(K, L): RngPad, RngPad -> RngUPolElt`

`DefiningPolynomial(K, L): FldPad, FldPad -> RngUPolElt`

`DefiningPolynomial(K, L): RngPadRes, RngPadRes -> RngUPolElt`

`DefiningPolynomial(K, L): RngPadResExt, RngPadRes -> RngUPolElt`

`DefiningPolynomial(K, L): RngPadResExt, RngPadResExt -> RngUPolElt`

Return the minimal polynomial of the generator of K over its coefficient ring L.

`DefiningMap(L): RngPad -> Map`

`DefiningMap(L): FldPad -> Map`

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

`HasDefiningMap(L): RngPad -> BoolElt, Map`

`DefiningMap(L): FldPad -> BoolElt, Map`

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): RngPad -> RngPad`

`PrimeRing(L): RngPadRes -> RngPadRes`

`PrimeRing(L): RngPadResExt -> RngPadRes`

`PrimeField(L): FldPad -> FldPad`

`pAdicRing(L): RngPad -> RngPad`

`pAdicRing(L): RngPadRes -> RngPadRes`

`pAdicRing(L): RngPadResExt -> RngPadRes`

`pAdicField(L): FldPad -> FldPad`

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

`BaseRing(L): RngPad -> RngPad`

`BaseRing(L): RngPadRes -> RngPadRes`

`BaseRing(L): RngPadResExt -> Rng`

`CoefficientRing(L): RngPad -> RngPad`

`CoefficientRing(L): RngPadRes -> RngPadRes`

`CoefficientRing(L): RngPadResExt -> Rng`

`BaseField(L): FldPad -> FldPad`

`CoefficientField(L): FldPad -> FldPad`

`BaseRing(L): FldPad -> FldPad`

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

`ResidueClassField(L): RngPad -> FldFin, Map`

`ResidueClassField(L): RngPadRes -> FldFin, Map`

`ResidueClassField(L): RngPadResExt -> FldFin, Map`

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

`ResidueSystem(R): RngPad -> [RngPadElt]`

`ResidueSystem(R): RngPadRes -> [RngPadEltRes]`

`ResidueSystem(R): RngPadResExt -> [RngPadEltResExt]`

`ResidueSystem(R): FldPad -> [FldPadElt]`

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): RngPad -> RngPadElt`

`UniformizingElement(L): RngPadRes -> RngPadResElt`

`UniformizingElement(L): RngPadResExt -> RngPadResExtElt`

`UniformizingElement(L): FldPad -> FldPadElt`

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

`L . 1: RngPad -> RngPadElt`

`L . 1: RngPadRes -> RngPadResElt`

`L . 1: RngPadResExt -> RngPadResExtElt`

`L . 1: FldPad -> FldPadElt`

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): RngPad -> RngIntElt`

`Precision(L): RngPadRes -> RngIntElt`

`Precision(L): RngPadResExt -> RngIntElt`

`Precision(L): FldPad -> RngIntElt`

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): RngPad -> BoolElt`

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

`HasRootOfUnity(L, n): RngPad, RngIntElt -> BoolElt`

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

`Discriminant(R): RngPad -> RngPadElt`

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): RngPad, RngPad -> RngPadElt`

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

`AdditiveGroup(R): RngPadRes -> GrpAb, Map`

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

Example: Elementary Invariants of Local Rings `(ex-899921)`

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
```

AbsoluteRootNumber(K): FldPad -> FldCycElt

The root number of a p-adic extension $K/\mathbb{Q}_p$ can be computed as in §3.3 of [JR06], with the result returned (using their convention and not Deligne's) as an element of $\mathbb{Q}(\zeta_4)$.

RootNumber(K): FldPad -> FldCycElt

The root number of a relative p-adic extension can be computed by using `AbsoluteRootNumber` and an induction relation.

Example: Root Numbers of p-adic Extensions (ex-f6c8be)

We compute the root numbers of some p-adic extensions.

```
> K := ext<pAdicField(2,20)|Polynomial([10,0,1])>; K;
Totally ramified extension defined by the polynomial x^2 + 10
 over 2-adic field mod 2^20
> RootNumber(K);
-zeta_4
> K := pAdicField(3,40);
> A := AllExtensions(K,3);
> _<x> := PolynomialRing(Integers(K)); // printing
> [<RootNumber(FieldOfFractions(a)),DefiningPolynomial(a)> : a in A];
[
    <zeta_4, x^3 + 3*x + 3>,
    <-zeta_4, x^3 + 6*x + 3>,
    <1, x^3 + 3*x^2 + 3>,
    <1, x^3 + 6*x^2 + 3>,
    <1, x^3 + 6*x^2 + 12>,
    <1, x^3 + 6*x^2 + 21>,
    <zeta_4, x^3 + 3>,
    <zeta_4, x^3 + 12>,
    <zeta_4, x^3 + 21>,
    <1, x^3 + 2*x + 1>
]
```

We take an example where the relative and absolute root numbers differ.

```
> K := pAdicField(3,20);
> L := ext<K|Polynomial([3,0,1])>; // both ramified
> M := ext<L|Polynomial([L.1,0,1])>; // both ramified
> RootNumber(M); // relative for M/L
zeta_4
> AbsoluteRootNumber(M); // absolute for M/Q3
-zeta_4
> RootNumber(L); // intermediate field
zeta_4
> X := AbsoluteTotallyRamifiedExtension(Integers(M));
> RootNumber(FieldOfFractions(X));
-zeta_4
```

49.6 Operations on Structures

AssignNames(~L, S): RngPad, SeqEnum

AssignNames(~L, S): RngPadResExt, SeqEnum

AssignNames(~L, S): FldPad, SeqEnum

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): RngPad -> RngIntElt

Characteristic(L): RngPadRes -> RngIntElt

Characteristic(L): RngPadResExt -> RngIntElt

`Characteristic(L): FldPad -> RngIntElt`

The characteristic of the local ring or field L.

`# L: RngPad -> RngIntElt`

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): RngPad, RngIntElt -> RngPadElt`

`Name(L, k): RngPadRes, RngIntElt -> RngPadResElt`

`Name(L, k): RngPadResExt, RngIntElt -> RngPadResExtElt`

`Name(L, k): FldPad, RngIntElt -> FldPadElt`

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): RngPad, Any -> RngPad`

`ChangePrecision(L, k): RngPad, Infty -> RngPad`

`ChangePrecision(~L, k): RngPad, Infty -> RngPad`

`ChangePrecision(L, k): RngPad, RngIntElt -> RngPad`

`ChangePrecision(~L, k): RngPad, RngIntElt -> RngPad`

`ChangePrecision(L, k): RngPadRes, RngIntElt -> RngPadRes`

`ChangePrecision(~L, k): RngPadRes, RngIntElt -> RngPadRes`

`ChangePrecision(L, k): RngPadResExt, RngIntElt -> RngPadResExt`

`ChangePrecision(~L, k): RngPadResExt, RngIntElt -> RngPadResExt`

`ChangePrecision(L, k): FldPad, RngIntElt -> FldPad`

`ChangePrecision(~L, k): FldPad, RngIntElt -> FldPad`

`ChangePrecision(L, k): FldPad, Any -> FldPad`

`ChangePrecision(L, k): FldPad, Infty -> FldPad`

`ChangePrecision(~L, k): FldPad, Infty -> FldPad`

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: RngPad, RngPad -> BoolElt`

`L eq K: RngPadRes, RngPadRes -> BoolElt`

`L eq K: RngPadResExt, RngPadResExt -> BoolElt`

`L eq K: FldPad, FldPad -> BoolElt`

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

`L ne K: RngPad, RngPad -> BoolElt`

`L ne K: RngPadRes, RngPadRes -> BoolElt`

`L ne K: RngPadResExt, RngPadResExt -> BoolElt`

`L ne K: FldPad, FldPad -> BoolElt`

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

Example: Changing Precision of Local Structures `(ex-23c278)`

```
> 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;
875811540203010669327330989556197582050l\\
637136728223573481676774311194266786628l\\
592914886772632598876953125
> 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.

49.6.1 Ramification Predicates

IsRamified(R): RngPad -> BoolElt

IsRamified(R): FldPad -> BoolElt

IsUnramified(R): RngPad -> BoolElt

IsUnramified(R): FldPad -> BoolElt

IsTotallyRamified(R): RngPad -> BoolElt

IsTotallyRamified(R): FldPad -> BoolElt

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

IsTamelyRamified(R): RngPad -> BoolElt

IsTamelyRamified(R): FldPad -> BoolElt

IsWildlyRamified(R): RngPad -> BoolElt

IsWildlyRamified(R): FldPad -> BoolElt

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

49.7 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.

49.7.1 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): RngPad -> RngPadElt`

`Zero(L): RngPadRes -> RngPadResElt`

`Zero(L): RngPadResExt -> RngPadResExtElt`

`Zero(L): FldPad -> FldPadElt`

Given a local ring or field L, create the additive identity of L. Note that if L is an unbounded precision ring, 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. The valuation and hence the absolute precision of the zero of a field is infinite.

`One(L): RngPad -> RngPadElt`

`One(L): RngPadRes -> RngPadResElt`

`One(L): RngPadResExt -> RngPadResExtElt`

`One(L): FldPad -> FldPadElt`

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): RngPad -> RngPadElt`

`Random(L): RngPadRes -> RngPadResElt`

`Random(L): RngPadResExt -> RngPadResExtElt`

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): RngPad -> RngPadElt`

`Representative(L): RngPadRes -> RngPadResElt`

`Representative(L): RngPadResExt -> RngPadResExtElt`

`Representative(L): FldPad -> FldPadElt`

Return an element of the local ring or field L.

`elt<L | u>: RngPad, RngElt -> RngPadElt`

`elt<L | u>: RngPad, [RngElt] -> RngPadElt`

`L ! u: RngPad, RngElt -> RngPadElt`

`L ! u: RngPad, [RngElt] -> RngPadElt`

`elt<L | u>: FldPad, RngElt -> FldPadElt`

`elt<L | u>: FldPad, [RngElt] -> FldPadElt`

`L ! u: FldPad, RngElt -> FldPadElt`

`L ! u: FldPad, [RngElt] -> FldPadElt`

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 $\mathbb{Z}/p^m\mathbb{Z}$; (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>: RngPad, RngElt, RngIntElt -> RngPadElt`

`elt<L | u, r>: RngPad, [RngElt], RngIntElt -> RngPadElt`

`elt<L | u, r>: FldPad, RngElt, RngIntElt -> FldPadElt`

`elt<L | u, r>: FldPad, [RngElt], RngIntElt -> FldPadElt`

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>: RngPad, RngIntElt, RngElt, RngIntElt -> RngPadElt`

`elt<L | v, u, r>: RngPad, RngIntElt, [RngElt], RngIntElt -> RngPadElt`

`elt<L | v, u, r>: FldPad, RngIntElt, RngElt, RngIntElt -> FldPadElt`

`elt<L | v, u, r>: FldPad, RngIntElt, [RngElt], RngIntElt -> FldPadElt`

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): RngPadElt -> RngPadElt`

`BigO(x): FldPadElt -> FldPadElt`

`O(x): RngPadElt -> RngPadElt`

`O(x): FldPadElt -> FldPadElt`

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): RngPad -> RngPadElt`

`UniformizingElement(L): RngPadRes -> RndPadResElt`

`UniformizingElement(L): RngPadResExt -> RndPadResExtElt`

`UniformizingElement(L): FldPad -> FldPadElt`

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

Example: Element Constructors and Imprecise Zeros (ex-9e5bd3)

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)
```

(continues on next page)

(continued from previous page)

```
> 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*39443045261050590270586428264139311483660321755451150238513946533203 11 +
O(pi^198)
```

Example: Precision When Coercing Sequences (ex-b0924e)

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;
-157980184343196320136914558 7*R.1 - 68057573045897503903339 4769 + O(5^40)
> s := [-68057573045897503903339 4769, -157980184343196320136914558 7];
> R!s;
-13585890629962*R.1 + 47482939261481 + O(5^20)
> P`DefaultPrecision := 40;
> R!s;
-157980184343196320136914558 7*R.1 - 68057573045897503903339 4769 + 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.

49.7.2 Element Decomposers

ElementToSequence(x): RngPadElt -> [RngElt]
ElementToSequence(x): RngPadResElt -> [RngElt]
ElementToSequence(x): RngPadResExtElt -> [RngElt]
ElementToSequence(x): FldPadElt -> [RngElt]
Eltseq(x): RngPadElt -> [RngElt]
Eltseq(x): RngPadResElt -> [RngElt]
Eltseq(x): RngPadResExtElt -> [RngElt]
Eltseq(x): FldPadElt -> [RngElt]
Coefficients(x): RngPadElt -> [RngElt]
Coefficients(x): RngPadResElt -> [RngElt]
Coefficients(x): RngPadResExtElt -> [RngElt]
Coefficients(x): FldPadElt -> [RngElt]

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): RngPadElt, RngIntElt -> RngPadElt`

`Coefficient(x, i): RngPadResElt, RngIntElt -> RngPadResElt`

`Coefficient(x, i): RngPadResExtElt, RngIntElt -> RngElt`

`Coefficient(x, i): FldPadElt, RngIntElt -> FldPadElt`

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

Example: Galois Descent of a Polynomial (ex-716c79)

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)
```

49.8 Operations on Elements

49.8.1 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: RngPadElt -> RngPadElt`

`- x: RngPadResElt -> RngPadResElt`

`- x: RngPadResExtElt -> RngPadResExtElt`

`- x: FldPadElt -> FldPadElt`

The negative of the element x.

`x + y: RngPadElt, RngPadElt -> RngPadElt`

`x + y: RngPadResElt, RngPadResElt -> RngPadResElt`

`x + y: RngPadResExtElt, RngPadResExtElt -> RngPadResExtElt`

`x + y: FldPadElt, FldPadElt -> FldPadElt`

The sum of the elements x and y.

`x - y: RngPadElt, RngPadElt -> RngPadElt`

`x - y: RngPadResElt, RngPadResElt -> RngPadResElt`

`x - y: RngPadResExtElt, RngPadResExtElt -> RngPadResExtElt`

`x - y: FldPadElt, FldPadElt -> FldPadElt`

The difference of the elements x and y.

`x * y: RngPadElt, RngPadElt -> RngPadElt`

`x * y: RngPadResElt, RngPadResElt -> RngPadResElt`

`x * y: RngPadResExtElt, RngPadResExtElt -> RngPadResExtElt`

`x * y: FldPadElt, FldPadElt -> FldPadElt`

The product of the elements x and y.

`x ^ k: RngPadElt, RngIntElt -> RngPadElt`

`x ^ k: RngPadResElt, RngIntElt -> RngPadResElt`

`x ^ k: RngPadResExtElt, RngIntElt -> RngPadResExtElt`

`x ^ k: FldPadElt, RngIntElt -> FldPadElt`

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: RngPadElt, RngPadElt -> RngPadElt`

`x div y: RngPadResElt, RngPadResElt -> RngPadResElt`

`x div y: RngPadResExtElt, RngPadResExtElt -> RngPadResExtElt`

`x div y: FldPadElt, FldPadElt -> FldPadElt`

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: RngPadElt, RngPadElt -> RngPadElt`

`x div:= y: RngPadResElt, RngPadResElt -> RngPadResElt`

`x div:= y: RngPadElt, RngPadElt -> RngPadElt`

`x div:= y: FldPadElt, FldPadElt -> FldPadElt`

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: RngPadElt, RngPadElt -> RngPadElt`

`x / y: FldPadElt, FldPadElt -> FldPadElt`

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): RngPadElt, RngPadElt -> BoolElt, RngPadElt`

`IsExactlyDivisible(x, y): RngPadResElt, RngPadResElt -> BoolElt, RngPadResElt`

`IsExactlyDivisible(x, y): RngPadResExtElt, RngPadResExtElt -> BoolElt, RngPadResExtElt`

`IsExactlyDivisible(x, y): FldPadElt, FldPadElt -> BoolElt, FldPadElt`

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

Example: Division of Non-units (ex-915090)

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)
```

49.8.2 Equality and Membership

It is possible to test whether two local ring or field elements are equal.

`x eq y: RngPadResElt, RngPadResElt -> BoolElt`

`x eq y: RngPadResExtElt, RngPadResExtElt -> BoolElt`

`x eq y: RngPadElt, RngPadElt -> BoolElt`

`x eq y: FldPadElt, FldPadElt -> BoolElt`

Given local ring or field elements x and y, return `true` if and only if x and y are identical in value.

`x ne y: RngPadResElt, RngPadResElt -> BoolElt`

`x ne y: RngPadResExtElt, RngPadResExtElt -> BoolElt`

`x ne y: RngPadElt, RngPadElt -> BoolElt`

`x ne y: FldPadElt, FldPadElt -> BoolElt`

Given local ring or field elements x and y, return `true` if and only if x and y are not identical in value.

`x in L: ., RngPad -> BoolElt`

`x in L: ., FldPad -> BoolElt`

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

`x notin L: ., RngPad -> BoolElt`

`x notin L: ., FldPad -> BoolElt`

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

Example: Constructing an Unramified Extension (ex-44ed45)

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 $\mathbb{Z}_p$. The quotient of $\mathbb{Z}_p[x]$ 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
```

49.8.3 Properties

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

`IsZero(x): RngPadElt -> BoolElt`

`IsZero(x): RngPadResElt -> BoolElt`

`IsZero(x): RngPadResExtElt -> BoolElt`

`IsZero(x): FldPadElt -> BoolElt`

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): RngPadElt -> BoolElt`

`IsOne(x): RngPadResElt -> BoolElt`

`IsOne(x): RngPadResExtElt -> BoolElt`

`IsOne(x): FldPadElt -> BoolElt`

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.

Note that this is always `false` for an element of a ring or field of unbounded precision.

`IsMinusOne(x): RngPadElt -> BoolElt`

`IsMinusOne(x): RngPadResElt -> BoolElt`

`IsMinusOne(x): RngPadResExtElt -> BoolElt`

`IsMinusOne(x): FldPadElt -> BoolElt`

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.

Note that this is always `false` for an element of a ring or field of unbounded precision.

`IsUnit(x): RngPadElt -> BoolElt`

`IsUnit(x): RngPadResElt -> BoolElt`

`IsUnit(x): RngPadResExtElt -> BoolElt`

`IsUnit(x): FldPadElt -> BoolElt`

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): RngPadElt -> BoolElt`

`IsIntegral(x): RngPadResElt -> BoolElt`

`IsIntegral(x): RngPadResExtElt -> BoolElt`

`IsIntegral(x): FldPadElt -> BoolElt`

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

49.8.4 Precision and Valuation

Note that the precision of an element may be modified.

`Parent(x): RngPadElt -> RngPad`

`Parent(x): RngPadResElt -> RngPadRes`

`Parent(x): RngPadResExtElt -> RngPadResExt`

`Parent(x): FldPadElt -> FldPad`

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

`Precision(x): RngPadElt -> RngIntElt`

`Precision(x): RngPadResElt -> RngIntElt`

`Precision(x): RngPadResExtElt -> RngIntElt`

`Precision(x): FldPadElt -> RngIntElt`

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): RngPadElt -> RngIntElt`

`AbsolutePrecision(x): RngPadResElt -> RngIntElt`

`AbsolutePrecision(x): RngPadResExtElt -> RngIntElt`

`AbsolutePrecision(x): FldPadElt -> RngIntElt`

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

`RelativePrecision(x): RngPadElt -> RngIntElt`

`RelativePrecision(x): RngPadResElt -> RngIntElt`

`RelativePrecision(x): RngPadResExtElt -> RngIntElt`

`RelativePrecision(x): FldPadElt -> RngIntElt`

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

`ChangePrecision(x, k): RngUPolElt, RngIntElt -> RngPadElt`

`ChangePrecision(~x, k): RngUPolElt, RngIntElt -> RngPadElt`

`ChangePrecision(x, k): RngPadElt, RngIntElt -> RngPadElt`

ChangePrecision(~x, k): RngPadElt, RngIntElt -> RngPadElt

ChangePrecision(x, k): FldPadElt, RngIntElt -> FldPadElt

ChangePrecision(~x, k): FldPadElt, RngIntElt -> FldPadElt

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): RngPadElt -> RngPadElt

Expand(x): FldPadElt -> FldPadElt

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): RngPadElt -> RngIntElt

Valuation(x): RngPadResElt -> RngIntElt

Valuation(x): RngPadResExtElt -> RngIntElt

Valuation(x): FldPadElt -> RngIntElt

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

Example: Relative and Absolute Precision (ex-c75a3c)

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.

Example: Pitfalls of Unbounded Precision (ex-a2d0b8)

As indicated above, IsOne for elements of an unbounded precision ring/field will always be false.

```
> IsOne(pAdicField(7)!1);
false
> IsMinusOne(pAdicField(7)!-1);
false
> IsZero(pAdicField(7)!0);
true
```

49.8.5 Logarithms and Exponentials

`Log(x): RngPadElt -> RngPadElt`

`Log(x): RngPadResElt -> RngPadResElt`

`Log(x): RngPadResExtElt -> RngPadResExtElt`

`Log(x): FldPadElt -> FldPadElt`

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): RngPadElt -> RngPadElt`

`Exp(x): RngPadResElt -> RngPadResElt`

`Exp(x): RngPadResExtElt -> RngPadResExtElt`

`Exp(x): FldPadElt -> FldPadElt`

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: Timing Logarithms and Exponentials (ex-bfa7a7)

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
> RelativePrecision(Exp(Log(x)) - x);
0
> x := b^2;
> time Exp(x);
1418565628*b^2 + 1334033745*b + 905945461
Time: 0.170
> RelativePrecision(Log(Exp(x)) - x);
0
> x := b^6;
> time Exp(x);
1700312013*b^2 - 72781965*b + 1129064707
Time: 0.020
```

(continues on next page)

(continued from previous page)

```
> RelativePrecision(Log(Exp(x)) - x);
0
```

49.8.6 Norm and Trace

`Norm(x): RngPadElt -> RngPadElt`

`Norm(x): RngPadResElt -> RngPadResElt`

`Norm(x): RngPadResExtElt -> RngElt`

`Norm(x): FldPadElt -> FldPadElt`

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

`Norm(x, R): RngPadElt, RngPad -> RngPadElt`

`Norm(x, R): RngPadResElt, RngPadRes -> RngPadResElt`

`Norm(x, R): RngPadResExtElt, Rng -> RngElt`

`Norm(x, R): FldPadElt, FldPad -> FldPadElt`

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.

`Trace(x): RngPadElt -> RngPadElt`

`Trace(x): RngPadResElt -> RngPadResElt`

`Trace(x): RngPadResExtElt -> RngElt`

`Trace(x): FldPadElt -> FldPadElt`

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): RngPadElt, RngPad -> RngPadElt`

`Trace(x, R): RngPadResElt, RngPadRes -> RngPadResElt`

`Trace(x, R): RngPadResExtElt, Rng -> RngElt`

`Trace(x, R): FldPadElt, FldPad -> FldPadElt`

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): RngPadElt -> RngUPolElt`

`MinimalPolynomial(x): RngPadResElt -> RngUPolElt`

`MinimalPolynomial(x): RngPadResExtElt -> RngUPolElt`

`MinimalPolynomial(x): FldPadElt -> RngUPolElt`

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

`MinimalPolynomial(x, R): RngPadElt, RngPad -> RngUPolElt`

`MinimalPolynomial(x, R): RngPadResElt, RngPadRes -> RngUPolElt`

`MinimalPolynomial(x, R): RngPadResExtElt, Rng -> RngUPolElt`

`MinimalPolynomial(x, R): FldPadElt, FldPad -> RngUPolElt`

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): RngPadElt -> RngUPolElt`

`CharacteristicPolynomial(x): RngPadResElt -> RngUPolElt`

`CharacteristicPolynomial(x): RngPadResExtElt -> RngUPolElt`

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

`CharacteristicPolynomial(x, R): RngPadElt, RngPad -> RngUPolElt`

`CharacteristicPolynomial(x, R): RngPadResElt, RngPadRes -> RngUPolElt`

`CharacteristicPolynomial(x, R): RngPadResExtElt, RngPadRes -> RngUPolElt`

`CharacteristicPolynomial(x, R): RngPadResExtElt, RngPadResExt -> RngUPolElt`

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): RngPadElt, RngIntElt -> RngPadElt`

`GaloisImage(x, i): RngPadResElt, RngIntElt -> RngPadResElt`

`GaloisImage(x, i): RngPadResExtElt, RngIntElt -> RngPadResExtElt`

`GaloisImage(x, i): FldPadElt, RngIntElt -> FldPadElt`

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

If the top part of the extension is totally ramified, x will be returned. If the top part of the extension is unramified of degree d, this function gives easy access to the cyclic automorphism group of order d. If the functionality of `GaloisImage` is not sufficient, `AutomorphismGroup` may be used instead.

Example: Arithmetic-Geometric Mean Point Counting `(ex-1dd0a7)`

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
```

`EuclideanNorm(x): RngPadResElt -> RngIntElt`

`EuclideanNorm(x): RngRadResExtElt -> RngIntElt`

Return the euclidean norm of the element x of a fixed precision local ring.

49.8.7 Power Relation (Algebraic Dependency)

By taking small linear combinations of powers of a p-adic number, one can attempt to determine if it satisfies an algebraic relation.

`PowerRelation(x, n): FldPadElt, RngIntElt -> RngUPolElt`

Given a p-adic number over Q_p, return a degree n relation for its powers. If this has small coefficients compared to the precision, then the element is likely an approximation to an algebraic number satisfying the return polynomial.

49.8.8 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): FldFinElt, RngPadResExt -> RngPadResExtElt`

`TeichmuellerLift(u, R): FldFinElt, RngPadRes -> RngPadResElt`

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.

49.9 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.

49.10 Roots of Elements

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

`SquareRoot(x): RngPadElt -> RngPadElt`

`SquareRoot(x): RngPadResElt -> RngPadResElt`

`SquareRoot(x): RngPadResExtElt -> RngPadResExtElt`

`SquareRoot(x): FldPadElt -> FldPadElt`

`Sqrt(x): RngPadElt -> RngPadElt`

`Sqrt(x): RngPadResElt -> RngPadResElt`

`Sqrt(x): RngPadResExtElt -> RngPadResExtElt`

`Sqrt(x): FldPadElt -> FldPadElt`

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): RngPadElt -> BoolElt, RngPadElt`

`IsSquare(x): RngPadResElt -> BoolElt, RngPadResElt`

`IsSquare(x): RngPadResExtElt -> BoolElt, RngPadResExtElt`

`IsSquare(x): FldPadElt -> BoolElt, FldPadElt`

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): RngPadElt -> RngPadElt`

`InverseSquareRoot(x): RngPadResElt -> RngPadResElt`

`InverseSquareRoot(x): RngPadResExtElt -> RngPadResExtElt`

`InverseSquareRoot(x): FldPadElt -> FldPadElt`

`InverseSqrt(x): RngPadElt -> RngPadElt`

`InverseSqrt(x): RngPadResElt -> RngPadResElt`

`InverseSqrt(x): RngPadResExtElt -> RngPadResExtElt`

`InverseSqrt(x): FldPadElt -> FldPadElt`

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): RngPadElt, RngPadElt -> RngPadElt`

`InverseSquareRoot(x, y): RngPadResElt, RngPadResElt -> RngPadResElt`

`InverseSquareRoot(x, y): RngPadResExtElt, RngPadResExtElt -> RngPadResExtElt`

`InverseSquareRoot(x, y): FldPadElt, FldPadElt -> FldPadElt`

`InverseSqrt(x, y): RngPadElt, RngPadElt -> RngPadElt`

`InverseSqrt(x, y): RngPadResElt, RngPadResElt -> RngPadResElt`

`InverseSqrt(x, y): RngPadResExtElt, RngPadResExtElt -> RngPadResExtElt`

`InverseSqrt(x, y): FldPadElt, FldPadElt -> FldPadElt`

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): RngPadElt, RngIntElt -> RngPadElt`

`Root(x, n): RngPadResElt, RngIntElt -> RngPadResElt`

`Root(x, n): RngPadResExtElt, RngIntElt -> RngPadResExtElt`

`Root(x, n): FldPadElt, RngIntElt -> FldPadElt`

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): RngPadElt, RngIntElt -> BoolElt, RngPadElt`

`IsPower(x, n): RngPadResElt, RngIntElt -> BoolElt, RngPadResElt`

`IsPower(x, n): RngPadResExtElt, RngIntElt -> BoolElt, RngPadResExtElt`

`IsPower(x, n): FldPadElt, RngIntElt -> BoolElt, FldPadElt`

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): RngPadElt, RngIntElt -> RngPadElt`

`InverseRoot(x, n): RngPadResElt, RngIntElt -> RngPadResElt`

`InverseRoot(x, n): RngPadResExtElt, RngIntElt -> RngPadResExtElt`

`InverseRoot(x, n): FldPadElt, RngIntElt -> FldPadElt`

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): RngPadElt, RngPadElt, RngIntElt -> RngPadElt`

`InverseRoot(x, y, n): RngPadResElt, RngPadResElt, RngIntElt -> RngPadResElt`

`InverseRoot(x, y, n): RngPadResExtElt, RngPadResExtElt, RngIntElt -> RngPadResExtElt`

`InverseRoot(x, y, n): FldPadElt, FldPadElt, RngIntElt -> FldPadElt`

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.

49.11 Polynomials

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

49.11.1 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): RngUPolElt, RngUPolElt -> RngUPolElt

Gcd(f, g): RngUPolElt, RngUPolElt -> RngUPolElt

GCD(f, g): RngUPolElt, RngUPolElt -> RngUPolElt

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: RngUPolElt, RngUPolElt -> RngUPolElt

f mod g: RngUPolElt, RngUPolElt -> RngUPolElt

LeastCommonMultiple(f, g): RngUPolElt, RngUPolElt -> RngUPolElt

Coefficient(f, i): RngUPolElt, RngIntElt -> RngElt

LeadingCoefficient(f): RngUPolElt -> RngElt

Derivative(f): RngUPolElt -> RngUPolElt

Evaluate(f, x): RngUPolElt, RngElt -> RngElt

Example: GCDs of p-adic Polynomials (ex-8d80f7)

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): RngUPolElt, RngIntElt -> RngUPolElt`
Shifts the valuation of each coefficient of f by n, ie. scales the polynomial by π^n.

49.11.2 Roots of Polynomials

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): RngUPolElt -> NwtnPgon`
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 *Newton Polygons* and are illustrated below.

`ValuationsOfRoots(f): RngUPolElt -> SeqEnum[<FldRatElt, RngIntElt>]`
Return a sequence containing pairs which are valuations of roots of f and the number of roots of f which have that valuation.

Example: Newton Polygons and Root Valuations (ex-8d9abc)

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> ]
```

(continues on next page)

(continued from previous page)

```
]
> [$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): RngUPolElt, RngPadElt -> RngPadElt

HenselLift(f, x): RngUPolElt, RngPadResElt -> RngPadResElt

HenselLift(f, x): RngUPolElt, RngPadResExtElt -> RngPadResExtElt

HenselLift(f, x): RngUPolElt, FldPadElt -> FldPadElt

HenselLift(f, x, k): RngUPolElt, RngPadElt, RngIntElt -> RngPadElt

HenselLift(f, x, k): RngUPolElt, RngPadResElt, RngIntElt -> RngPadResElt

HenselLift(f, x, k): RngUPolElt, RngPadResExtElt, RngIntElt -> RngPadResExtElt

HenselLift(f, x, k): RngUPolElt, FldPadElt, RngIntElt -> FldPadElt

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: Square Roots by Hensel Lifting (ex-88d396)

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);
```

(continues on next page)

(continued from previous page)

```
              ^
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.

Functions returning Roots

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

`Roots(f): RngUPolElt -> [ <RngPadElt, RngIntElt> ]`

`Roots(f, R): RngUPolElt, RngPad -> [ <RngPadElt, RngIntElt> ]`

```
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): RngUPolElt -> BoolElt, RngPadElt`

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: Ramified Quadratic Extensions of Z2 (ex-87cd32)

We generate the ramified extensions of $\mathbb{Z}_2$ 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
```

(continues on next page)

(continued from previous page)

```
>         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 $\mathbb{Q}_2$ of degree 2 are in bijection to the 7 non-trivial classes of $\mathbb{Q}_2^*/(\mathbb{Q}_2^*)^2$ and one of these classes yields the unramified extension.

49.11.3 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): RngUPolElt, [RngUPolElt] -> [RngUPolElt]`

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: Hensel Factorization of Polynomials `(ex-aacddc)`

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): RngUPolElt -> BoolElt`

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): RngUPolElt -> [ < RngUPolElt, RngIntElt > ]`

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): RngUPolElt -> [ < RngUPolElt, RngIntElt > ]`

`LocalFactorization(f): RngUPolElt -> [ < RngUPolElt, RngIntElt >]`

```
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 squarefree. 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): RngUPolElt -> RngIntElt`

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): RngUPolElt, RngUPolElt -> BoolElt`

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

`Distance(f, g): RngUPolElt, RngUPolElt -> RngIntElt`

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: Suggested Precision for Factorization (ex-f1ae2a)

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>
```

(continues on next page)

(continued from previous page)

```
]
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: Local Factorizations and Galois Groups (ex-836762)

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.

SplittingField(f): RngUPolElt[FldPad] -> FldPad, SeqEnum

SplittingField(f): RngUPolElt[RngPad] -> FldPad, SeqEnum

```
Std: BoolElt                    Default: true
```

Splitting field F of a squarefree polynomial f over a p-adic ring or field K, and the roots of f in F. The optional parameter **Std** (true by default) specifies whether F/K should be converted to a standard form — at most one ramified extension over one unramified extension of K.

Example: Splitting Field over a p-adic Field (ex-f42dd9)

```
> K:=pAdicField(3,20);
> R<x>:=PolynomialRing(K);
> F:=SplittingField(x^9-3);
> Degree(F,K), RamificationDegree(F,K), InertiaDegree(F,K);
54 54 1
```

49.12 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): RngPad -> [Map]

Automorphisms(L): FldPad -> [Map]

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

Automorphisms(K, k): FldPad, FldPad -> [Map]

Automorphisms(K, k): RngPad, RngPad -> [Map]

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): RngPad -> GrpPerm, Map

AutomorphismGroup(L): FldPad -> GrpPerm, Map

Return the automorphism group acting on L over its p-adic sub-field $\mathbb{Q}_p$ 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): RngPad, RngPad -> GrpPerm, Map

AutomorphismGroup(K, k): FldPad, FldPad -> GrpPerm, Map

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): RngPad -> BoolElt

IsNormal(K): FldPad -> BoolElt

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

IsNormal(K, k): RngPad, RngPad -> BoolElt

IsNormal(K, k): FldPad, FldPad -> BoolElt

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

IsAbelian(K, k): FldPad, FldPad -> BoolElt

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

Continuations(m, L): Map, RngPad -> [Map]

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

IsIsomorphic(E, K): RngPad, RngPad -> BooElt

IsIsomorphic(E, K): FldPad, FldPad -> BooElt

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

Example: Automorphisms of a Local Ring (ex-b62260)

We define an extension of $\mathbb{Z}_2$ 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
```

GaloisGroup(f): RngUPolElt[FldPad] -> GrpPerm, SeqEnum, UserProgram

GaloisGroup(f): RngUPolElt[RngPad] -> GrpPerm, SeqEnum, UserProgram

Computes the Galois group G of a squarefree polynomial f over a p-adic ring or field K. Returns G as a permutation group on the roots of f in its splitting field F, the roots themselves, and a map $G \to \mathrm{Aut}(F/K)$.

Example: Galois Group of a Local Polynomial (ex-286b99)

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> G,r,act:=GaloisGroup(x^4-2);
> G;                                        // permutation group on roots in r
Permutation group G acting on a set of cardinality 4
(1, 4, 2, 3)
(1, 2)
> GroupName(G);
D4
> F<pi>:=Universe(r); F;                            // splitting field F of f
Totally ramified extension defined by the polynomial x^8 + 8*x^7 + 24*x^6 +
   32*x^5 + 18*x^4 + 8*x^3 + 12*x^2 + 8*x + 2
 over 2-adic field mod 2^20
> sigma:=act(G.1);                                  // an automorphism of F/K
> sigma;
Mapping from: FldPad: F to FldPad: F given by a rule [no inverse]
> sigma(F.1);
(43690*pi^7 - 5*pi^6 + 43677*pi^5 - 15*pi^4 + 43687*pi^3 + 43684*pi - 3)*pi +
   O(pi^136)
```

49.13 Completions

Local rings can be obtained by completing an order at a prime ideal (see Chapter Number Fields and Completion(K, P): FldAlg, RngOrdIdl → FldLoc, Map).

Completion(O, P): RngOrd, RngOrdIdl -> RngPad, Map

Completion(K, P): FldNum, RngOrdIdl -> FldPad, Map

```
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): RngOrdIdl, RngIntElt -> RngPad, Map

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

Example: Completion of a Number Field (ex-2bb9ae)

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
```

(continues on next page)

(continued from previous page)

```
> mC(K.1);
(46564489*$.1 - 47959419)*C.1 - 116434149*$.1 - 61099304 + O(C.1^20)
> delta := (K.1 @ mC @@ mC) - K.1;
> delta;
8337821493402521350488*K.1^5 - 690735069600568964644 32*K.1^4 +
    189847416443444330877726*K.1^3 - 45336153029197695133787 6*K.1^2 +
    336979647814116799276099*K.1 - 267520869714197002579071
> // Check the accuracy of the mappings using the valuation of the difference
> Valuation(delta, lp[2][1]);
18
> C`DefaultPrecision := 30;
> mC(K.1);
(1090965976127*$.1 - 1208477074641)*C.1 - 589359803563*$.1 + 288063654676 +
O(C.1^30)
> delta := (K.1 @ mC @@ mC) - K.1;
> delta;
-619800242441603716728687734334907 83*K.1^5 +
    118979680306480309259329108876896 8754*K.1^4 -
    320235394693319058886430918065386 8957*K.1^3 +
    753738692804616458073114503187201 7049*K.1^2 -
    551129700293668257996421058601330 8810*K.1 +
    443809944480643131358253343594109 8722
> Valuation(delta, lp[2][1]);
28
> C`DefaultPrecision := 10;
> mC(K.1);
(-7708*$.1 + 7759)*C.1 + 4747*$.1 - 5859 + O(C.1^10)
> delta := (K.1 @ mC @@ mC) - K.1;
> delta;
1908210240*K.1^5 - 7326424608*K.1^4 + 16701662320*K.1^3 - 35965440540*K.1^2 +
    41324075079*K.1 - 30476856505
> Valuation(delta, lp[2][1]);
8
```

49.14 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.

49.14.1 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): RngPad -> SeqEnum

`PrincipalUnitGroupGenerators(R): FldPad -> SeqEnum`

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

`PrincipalUnitGroup(R): RngPad -> GrpAb, Map`

`PrincipalUnitGroup(R): RngPadResExt -> GrpAb, Map`

`PrincipalUnitGroup(R): RngPadRes -> GrpAb, Map`

The principal units of a p-adic ring or field R are elements of the form $1 + \pi\mathbb{Z}_R$ where π is a uniformizing element of R and $\mathbb{Z}_R$ 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): RngPad -> GrpAb, Map`

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): FldPad -> GrpAb, Map`

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

`UnitGroupGenerators(R): RngPad -> SeqEnum`

`UnitGroupGenerators(R): RngPadRes -> SeqEnum`

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

`UnitGroupGenerators(F): FldPad -> SeqEnum`

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

`pSelmerGroup(p, F): RngIntElt, FldPad -> GrpAb, Map`

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.

49.14.2 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): FldPad, Map -> GrpAb, Map`

`NormGroup(R, m): RngPad, Map -> GrpAb, Map`

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): FldPad, Map, RngElt -> BoolElt, RngElt`

`NormEquation(R, m, b): RngPad, Map, RngElt -> BoolElt, RngElt`

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): Map, Map, GrpAb -> GrpAb, Map`

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): Map, Map, GrpAb -> GrpAb`

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): Map, Map -> GrpAb`

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.

49.14.3 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): Map, GrpAb -> FldAb`

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): Map, GrpAb -> RngIntElt`

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.

49.15 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): RngPad, RngIntElt -> [RngPad]`

`AllExtensions(R, n): FldPad, RngIntElt -> [RngPad]`

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

Given a p-adic ring or field R and some positive integer n, compute all the extensions of R of degree n. At least one extension is given in every isomorphism class. The optional parameters can be used to impose restrictions on the fields returned. Note that j and vD must be unspecified (as -1) when $F \neq 1$ or when E and F are both not given.

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.

vD
: specifies the valuation of the discriminant, -1 implies no restriction.

j
: specifies the valuation of the discriminant via the formula $vD := n + j - 1$.

Galois
when set to **true**, limits the extensions to only list normal extensions.

NumberOfExtensions(R, n): RngPad, RngIntElt -> RngIntElt

```
E     : RngIntElt                    Default: 0
F     : RngIntElt                    Default: 0
Galois: BoolElt                      Default: false
vD    : RngIntElt                    Default: -1
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. Note that j and vD must be unspecified (as -1) when $F \neq 1$ or when E and F are both not given.

Note that the count will not be the same as **AllExtensions**, as the latter need only be up to isomorphism.

OreConditions(R, n, j): RngPad, RngIntElt, RngIntElt -> BoolElt

OreConditions(R, n, j): FldPad, RngIntElt, RngIntElt -> BoolElt

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

Example: All Extensions of Degree 9 over Q3 (ex-ea2536)

We follow Examples 9.1 and 9.2 from [PR01].

There are 54 (totally ramified) extensions of degree 9 and discriminant 3^{9+4-1} over $\mathbb{Q}_3$. There are six generating polynomials, each defining nine isomorphism classes. The possible (nontrivial) subfields of these have degree 3 and $j_0 = 1$, of which there are two defining polynomials each with three isomorphism classes. Each of these degree 3 fields then admits two extensions with $j_1 = 1$, which give six isomorphism classes. This gives a total of 27 degree 9 extensions that have a subfield of degree 3.

```
> R := pAdicRing(3,20);
> _<x> := PolynomialRing(R); // for printing
> NumberOfExtensions(R,9 : F:=1,j:=4);
54
> A9 := AllExtensions(R,9 : F:=1,j:=4);
> [DefiningPolynomial(a) : a in A9];
[ x^9 + 3*x^4 + 3, x^9 + 6*x^4 + 3,
  x^9 + 3*x^4 + 3*x^3 + 3, x^9 + 6*x^4 + 3*x^3 + 3,
  x^9 + 3*x^4 + 6*x^3 + 3, x^9 + 6*x^4 + 6*x^3 + 3 ]
> A3 := AllExtensions(R,3 : F:=1,j:=1);
> NumberOfExtensions(A3[1],3 : F:=1,j:=1);
6
> [DefiningPolynomial(a) : a in A3];
[ x^3 + 3*x + 3, x^3 + 6*x + 3 ]
> _<pi> := A3[1];
> _<y> := PolynomialRing(A3[1]);
> B3 := AllExtensions(A3[1],3 : F:=1,j:=1);
> [DefiningPolynomial(f) : f in B3];
[ y^3 + pi*y + pi, y^3 + 2*pi*y + pi ]
```

The other example concerns degree 10 extensions of $\mathbb{Q}_5$. Here there are 1818 total extensions, of which 1 is unramified and 2 have ramification degree 2, while 605 have ramification degree 5 and 1210 are totally ramified. With ramification degree 5, there are 145 defining polynomials over the unramified quadratic field, split into five j-groupings. As noted in [PR01], there is a further splitting in the $j = 4$ grouping. Similarly, there are 145 defining polynomials over either of the two tamely ramified extensions of degree 2 over $\mathbb{Q}_5$. The resulting fields are in fact isomorphic in

pairs, but the `AllExtensions` function still lists both fields in each pair. So it returns 438 fields (1+2+145+290) rather than the stated 293 isomorphism classes (1+2+145+145).

```
> R := pAdicRing(5,20);
> NumberOfExtensions(R,10);
1818
> [NumberOfExtensions(R,10 : E:=e) : e in Divisors(10)];
[ 1, 2, 605, 1210 ]
> U := UnramifiedExtension(R,2);
> [#AllExtensions(U,5 : E:=5,j:=j0) : j0 in [1..5]];
[ 24, 24, 24, 48, 25 ]
> // compare the above/below to (#K)/N in Pauli-Roblot
> [#AllExtensions(R,10 : E:=10,j:=j0): j0 in [1..10]];
[ 8, 8, 8, 16, 0, 40, 40, 80, 40, 50 ] // twice P-R
```

49.16 Exact p-Adic Rings

49.16.1 Introduction

The exact p-adic rings and fields use representations which contain a kind and a list of dependencies. These rings and their elements can then be approximated to as much precision as the objects they depend on allow, some to arbitrary precision depending on their construction. Approximations are calculated to some epoch n which is equivalent to precision 2^n. Such representations are often referred to as "lazy" as the work of computing to precision is delayed until it is necessary. The exact p-adic rings and fields use the inexact p-adic rings and fields described in the majority of this chapter for approximations.

For example, an element of an exact p-adic ring resulting from the addition of 2 elements will store that it is the sum of these two elements and approximations to this sum are computed by adding the corresponding approximations of the summands, when such approximations are required.

An advantage of this lazy representation is that a precision does not have be decided before computations are started. A disadvantage of this exact representation occurs when approximations are required to incredibly high precision, some precisions which may be considerably time and space expensive.

Exact p-adic rings have type `RngXPad`, exact p-adic fields `FldXPad`, their elements `RngXPadElt` and `FldXPadElt` respectively. There is also a dedicated polynomial ring type `RngUPolXPad` whose elements have type `RngUPolXPadElt`.

For further information see [Dor21].

`IsExactpAdic(x): Any -> BoolElt`

Returns `true` if x is an exact p-adic object, such as a ring, field, element, or polynomial.

49.16.2 Exact p-adic Rings and Fields

In this section we describe the construction and functionality of the exact p-adic rings and fields.

Construction of Exact p-adic Rings and Fields

`pAdicField(p : parameters): RngIntElt -> FldXPad`

```
Exact : BoolElt                    Default: false
Global: BoolElt                    Default: true
```

Returns $\mathbb{Q}_p$ as an exact p-adic field if `Exact` is set to `true` otherwise returns an inexact p-adic field. If `Global` is set to `false` then a new object is constructed rather than reusing an existing object.

`pAdicRing(p : parameters): RngIntElt -> FldXPad`

```
Exact : BoolElt                    Default: false
Global: BoolElt                    Default: true
```

Returns $\mathbb{Z}_p$ as an exact p-adic integer ring if `Exact` is set to `true` otherwise returns an inexact p-adic field. If `Global` is set to `false` then a new object is constructed rather than reusing an identical existing object.

`ext<K | f>: FldXPad, RngUPolElt -> RngXPad`

`ext<K | f>: FldXPad, RngUPolXPadElt -> RngXPad`

`ext<K | f>: FldXPad, [RngXPadElt] -> RngXPad`

`ext<K | d>: FldXPad, RngIntElt -> RngXPad`

`ext<K | f>: RngXPad, RngUPolElt -> RngXPad`

`ext<K | f>: RngXPad, RngUPolXPadElt -> RngXPad`

`ext<K | f>: RngXPad, [RngXPadElt] -> RngXPad`

`ext<K | d>: RngXPad, RngIntElt -> RngXPad`

Construct the extension defined by the polynomial f, which must be inertial or Eisenstein, or an unramified extension of degree d.

Example: Constructing Exact p-adic Fields (ex-72b734)

The construction of an exact p-adic field is shown as well as extensions, both unramified and ramified.

```
> Qp := pAdicField(11 : Exact);
> U := ext<Qp | 3>;
> U;
Unramified extension of degree 3 of 11-adic field (exact)
> R := ext<U | Polynomial([11, 11^2, 1])>;
> R;
Totally ramified extension of degree 2 of Unramified extension of degree 3 of
11-adic field (exact)
> ext<Qp | Polynomial([3, 5, 1])>;
Unramified extension of degree 2 of 11-adic field (exact)
> ext<Qp | Polynomial([11, 2*11^2, 1])>;
Totally ramified extension of degree 2 of 11-adic field (exact)
```

Related Structures

`RingOfIntegers(L): FldXPad -> RngXPad`

`RingOfIntegers(L): RngXPad -> RngXPad`

`IntegerRing(L): FldXPad -> RngXPad`

`Integers(L): FldXPad -> RngXPad`

`IntegerRing(L): RngXPad -> RngXPad`

`Integers(L): RngXPad -> RngXPad`

The ring of integers of the exact p-adic field L.

`FieldOfFractions(L): FldXPad -> FldXPad`

`FieldOfFractions(L): RngXPad -> FldXPad`

The field of fractions of the exact p-adic ring L.

`BaseField(F): FldXPad -> FldXPad`

`CoefficientField(F): FldXPad -> FldXPad`

Returns the base field of the exact p-adic field F when F is an extension, or F when F is the prime field.

`BaseRing(R): RngXPad -> RngXPad`

`CoefficientRing(R): RngXPad -> RngXPad`

Returns the base ring of the exact p-adic ring R when R is an extension, or R when R is the prime ring.

`ResidueClassField(L): FldXPad -> FldFin, Map`

`ResidueClassField(L): RngXPad -> FldFin, Map`

The residue class field F of the exact p-adic ring or field K, as a finite field, and the map from K to F.

`pAdicQuotientRing(L, k): FldXPad, RngIntElt -> RngPadRes, Map`

`pAdicQuotientRing(L, k): RngXPad, RngIntElt -> RngPadRes, Map`

`quo<L | x>: RngXPad, Any -> RngPadRes, Map`

`quo<L | x>: FldXPad, Any -> RngPadRes, Map`

The quotient of the exact p-adic ring or field L by $L!x$, or by π^k where π is the uniformizing element of L.

`InfinitePrecisionApproximation(K): FldXPad -> FldPad`

`InfinitePrecisionApproximation(K): RngXPad -> RngPad`

The p-adic field of type `FldPad` or the p-adic ring of type `RngPad` with `Precision Infinity()` compatible with all finite–precision approximations of the exact p-adic field or ring K.

`R eq T: RngXPad, RngXPad -> BoolElt`

`R eq T: FldXPad, FldXPad -> BoolElt`

Return whether the exact p-adic rings or fields R and T are equal.

Example: Structures Related to Exact p-adic Rings (ex-34e873)

```
> Qp := pAdicField(5 : Exact);
> Zp := Integers(Qp);
> FieldOfFractions(Zp) eq Qp;
true
> L := ext<ext<Zp | 5> | Polynomial([5, 5^3, 5^2, 1])>;
> L;
Totally ramified extension of degree 3 of Unramified extension of degree 5 of
5-adic ring (exact)
> BaseRing(L);
Unramified extension of degree 5 of 5-adic ring (exact)
> ResidueClassField(L);
Finite field of size 5^5
Mapping from: RngXPad: L to GF(5^5)
> InfinitePrecisionApproximation(L);
Totally ramified extension defined by a map over Unramified extension defined by
a map over 5-adic ring
> pAdicQuotientRing(L, 20);
Quotient of Totally ramified extension of Unramified extension of Quotient of
the 5-adic ring modulo the ideal generated by 5^7 modulo x^5 + 4*x + 3 modulo
x^3 + 25*x^2 + 125*x + 5 and $.1^20
Mapping from: RngXPad: L to ChangePrecision(TotallyRamifiedExtension(
UnramifiedExtension(pAdicQuotientRing(5, 7), Polynomial(pAdicQuotientRing(5, 7),
\\[3, 4, 0, 0, 0, 1])), Polynomial([UnramifiedExtension(pAdicQuotientRing(5, 7),
Polynomial(pAdicQuotientRing(5, 7), \\[3, 4, 0, 0, 0, 1])) | [5], [125], [25],
[1]])), 20)
```

Generating Elements

`R . i: FldXPad, RngIntElt -> FldXPadElt`

`R . i: RngXPad, RngIntElt -> RngXPadElt`

`Name(R, i): RngXPad, RngIntElt -> RngXPadElt`

`Name(R, i): FldXPad, RngIntElt -> FldXPadElt`

Given an exact p-adic ring or field R and a positive integer i return the i-th generator of R.

`AssignNames(~R, S): RngXPad, SeqEnum[MonStgElt]`

`AssignNames(~R, S): FldXPad, SeqEnum[MonStgElt]`

Given an exact p-adic ring or field R assign the string in the sequence S of length 1 to be the name used when printing elements of R. On construction of R, `R<x>` can be used to assign the string `"x"` as the name to be used when printing in which case the identifier `x` will be assigned the value of `R.1`.

`Generator(R): FldXPad -> FldXPadElt`

`Generator(R): RngXPad -> RngXPadElt`

Returns the generator of the exact p-adic ring or field R over its base field. For extensions R, a root of its defining polynomial is returned, otherwise 1.

`UniformizingElement(R): FldXPad -> FldXPadElt`

`UniformizingElement(R): RngXPad -> RngXPadElt`

Returns an element of valuation 1 of the exact p-adic ring or field R.

`ResidueGenerator(R): FldXPad -> FldXPadElt`

`ResidueGenerator(R): RngXPad -> RngXPadElt`

Returns an element of valuation 0 whose residue class generates the residue class field of the exact p-adic ring or field R over $\mathbb{F}_p$.

`AbsoluteGenerator(R): FldXPad -> FldXPadElt`

`AbsoluteGenerator(R): RngXPad -> RngXPadElt`

An element which generates the exact p-adic ring or field R over its prime subfield. This will be either 1, a uniformizing element, a residue generator, or the sum of the two, depending on the inertia and ramification degrees.

Example: Generating Elements of Exact p-adic Rings (ex-1abc7d)

```
> Zp := pAdicRing(7 : Exact);
> Zp.1;
1 + O(7^20)
> UniformizingElement(Zp);
7 + O(7^20)
> ResidueGenerator(Zp);
1 + O(7^20)
> L := ext<ext<Zp | Polynomial([7, 77, 7*17, 1])> | 4>;
> BL := BaseRing(L);
> L.1;
L.1 + O($.1^20)
> Generator(L);
L.1 + O($.1^20)
> UniformizingElement(L);
BL.1 + O(BL.1^20)
> ResidueGenerator(L);
L.1 + O($.1^20)
> AbsoluteGenerator(L);
L.1 + BL.1 + O(BL.1^20)
```

Invariants

`Prime(L): FldXPad -> RngIntElt`

`Prime(L): RngXPad -> RngIntElt`

The (rational) prime p having valuation equal to the ramification degree of the exact p-adic ring or field L.

`Degree(L, K): FldXPad, FldXPad -> RngIntElt`

`Degree(L): FldXPad -> RngIntElt`

`Degree(L): RngXPad -> RngIntElt`

`Degree(L, K): RngXPad, RngXPad -> RngIntElt`

The degree of the extension L over the exact p-adic ring or field K if given or the base ring of L otherwise.

`InertiaDegree(L, K): FldXPad, FldXPad -> RngIntElt`

`InertiaDegree(L): FldXPad -> RngIntElt`

`InertiaDegree(L, K): RngXPad, RngXPad -> RngIntElt`

`InertiaDegree(L): RngXPad -> RngIntElt`

The inertia degree of the extension L over the exact p-adic ring or field K if given or the base ring of L otherwise.

`RamificationDegree(L, K): FldXPad, FldXPad -> RngIntElt`

`RamificationDegree(L): FldXPad -> RngIntElt`

`RamificationIndex(L): FldXPad -> RngIntElt`

`RamificationIndex(L, K): FldXPad, FldXPad -> RngIntElt`

`RamificationDegree(L): RngXPad -> RngIntElt`

`RamificationDegree(L, K): RngXPad, RngXPad -> RngIntElt`

`RamificationIndex(L): RngXPad -> RngIntElt`

`RamificationIndex(L, K): RngXPad, RngXPad -> RngIntElt`

The ramification degree of the extension L over the exact p-adic ring or field K if given or the base ring of L otherwise.

`DefiningPolynomial(R): RngXPad -> RngUPolElt`

`DefiningPolynomial(R): FldXPad -> RngUPolElt`

`Exact: BoolElt` `Default: true`

Given an exact p-adic ring or field R returns the defining polynomial of R with type `RngUPolElt`, unless `Exact` is set to `true`, in which case the polynomial is returned with type `RngUPolXPadElt`.

`AbsoluteDegree(F): FldXPad -> RngIntElt`

`AbsoluteInertiaDegree(L): FldXPad -> RngIntElt`

`AbsoluteInertiaIndex(L): FldXPad -> RngIntElt`

`AbsoluteRamificationDegree(L): FldXPad -> RngIntElt`

`AbsoluteRamificationIndex(L): FldXPad -> RngIntElt`

`AbsoluteDegree(F): RngXPad -> RngIntElt`

`AbsoluteInertiaDegree(L): RngXPad -> RngIntElt`

`AbsoluteInertiaIndex(L): RngXPad -> RngIntElt`

`AbsoluteRamificationDegree(L): RngXPad -> RngIntElt`

`AbsoluteRamificationIndex(L): RngXPad -> RngIntElt`

The degree, inertia degree, or ramification degree of the exact p-adic ring or field L over its prime subfield.

`DiscriminantValuation(L): FldXPad -> RngIntElt`

`DiscriminantValuation(L, K): FldXPad, FldXPad -> RngIntElt`

`DiscriminantValuation(L, K): RngXPad, RngXPad -> RngIntElt`

`DiscriminantValuation(L): RngXPad -> RngIntElt`

The valuation of the discriminant of the extension L/K of exact p-adic rings or fields. This is computed as the height of the leftmost vertex of the ramification polygon (and so avoids actually computing the discriminant). If it is not specified K defaults to the base ring of L.

RamificationPolygon(L): FldXPad -> NwtnPgon

RamificationPolygon(L, K): FldXPad, FldXPad -> NwtnPgon

RamificationPolygon(L, K): RngXPad, RngXPad -> NwtnPgon

RamificationPolygon(L): RngXPad -> NwtnPgon

RamificationPolygon(f): RngUPolXPadElt[FldXPad] -> NwtnPgon

RamificationPolygon(f): RngUPolXPadElt[RngXPad] -> NwtnPgon

Constructs the ramification polygon of the Eisenstein or inertial polynomial f over an exact p-adic ring or field. The input can instead be an extension L/K, where K defaults to the base ring of L if it is not specified.

The locations and slopes of the vertices correspond to different subfields fixed by different ramification subgroups of the Galois group. For example a horizontal face corresponds to an unramified subfield, and a face of slope -1 corresponds to a tame subfield. This is represented as a Newton Polygon (type NwtnPgon).

Example: Invariants of Exact p-adic Fields (ex-5789a2)

```
> Qp := pAdicField(17 : Exact);
> Prime(Qp);
17
> Degree(Qp);
1
> L := ext<ext<Qp | 7> | Polynomial([3*17, 4*17^2, 2*17, 1])>;
> Prime(L);
17
> Degree(L);
3
> Degree(L, Qp);
21
> Degree(L, BaseField(L));
3
> InertiaDegree(L);
1
> RamificationDegree(L);
3
> DefiningPolynomial(L);
(1 + O(17^20))*$.1^3 + (2*17 + O(17^21))*$.1^2 + (4*17^2 + O(17^22))*$.1 + 3*17
    + O(17^21)
> Parent($1);
Univariate Polynomial Ring over Unramified extension of degree 7 of 17-adic
field (exact)
> DefiningPolynomial(CoefficientField(L));
(1 + O(17^20))*$.1^7 + O(17^2048)*$.1^6 + O(17^2048)*$.1^5 + O(17^2048)*$.1^4 +
    O(17^2048)*$.1^3 + O(17^2048)*$.1^2 + (12 + O(17^20))*$.1 + 14 + O(17^20)
> DefiningPolynomial(L : Exact);
(1 + O(17^20))*$.1^3 + (2*17 + O(17^21))*$.1^2 + (4*17^2 + O(17^22))*$.1 + 3*17
    + O(17^21)
> Parent($1);
Univariate polynomial ring (exact) over Unramified extension of degree 7 of
17-adic field (exact)
> AbsoluteDegree(L);
21
> DiscriminantValuation(L);
2
> DiscriminantValuation(L, Qp);
2
> RamificationPolygon(L);
```

(continues on next page)

(continued from previous page)

```
Newton Polygon with vertices {(1, 2), (3, 0)} and defining points {(1, 2), (2,
1), (3, 0)}
```

49.16.3 Exact p-adic Elements

In this section we describe the construction and functionality related to elements of exact p-adic rings and fields.

GetExactpAdicsPrintPrecision() -> RngIntElt

Returns the precision to which exact p-adic objects are printed by default.

SetExactpAdicsPrintPrecision(k): Infty

SetExactpAdicsPrintPrecision(k): RngIntElt

Set the precision to which exact p-adic objects are printed by default to the integer k or ∞.

K ! x: FldXPad, Any -> FldXPadElt

R ! x: RngXPad, Any -> RngXPadElt

elt<K | x>: FldXPad, Any -> FldXPadElt

elt<R | x>: RngXPad, Any -> RngXPadElt

Elements of an exact p-adic ring or field may be coerced from:

- Integers or rationals
- Any base field
- The residue class field
- Any compatible finite-precision p-adic field, ring, or quotient ring

In the latter two cases, the coerced element has its precision capped to the precision of x, so is not really "exact". To select a full-precision lift, use `CoerceAndLift(K,x)`.

AbsolutePrecision(x): FldXPadElt -> RngIntElt

AbsolutePrecision(x): RngXPadElt -> RngIntElt

The precision of the exact p-adic element x relative to valuation 0.

RelativePrecision(x): FldXPadElt -> RngIntElt

RelativePrecision(x): RngXPadElt -> RngIntElt

The precision of the exact p-adic element x relative to its weak valuation. The relative precision is zero if and only if x is weakly zero.

Valuation(x): FldXPadElt -> RngIntElt

Valuation(x): RngXPadElt -> RngIntElt

The valuation of the exact p-adic element x. Note that if x is weakly zero, its epoch will be increased until it is no longer weakly zero, or until its weak valuation is infinite. If x is actually zero, this could compute forever.

WeakValuation(x): RngXPadElt -> RngIntElt

WeakValuation(x): FldXPadElt -> RngIntElt

A lower bound on the valuation of the exact p-adic element x. It is equal to the true valuation unless x is weakly zero.

ShiftValuation(x, n): FldXPadElt, RngIntElt -> FldXPadElt

ShiftValuation(x, n): RngXPadElt, RngIntElt -> RngXPadElt

Multiply the exact p-adic element x by the mth power of the uniformizing element. This returns an element whose valuation is m more than that of x.

ValuationEq(x, n): FldXPadElt, RngIntElt -> BoolElt

ValuationEq(x, n): FldXPadElt, Infty -> BoolElt

`ValuationEq(x, n): RngXPadElt, RngIntElt -> BoolElt`
`ValuationEq(x, n): RngXPadElt, Infty -> BoolElt`
`ValuationNe(x, n): FldXPadElt, RngIntElt -> BoolElt`
`ValuationNe(x, n): FldXPadElt, Infty -> BoolElt`
`ValuationNe(x, n): RngXPadElt, RngIntElt -> BoolElt`
`ValuationNe(x, n): RngXPadElt, Infty -> BoolElt`
`ValuationGe(x, n): FldXPadElt, RngIntElt -> BoolElt`
`ValuationGe(x, n): FldXPadElt, Infty -> BoolElt`
`ValuationGe(x, n): RngXPadElt, RngIntElt -> BoolElt`
`ValuationGe(x, n): RngXPadElt, Infty -> BoolElt`
`ValuationGt(x, n): FldXPadElt, Infty -> BoolElt`
`ValuationGt(x, n): FldXPadElt, RngIntElt -> BoolElt`
`ValuationGt(x, n): RngXPadElt, RngIntElt -> BoolElt`
`ValuationGt(x, n): RngXPadElt, Infty -> BoolElt`
`ValuationLe(x, n): FldXPadElt, Infty -> BoolElt`
`ValuationLe(x, n): FldXPadElt, RngIntElt -> BoolElt`
`ValuationLe(x, n): RngXPadElt, RngIntElt -> BoolElt`
`ValuationLe(x, n): RngXPadElt, Infty -> BoolElt`
`ValuationLt(x, n): FldXPadElt, RngIntElt -> BoolElt`
`ValuationLt(x, n): FldXPadElt, Infty -> BoolElt`
`ValuationLt(x, n): RngXPadElt, Infty -> BoolElt`
`ValuationLt(x, n): RngXPadElt, RngIntElt -> BoolElt`

Compare the valuation of the exact p-adic element x to n. These always terminate.

`IsUnit(x): RngXPadElt -> BoolElt`
`IsUnit(x): FldXPadElt -> BoolElt`

Return `true` if the exact p-adic element x is a unit.

`IsIntegral(x): RngXPadElt -> BoolElt`
`IsIntegral(x): FldXPadElt -> BoolElt`

Returns `true` if the exact p-adic element x is an integer.

`IsWeaklyZero(x): StrAnyXPadElt -> BoolElt`
`IsWeaklyEqual(x, y): StrAnyXPadElt, StrAnyXPadElt -> BoolElt`

Returns `true` if the exact p-adic element x, respectively $x - y$, is zero up to its current precision. If x, respectively $x - y$, is zero then it is weakly zero, but not necessarily the other way around.

`IsDefinitelyZero(x): StrAnyXPadElt -> BoolElt`
`IsDefinitelyEqual(x, y): StrAnyXPadElt, StrAnyXPadElt -> BoolElt`

Returns `true` if the exact p-adic element x is definitely zero, respectively, x and y are identical or both definitely zero.

`CoerceAndLift(S, x): StrAnyXPad, Any -> StrAnyXPadElt`

Coerce x into the exact p-adic structure S, and ensure its precision is unbounded. If x is a finite-precision object, then ordinary coercion `S!x` will have capped precision whereas `CoerceAndLift(S, x)` will "lift" this to have unbounded precision

Arithmetic with Elements

`x * y: FldXPadElt, FldXPadElt -> FldXPadElt`
`x * y: RngXPadElt, RngXPadElt -> RngXPadElt`
`x + y: FldXPadElt, FldXPadElt -> FldXPadElt`
`x + y: RngXPadElt, RngXPadElt -> RngXPadElt`

```
- y: FldXPadElt -> FldXPadElt
- y: RngXPadElt -> RngXPadElt
x - y: RngXPadElt, RngXPadElt -> RngXPadElt
x - y: FldXPadElt, FldXPadElt -> FldXPadElt
x ^ n: FldXPadElt, RngIntElt -> FldXPadElt
x ^ n: RngXPadElt, RngIntElt -> RngXPadElt
x / y: FldXPadElt, FldXPadElt -> FldXPadElt
x / y: RngXPadElt, RngXPadElt -> RngXPadElt
x div y: RngXPadElt, RngXPadElt -> RngXPadElt
Quotrem(x, y): RngXPadElt, RngXPadElt -> RngXPadElt, RngXPadElt
GCD(x, y): FldXPadElt, FldXPadElt -> FldXPadElt
Gcd(x, y): FldXPadElt, FldXPadElt -> FldXPadElt
GreatestCommonDivisor(x, y): FldXPadElt, FldXPadElt -> FldXPadElt
GCD(x, y): RngXPadElt, RngXPadElt -> RngXPadElt
Gcd(x, y): RngXPadElt, RngXPadElt -> RngXPadElt
GreatestCommonDivisor(x, y): RngXPadElt, RngXPadElt -> RngXPadElt
```

Given exact p-adic elements x and y return their greatest common divisor.

```
ExtendedGreatestCommonDivisor(x, y): FldXPadElt, FldXPadElt -> FldXPadElt, FldXPadElt,
                                     FldXPadElt
XGCD(x, y): FldXPadElt, FldXPadElt -> FldXPadElt, FldXPadElt, FldXPadElt
Xgcd(x, y): FldXPadElt, FldXPadElt -> FldXPadElt, FldXPadElt, FldXPadElt
ExtendedGreatestCommonDivisor(x, y): RngXPadElt, RngXPadElt -> RngXPadElt, RngXPadElt,
                                     RngXPadElt
XGCD(x, y): RngXPadElt, RngXPadElt -> RngXPadElt, RngXPadElt, RngXPadElt
Xgcd(x, y): RngXPadElt, RngXPadElt -> RngXPadElt, RngXPadElt, RngXPadElt
```

Given exact p-adic elements x and y return their greatest common divisor g as well as a and b such that $g = ax + by$.

Example: Arithmetic with Exact p-adic Elements (ex-bdd87d)

```
> Zp := pAdicRing(19 :  Exact);
> x := Zp!3948701;
> y := Zp!748873*19;
> x;
3948701 + O(19^20)
> y;
14228587 + O(19^20)
> FieldOfFractions(Zp)!x;
3948701 + O(19^20)
> FieldOfFractions(Zp)!y;
748873*19 + O(19^21)
> AbsolutePrecision(x);
32
> RelativePrecision(x);
32
> Valuation(x);
0
> WeakValuation(x);
0
> ShiftValuation(x, 5);
9777374597399 + O(19^20)
> ValuationGe($1, 5);
```

(continues on next page)

(continued from previous page)

```
true
> IsUnit(x);
true
> IsIntegral(x);
true
> Valuation(y);
1
> x + y;
18177288 + O(19^20)
> x*y;
56184435715487 + O(19^20)
> Valuation(y^5);
5
> x/y;
-1467808620217597850228656 6*19^-1 + O(19^19)
> Quotrem(x, y);
O(19^20) 3948701 + O(19^20)
> Quotrem(y, x);
153721219306072143297374 77 + O(19^20) O(19^20)
> Gcd(x, y);
3948701 + O(19^20)
> Xgcd(19*x, y);
75025319 + O(19^20) 1 + O(19^20) O(19^20)
> IsWeaklyZero(Zp!0);
true
> IsDefinitelyZero(Zp!0);
false
> IsDefinitelyZero(FieldOfFractions(Zp)!0);
true
> IsDefinitelyEqual(x, x);
true
> CoerceAndLift(Zp, 0);
O(19^20)
> IsWeaklyZero($1);
true
> IsDefinitelyZero($2);
false
> CoerceAndLift(FieldOfFractions(Zp), 0);
0
> IsDefinitelyZero($1);
true
```

49.16.4 Polynomials over Exact p-adic Rings and Fields

While polynomial rings of type `RngUPol` and polynomials of type `RngUPolElt` can be constructed in the regular way over exact p-adic rings and fields there is also a specific type of polynomial rings which can be constructed over the exact p-adic rings and fields. These polynomial rings have type `RngUPolXPad` and their elements have type `RngUPolXPadElt`.

Exact Polynomial Rings

In this section we describe the construction and functionality for polynomial rings of type `RngUPolXPad`.

`PolynomialRing(R : parameters): Rng -> RngUPol`

```
Global: BoolElt                    Default: true
Exact : BoolElt                    Default: false
```

A polynomial ring specifically to parent polynomials with exact p-adic coefficients exists to be able to implement new kinds of polynomials which are best implemented by giving approximations of the entire polynomial wholesale, instead of giving each coefficient individually. This ring is constructed when the `Exact` parameter is set to `true`, otherwise an ordinary polynomial ring over the exact p-adic ring or field R is constructed.

`R eq T: RngUPolXPad, RngUPolXPad -> BoolElt`

Given polynomial rings R and T over exact p-adic rings or fields, return whether R and T are equal.

`BaseRing(R): RngUPolXPad -> Rng`

`CoefficientRing(R): RngUPolXPad -> Rng`

Returns the coefficient ring of the polynomial ring R over an exact p-adic ring or field.

`R . i: RngUPolXPad, RngIntElt -> RngXPadElt`

`Name(R, i): RngUPolXPad, RngIntElt -> RngUPolXPadElt`

`Generator(R): RngUPolXPad -> RngUPolXPadElt`

Returns the polynomial variable of the polynomial ring R over an exact p-adic ring of field.

`AssignNames(~R, S): RngUPolXPad, SeqEnum[MonStgElt]`

Given a polynomial ring over an exact p-adic ring or field R assign the string in the sequence S of length 1 to be the name used for the variable of R when printing elements of R. On construction of R, `R<x>` can be used to assign the string `"x"` as the name to be used when printing in which case the identifier `x` will be assigned the value of `R.1`.

Example: Exact p-adic Polynomial Rings (ex-b44a56)

```
> Zp := pAdicRing(7 : Exact);
> PolynomialRing(Zp);
Univariate Polynomial Ring over 7-adic ring (exact)
> PolynomialRing(Zp : Exact);
Univariate polynomial ring (exact) over 7-adic ring (exact)
```

Polynomials

In these next sections we describe the construction and functionality for polynomials of type `RngUPolXPadElt`.

`R ! f: RngUPolXPad, Any -> RngUPolXPadElt`

Polynomials with exact p-adic coefficients can be constructed from:

- Other polynomials, either exact or ordinary, over any exact p-adic ring.
- Sequences of coefficients coercible to the base ring.
- Anything coercible to the base ring. If you are coercing from finite-precision rings, `CoerceAndLift(R,f)` may be appropriate.

`BaseRing(f): RngUPolXPadElt -> Rng`

`CoefficientRing(f): RngUPolXPadElt -> Rng`

Returns the coefficient ring of the polynomial f over an exact p-adic ring or field.

`CanChangeRing(f, R): RngUPolXPadElt, Rng -> BoolElt, RngUPolXPadElt`

`ChangeRing(f, R): RngUPolXPadElt, Rng -> RngUPolXPadElt`

Given a polynomial f over an exact p-adic ring or field and a ring R, return whether the coefficients of f can be coerced into R and the polynomial with those coefficients in R if so.

`Degree(f): RngUPolXPadElt -> RngIntElt`

Given a polynomial f returns the true degree of f. That is, the corresponding leading coefficient is known to be non-zero.

`WeakDegree(f): RngUPolXPadElt -> RngIntElt`

Given a polynomial f over an exact p-adic ring or field, returns an upper bound on the degree of f. That is, the corresponding leading coefficient may be weakly zero.

`Coefficient(f, i): RngUPolXPadElt, RngIntElt -> RngElt`

`Coefficients(f): RngUPolXPadElt -> SeqEnum`

Given a polynomial f return the ith coefficient and all coefficients of f respectively.

`ExactPolynomial(f): RngUPolXPadElt -> RngUPolXPadElt`

`ExactPolynomial(f): RngUPolElt -> RngUPolXPadElt`

Given a polynomial f of type `RngUPolElt` over an exact p-adic ring or field, convert f into a polynomial of type `RngUPolXPadElt`.

`Evaluate(f, x): RngUPolXPadElt, Any -> RngAnyXPadElt`

Given a polynomial f over an exact p-adic ring or field and an element x coercible into the coefficient ring of f return the value of f when the variable is replaced by x.

`Derivative(f, m): RngUPolXPadElt, RngIntElt -> RngUPolXPadElt`

`Derivative(f): RngUPolXPadElt -> RngUPolXPadElt`

Given a polynomial f over an exact p-adic ring or field and an integer m, return the mth derivative of f or the first derivative if m is not specified.

`Discriminant(f): RngUPolXPadElt -> RngAnyXPadElt`

Given a polynomial f over an exact p-adic ring or field, return the discriminant, the product of the differences between the roots of f.

`Resultant(f, g): RngUPolXPadElt, RngUPolXPadElt -> RngAnyXPadElt`

Given polynomials f and g over an exact p-adic ring or field, return their resultant.

`IsInertial(f): RngUPolXPadElt -> BoolElt`

`IsEisenstein(f): RngUPolXPadElt -> BoolElt`

Given a polynomial f over an exact p-adic ring or field, return whether the polynomial is inertial or Eisenstein, respectively.

`IsWeaklyZero(f): StrAnyXPadElt -> BoolElt`

`IsWeaklyEqual(f, g): StrAnyXPadElt, StrAnyXPadElt -> BoolElt`

Returns `true` if the exact p-adic polynomial f, respectively $f-g$, is zero up to its current precision. If f, respectively $f-g$, is zero then it is weakly zero, but not necessarily the other way around.

`IsDefinitelyZero(f): StrAnyXPadElt -> BoolElt`

`IsDefinitelyEqual(f, g): StrAnyXPadElt, StrAnyXPadElt -> BoolElt`

Returns `true` if the exact p-adic polynomial f is definitely zero, respectively, f and g are identical or both definitely zero.

`CoerceAndLift(S, x): StrAnyXPad, Any -> StrAnyXPadElt`

Coerce x into the exact p-adic structure S, and ensure its precision is unbounded. If x is a finite-precision object, then ordinary coercion `S!x` will have capped precision whereas `CoerceAndLift(S, x)` will "lift" this to have unbounded precision

Arithmetic

`x * y: RngUPolXPadElt, RngUPolXPadElt -> RngUPolXPadElt`

`- y: RngUPolXPadElt -> RngUPolXPadElt`

`x + y: RngUPolXPadElt, RngUPolXPadElt -> RngUPolXPadElt`

`x - y: RngUPolXPadElt, RngUPolXPadElt -> RngUPolXPadElt`

`x / c: RngUPolXPadElt, RngElt -> RngUPolXPadElt`

`x ^ n: RngUPolXPadElt, RngIntElt -> RngUPolXPadElt`

`Quotrem(x, y): RngUPolXPadElt, RngUPolXPadElt -> RngUPolXPadElt`

`x div y: RngUPolXPadElt, RngUPolXPadElt -> RngUPolXPadElt`

`x mod y: RngUPolXPadElt, RngUPolXPadElt -> RngUPolXPadElt`

Example: Arithmetic with Exact p-adic Polynomials (ex-88ff30)

```
> Qp := pAdicField(11 : Exact);
> f := Polynomial(Qp, [3241, 245, 134, 1345, 125, 1542]);
> f;
(1542 + O(11^20))*$.1^5 + (125 + O(11^20))*$.1^4 + (1345 + O(11^20))*$.1^3 +
    (134 + O(11^20))*$.1^2 + (245 + O(11^20))*$.1 + 3241 + O(11^20)
> f := ExactPolynomial(f); f;
(1542 + O(11^20))*$.1^5 + (125 + O(11^20))*$.1^4 + (1345 + O(11^20))*$.1^3 +
    (134 + O(11^20))*$.1^2 + (245 + O(11^20))*$.1 + 3241 + O(11^20)
> BaseRing(f);
11-adic field (exact)
> CanChangeRing(f, Integers(Qp));
true (1542 + O(11^20))*$.1^5 + (125 + O(11^20))*$.1^4 + (1345 + O(11^20))*$.1^3
    + (134 + O(11^20))*$.1^2 + (245 + O(11^20))*$.1 + 3241 + O(11^20)
> Degree(f);
5
> WeakDegree(ExactPolynomial(f));
5
> Coefficient(f, 2);
134 + O(11^20)
> Evaluate(f, Qp.1);
6632 + O(11^20)
> Derivative(f, 2);
(30840 + O(11^20))*$.1^3 + (1500 + O(11^20))*$.1^2 + (8070 + O(11^20))*$.1 + 268
    + O(11^20)
> Discriminant(f);
-77083727534419762054 + O(11^20)
> Resultant(f, Derivative(f));
213641244987848541309 + O(11^20)
> IsInertial(f);
true
> IsEisenstein(f);
false
> IsWeaklyZero(PolynomialRing(Qp : Exact)!0);
true
> IsDefinitelyZero(PolynomialRing(Qp : Exact)!0);
true
> IsWeaklyZero(f-f);
true
> IsDefinitelyZero(f-f);
false
> IsDefinitelyEqual(f, f);
true
```

Factorization and Roots

The functionality in this section is related to taking roots and computing factorizations of polynomials. For some of these intrinsics there are restrictions on the input polynomial regarding multiple factors. Since no finite precision p-adic algorithm can provbably identify equal roots multiple factors are not checked for and some intrinsics may run forever on input having multiple factors.

`NewtonPolygon(f): RngUPolXPadElt[RngXPad] -> NwtnPgon`

`NewtonPolygon(f): RngUPolXPadElt[FldXPad] -> NwtnPgon`

Returns the Newton polygon of the polynomial f over an exact p-adic ring or field.

`Roots(f, R): RngUPolElt, FldXPad -> SeqEnum`

`Roots(f, R): RngUPolElt, RngXPad -> SeqEnum`

`Roots(f, R): RngUPolXPadElt, RngXPad -> SeqEnum`

`Roots(f, R): RngUPolXPadElt, FldXPad -> SeqEnum`

`Roots(f): RngUPolXPadElt[FldXPad] -> SeqEnum`

`Roots(f): RngUPolXPadElt[RngXPad] -> SeqEnum`

```
Max: Infty                    Default: Infinity()
```

Given a polynomial f over an exact p-adic ring or field, return the roots of f in the exact p-adic field or ring R or the base ring of f if R is not specified. The polynomial f must have no repeated roots (without further assumptions on the structure of the polynomial, no finite-precision p-adic algorithm can provably identify a set of equal roots).

`HasRoot(f): RngUPolXPadElt[RngXPad] -> BoolElt, RngXPadElt`

`HasRoot(f): RngUPolXPadElt[FldXPad] -> BoolElt, FldXPadElt`

`HasRoot(f): RngUPolElt[RngXPad] -> BoolElt, RngXPadElt`

`HasRoot(f): RngUPolElt[FldXPad] -> BoolElt, FldXPadElt`

`HasRoot(f, R): RngUPolElt, FldXPad -> BoolElt, FldXPadElt`

`HasRoot(f, R): RngUPolElt, RngXPad -> BoolElt, RngXPadElt`

`HasRoot(f, R): RngUPolXPadElt, FldXPad -> BoolElt, FldXPadElt`

`HasRoot(f, R): RngUPolXPadElt, RngXPad -> BoolElt, RngXPadElt`

Given a polynomial f over an exact p-adic ring or field, return whether f has a root in the exact p-adic ring or field R or the base ring of f if R is not specified. If f does have such a root then also return one root. The polynomial f must have no repeated roots (see *Roots* above).

`Factorization(f, R): RngUPolXPadElt, FldXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorisation(f, R): RngUPolXPadElt, FldXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorisation(f, R): RngUPolXPadElt, RngXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorization(f, R): RngUPolXPadElt, RngXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorization(f, R): RngUPolElt, RngXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorisation(f, R): RngUPolElt, RngXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorisation(f, R): RngUPolElt, FldXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorization(f, R): RngUPolElt, FldXPad -> SeqEnum, RngXPadElt, SeqEnum`

`Factorization(f): RngUPolXPadElt[RngXPad] -> SeqEnum, RngXPadElt, SeqEnum`

`Factorization(f): RngUPolXPadElt[FldXPad] -> SeqEnum, FldXPadElt, SeqEnum`

`Factorisation(f): RngUPolXPadElt[RngXPad] -> SeqEnum, RngXPadElt, SeqEnum`

`Factorisation(f): RngUPolXPadElt[FldXPad] -> SeqEnum, FldXPadElt, SeqEnum`

```
Certificates : BoolElt                    Default: false
Extensions   : BoolElt                    Default: false
Max          : Infty                      Default: Infinity()
DegreeDivides: RngIntElt                  Default: 0
DegreeGe     : RngIntElt                  Default: 1
```

Given a polynomial f over an exact p-adic ring or field, return the factorization of f over R if given or the coefficient ring of f otherwise. The polynomial f must have no repeated factors (for similar reasons to *Roots* above). A scaling factor is also returned, so that the product of the factors equals f multiplied by the scaling factor.

If the parameter `Certificates` is set to `true`, a corresponding sequence of certificates for each factor is also returned. If the parameter `Extensions` is set to `true`, (which implies `Certificates`) then each certificate also includes the extension defined by the factor.

```
IsIrreducible(f): RngUPolXPadElt[RngXPad] -> BoolElt, Rec
IsIrreducible(f): RngUPolXPadElt[FldXPad] -> BoolElt, Rec
IsIrreducible(f): RngUPolElt[RngXPad] -> BoolElt, Rec
IsIrreducible(f): RngUPolElt[FldXPad] -> BoolElt, Rec
```

```
Extension  : BoolElt                    Default: false
Certificate: BoolElt                    Default: false
```

Given a polynomial f over an exact p-adic ring or field, return whether f is irreducible. It must have at least one non-repeated factor (for similar reasons to *Roots* above). If the parameters `Extension` or `Certificate` are given, the second return value is as in *Factorization*.

```
IsHenselLiftable(f, x): RngUPolElt, FldXPadElt -> BoolElt, FldXPadElt
IsHenselLiftable(f, x): RngUPolElt, RngXPadElt -> BoolElt, RngXPadElt
IsHenselLiftable(f, x): RngUPolXPadElt, FldXPadElt -> BoolElt, FldXPadElt
IsHenselLiftable(f, x): RngUPolXPadElt, RngXPadElt -> BoolElt, RngXPadElt
```

Given a polynomial f over an exact p-adic ring or field and an element x of an exact p-adic ring or field return whether x is close enough to a root of f to be Hensel lifted to that root. If so, returns the lifted root.

```
RamificationResidualPolynomial(f, face): RngUPolElt[FldXPad], NwtnPgonFace ->
                                RngUPolElt
RamificationResidualPolynomial(f, face): RngUPolElt[RngXPad], NwtnPgonFace ->
                                RngUPolElt
RamificationResidualPolynomial(f, face): RngUPolXPadElt[FldXPad], NwtnPgonFace ->
                                RngUPolElt
RamificationResidualPolynomial(f, face): RngUPolXPadElt[RngXPad], NwtnPgonFace ->
                                RngUPolElt
```

Given a polynomial f over an exact p-adic ring or field, return the residual polynomial associated to the given $face$ of the ramifcation polygon of f.

```
RamificationResidualPolynomials(f): RngUPolElt[FldXPad] -> SeqEnum, NwtnPgon
RamificationResidualPolynomials(f): RngUPolElt[RngXPad] -> SeqEnum, NwtnPgon
RamificationResidualPolynomials(f): RngUPolXPadElt[FldXPad] -> SeqEnum, NwtnPgon
RamificationResidualPolynomials(f): RngUPolXPadElt[RngXPad] -> SeqEnum, NwtnPgon
```

Given an inertial or Eisenstein polynomial f over an exact p-adic ring or field, return the residual polynomials assiociated to each face of the ramification polygon of f, and the ramification polygon itself.

Example: Factorization over Exact p-adic Fields (ex-95a032)

```
> Qp := pAdicField(17 : Exact);
> P<x> := PolynomialRing(Qp : Exact);
> f := (x - 14)*(x - Qp.1)*(x - Qp.1 - 1);
> NewtonPolygon(f);
Newton Polygon with vertices {(0, 0), (3, 0)} and defining points {(0, 0), (3,
0)}
```

(continues on next page)

(continued from previous page)

```
> HasRoot(f);
true 14 + O(17^20)
> Roots(f);
[ <14 + O(17^20), 1>, <2 + O(17^20), 1>, <1 + O(17^20), 1> ]
> Factorization(f);
[ <(1 + O(17^20))*x - 14 + O(17^20), 1>, <(1 + O(17^20))*x - 2 + O(17^20), 1>,
<(1 + O(17^20))*x - 1 + O(17^20), 1> ]
1 + O(17^20)
> IsIrreducible(f);
false
> IsHenselLiftable(f, Qp!14);
true 14 + O(17^20)
> IsHenselLiftable(f, Qp!10);
false
> RamificationResidualPolynomials(x^3 + 3*17*x^2 + 2*17^2*x + 5*17);
[
    $.1^2 + 3*$.1 + 3
]
Newton Polygon with vertices {(1, 2), (3, 0)} and defining points {(1, 2), (2,
    1), (3, 0)}
```

49.17 Bibliography

- [C+05] H. Cohen and others. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and Its Applications. CRC Pr Llc, 2005.
- [Dor21] Christopher Doris. Exact p-adic computation in Magma. *Journal of Symbolic Computation*, 104:476493, May 2021. URL: http://dx.doi.org/10.1016/j.jsc.2020.08.005, doi:10.1016/j.jsc.2020.08.005.
- [JR06] J. W. Jones and D. P. Roberts. A database of local fields. *J. Symb. Comput.*, 41(1):80–97, 2006.
- [PR01] S. Pauli and X.-F. Roblot. On the computation of all extensions of a p-adic field of a given degree. *Math. Comp.*, 70(236):1641–1659, 2001.
- [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.

Chapter 50

General p-Adic Extensions

50.1 Introduction

The local fields described in this chapter are extensions of any p-adic local field in Magma by any irreducible polynomial over that field. The polynomial defining the extension is not required to be inertial or eisenstein, in contrast to the (older) extensions of p-adic fields. This allows 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.

Only fields are implemented in this way – no construction of a ring of integers is provided (although `IntegralBasis` gives a basis for it as a module over the base ring).

These local fields have type `RngLocA` with elements of type `RngLocAElt`, while the local fields described in the previous chapter *p-Adic Rings and Their Extensions* have type `FldPad`.

These fields (of type `RngLocA`) can be converted to the other representation (type `FldPad`) using `RamifiedRepresentation`, which returns an isomorphism between them. This can be used for calculations not supported for `RngLocA`.

50.2 Constructions

The main construction is to extend a local field (of either type) by adjoining the root of a univariate polynomial. Another construction is of the subfield generated by given elements.

`LocalField(L, f): FldPad, RngUPolElt -> RngLocA`

`LocalField(L, f): RngLocA, RngUPolElt -> RngLocA`

Construct a local field F as an extension of the local field L by the polynomial f over L.

Example: One-Step Extension from a Two-Step Field `(ex-827f9a)`

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 >: RngLocA, RngLocAElt, ..., RngLocAElt -> RngLocA

sub< L | S>: RngLocA, SeqEnum -> RngLocA

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: Subfield Generated by an Element (ex-0823ac)

```
> 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)
```

50.3 Operations with Fields

BaseRing(L): RngLocA -> Rng

CoefficientRing(L): RngLocA -> Rng

Return the coefficient field of the local field L. This is the field which was extended to construct L.

DefiningPolynomial(L): RngLocA -> RngUPolElt

Return the polynomial used to define the local field L as an extension of its coefficient field.

Degree(L): RngLocA -> RngIntElt

Return the degree of the local field L, that is, the degree of its defining polynomial.

Degree(L, R): RngLocA, Rng -> RngIntElt

Return the degree of L as an extension of R where R is some coefficient ring of L.

InertiaDegree(L): RngLocA -> RngIntElt

RamificationDegree(L): RngLocA -> RngIntElt

RamificationIndex(L): RngLocA -> RngIntElt

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): RngLocA -> RngIntElt

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): RngLocA -> RngElt`

Return the prime of the local field L. This is the same as the prime of the coefficient field of L.

Example: Coefficient Ring and Defining Polynomial (ex-ef970a)

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): RngLocA -> RngUPolRes`

Return the polynomial quotient ring which is isomorphic to the local field L and is used to represent L.

`RamifiedRepresentation(L): RngLocA -> FldPad, Map`

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: Two-Step Representation of a Local Field (ex-a41b45)

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 +
    41709273235569453711334820170343703378866 3*$.1^2 +
    58619460221835676233637925289507569853713 7*$.1 -
    13009411322463399816688753375590121409659 7
 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 + (10312136871155901743985045546 31*7^-1 + O(7^35))*$.1^2 +
    O(7^37)*$.1 + 13128487736606729510635017356 8*7 + O(7^36)
```

`AssignNames(~L, S): RngLocA, SeqEnum[MonStgElt]`

Assign the name in the sequence S to the generator of the extension defining the local field L.

`Name(L, i): RngLocA, RngIntElt -> RngLocAElt`

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): RngLocA -> RngElt`

Return the discriminant of the local field L.

`ResidueClassField(L): RngLocA -> Rng, Map`

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): RngLocA, Map -> RngLocA, Map, Map`

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.

50.3.1 Predicates on Fields

`IsRamified(L): RngLocA -> BoolElt`

Return whether the local field L has a non trivial ramified subfield, that is, the ramification degree of L is greater than 1.

`IsTamelyRamified(L): RngLocA -> BoolElt`

`IsWildlyRamified(L): RngLocA -> BoolElt`

Return whether the local field L is tamely or wildly ramified.

`IsTotallyRamified(L): RngLocA -> BoolElt`

Return whether the local field L is a totally ramified extension, that is, L has a trivial inertial subfield.

`IsUnramified(L): RngLocA -> BoolElt`

Return whether the local field L is equal to its inertial subfield.

50.4 Maximal Order

`IntegralBasis(L): RngLocA -> SeqEnum`

Return a basis for the maximal order of the local field L.

`IsIntegral(a): RngLocAElt -> BoolElt, SeqEnum`

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: Integral Basis of a Local Field (ex-9bf4a2)

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) ]
```

50.5 Homomorphisms

`hom< L -> R | a >: RngLocA, Rng, RngElt -> Map`

`hom< L -> R | cfm, a >: RngLocA, Rng, Map, RngElt -> Map`

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.

50.6 Automorphisms and Galois Theory

`FrobeniusAutomorphism(L): RngLocA -> Map`

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): RngLocA -> Grp, Map`

Return the automorphism group of the local field L and a map from the group to the parent of automorphisms of L.

`DecompositionGroup(L): RngLocA -> GrpPerm`

`InertiaGroup(L): RngLocA -> GrpPerm`

`RamificationGroup(L, i): RngLocA, RngIntElt -> GrpPerm`

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): RngLocA, GrpPerm -> RngLocA`

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: Automorphisms and Inertia Groups (ex-180d7f)

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)*$.1^4 +
    (10359052517489881294588814641 23*7^-1 + O(7^35))*$.1^3 + O(7^36)*$.1^2 -
    (10094439077108649085019837355 01 + O(7^36))*$.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)
```

50.7 Elements Operations

`L ! r: RngLocA, Any -> RngLocAElt`

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: RngLocA, RngIntElt -> RngLocAElt`
Return the generator of the local field L. The only valid input for i is 1.

`InertialElement(L): RngLocA -> RngLocAElt`
Return a generator for the inertial subfield of the local field L.

`UniformizingElement(L): RngLocA -> RngLocAElt`
Return an element of the local field L of valuation 1.

50.7.1 Arithmetic

`a * b: RngLocAElt, RngLocAElt -> RngLocAElt`

`a + b: RngLocAElt, RngLocAElt -> RngLocAElt`

`a - b: RngLocAElt, RngLocAElt -> RngLocAElt`

`- a: RngLocAElt -> RngLocAElt`

`a ^ n: RngLocAElt, RngIntElt -> RngLocAElt`

`a / b: RngLocAElt, RngLocAElt -> RngLocAElt`

50.7.2 Predicates on Elements

`a eq b: RngLocAElt, RngLocAElt -> BoolElt`
Return whether the local field elements a and b are considered equal.

`IsOne(a): RngLocAElt -> BoolElt`
`IsMinusOne(a): RngLocAElt -> BoolElt`
Return whether the local field element a is known to be 1 or -1 to the precision of the field.

`IsWeaklyZero(a): RngLocAElt -> BoolElt`
Return whether the local field element a is not known to be non zero.

`IsZero(a): RngLocAElt -> BoolElt`
Return whether the local field element a is known to be zero.

50.7.3 Other Operations on Elements

`Valuation(a): RngLocAElt -> RngExtReElt`
The valuation of the element a in a local field.

`RelativePrecision(a): RngLocAElt -> RngExtReElt`
The relative precision of the element a in a local field.

`Eltseq(a): RngLocAElt -> SeqEnum`
Return the coefficients of powers of the generator of the parent of a in a.

`RepresentationMatrix(a): RngLocAElt -> AlgMatElt`
The representation matrix of the element a of a local field.

`Norm(a): RngLocAElt -> RngElt`
`Norm(a, F): RngLocAElt, Rng -> RngElt`
`Trace(a): RngLocAElt -> RngElt`
`Trace(a, F): RngLocAElt, Rng -> RngElt`
`MinimalPolynomial(a): RngLocAElt -> RngUPolElt`

`MinimalPolynomial(a, F): RngLocAElt -> RngUPolElt`

Return the norm, trace, or minimal polynomial of a. If a coefficient field F of the parent L of a is given then the norm, trace, or minimal polynomial will be that of a as an element of L represented as an extension of F.

Example: Uniformizing and Inertial Elements (ex-0bf192)

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) ]
```

50.8 Polynomial Factorization

Polynomials over local fields can be factored and their roots computed.

`Factorization(f): RngUPolElt[RngLocA] -> SeqEnum, RngElt, Any`

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): RngUPolElt[RngLocA] -> RngIntElt`

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): RngUPolElt[RngLocA] -> SeqEnum`

`Roots(f, R): RngUPolElt, RngLocA -> SeqEnum`

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: Factorization over a Local Field (ex-579903)

```
> 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 +
```

(continues on next page)

(continued from previous page)

```
        (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)
```

Chapter 51

Power, Laurent, and Puiseux Series

51.1 Introduction

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

51.1.1 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`.

51.1.2 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.

51.1.3 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

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

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

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

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

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

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

51.1.4 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 $v \in \mathbb{Q}$ 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 $p = v + r$.

51.1.5 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.

51.1.6 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!

51.1.7 Polynomials over Series Rings

For a discussion of operations for polynomials over series rings see Chapter *Newton Polygons* and Section *Polynomials over Series Rings*.

51.2 Creation Functions

51.2.1 Creation of Structures

`PowerSeriesRing(R): Rng -> RngSerPow`

`PowerSeriesRing(R, p): Rng, RngIntElt -> RngSerPow`

```
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): Rng -> RngSerLaur`

`LaurentSeriesRing(R, p): Rng, RngIntElt -> RngSerLaur`

```
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): Rng -> RngSerPuis`

`PuiseuxSeriesRing(R, p): Rng, RngIntElt -> RngSerPuis`

```
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: Global and Non-Global Series Rings (ex-f4cf13)

We demonstrate the difference between global and non-global rings. We first create the global power series ring over $\mathbb{Q}$ 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
```

51.2.2 Special Options

`AssertAttribute(S, "DefaultPrecision", n): RngSer, MonStgElt, RngIntElt`

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"): RngSer, MonStgElt -> BoolElt, RngIntElt`

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"]): RngSer, [ MonStgElt ]`

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): RngSer, RngIntElt -> RngSerElt`

`S . 1: RngSer, RngIntElt -> RngSerElt`

`Name(S, 1): RngSer, RngIntElt -> RngSerElt`

`S . 1: RngSer, RngIntElt -> RngSerElt`

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.

51.2.3 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: RngSer, RngInt -> RngSerElt`

`UniformizingElement(R): RngSer -> RngSerElt`

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

`elt< R | v, [ a1, ..., a_d], p >: RngIntElt, SeqEnum, RngIntElt -> RngSerElt`

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: RngSer, SeqEnum -> RngSerElt`

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): RngSerElt -> RngIntElt`

`O(f): RngSerElt -> RngIntElt`

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): RngSer -> RngSerElt`

`One(Q): RngSer -> RngSerElt`

`Identity(Q): RngSer -> RngSerElt`

`Identity(Q): RngSer -> RngSerElt`

`Zero(Q): RngSer -> RngSerElt`

`Zero(Q): RngSer -> RngSerElt`

`Representative(Q): RngSer -> RngSerElt`

`Representative(Q): RngSer -> RngSerElt`

51.3 Structure Operations

51.3.1 Related Structures

`Parent(R): RngSer -> Pow`

`Category(R): RngSer -> Cat`

`BaseRing(R): RngSer -> Rng`

`CoefficientRing(R): RngSer -> Rng`

Return the coefficient ring of the series ring R.

`IntegerRing(R): RngSer -> RngSerPow`

`Integers(R): RngSer -> RngSerPow`

`RingOfIntegers(R): RngSer -> RngSerPow`

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

`FieldOfFractions(R): RngSer -> RngSerLaur`

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

`ChangePrecision(R, r): RngSer, Any -> RngSer`

`ChangePrecision(~R, r): RngSer, Any -> RngSer`

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

`ChangeRing(R, C): RngSer, Rng -> RngSer, Map`

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

`ResidueClassField(R): RngSer -> Rng, Map`

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.

51.3.2 Invariants

`Characteristic(R): RngSer -> RngIntElt`

`Precision(R): RngSer -> ExtReElt`

`GetPrecision(R): RngSer -> ExtReElt`

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.

51.3.3 Ring Predicates and Booleans

`IsCommutative(Q): RngSer -> BoolElt`

`IsUnitary(Q): RngSer -> BoolElt`

`IsFinite(Q): RngSer -> BoolElt`

`IsOrdered(Q): RngSer -> BoolElt`

`IsField(Q): RngSer -> BoolElt`

`IsEuclideanDomain(Q): RngSer -> BoolElt`

`IsPID(Q): RngSer -> BoolElt`

`IsUFD(Q): RngSer -> BoolElt`

`IsDivisionRing(Q): RngSer -> BoolElt`

`IsEuclideanRing(Q): RngSer -> BoolElt`

`IsPrincipalIdealRing(Q): RngSer -> BoolElt`

`IsDomain(Q): RngSer -> BoolElt`

`R eq S: RngSer, RngSer -> BoolElt`

`R ne S: RngSer, RngSer -> BoolElt`

51.4 Basic Element Operations

51.4.1 Parent and Category

`Parent(r): RngSerElt -> RngSer`

`Category(r): RngSerElt -> Cat`

51.4.2 Arithmetic Operators

`+ b: RngSerElt -> RngSerElt`

`- b: RngSerElt -> RngSerElt`

`a + b: RngSerElt, RngSerElt -> RngSerElt`

`a - b: RngSerElt, RngSerElt -> RngSerElt`

`a * b: RngSerElt, RngSerElt -> RngSerElt`

`a ^ k: RngSerElt, RngIntElt -> RngSerElt`

`a div b: RngSerPowElt, RngSerPowElt -> RngSerPowElt`

`a / b: RngSerElt, RngSerElt -> RngSerElt`

51.4.3 Equality and Membership

`a eq b: RngSerElt, RngSerElt -> BoolElt`

`a ne b: RngSerElt, RngSerElt -> BoolElt`

`a in R: RngSerElt, RngSer -> BoolElt`

`a notin R: RngSerElt, RngSer -> BoolElt`

51.4.4 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): RngSerElt -> BoolElt`
`IsOne(a): RngSerElt -> BoolElt`
`IsMinusOne(a): RngSerElt -> BoolElt`
`IsNilpotent(x): RngSerElt -> BoolElt`
`IsIdempotent(x): RngSerElt -> BoolElt`
`IsUnit(a): RngSerElt -> BoolElt`
`IsZeroDivisor(x): RngSerElt -> BoolElt`
`IsRegular(x): RngSerElt -> BoolElt`
`IsIrreducible(x): RngSerElt -> BoolElt`
`IsPrime(x): RngSerElt -> BoolElt`
`IsWeaklyZero(f): RngSerElt -> BoolElt`

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): RngSerElt, RngSerElt -> BoolElt`

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): RngSerElt, RngSerElt -> BoolElt`

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.

51.4.5 Precision

`AbsolutePrecision(f): RngSerElt -> RngIntElt`

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): RngSerElt -> RngIntElt`

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): RngSerElt, RngIntElt -> RngSerElt`

`ChangePrecision(f, r): RngSerElt, Infty -> RngSerElt`

`ChangePrecision(~f, r): RngSerElt, RngIntElt`

`ChangePrecision(~f, r): RngSerElt, Infty`

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

51.4.6 Coefficients and Degree

`Coefficients(f): RngSerElt -> [ RngElt ], RngIntElt, RngIntElt`

`ElementToSequence(f): RngSerElt -> [ RngElt ], RngIntElt, RngIntElt`

`Eltseq(f): RngSerElt -> [ RngElt ], RngIntElt, RngIntElt`

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): RngSerElt, RngElt -> RngElt`

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): RngSerElt -> RngElt`

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): RngSerElt -> RngElt`

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): RngSerElt -> RngSerElt`

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

`ExponentDenominator(f): RngMSerElt -> RngElt`

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): RngSerElt -> RngIntElt`

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): RngSerElt -> RngIntElt`

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): RngSerElt -> RngIntElt`

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

51.4.7 Evaluation and Derivative

`Derivative(f): RngSerElt -> RngSerElt`

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): RngSerElt, RngIntElt -> RngSerElt`

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): RngSerElt -> RngSerElt`

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): RngSerElt, RngElt -> RngElt`

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): RngSerElt -> RngSerElt`

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.

51.4.8 Square Root

`SquareRoot(f): RngSerElt -> RngSerElt`

`Sqrt(f): RngSerElt -> RngSerElt`

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

51.4.9 Composition and Reversion

`Composition(f, g): RngSerElt, RngSerElt -> RngSerElt`

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): RngSerElt -> RngSerElt`

`Reverse(f): RngSerElt -> RngSerElt`

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): RngSerElt, RngSerElt -> RngSerElt`

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: Composition and Reversion of Series (ex-cad173)

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))
```

51.5 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.

51.5.1 Exponential and Logarithmic Functions

`Exp(f): RngSerElt -> RngSerElt`

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

`Log(f): RngSerElt -> RngSerElt`

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

Example: Bernoulli Numbers from Generating Functions (ex-739f49)

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 $O(x^{20})$, 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 +
```

(continues on next page)

(continued from previous page)

```
    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 -
    3617/1067062842880000*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);
-1659638064056855722985212308807713420665866430280667189235265099315533164122O\\
960084014956088135770921465025323942809207851857992860213463783252745409096420\\
932509953165466735675485979034817619983727209844291081908145597829674980159889\\
976244240633746601120703300698329029710482600069717866917229113749797632930033\\
559794717838407415772796504419464932337498642714226081743688706971990010734262\\
076881238322867559275748219588404488023034528296023051638858467185173202483888\\
794342720837413737644410765563213220043477396887812891242952336301344808165757\\
942109887803692579439427973561487863524556256869403384306433922049078300720480\\
361757680714198044230522015775475287075315668886299978958150756677417180004362\\
98145439661364661232701978414174049983546l/8365830
> n;
500
```

51.5.2 Trigonometric Functions and their Inverses

Sin(f): RngSerElt -> RngSerElt

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): RngSerElt -> RngSerElt

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): RngSerElt -> RngSerElt

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): RngSerElt -> RngSerElt

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

Arcsin(f): RngSerElt -> RngSerElt

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

Arccos(f): RngSerElt -> RngSerElt

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

`Arctan(f): RngSerElt -> RngSerElt`

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

51.5.3 Hyperbolic Functions and their Inverses

`Sinh(f): RngSerElt -> RngSerElt`

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

`Cosh(f): RngSerElt -> RngSerElt`

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

`Tanh(f): RngSerElt -> RngSerElt`

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

`Argsinh(f): RngSerElt -> RngSerElt`

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

`Argcosh(f): RngSerElt -> RngSerElt`

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

`Argtanh(f): RngSerElt -> RngSerElt`

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

51.6 The Hypergeometric Series

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

`HypergeometricSeries(a, b, c, z): RngElt, RngElt, RngElt, RngElt -> RngElt`

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

51.7 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): RngUPolElt[RngSer], SeqEnum[RngUPolElt] -> [RngUPolElt]`

`HenselLift(f, L): RngUPolElt[RngSerExt], SeqEnum[RngUPolElt] -> [RngUPolElt]`

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): RngUPolElt[RngSerPow[FldFin]] -> [ < RngUPolElt[RngSerPow],
              RngIntElt > ], RngSerPowElt
```

```
Factorization(f): RngUPolElt[RngSerLaur[FldFin]] -> [ < RngUPolElt[RngSerLaur],
              RngIntElt > ], RngSerLaurElt
```

`Factorization(f): RngUPolElt[RngSerExt] -> [ < RngUPolElt[RngSerExt], RngIntElt > ], RngSerExtElt`

```
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: Factorization over Series Rings (ex-32baab)

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
```

(continues on next page)

(continued from previous page)

```
        + 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)>
]
```

51.8 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.

51.8.1 Constructions of Extensions

UnramifiedExtension(R, f): RngSerPow[FldFin], RngUPolElt -> RngSerExt

UnramifiedExtension(R, f): RngSerLaur[FldFin], RngUPolElt -> RngSerExt

UnramifiedExtension(R, f): RngSerExt, RngUPolElt -> RngSerExt

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): RngSerPow[FldFin], RngUPolElt -> RngSerExt

TotallyRamifiedExtension(R, f): RngSerLaur[FldFin], RngUPolElt -> RngSerExt

TotallyRamifiedExtension(R, f): RngSerExt, RngUPolElt -> RngSerExt

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): RngSerExt, RngIntElt -> RngSerExt

ChangePrecision(~E, r): RngSerExt, RngIntElt

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): RngSerExt -> RngSerExt

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

Example: Two-Step Extension of a Series Ring (ex-0fa010)

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
```

(continues on next page)

(continued from previous page)

```
> 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)
```

51.8.2 Operations on Extensions

Precision(E): RngSerExt -> RngIntElt

GetPrecision(E): RngSerExt -> RngIntElt

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

CoefficientRing(E): RngSerExt -> Rng

BaseRing(E): RngSerExt -> Rng

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

DefiningPolynomial(E): RngSerExt -> RngUPolElt

The polynomial used to define the extension E.

InertiaDegree(E): RngSerExt -> RngIntElt

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

RamificationIndex(E): RngSerExt -> RngIntElt

RamificationDegree(E): RngSerExt -> RngIntElt

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

ResidueClassField(E): RngSerExt -> FldFin

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

UniformizingElement(E): RngSerExt -> RngSerExtElt

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

IntegerRing(E): RngSerExt -> RngSerExt

Integers(E): RngSerExt -> RngSerExt

RingOfIntegers(E): RngSerExt -> RngSerExt

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: RngSerExt, RngSerExt -> BoolElt

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

`E . i: RngSerExt, RngIntElt -> RngSerExtElt`

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): RngSerExt, [ MonStgElt ]`

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: Operations on Series Ring Extensions (ex-b88c0a)

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)
```

51.8.3 Elements of Extensions

`x * y: RngSerExtElt, RngSerExtElt -> RngSerExtElt`

`x + y: RngSerExtElt, RngSerExtElt -> RngSerExtElt`

`x - y: RngSerExtElt, RngSerExtElt -> RngSerExtElt`

`- x: RngSerExtElt -> RngSerExtElt`

`x ^ n: RngSerExtElt, RngIntElt -> RngSerExtElt`

`x div y: RngSerExtElt, RngSerExtElt -> RngSerExtElt`

`x / y: RngSerExtElt, RngSerExtElt -> RngSerExtElt`

`x eq y: RngSerExtElt, RngSerExtElt -> BoolElt`

`IsZero(e): RngSerExtElt -> BoolElt`

`IsOne(e): RngSerExtElt -> BoolElt`

`IsMinusOne(e): RngSerExtElt -> BoolElt`

`IsUnit(e): RngSerExtElt -> BoolElt`

`Valuation(e): RngSerExtElt -> RngIntElt`

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): RngSerExtElt -> RngIntElt`

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): RngSerExtElt -> RngIntElt`

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): RngSerExtElt -> [ RngElt ]`

`Eltseq(e): RngSerExtElt -> [ RngElt ]`

`ElementToSequence(e): RngSerExtElt -> [ RngElt ]`

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: Arithmetic in Series Ring Extensions (ex-77b403)

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
```

(continues on next page)

(continued from previous page)

```
> 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) ]
```

51.8.4 Optimized Representation

`OptimizedRepresentation(E): RngSerExt -> RngSer, Map`

`OptimisedRepresentation(E): RngSerExt -> RngSer, Map`

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: Optimized Representation of an Extension `(ex-95694d)`

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
```

51.9 Bibliography

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

Chapter 52

Lazy Power Series Rings

52.1 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
```

52.2 Creation of Lazy Series Rings

Both univariate and multivariate lazy series rings can be created.

`LazyPowerSeriesRing(C, n): Rng, RngIntElt -> RngPowLaz`

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): RngPowLaz, Rng -> RngPowLaz, Map

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 a series s in L to a series whose coefficients are those of s coerced into C.

Example: Creating Lazy Power Series Rings (ex-085add)

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
```

52.3 Functions on Lazy Series Rings

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

R . i: RngPowLaz, RngIntElt -> RngPowLazElt

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

AssignNames(~R, S): RngPowLaz, [MonStgElt]

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): RngPowLaz -> Rng

CoefficientRing(R): RngPowLaz -> Rng

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): RngPowLaz -> RngIntElt

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

R1 eq R2: RngPowLaz, RngPowLaz -> BoolElt

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: Names and Properties of a Lazy Series Ring (ex-aee740)

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
```

52.4 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.

52.4.1 Creation of Finite Lazy Series

`R ! c: RngPowLaz, RngElt -> RngPowLazElt`

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: RngPowLaz, RngPowLazElt -> RngPowLazElt`

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: RngPowLaz, [RngElt] -> RngPowLazElt`

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): RngPowLaz, RngMPolElt -> RngPowLazElt`

`LazySeries(R, f): RngPowLaz, FldFunRatElt -> RngPowLazElt`

`LazySeries(R, f): RngPowLaz, RngUPolElt -> RngPowLazElt`

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: Creating Finite Lazy Series (ex-60e042)

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
```

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>: RngPowLaz, Map -> RngPowLazElt`

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: Univariate Lazy Series from a Map (ex-6e6661)

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: Multivariate Lazy Series from a Map (ex-53315d)

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
```

52.4.2 Arithmetic with Lazy Series

All the usual arithmetic operations are possible for lazy series.

`s + t: RngPowLazElt, RngPowLazElt -> RngPowLazElt`
: The sum of the two lazy series s and t.

`- s: RngPowLazElt -> RngPowLazElt`
: The negation of the lazy series s.

`s - t: RngPowLazElt, RngPowLazElt -> RngPowLazElt`
: The difference between lazy series s and t.

`s * t: RngPowLazElt, RngPowLazElt -> RngPowLazElt`
: The product of the lazy series s and t.

`s + r: RngPowLazElt, RngElt -> RngPowLazElt`

`r + s: RngElt, RngPowLazElt -> RngPowLazElt`

The sum of the lazy series s and the element r of the coefficient ring of the parent ring of s.

`c * s: RngElt, RngPowLazElt -> RngPowLazElt`

`s * c: RngPowLazElt, RngElt -> RngPowLazElt`

The product of the lazy series s and the element c of the coefficient ring of the parent ring of s.

`s * n: RngPowLazElt, [RngIntElt] -> RngPowLazElt`

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: RngPowLazElt, RngIntElt -> RngPowLazElt`

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: Arithmetic with Lazy Series (ex-75ff3a)

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
```

52.4.3 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): RngPowLazElt, RngIntElt -> RngElt`

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): RngPowLazElt, SeqEnum -> RngElt`

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): RngPowLazElt, RngIntElt -> [RngElt]`

`Coefficients(s, l, n): RngPowLazElt, RngIntElt, RngIntElt -> [RngElt]`

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): RngPowLazElt -> Any`

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): RngPowLazElt, RngIntElt`

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): RngPowLazElt, RngIntElt, RngIntElt`

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): RngPowLazElt -> RngElt`

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): RngPowLazElt -> RngPowLazElt`

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: Coefficients by the Spiral Algorithm (ex-eaeab7)

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 -
```

(continues on next page)

(continued from previous page)

```
    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): RngPowLazElt, [RngIntElt] -> SeqEnum

CoefficientsNonSpiral(s, n): RngPowLazElt, RngIntElt -> SeqEnum

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): RngPowLazElt, [RngIntElt], [RngIntElt] -> RngIntElt

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: Coefficients by the Non-Spiral Algorithm (ex-31429a)

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, 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.

52.4.4 Predicates on Lazy Series

`s eq t: RngPowLazElt, RngPowLazElt -> BoolElt`

Return `true` if the lazy series s and t are exactly the same series.

`IsZero(s): RngPowLazElt -> BoolElt`

Return `true` if the lazy series s was created as the zero series.

`IsOne(s): RngPowLazElt -> BoolElt`

Return `true` if the lazy series s was created as the one series.

`IsMinusOne(s): RngPowLazElt -> BoolElt`

Return `true` if the lazy series s was created as the minus one series.

`IsUnit(s): RngPowLazElt -> BoolElt`

Return `true` if the lazy series s is a unit.

`IsWeaklyZero(s, n): RngPowLazElt, RngIntElt -> BoolElt`

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): RngPowLazElt, RngPowLazElt, RngIntElt -> BoolElt`

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`.

52.4.5 Other Functions on Lazy Series

`Derivative(s): RngPowLazElt -> RngPowLazElt`

`Derivative(s, v): RngPowLazElt, RngIntElt -> RngPowLazElt`

`Derivative(s, v, n): RngPowLazElt, RngIntElt, RngIntElt -> RngPowLazElt`

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): RngPowLazElt -> RngPowLazElt`

`Integral(s, v): RngPowLazElt, RngIntElt -> RngPowLazElt`

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): RngPowLazElt, RngPowLazElt -> RngPowLazElt`

`Evaluate(s, T): RngPowLazElt, [RngPowLazElt] -> RngPowLazElt`

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: Evaluating a Lazy Series `(ex-f9b608)`

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): RngPowLazElt -> RngPowLazElt

Sqrt(s): RngPowLazElt -> RngPowLazElt

The square root of the lazy series s.

IsSquare(s): RngPowLazElt -> BoolElt, RngPowLazElt

Return **true** if the lazy series s is a square and the square root if so.

PolynomialCoefficient(s, i): RngPowLazElt, RngIntElt -> RngPowLazElt

PolynomialCoefficient(s, i): RngPowLazElt, [RngIntElt] -> RngPowLazElt

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.

Chapter 53

Algebraic Power Series Rings

53.1 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].

53.2 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.

53.2.1 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.

53.2.2 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.

53.3 Constructors

Using constructors one can construct power series starting from polynomial data or using other power series recursively.

`PolyToSeries(s): RngMPolElt -> RngPowAlgElt`

Given a multivariate polynomial s, returns the series representation of s.

`AlgebraicPowerSeries(dp, ip, L, e): RngUPolElt, RngMPolElt, Lat, RngIntElt -> RngPowAlgElt`

`AlgebraicPowerSeries(dp, ip, e): RngUPolElt, RngMPolElt, RngIntElt -> RngPowAlgElt`

`AlgebraicPowerSeries(dp, ip): RngUPolElt, RngMPolElt -> RngPowAlgElt`

`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): Tup, SeqEnum, SeqEnum -> RngPowAlgElt

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 $\underline{x}^{\mu} \mapsto \prod_i \mathtt{v[i]}^{\langle \mathtt{nu[i]},\mu\rangle}$. 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): RngUPolElt -> RngPowAlgElt

```
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.

53.3.1 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): RngUPolElt -> Tup, SeqEnum, RngIntElt

```
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: Constructors for Algebraic Power Series `(ex-da4682)`

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 $\mathbb{Z}(\frac{1}{5}, -\frac{2}{5}) + \mathbb{Z}(\frac{2}{5}, \frac{1}{5})$ but its "expansions" are polynomials supported on $\mathbb{Z}(1, -2) + \mathbb{Z}(2, 1)$.

```
> 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 $\mathbb{Q}$.

```
> 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 $\mathbb{Q}$ by `gamma_0` s.t. $\texttt{gamma_0}^2 - \texttt{gamma_0} + 1 = 0$ and an extension of the exponent lattice to $\mathbb{Z}(\frac{1}{3}, \frac{1}{3}) + \mathbb{Z}(\frac{1}{3}, -\frac{2}{3})$. 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 $\mathbb{Q}[[x, y]] \to \mathbb{Q}[[x, y]]$ 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`.

53.4 Accessors and Expansion

The following functions provide an interface to conveniently extract information from a power series defined as above.

`Domain(s): RngPowAlgElt -> RngMPol`

Return the multivariate polynomial ring that is used for approximating the series s by its truncations.

ExponentLattice(s): RngPowAlgElt -> Tup

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): RngPowAlgElt -> RngUPolElt

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): RngPowAlgElt -> RngIntElt

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): RngPowAlgElt, RngIntElt -> BoolElt, RngMPolElt

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: Accessing an Algebraic Power Series (ex-4231eb)

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 $\mathbb{Q}(s)[[t^{1/3}]]$, because it is approximated in $\mathbb{Q}(s)[t]$ and has exponent lattice $\frac{1}{3}\mathbb{Z}$. A defining polynomial in $\mathbb{Q}(s)[t][u]$ was also computed. (Recall that `s3` has been defined recursively.) The order is $\frac{4}{3}$ which we know already from a previous expansion.

53.5 Arithmetic

There are functions to perform basic arithmetic operations (addition, subtraction etc.) on power series.

AlgComb(c, ss): RngMPolElt, SeqEnum -> RngPowAlgElt

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: RngPowAlgElt, RngPowAlgElt -> RngPowAlgElt`

`s - t: RngPowAlgElt, RngPowAlgElt -> RngPowAlgElt`

`s * t: RngPowAlgElt, RngPowAlgElt -> RngPowAlgElt`

Add, subtract, or multiply two power series.

Example: Substituting Series into Polynomials (ex-46bd4b)

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
```

53.6 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): RngPowAlgElt -> BoolElt`

Decides if the series is zero.

`s eq t: RngPowAlgElt, RngPowAlgElt -> BoolElt`

Decides if two series are equal.

`IsPolynomial(s): RngPowAlgElt -> BoolElt, RngMPolElt`

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: Testing Whether a Series Is Polynomial (ex-e6b227)

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
```

53.7 Modifiers

The following functions modify the representation of the power series or apply a simple automorphism.

ScaleGenerators(s, ls): RngPowAlgElt, SeqEnum -> RngPowAlgElt

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): RngPowAlgElt, RngMPol -> RngPowAlgElt

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): RngPowAlgElt -> RngPowAlgElt

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: Scaling the Generators of a Series (ex-e60c95)

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 $\mathbb{Q}(i)[[u, v]]$.

```
> 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
```

53.8 Bibliography

- [Bec07] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007. http://www.ricam.oeaw.ac.at/publications/reports/.

Chapter 54

Valuation Rings

54.1 Introduction

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

54.2 Creation Functions

54.2.1 Creation of Structures

`ValuationRing(Q, p): FldRat, RngIntElt -> RngVal`

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, $r = \frac{x}{y} \in \mathbb{Q}$ such that $p \not| y$.

`ValuationRing(F, f): FldFunRat, RngUPolElt -> RngVal`

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): FldFunRat -> RngVal`

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 $\deg(h) \geq \deg(g)$.

`ValuationRing(K, p): FldNum, RngOrdIdl -> RngVal`

`ValuationRing(K, p): FldFun, RngFunOrdIdl -> RngVal`

Given an algebraic number field or function field K and a prime ideal p contained in K, construct the valuation ring R corresponding to the valuation given by the prime ideal p.

54.2.2 Creation of Elements

`V ! r: RngVal, FldFunElt -> RngValElt`

`V ! r: RngVal, FldRatElt -> RngValElt`

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.

54.3 Structure Operations

54.3.1 Related Structures

`Category(V): RngVal -> Cat`

`Parent(V): RngVal -> Pow`

`PrimeRing(V): RngVal -> RngVal`

`Center(V): RngVal -> RngVal`

`FieldOfFractions(V): RngVal -> Rng`

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

54.3.2 Numerical Invariants

`Characteristic(V): RngVal -> RngIntElt`

54.4 Element Operations

54.4.1 Arithmetic Operations

`+ v: RngValElt -> RngValElt`

`- v: RngValElt -> RngValElt`

`v + w: RngValElt, RngValElt -> RngValElt`

`v - w: RngValElt, RngValElt -> RngValElt`

`v * w: RngValElt, RngValElt -> RngValElt`

`v ^ k: RngValElt, RngIntElt -> RngValElt`

`v / w: RngValElt, RngValElt -> RngValElt`

`v +:= w: RngValElt, RngValElt -> RngValElt`

`v -:= w: RngValElt, RngValElt -> RngValElt`

`v *:= w: RngValElt, RngValElt -> RngValElt`

`v div w: RngValElt, RngValElt -> RngValElt`

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.

54.4.2 Equality and Membership

`v eq w: RngValElt, RngValElt -> BoolElt`

`v ne w: RngValElt, RngValElt -> BoolElt`

`v in V: RngValElt, Rng -> BoolElt`

`v notin V: RngValElt, Rng -> BoolElt`

54.4.3 Parent and Category

`Parent(v): RngValElt -> RngVal`

`Category(v): RngValElt -> Cat`

54.4.4 Predicates on Ring Elements

`IsZero(n): RngValElt -> BoolElt`

`IsOne(n): RngValElt -> BoolElt`

`IsMinusOne(n): RngValElt -> BoolElt`

`IsNilpotent(n): RngValElt -> BoolElt`

`IsIdempotent(n): RngValElt -> BoolElt`

`IsUnit(n): RngValElt -> BoolElt`

`IsZeroDivisor(n): RngValElt -> BoolElt`

`IsRegular(n): RngVal -> BoolElt`

54.4.5 Other Element Functions

`EuclideanNorm(v): RngValElt -> RngIntElt`

`Valuation(v): RngValElt -> RngIntElt`

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

`Quotrem(v, w): RngValElt, RngValElt -> RngValElt, RngValElt`

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): RngValElt, RngValElt -> RngValElt`

`Gcd(v, w): RngValElt, RngValElt -> RngValElt`

`GCD(v, w): RngValElt, RngValElt -> RngValElt`

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 and u is the uniformizing element of V (with valuation $\phi(u) = 1$).

`ExtendedGreatestCommonDivisor(v, w): RngValElt, RngValElt -> RngValElt, RngValElt, RngValElt`

`Xgcd(v, w): RngValElt, RngValElt -> RngValElt, RngValElt, RngValElt`

`XGCD(v, w): RngValElt, RngValElt -> RngValElt, RngValElt, RngValElt`

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 and u is the uniformizing element of V (with valuation $\phi(u) = 1$).

Chapter 55

Galois Rings

55.1 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 $\mathbb{Z}_{p^a}$ (where p is prime) by a monic polynomial $D \in \mathbb{Z}[x]$ which is irreducible modulo p. Thus R is presented as the polynomial quotient ring $\mathbb{Z}_{p^a}[x]/\langle D \rangle$ and is usually written as $\mathrm{GR}(p^a, d)$, 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 $\mathbb{Z}_p[x]/\langle D \rangle$, where D is here considered as a polynomial in $\mathbb{Z}_p[x]$ (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 $\mathbb{Z}_{p^a}$ 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 $\mathbb{Z}_{p^a}$. For a 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 $\mathbb{Z}_p[x]/\langle D \rangle$. 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.

55.2 Creation Functions

55.2.1 Creation of Structures

`GaloisRing(q, d): RngIntElt, RngIntElt -> RngGal`

`GR(q, d): RngIntElt, RngIntElt -> RngGal`

Given integers q, $d \geq 1$, where $q = p^a$ for prime p and $a \geq 1$, create the default Galois ring $\mathrm{GR}(p^a, d)$. The defining polynomial used to construct the ring will be that used for $\mathbf{F}_{p^d}$, lifted to $\mathbb{Z}_{p^a}$. 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): RngIntElt, RngIntElt, RngIntElt -> RngGal`

`GR(p, a, d): RngIntElt, RngIntElt, RngIntElt -> RngGal`

`Check: BoolElt`	`Default: true`

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

`GR(q, D): RngIntElt, RngUPol -> RngGal`

Given an integer q, where $q = p^a$ for prime p and $a \geq 1$, and a monic polynomial D over $\mathbb{Z}$ such that D is irreducible mod p, create the Galois ring $R = \mathrm{GR}(p^a, D)$. The coefficients of D are reduced modulo p^a, and R is constructed to behave like the polynomial quotient ring $\mathbb{Z}_{p^a}[x]/\langle D \rangle$. 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): RngIntElt, RngIntElt, RngUPol -> RngGal`

`GR(p, a, D): RngIntElt, RngIntElt, RngUPol -> RngGal`

Check: BoolElt	Default: true

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

55.2.2 Names

`AssignNames(~R, [f]): RngGal, [ MonStgElt ]`

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): RngGal, RngIntElt -> RngGalElt`

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

55.2.3 Creation of Elements

`R . 1: RngGal -> RngGalElt`

`Generator(R): RngGal -> RngGalElt`

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

`R ! a: RngGal, RngElt -> RngGalElt`

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 $\mathbb{Z}_{p^a}$ of R, to be identified with the corresponding element of R.

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

`One(R): RngGal -> RngGalElt`

`Identity(R): RngGal -> RngGalElt`

`Zero(R): RngGal -> RngGalElt`

`Representative(R): RngGal -> RngGalElt`

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

`Random(R): RngGal -> RngGalElt`

Create a pseudo-random element of Galois ring R.

55.2.4 Sequence Conversions

`ElementToSequence(a): RngGalElt -> [ RngIntResElt ]`

`Eltseq(a): RngGalElt -> [ RngIntResElt ]`

Given an element a of the Galois ring R, return the sequence of coefficients $[a_0, \ldots, a_{n-1}]$ in the base ring $\mathbb{Z}_{p^a}$ of R (where R is $\mathbb{Z}_{p^a}[x]/\langle D \rangle$), 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: Defining a Galois Ring (ex-5d69d7)

We can define the Galois ring $\mathrm{GR}(2^3, 2)$ 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 ]
```

55.3 Structure Operations

55.3.1 Related Structures

`Category(R): RngGal -> Cat`

`Parent(R): RngGal -> PowerStructure`

`Centre(R): RngGal -> RngGal`

`PrimeRing(R): RngGal -> RngGal`

`PrimeField(R): RngGal -> RngGal`

`FieldOfFractions(R): RngGal -> RngGal`

55.3.2 Numerical Invariants

`Characteristic(R): RngGal -> RngIntElt`

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

`# R: RngGal -> RngIntElt`

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

`Degree(R): RngGal -> RngIntElt`

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

`ResidueField(R): RngGal -> FldFin`

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

55.3.3 Ring Predicates and Booleans

The following functions are described for rings in general in Section Predicates and Boolean Operations.

`IsCommutative(R): RngGal -> BoolElt`

`IsUnitary(R): RngGal -> BoolElt`

`IsFinite(R): RngGal -> BoolElt`

`IsOrdered(R): RngGal -> BoolElt`

`IsField(R): RngGal -> BoolElt`

`IsEuclideanDomain(R): RngGal -> BoolElt`

`IsPID(R): RngGal -> BoolElt`

`IsUFD(R): RngGal -> BoolElt`

`IsDivisionRing(R): RngGal -> BoolElt`

`IsEuclideanRing(R): RngGal -> BoolElt`

`IsPrincipalIdealRing(R): RngGal -> BoolElt`

`IsDomain(R): RngGal -> BoolElt`

`R eq G: RngGal, Rng -> BoolElt`

`R ne G: RngGal, Rng -> BoolElt`

55.4 Element Operations

See also Section Generic Element Functions.

55.4.1 Arithmetic Operators

`+ a: RngGalElt -> RngGalElt`

`- a: RngGalElt -> RngGalElt`

`a + b: RngGalElt, RngGalElt -> RngGalElt`

`a - b: RngGalElt, RngGalElt -> RngGalElt`

`a * b: RngGalElt, RngGalElt -> RngGalElt`

`a ^ k: RngGalElt, RngIntElt -> RngGalElt`

`a +:= b: RngGalElt, RngGalElt -> RngGalElt`

`a -:= b: RngGalElt, RngGalElt -> RngGalElt`

`a *:= b: RngGalElt, RngGalElt -> RngGalElt`

55.4.2 Euclidean Operations

`a div b: RngGalElt, RngGalElt -> RngGalElt`

`a mod b: RngGalElt, RngGalElt -> RngGalElt`

`Quotrem(a, b): RngGalElt, RngGalElt -> RngGalElt`

`GCD(a, b): RngGalElt, RngGalElt -> RngGalElt`

`LCM(a, b): RngGalElt, RngGalElt -> RngGalElt`

`XGCD(a, b): RngGalElt, RngGalElt -> RngGalElt, RngGalElt, RngGalElt`

55.4.3 Equality and Membership

`a eq b: RngGalElt, RngGalElt -> BoolElt`

`a ne b: RngGalElt, RngGalElt -> BoolElt`

`a in R: RngGalElt, Rng -> BoolElt`

`a notin R: RngGalElt, Rng -> BoolElt`

55.4.4 Parent and Category

`Parent(a): RngGalElt -> RngGal`

`Category(a): RngGalElt -> Cat`

55.4.5 Predicates on Ring Elements

`IsZero(a): RngGalElt -> BoolElt`

`IsOne(a): RngGalElt -> BoolElt`

`IsMinusOne(a): RngGalElt -> BoolElt`

IsNilpotent(a): RngGalElt -> BoolElt

IsIdempotent(a): RngGalElt -> BoolElt

IsUnit(a): RngGalElt -> BoolElt

IsZeroDivisor(a): RngGalElt -> BoolElt

Example: Groebner Bases over Galois Rings (ex-6cf97d)

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 Gröbner Bases for much more information on such Gröbner bases.

Chapter 56

Newton Polygons

56.1 Introduction

This chapter introduces functions which allow the construction 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.

56.2 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.

56.2.1 Creation of Newton Polygons

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

`NewtonPolygon(f): RngMPolElt -> NwtnPgon`

```
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): RngUPolElt -> NwtnPgon

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): RngUPolElt, RngOrdIdl -> NwtnPgon

NewtonPolygon(f, p): RngUPolElt, RngFunOrdIdl -> NwtnPgon

NewtonPolygon(f, p): RngUPolElt, RngIntElt -> NwtnPgon

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): RngUPolElt, PlcFunElt -> NwtnPgon

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): Crv -> NwtnPgon

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

NewtonPolygon(V): SeqEnum -> NwtnPgon

NewtonPolygon(V): SetEnum -> NwtnPgon

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): NwtnPgon -> SeqEnum`

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: Creating Newton Polygons (ex-4f67e3)

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.

56.2.2 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): NwtnPgon -> SeqEnum`

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 *Creation of Newton Polygons*). The faces are listed anticlockwise starting with the face with its left endpoint being the lowest of the leftmost points.

`InnerFaces(N): NwtnPgon -> SeqEnum`

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): NwtnPgon -> SeqEnum`

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

`OuterFaces(N): NwtnPgon -> SeqEnum`

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): NwtnPgon -> SeqEnum`

The faces of the compact convex hull of P_N.

Example: Faces of a Newton Polygon (ex-a15e3f)

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): NwtnPgon -> SeqEnum`

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

`InnerVertices(N): NwtnPgon -> SeqEnum`

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

`LowerVertices(N): NwtnPgon -> SeqEnum`

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

`OuterVertices(N): NwtnPgon -> SeqEnum`

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

`AllVertices(N): NwtnPgon -> SeqEnum`

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

Example: Vertices of a Newton Polygon (ex-534640)

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;
```

(continues on next page)

(continued from previous page)

```
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): NwtnPgonFace -> SeqEnum`

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

`FacesContaining(N, p): NwtnPgon, Tup -> SeqEnum`

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

Example: End Vertices and Containing Faces `(ex-8ed2f7)`

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> ]
```

(continues on next page)

(continued from previous page)

```
> 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): NwtnPgonFace -> Tup`

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): NwtnPgon -> [ Tup ]`

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

`Weight(F): NwtnPgonFace -> RngIntElt`

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): NwtnPgon -> SeqEnum`

The slopes of the faces of the newton polygon N.

`InnerSlopes(N): NwtnPgon -> SeqEnum`

`LowerSlopes(N): NwtnPgon -> SeqEnum`

`AllSlopes(N): NwtnPgon -> SeqEnum`

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

Example: Gradient Vector and Weight of a Face (ex-260949)

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
```

56.2.3 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): NwtnPgon, Tup -> BoolElt`

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): NwtnPgon, Tup -> BoolElt`

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): NwtnPgon, Tup -> BoolElt`

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): NwtnPgon, Tup -> BoolElt`

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): NwtnPgon, Tup -> BoolElt`

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

56.3 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): NwtnPgon -> BoolElt`

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

`Polynomial(N): NwtnPgon -> RngElt`

The polynomial used to define the polygon N.

`ParentRing(N): NwtnPgon -> Rng`

The parent ring of the polynomial of the polygon N.

`IsNewtonPolygonOf(N, f): NwtnPgon, RngElt -> BoolElt`

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

`FaceFunction(F): NwtnPgonFace -> RngElt`

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): NwtnPgonFace -> BoolElt`

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

`IsDegenerate(N): NwtnPgon -> BoolElt`

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

`IsNondegenerate(F): NwtnPgonFace -> BoolElt`

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

`IsNondegenerate(N): NwtnPgon -> BoolElt`

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

56.4 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): RngUPolElt -> [ < RngIntElt, RngIntElt > ]`

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

`ValuationsOfRoots(f, p): RngUPolElt, RngIntElt -> [ < RngIntElt, RngIntElt > ]`

`ValuationsOfRoots(f, p): RngUPolElt, RngOrdIdl -> [ < RngIntElt, RngIntElt > ]`

`ValuationsOfRoots(f, p): RngUPolElt, PlcFunElt -> [ < RngIntElt, RngIntElt > ]`

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.

56.5 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): MonStgElt, RngIntElt`

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`.

56.5.1 Operations not associated with Duval's Algorithm

`PuiseuxExpansion(f, n): RngUPolElt, RngIntElt -> SeqEnum[RngSerPuisElt]`

```
PreciseRoot   : BoolElt                    Default: false
TestSquarefree: BoolElt                    Default: true
NoExtensions  : BoolElt                    Default: false
LowerFaces    : BoolElt                    Default: true
OneRoot       : BoolElt                    Default: false
Newton        : Verbose                    Default: 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(K): Fld → FldAC`), so that all roots may be found.

`ExpandToPrecision(f, c, n): RngUPolElt, RngSerElt, RngIntElt -> RngSerElt`

```
PreciseRoot   : BoolElt                    Default: false
TestSquarefree: BoolElt                    Default: true
Newton        : Verbose                    Default: 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): RngUPolElt, RngIntElt, RngIntElt -> RngSerElt`

```
Newton: Verbose                    Default: 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: Puiseux Expansions to Higher Precision (ex-d1b398)

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 -
```

(continues on next page)

(continued from previous page)

```
    2096081963188224*a^63 - 25814314231136256*a^67 -
    319605795242639360*a^71 - 3975750610806374400*a^75 -
    4966629993807347712 0*a^79 - 622818862289639178240*a^83 -
    78372470789596879257 60*a^87 - 9893104646049113309184 0*a^91 -
    12524249498721749827 58400*a^95 - 15897106567806080658702336*a^99
    + O(a^103)
Time: 0.410
```

IsPartialRoot(f, c): RngUPolElt, RngSerElt -> BoolElt

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

IsUniquePartialRoot(f, c): RngUPolElt, RngSerElt -> BoolElt

```
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: Checking That a Partial Root Can Be Expanded (ex-5113cd)

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
```

(continues on next page)

(continued from previous page)

```
> 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): RngSerElt -> SeqEnum`

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): RngSerElt, RngSerElt -> SeqEnum`

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: Exponents of Puiseux Expansions (ex-cc0585)

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)),
```

(continues on next page)

(continued from previous page)

```
    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]);
[]
```

56.5.2 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): RngUPolElt, RngIntElt -> SeqEnum

```
Version       : MonStgElt                    Default: "Rational"
TestSquarefree: BoolElt                      Default: true
NoExtensions  : BoolElt                      Default: false
LowerFaces    : BoolElt                      Default: true
OneRoot       : BoolElt                      Default: false
Newton        : Verbose                      Default: 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): Tup -> SeqEnum`

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

`PuiseuxToParametrization(S): RngSerElt -> Tup`

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: Duval's Algorithm for Puiseux Expansions (ex-2dca42)

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);
```

(continues on next page)

(continued from previous page)

```
> 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
```

(continues on next page)

(continued from previous page)

```
> 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)
]
```

(continues on next page)

(continued from previous page)

```
> 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]);
```

(continues on next page)

(continued from previous page)

```
> 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);
Time: 0.439
> 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 D := DuvalPuiseuxExpansion(f, 13);
Time: 0.009
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.200
```

56.5.3 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): RngUPolElt -> [<RngSerElt, RngIntElt>]`

`Roots(f, n): RngUPolElt, RngIntElt -> [<RngSerElt, RngIntElt>]`

```
Newton: Verbose                    Default: 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): RngUPolElt -> BoolElt, RngSerElt`

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: Roots of Polynomials over Puiseux Series (ex-0c4847)

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 :
```

(continues on next page)

(continued from previous page)

```
[
   <-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>
]
```

56.6 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.

Chapter 57

Series Rings over p-Adic Rings

57.1 Introduction

Thanks to Xavier Caruso and David Lubicz, Magma now contains an implementation of linear algebra over $\mathbf{Z}_p[[u]]$ and related rings [CL12]. This is a first step in various calculations with p-adic Hodge theory, to be able to compute with modules over suitable approximations in these rings.

The package deliberately builds a wrapper level through user-defined types in Magma. While one could alternatively just do everything in terms of power (and Laurent) series rings over the p-adics directly, this extra layer is hoped to make it easier to use the functionality in a coherent way. Some of the common functions applicable to power series, matrices, and modules have been copied over, but not all of them. The underlying attributes can be accessed directly in the cases where there is no equivalent on the types described here. Most notably, vectors and matrices that are user-defined types do not seem yet to be accessible via $v[i]$ or $M[i,j]$.

The package has seen limited testing, with not every possibility of precision bounds (in both the p-adic and power series aspects) being considered in all cases, nor has much work been done for base rings that are extensions of $\mathbf{Z}_p$.

57.1.1 Background

Let R be a discrete valuation ring with residue field of characteristic p, and let ν be real. Recall that $f(u) = \sum_i a_i u^i$ where $a_i \in \mathrm{Frac}(R)$ converges for $|u| < 1/p^\nu$ if and only if $\inf_i v_R(i) + i\nu$ is bounded.

When ν is rational, this infinimum is attained. It is called the Gauss valuation, denoted by $v_\nu(f)$, and the Weierstrass degree $d_\nu(f)$ is the minimal i that realises this (by convention, the Weierstrass degree of the zero element is $-\infty$). An element is *distinguished* if its Gauss valuation is 0. We have that $v_\nu(fg) = v_\nu(f) + v_\nu(g)$ and $v_\nu(f+g) \geq \min\big(v_\nu(f), v_\nu(g)\big)$, while $d_\nu(fg) = d_\nu(f)d_\nu(g)$.

As it is not particularly clear how to represent elements of a convergent series ring in finite form, so we instead work with the "slope" ring of elements of nonnegative Gauss valuation for which $v_\nu(f) \geq 0$, namely

$$S^\nu(R) = \left\{ \sum_{i=0}^{\infty} a_i u^i | v_R(a_i) + i \cdot \nu \geq 0, a_i \in \mathrm{Frac}(R) \right\}.$$

For instance, when $\nu = 0$, this ring is just the power series ring over R.

There are two related rings of interest. The first extends the coefficients to allow them to have negative valuation in $\mathrm{Frac}(R)$, namely $S_p^\nu(R) = S^\nu[1/\pi]$ where π uniformises R. The second is a type of Laurent extension, namely writing $\nu = a/b$ in lowest terms, we have $S_u^\nu(R) = S^\nu[\pi^a/u^b]$.

For instance, the polynomial u^2/p is in $S^{1/2}(\mathbf{Q}_p)$ and $S^\nu(\mathbf{Q}_p)$ for all ν, and inverting it gives us $p/u^2 \in S_u^{1/2}(\mathbf{Q}_p)$. Note that as a power series u^2/p clearly has infinite radius of convergence, even though it is in not in the slope ring $S^0(\mathbf{Q}_p)$.

Another example, $p + u$ has Gauss valuation $\min(1, \nu)$, and thus the Weierstrass degree is 0 for $\nu \geq 1$ but degree 1 for $\nu < 1$.

Both of these "completion" rings are Euclidean, which allows us to construct a module theory over them. While internally Magma has generic code for handling modules over Euclidean rings, with these user-defined types the component parts are all taken from boilerplate package-level implementations.

57.1.2 Basic Operations

Here are the basic creations and operations for S^ν rings and elements. All of these have a direct analogue for S_p^ν and S_u^ν rings.

Creation Intrinsics

SnuRing(F, nu): FldPad, FldRatElt -> SnuRng

SpRing(F, nu): FldPad, FldRatElt -> SpRng

SuRing(F, nu): FldPad, FldRatElt -> SuRng

```
Precision: RngIntElt                    Default: Infinity()
```

Given a p-adic field (or ring) F and a slope ν (which can be rational or integral), create the slope ring (or the S_p^ν or S_u^ν ring). The precision refers to that of the power series ring.

SnuRing(F): FldPad -> SnuRng

SpRing(F): FldPad -> SpRng

SuRing(F): FldPad -> SuRng

SnuRing(p, e): RngIntElt, RngIntElt -> SnuRng

SnuRing(p): RngIntElt -> SnuRng

SpRing(p, e): RngIntElt, RngIntElt -> SpRng

SpRing(p): RngIntElt -> SpRng

SuRing(p, e): RngIntElt, RngIntElt -> SuRng

SuRing(p): RngIntElt -> SuRng

```
nu       : FldRatElt                    Default: 0
Precision: RngIntElt                    Default: Infinity()
```

The first three are the same as above, but with ν as a parameter. The others create the slope ring over $\mathbf{Q}_p$, with e being the precision of the p-adic ring in this case.

SnuRing(S, nu): RngSerPow, FldRatElt -> SnuRng

SpRing(S, nu): RngSerPow, FldRatElt -> SpRng

SuRing(S, nu): RngSerLaur, FldRatElt -> SuRng

Given a power-series ring over a p-adic ring or field and a rational number (or integer), create the slope ring. For the S_u^ν ring, one gives a Laurent series ring.

SnuRing(S): SpRng -> SnuRng

SpRing(S): SnuRng -> SpRng

SuRing(S): SuRng -> SnuRng

SuRing(S): SnuRng -> SuRng

Given an S_p^ν-ring, create the associated S^ν-ring, and vice-versa. Similarly with an S_u^ν-ring.

Access Functions

R **eq** S: SnuRng, SnuRng -> BoolElt

R **ne** S: SnuRng, SnuRng -> BoolElt

R **eq** S: SpRng, SpRng -> BoolElt

`R ne S: SpRng, SpRng -> BoolElt`

`R eq S: SuRng, SuRng -> BoolElt`

`R ne S: SuRng, SuRng -> BoolElt`

Two slope rings are equal if their underlying p-adic power series rings are the same (including precisions), and the slopes ν are the same.

`Slope(S): SnuRng -> RngIntElt`

`Slope(S): SpRng -> RngIntElt`

`Slope(S): SuRng -> RngIntElt`

The slope of the S^ν ring.

`Precision(S): SnuRng -> RngIntElt`

`Precision(S): SpRng -> RngIntElt`

`Precision(S): SuRng -> RngIntElt`

The precision of the underlying power-series ring.

`CoefficientRing(S): SnuRng -> RngIntElt`

`CoefficientRing(S): SpRng -> RngIntElt`

`CoefficientRing(S): SuRng -> RngIntElt`

The underlying p-adic ring of the S^ν ring.

57.1.3 Element Operations

The full range of addition, subtraction, multiplication, division (where applicable) and powering are available, similarly with equality.

`Parent(f): SnuElement -> SnuRng`

`Parent(f): SpElement -> SpRng`

`Parent(f): SuElement -> SuRng`

Return the S^ν ring to which the element belongs.

`IsWeaklyZero(f): SnuElement -> BoolElt`

`IsWeaklyZero(f): SpElement -> BoolElt`

`IsWeaklyZero(f): SuElement -> BoolElt`

This is the proper way to test equality with elements of S^ν. It checks whether every (known) coefficient is zero to within the p-adic precision.

Valuation and Degree

`GaussValuation(f): SnuElement -> FldRatElt`

`GaussValuation(f): SpElement -> FldRatElt`

`GaussValuation(f): SuElement -> FldRatElt`

`WeierstrassDegree(f): SnuElement -> FldRatElt`

`WeierstrassDegree(f): SpElement -> FldRatElt`

`WeierstrassDegree(f): SuElement -> FldRatElt`

The Gauss valuation and Weierstrass degree respectively return these values.

`IsDistinguished(f): SnuElement -> BoolElt`

The `IsDistinguished` intrinsic returns whether the Gauss valuation is zero (this intrinsic is only for S^ν elements).

`LeadingTerm(f): SnuElement -> RngSerPowElt`

`LeadingTerm(f): SpElement -> RngSerPowElt`

`LeadingTerm(f): SuElement -> RngSerLaurElt`

Gives the first non(weakly)zero term in the S_p^ν-element, as a power series (or Laurent series). Useful in particular when examining vectors or matrices of such.

`WeierstrassTerm(f): SnuElement -> RngSerPowElt`

`WeierstrassTerm(f): SpElement -> RngSerPowElt`

`WeierstrassTerm(f): SuElement -> RngSerLaurElt`

Gives the term corresponding to the Weierstrass degree of the S_p^ν-element, as a power series (or Laurent series). Useful in particular when examining vectors or matrices of such.

`O(x): SnuElement -> SnuElement`

`O(x): SpElement -> SpElement`

`O(x): SuElement -> SuElement`

Gives the big-Oh value of the argument.

Example: Basic Operations in a Slope Ring (ex-c0e1cb)

Here are some basic operations with a ring of slope 1.

```
> S<u> := SnuRing (pAdicField (5, 15), 1); // slope 1 over 5-adics
> assert Slope(S) eq 1;
> f := 5*u + u^2 + 2*u^3 + u^5 + O(u^10);
> g := u^4/25 + u^6 + u^8 + O(u^10);
> GaussValuation (f); // both have Gauss valuation 2
2
> GaussValuation (g); // both have Gauss valuation 2
2
> q := g/f; q; // g has degree 4, so this division works
(5^-3 + O(5^12))*u^3 + (-5^-4 + O(5^11))*u^4 +
  (616*5^-5 + O(5^10))*u^5 + (-606*5^-6 + O(5^9))*u^6 +
  (9946*5^-7 + O(5^8))*u^7 + (-3761*5^-8 + O(5^7))*u^8 + O(u^9)
> LeadingTerm (q); // returned as a power series ring element
(5^-3 + O(5^12))*$.1^3
```

All the same functionality copies over to the S_p^ν rings, the main difference is that one can invert p directly there. There is automatic coercion from S^ν to S_p^ν rings in some cases, even when the slopes are not equal. The criterion is whether the result in the S_p^ν ring is valid.

```
> Sp<uu> := SpRing (pAdicField (5, 15), 1/2); // slope 1/2 over 5-adics
> assert Slope(Sp) eq 1/2;
> ff := 1/5 + uu + 5*uu^2 + O(uu^10);
> GaussValuation (ff);
-1
> gg := uu + 5^2*uu^2 + uu^3 - uu^6 + uu^9 + O(uu^10);
> h := S ! (ff * S!gg); // coerced back to the Snu-ring
> assert IsDistinguished (h);
> LeadingTerm (ff) * LeadingTerm (gg) eq LeadingTerm (h);
true
```

Similarly with S_u^ν rings, where Laurent series are now also allowed.

```
> Su<z> := SuRing (pAdicField (7, 25), 2/3); // slope 2/3 over 7-adics
> F := 7/z + 3*7 + z^2/7 + 7^2*z^3 + O(z^10);
> GaussValuation (F);
1/3
> WeierstrassTerm (F);
(7 + O(7^26))*$.1^-1
> SR<t> := SnuRing (Su);
```

(continues on next page)

(continued from previous page)

```
> t*z/7; // automatic coercion into Su
(7^-1 + O(7^24))*z^2 + O(z^20)
```

57.1.4 Euclidean Algorithm

Given $A, B \in S^\nu$ with $v_\nu(A) \geq v_\nu(B)$ (so in particular B is nonzero) the quotient-remainder algorithm returns Q, R such that $A = BQ+R$ with R a polynomial of degree less that $d_\nu(B)$. In general the precision loss in this operation becomes more acute with larger ν. This precision loss percolates throughout all computations. Note that the result in general depends on ν.

The Weierstrass preparation theorem takes a distinguished element $f \in S^\nu$ and writes $f = UP$ where U is invertible in $S^\nu(R)$ and P is a polynomial of degree $d_\nu(f)$. The units of S^ν are precisely those elements with Gauss valuation and Weierstrass degree both 0.

The ring S^ν is not itself Euclidean, as we can only properly quotient elements as A/B when $v_\nu(B) \geq v_\nu(A)$, and this property need not be preserved in the Euclidean steps. For instance with $\nu = 1$, starting with $A = pu^2 + p^4$ and $B = p^2u$ we get $Q = u/p$ and $R = p^4$, but then $v_\nu(R) = 4 > 3 = v_\nu(B)$, so that we cannot divide B by R.

However, the ability to scale by arbitrary powers of the uniformiser implies that the ring S^ν_p is Euclidean with respect to the Weierstrass degree, and similarly the ring S^ν_u is a discrete valuation ring (hence Euclidean) for the valuation v_ν.

The extended gcd in S^ν or S^ν_p takes A, B with $v(A) \geq v(B)$ and returns (G, H, w, x, y, z) has $Aw+Bx = G$ where $v(B) = v(G)$ and $Ay + Bz = H$ with $v(H) > v(A)$ with $wz - xy = 1$ (consecutive Euclidean steps). The gcd is G in the case where H is the zero polynomial (this will always be true over S^ν_p).

Since S^ν_u is a discrete valuation ring, the quotient-remainder $A = qB + r$ always has $r = 0$ when $v(B) < v(A)$, and else $q = 0$. The *canonical* element of valuation v in S^ν_u is $u^y p^z$ where $0 \leq y < \text{denom}(\nu)$ is as small as possible and $y\nu + z = v$. The gcd of two elements A and B can always be taken to be a canonical element of valuation equal to the smaller of the valuations of A and B.

Intrinsics

`WeierstrassPreparation(f): SnuElement -> SnuElement, SnuElement`

Given a distinguished element f of S^ν (one of Gauss valuation 0), write $f = UP$ where U is invertible in S^ν and P is a polynomial (whose degree equals the Weierstrass degree of f).

`Quotrem(A, B): SnuElement, SnuElement -> SnuElement, SnuElement`

Given S^ν elements A and B with $v(A) \geq v(B)$, determine Q, R with $A = BQ + R$ where B is a polynomial whose degree is less than the Weierstrass degree of B.

`Quotrem(A, B): SpElement, SpElement -> SpElement, SpElement`

Given A and (nonzero) B which are S^ν_p elements, determine Q, R with $A = BQ + R$ where B is a polynomial whose degree is less than the Weierstrass degree of B.

`Quotrem(A, B): SuElement, SuElement -> SuElement, SuElement`

Given A and (nonzero) B which are S^ν_u elements, determine Q, R with $A = BQ + R$. Here when $v(A) \geq v(B)$ we have $Q = A/B$ and $R = 0$, and when $v(A) < v(B)$ we have $Q = 0$ and $R = A$.

`ExtendedGcd(A, B): SnuElement, SnuElement -> SnuElement, SnuElement, SnuElement, SnuElement, SnuElement, SnuElement`

Given S^ν elements A and B with $v(A) \geq v(B)$, try to compute the gcd. This returns a 6-tuple (G, H, w, x, y, z) with $Aw + Bx = G$, $Cw + Dy = H$, and $wz - xy = 1$. When $H = 0$ then G is the gcd. Alternatively, $v(H)$ will be some element with $v(H) > v(A)$, which precludes the Euclidean algorithm from progressing.

`ExtendedGcd(A, B): SpElement, SpElement -> SpElement, SpElement, SpElement, SpElement, SpElement, SpElement`

Given S^ν_p elements A and B compute the extended gcd. This returns a 6-tuple (G, H, w, x, y, z) with $Aw + Bx = G$, $Cw + Dy = H = 0$, and $wz - xy = 1$, where G is the gcd.

ExtendedGcd(A, B): SuElement, SuElement -> SuElement, SuElement, SuElement, SuElement, SuElement, SuElement

Given S_u^ν elements A and B compute the extended gcd. Since S_u^ν is a DVR, the gcd can be taken to be the CanonicalElement of the smaller valuation of A and B.

CanonicalElement(S, v): SuRng, FldRatElt -> SuElement

The *canonical* element of valuation v in S_u^ν is $u^y p^z$ where $0 \leq y < \text{denom}(\nu)$ is as small as possible and $y\nu + z = v$.

Example: Euclidean Operations on Series (ex-db44fa)

Here are some basic examples of Euclidean operations.

```
> S<u> := SnuRing (pAdicField (5, 15), 1); // slope 1 over 5-adics
> f := 5*u + u^2 + 2*u^3 + u^5 + O(u^10);
> g := u^4/25 + u^6 + u^8 + O(u^10);
> q := g/f;
> assert IsDistinguished (q);
> U, P := WeierstrassPreparation (q); // loses a lot of precision
> U;
1 + O(5^15) + (-5^-1 + O(5^14))*u + (616*5^-2 + O(5^13))*u^2 +
 (-606*5^-3 + O(5^12))*u^3 + (9946*5^-4 + O(5^11))*u^4 +
 (-3761*5^-5 + O(5^10))*u^5 + O(u^6)
> P;
(5^-3 + O(5^12))*u^3 + O(u^6)
> q, r := Quotrem (g, f); assert r eq 0; assert q eq g/f;
> q; // same as above, though is O(u^10) not O(u^9)
(5^-3 + O(5^12))*u^3 + (-5^-4 + O(5^11))*u^4 +
  (616*5^-5 + O(5^10))*u^5 + (-606*5^-6 + O(5^9))*u^6 +
  (9946*5^-7 + O(5^8))*u^7 + (-3761*5^-8 + O(5^7))*u^8 +
  (-172699*5^-9 + O(5^6))*u^9 + O(u^10)
> q, r := Quotrem (f, g); // this is valid, as the valuations are equal
> q, r; assert f eq g*q + r;
(5^2 + O(5^17))*u + (-5^4 + O(5^17))*u^3 + (24*5^4 + O(5^17))*u^5 +
  (-23*5^6 + O(5^17))*u^7 + (551*5^6 + O(5^17))*u^9 + O(u^10)
(5 + O(5^16))*u + (1 + O(5^15))*u^2 + (2 + O(5^15))*u^3 + O(u^10)
> G, H, A, B := ExtendedGcd (f, g); assert H eq 0;
> G;
(-5 + O(5^16))*u + (-1 + O(5^15))*u^2 + (-2 + O(5^15))*u^3 + O(u^10)
> assert A*f + B*g eq G;
```

```
> f1 := 5*u^2 + 5^4;     // example from the text
> g1 := 5^2*u;           // where ExtendedGcd fails
> G, H := ExtendedGcd (f1, g1); assert H ne 0;
> H;
-5^4 + O(5^24) + O(u^20)
> GaussValuation (H); GaussValuation (g1);
4 3
```

The above will also work in S_p^ν. As noted above, the corresponding commands for S_u^ν rings are particularly simplified.

```
> Su<z> := SuRing (S);
> F := Su!f; assert GaussValuation(F) eq 2;
> G := Su!g; assert GaussValuation(G) eq 2;
> Q, R := Quotrem (F, G); assert R eq 0;
> g, _, A, B, C, D := ExtendedGcd (F, G);
> g; assert g eq CanonicalElement (Su, 2);
```

(continues on next page)

(continued from previous page)

```
5^2 + O(5^17) + O(z^20)
> assert A*F + B*G eq g;
> assert C*F + D*G eq 0;
> assert A*D - B*C eq 1;
```

57.2 Matrices and Modules

Since S_p^ν and S_u^ν are Euclidean, we can work with canonical forms and modules over these rings. We consider S_p^ν first.

57.2.1 Matrices

The row-echelon form of a matrix over S_p^ν has pivot entries which are polynomials of the form

$$u^d + \sum_{j=0}^{d-1} b_j u^j \qquad \text{where} \qquad v(b_j) + \nu j > \nu d$$

for j in the sum, so that the Weierstrass degree is d and the Gauss valuation is $d\nu$.

The Hermite form additionally assures that the other entries in the columns with pivots are polynomials of smaller degree, via taking Quotrem. This can be made unique via suitable choices, and so provides a way of testing equality of modules.

The Smith form is computed by creating a matrix of pivots corresponding to a Hermite on the matrix and then a Hermite on the transpose. One also demands recursive divisibility of the resulting diagonal terms.

Over S_u^ν the picture is the same, except that pivot entries of the echelon form are all `CanonicalElement`s, with similar changes thereon. E.g., the Hermite form will have elements of smaller valuation above the pivots.

SpMatrixSpace(S, r, c): SpRng, RngIntElt, RngIntElt -> SpMatRng

SuMatrixSpace(S, r, c): SuRng, RngIntElt, RngIntElt -> SuMatRng

Given an S_p^ν of S_u^ν ring and a (nonnegative) number of rows and columns, construct the matrix space.

IdentityMatrix(S, n): SpRng, RngIntElt -> SpMat

IdentityMatrix(S): SpMatRng -> SpMat

ZeroMatrix(S): SpMatRng -> SpMat

ZeroMatrix(S, n): SpRng, RngIntElt -> SpMat

ZeroMatrix(S, r, c): SpRng, RngIntElt, RngIntElt -> SpMat

IdentityMatrix(S, n): SuRng, RngIntElt -> SuMat

IdentityMatrix(S): SuMatRng -> SuMat

ZeroMatrix(S): SuMatRng -> SuMat

ZeroMatrix(S, n): SuRng, RngIntElt -> SuMat

ZeroMatrix(S, r, c): SuRng, RngIntElt, RngIntElt -> SuMat

Different ways of obtaining the zero matrix or identity matrix (if applicable) of a matrix ring.

SpMatrix(A): SeqEnum -> SpMat

SpMatrix(r, c, A): RngIntElt, RngIntElt, SeqEnum -> SpMat

SpMatrix(v): SeqEnum -> SpMat

SpMatrix(A): SeqEnum -> SpMat

SuMatrix(r, c, A): RngIntElt, RngIntElt, SeqEnum -> SuMat

`SuMatrix(v): SeqEnum -> SuMat`

Various ways of obtaining an `SpMat` or `SuMat`. The first takes a sequence of sequences of `SpElements` or `SuElements`, the second takes a row and column count and a sequence of `SpElements` or `SuElements`, and the third takes a sequence of `SpVecs` or `SuVecs`.

`GaussValuations(M): SpMat -> SeqEnum`

`GaussValuations(M): SuMat -> SeqEnum`

Returns a sequence of sequences of the Gauss valuations of the elements of a matrix. Since these can be `ExtendedReals`, the valuations themselves are put in an array rather than a matrix.

`WeierstrassDegrees(M): SpMat -> SeqEnum`

`WeierstrassDegrees(M): SuMat -> SeqEnum`

Returns a sequence of sequences of the Weierstrass degrees of the elements of a matrix. Since these can be `ExtendedReals`, the valuations themselves are put in an array rather than a matrix.

`LeadingTerms(M): SpMat -> Mtrx`

`LeadingTerms(M): SuMat -> Mtrx`

Returns a matrix (defined over the underlying power or Laurent series ring) which has the leading term of each element.

`WeierstrassTerms(M): SpMat -> Mtrx`

`WeierstrassTerms(M): SuMat -> Mtrx`

Returns a matrix (defined over the underlying power or Laurent series ring) which has the Weierstrass term of each element.

`IsWeaklyZero(M): SpMat -> BoolElt`

`IsWeaklyZero(M): SuMat -> BoolElt`

Returns whether every element in the matrix is nearly zero.

`EchelonForm(M): SpMat -> SpMat, SpMat`

`EchelonForm(M): SuMat -> SuMat, SuMat`

Transform: BoolElt	Default: false

Returns the (row) echelon form E, and (if desired) the transform T such that $E = TM$.

`HermiteForm(M): SpMat -> SpMat, SpMat`

`HermiteForm(M): SuMat -> SuMat, SuMat`

Transform: BoolElt	Default: false

Returns the Hermite form H, and (if desired) the transform T such that $H = TM$.

`SmithForm(M): SpMat -> SpMat, SpMat, SpMat`

`SmithForm(M): SuMat -> SuMat, SuMat, SuMat`

Returns the Smith form S, and (if desired) the transforms P, Q such that $S = PMQ$.

`Kernel(M): SpMat -> SpSpc`

`Kernel(M): SuMat -> SuSpc`

`Image(M): SpMat -> SpSpc`

`Image(M): SuMat -> SuSpc`

Return the kernel or image of an S_p^ν or S_u^ν matrix as a module space. The `Image` also has a `Transform` parameter that gives the inclusion.

Example: Matrices over Sp Rings (ex-8c19ad)

Here are some basic examples of operations with matrices over S_p^ν.

```
> S<u> := SpRing (pAdicField (3, 20), 1/2); // slope 1/2
> M := SpMatrix (2, 3, [ S | u^2/3,u/3^2,0, u^3,3^2*u,3*u ]);
> E, T := EchelonForm (M : Transform); assert E eq T*M;
> LeadingTerms (E);
[(3^-1 + O(3^19))*$.1^2  (3^-2 + O(3^18))*$.1  O($.1^20)]
[O($.1^20)               (3^2 + O(3^18))*$.1   (3 + O(3^21))*$.1]
> WeierstrassTerms (E); // not the same
[(3^-1 + O(3^19))*$.1^2  (3^-2 + O(3^18))*$.1    O($.1^20)]
[O($.1^20)              -(3^-1 + O(3^19))*$.1^2  (3 + O(3^21))*$.1]
```

```
> S<u> := SpRing (pAdicField (5, 10), 3/2); // slope 3/2
> M := SpMatrix (3,5,[S|5^3,u,u^2/5,0,0, 0,0,25,u^3/25,3*5, 0,0,0,0,5^2]);
> H, T := HermiteForm (M : Transform); assert H eq T*M;
> WeierstrassTerms(H);
[5^3 + O(5^13)  $.1        O($.1^20)     -(5^-5 + O(5^5))*$.1^5  O($.1^20)]
[O($.1^20)      O($.1^20)  5^2 + O(5^9)  (5^-2 + O(5^8))*$.1^3   O($.1^20)]
[O($.1^20)      O($.1^20)  O($.1^20)     O($.1^20)               5^2 + O(5^12)]
> X, P, Q := SmithForm (M : Transform); assert P*M*Q eq X;
> WeierstrassTerms(X); // note that 5^3 divides 5^2 in this ring!
[5^3 + O(5^13)  O($.1^20)     O($.1^20)      O($.1^20)  O($.1^20)]
[O($.1^20)      5^2 + O(5^9)  O($.1^20)      O($.1^20)  O($.1^20)]
[O($.1^20)      O($.1^20)     5^2 + O(5^12)  O($.1^20)  O($.1^20)]
```

Example: Matrices over Su Rings `(ex-970ed4)`

Here are similar examples over S_u^ν.

```
> S<u> := SuRing (pAdicField (5, 15), 2/3); // slope 2/3
> M := SuMatrix (2, 3, [ S | 5^2/u,0,u^2, u,5*u^2,5^3/u^2 ]);
> E, T := EchelonForm (M : Transform); assert E eq T*M;
> LeadingTerms (E);
[$.1        (5 + O(5^16))*$.1^2   (5^3 + O(5^18))*$.1^-2]
[O($.1^20)  5^3 + O(5^15)         (5^5 + O(5^20))*$.1^-4]
> WeierstrassTerms (E); // not the same, in lower right
[$.1        (5 + O(5^16))*$.1^2  (5^3 + O(5^18))*$.1^-2]
[O($.1^20)  5^3 + O(5^15)        -$.1^2]
```

```
> A := [S | 25/u,u,u^2/5,0,0, 0,0,25,125/u^2,50*u, 0,0,0,0,5*u^2];
> M := SuMatrix (3, 5, A);
> H, T := HermiteForm (M : Transform); assert H eq T*M;
> W := WeierstrassTerms(H); W;
[$.1^2 (5^-2 + O(5^13))*$.1^4 (5^-3 + O(5^12))*$.1^5 O($.1^20) O($.1^20)]
[O($.1^20) O($.1^20) 5^2 + O(5^17) (5^3 + O(5^18))*$.1^-2 O($.1^20)]
[O($.1^20) O($.1^20) O($.1^20) O($.1^20) (5 + O(5^16))*$.1^2]
> assert CanonicalElement (S, 4/3) eq S!W[1,1];
> assert CanonicalElement (S, 2) eq S!W[2,3];
> assert CanonicalElement (S, 7/3) eq S!W[3,5];
> X, P, Q := SmithForm (M : Transform); assert X eq P*M*Q;
> WeierstrassTerms(X);
[(5^-1 + O(5^14))*$.1^2 O($.1^20) O($.1^20) O($.1^20) O($.1^20)]
[O($.1^20) (5 + O(5^16))*$.1 O($.1^20) O($.1^20) O($.1^20)]
[O($.1^20) O($.1^20) (5 + O(5^16))*$.1^2 O($.1^20) O($.1^20)]
```

57.2.2 Modules

These are implemented as "vectors" over S_p^ν or S_u^ν.

`SpSpace(R, n): SpRng, RngIntElt -> SpSpc`

`SuSpace(R, n): SuRng, RngIntElt -> SuSpc`

Given an S_p^ν or S_u^ν ring and a nonnegative integer, construct the module of that degree.

`SpSpace(M): SpMat -> SpSpc`

`SuSpace(M): SuMat -> SuSpc`

Given an S_p^ν or S_u^ν matrix, construct the associated module space. This always computes the Hermite form to use as the basis.

`SpSpace(v): SeqEnum -> SpSpc`

`SuSpace(v): SeqEnum -> SuSpc`

Given a sequence of `SpVec`s or `SuVec`s, construct the module space generated by them. Again this always computes the Hermite form for the basis.

`Ambient(S): SpSpc -> SpSpc`

`Ambient(S): SuSpc -> SuSpc`

Given a module space, return the (full) ambient space of dimension equal to the degree.

`Parent(v): SpVec -> SpSpc`

`Parent(v): SuVec -> SuSpc`

The Parent of an `SpVec` or `SuVec` is always its `Ambient` space.

`ZeroVector(S): SpSpc -> SpVec`

`ZeroVector(S): SuSpc -> SuVec`

Return the zero vector of a module space.

`SpVector(e): SeqEnum -> SpVec`

`SuVector(e): SeqEnum -> SuVec`

Given a sequence of S_p^ν or S_u^ν elements, construct the vector of them.

`Basis(S): SpSpc -> SeqEnum`

`Basis(S): SuSpc -> SeqEnum`

Given a module space, return a basis as a sequence of vectors.

`BasisMatrix(S): SpSpc -> SpMat`

`BasisMatrix(S): SuSpc -> SuMat`

Given a module space, return a basis as a matrix.

`Dimension(S): SpSpc -> RngIntElt`

`Dimension(S): SuSpc -> RngIntElt`

`Degree(S): SpSpc -> RngIntElt`

`Degree(S): SuSpc -> RngIntElt`

The dimension and degree of a module space.

`LeadingTerms(v): SpVec -> ModTupRngElt`

`LeadingTerms(v): SuVec -> ModTupRngElt`

The leading term of each element in an `SpVec` or `SuVec`.

`WeierstrassTerms(v): SpVec -> ModTupRngElt`

`WeierstrassTerms(v): SuVec -> ModTupRngElt`

The Weierstrass term of each element in an `SpVec` or `SuVec`.

`GaussValuations(v): SpVec -> SeqEnum`

GaussValuations(v): SuVec -> SeqEnum

Returns a sequence of sequences of the Gauss valuations of the elements of a matrix. Since these can be **ExtendedReals**, the valuations themselves are put in an array rather than a vector.

WeierstrassDegrees(v): SpVec -> SeqEnum

WeierstrassDegrees(v): SuVec -> SeqEnum

Returns a sequence of sequences of the Weierstrass degrees of the elements of a vector. Since these can be **ExtendedReals**, the valuations themselves are put in an array rather than a vector.

IsWeaklyZero(v): SpVec -> BoolElt

IsWeaklyZero(v): SuVec -> BoolElt

Return whether an **SpVec** or **SuVec** is weakly zero.

M1 + M2: SpSpc, SpSpc -> SpSpc

M1 + M2: SuSpc, SuSpc -> SuSpc

DirectSum(M1, M2): SpSpc, SpSpc -> SpSpc

DirectSum(M1, M2): SuSpc, SuSpc -> SuSpc

M1 **meet** M2: SpSpc, SpSpc -> SpSpc

M1 **meet** M2: SuSpc, SuSpc -> SuSpc

The sum and intersection of two module spaces. The use of **DirectSum** requires the summands to be disjoint.

S * M: SpSpc, SpMat -> SpSpc

S * M: SuSpc, SuMat -> SuSpc

The transformation of a module space by a matrix. The basis of the resulting space will be given in Hermite form.

IsConsistent(M, v): SpMat, SpVec -> BoolElt, SpVec

IsConsistent(M, v): SuMat, SuVec -> BoolElt, SuVec

IsConsistent(M, e): SpMat, SeqEnum -> BoolElt, SeqEnum

IsConsistent(M, e): SuMat, SeqEnum -> BoolElt, SeqEnum

IsConsistent(M, W): SpMat, SpMat -> BoolElt, SpMat, SpMat

IsConsistent(M, W): SuMat, SpMat -> BoolElt, SuMat, SuMat

v **in** M: SpVec, SpMat -> BoolElt, SpVec

v **in** M: SuVec, SpMat -> BoolElt, SuVec

Determines whether $xM = v$ is solvable, and it so returns such an x. The second version takes an a sequence of vectors, and the third takes a matrix (and also includes the kernel as a third return value).

IsSubspace(A, B): SpSpc, SpSpc -> BoolElt, SpMat

IsSubspace(A, B): SuSpc, SuSpc -> BoolElt, SuMat

Determines whether A is a subspace of B, and if so gives an inclusion map. Equality can be determined via symmetric inclusion, or alternatively via the Hermite form (which should be unique) of the basis matrix.

Example: Modules over Sp Rings (ex-3f4950)

Here are some basic examples of usage for modules.

```
> S<u> := SpRing (pAdicField (5, 25), 1/2); // slope 1/2
> A := SpSpace (S, 3);
> A.1;
[ 1 + O(5^25) + O(u^20), O(u^20), O(u^20) ]
> M := SpMatrix (2, 3, [ S | 5*u,5,u^2, 1/5,u^3,u ]);
> B := SpSpace (M); assert Dimension(B) eq 2;
> LeadingTerms(BasisMatrix(B));
```

(continues on next page)

(continued from previous page)

```
[5^-1 + O(5^24) O($.1^20)    $.1]
[O($.1^20)      -5 + O(5^26) (24 + O(5^25))*$.1^2]
> B eq SpSpace (VerticalJoin (M, M)); // also gives inclusion
true
> A meet B eq B;
true
> Image(M) eq B; // by definition
true
> IsConsistent (M, A.1);
false
> IsConsistent (M, B.1);
true
> SpSpace([B.1]) + SpSpace([B.2]) eq B;
true
> SpSpace([B.1]) + SpSpace([u * B.2]) eq B; // u does not invert
false
> SpSpace([1/5 * B.1]) + SpSpace([5 * B.2]) eq B;
true
> sub<B|[B.1+B.2]> + sub<B|[B.2-B.1]> eq sub<B|[B.1,B.2]>;
true
```

Example: Module with a Nontrivial Kernel (ex-76e436)

Here is an example over S_p^ν with a nontrivial kernel.

```
> S<u> := SpRing (pAdicField (7, 30), 1); // slope 1
> M := SpMatrix(3,5,[S|1,0,u,u^2,u,  u^2/7,0,1,0,0, 0,0,1,0,u^4/7^2]);
> ROWS := Rows(M);
> V := u * ROWS[1] + u^2/7 * ROWS[3]; // create a kernel
> MAT := SpMatrix(ROWS cat [Universe(ROWS)|V]); // 4 x 5
> Dimension (SpSpace (MAT)); // 3
3
> K := Kernel (MAT); K;
SpSpace of degree 4, dimension 1 over Sp-slope ring of slope 1 given
 by Power series ring in $.1 over pAdicField(7, 30)
Basis given by:
[ (-1 + O(7^30))*u + O(u^21), O(u^21),
  (-7^-1 + O(7^29))*u^2 + O(u^21), 1 + O(7^30) + O(u^21) ]
> KK := Kernel(BasisMatrix(K)); KK;
SpSpace of degree 1, dimension 0 over Sp-slope ring of slope 1 given
 by Power series ring in $.1 over pAdicField(7, 30)
> assert ZeroVector(Ambient(KK)) in KK;
```

Example: Modules over Su Rings (ex-9b9567)

Here is an example over S_u^ν.

```
> S<u> := SuRing (pAdicField (7, 20), 5/4); // slope 5/4
> A := SuSpace (S, 3);
> B := SuSpace ([A.1, A.3]); // dimension 2
> SuSpace([B.1]) + SuSpace([7^5/u^4 * B.2]) eq B; // slope 5/4
true
> Basis(sub<A|[A.1+A.3]>)[1] in B;
true
> M := SuMatrix (2, 3, [S | 7^2/u,7,5*7*u, 7,u^3,u^2 ]);
> V := SuSpace(M);
> I := V meet B; assert Dimension(I) eq 1;
```

(continues on next page)

(continued from previous page)

```
> v := Basis(I)[1]; assert v in I;
> WeierstrassTerms(v);
(7 + O(7^21) O($.1^20) $.1^2)
> WeierstrassDegrees(v);
[ 0, -Infinity, 2 ]
> I + B eq B; assert v in B;
true
> J := VerticalJoin (M, M);
> SetSeed(1); // ensure example works
> JJ := (SuMatrixSpace(S,4,4)!RandomSLnZ(4,8,16)) * J;
> assert Dimension(Kernel(JJ)) eq 2;
> assert Dimension(Image(JJ)) eq 2;
```

Note that the final commands, with the dimension of the kernel/image of the given space, can sometimes fail for various reasons involving precision loss.

57.3 Bibliography

- [CL12] X. Caurso and D. Lubicz. Linear algebra over $Z_p[[u]]$ and related rings. *LMS Journal of Computation and Mathematics*, 2012.

Chapter 58

Local Galois Representations

58.1 Overview

This package provides functionality for working with Galois representations

$$\mathrm{Gal}(\bar{K}/K) \longrightarrow \mathrm{GL}_m(\mathbb{C})$$

over p-adic fields K (type `FldPad`). The representations we consider are precisely those on the 'Galois side' of the local Langlands correspondence, or, technically, the *Frobenius-semisimple Weil-Deligne representations* over K. We refer the reader to Tate's article [Tat79, §4] for their basic properties.

In arithmetic geometry, such representations arise from l-adic étale cohomology of algebraic varieties over K for $l \neq p$. Magma includes Galois representations that come from finite Galois groups of p-adic extensions, local components of Dirichlet characters and Artin representations, local Galois representations coming from a Tate module of an elliptic curve or a modular form, as well as various constructions to produce new representations from existing ones, such as direct sums, tensor products, induction, restriction, and semisimplification.

58.1.1 Notation and Printing

Suppose K is a finite extension of $\mathbb{Q}_p$, with ring of integers O, maximal ideal m, and residue field $k = O/m \cong \mathbb{F}_q$. The absolute Galois group of K fits into an exact sequence

$$\begin{array}{ccccccccc} 1 & \longrightarrow & I_K & \longrightarrow & \mathrm{Gal}(\bar{K}/K) & \longrightarrow & \mathrm{Gal}(\bar{k}/k) & \longrightarrow & 1, \\ & & & & \sigma & \mapsto & \sigma \bmod m & & \end{array}$$

where I_K is the *inertia group* of K. The group $\mathrm{Gal}(\bar{k}/k)$ is topologically generated by the automorphism $x \mapsto x^q$, and any of its lifts to $\mathrm{Gal}(\bar{K}/K)$ is called an *arithmetic Frobenius element*, denoted Frob_K.

The *Weil group* W_K *of* K is a subgroup of $\mathrm{Gal}(\bar{K}/K)$ generated by the inertia group I_K and any Frobenius element. It fits in the same exact sequence as above, except that the profinite group $\mathrm{Gal}(\bar{k}/k) \cong \hat{\mathbb{Z}}$ is replaced by a copy of $\mathbb{Z}$ with discrete topology. A *Weil representation* is a continuous representation

$$\rho : W_K \longrightarrow \mathrm{GL}_m(\mathbb{C}).$$

By continuity, the image of inertia $\rho(I_K)$ is finite, and ρ is said to be *unramified* if it is trivial. We will always assume that ρ is *Frobenius-semisimple*, that is $\rho(\mathrm{Frob}_K)$ is a semisimple endomorphism of $\mathbb{C}^m$. Every such representation ρ is a direct sum of irreducible representations of the form

$$\psi \otimes R,$$

where $\psi : I_K \mapsto 1, \mathrm{Frob}_K \mapsto \alpha \in \mathbb{C}^\times$ is an unramified 1-dimensional character (uniquely determined by $\alpha \in \mathbb{C}^\times$) and R is a representation of $\mathrm{Gal}(F/K)$ for some finite Galois extension F/K.

Weil representations occur naturally as local components of Artin representations, and as representations associated to (H^1 of) elliptic curves and abelian varieties over K with potentially good reduction. (By the

Néron-Ogg-Shafarevich criterion, potentially good reduction is *equivalent* to $\rho(I_K)$ being finite.) To deal with arbitrary reduction behaviour one considers, more generally, *Weil-Deligne* representations. These come from Galois representations

$$\rho : \mathrm{Gal}(\bar{K}/K) \longrightarrow \mathrm{GL}_m(\mathbb{C})$$

that can be described as follows.

As before, ρ is a direct sum of indecomposable representations, and an indecomposable one is now of the form (cf. [Tat79, 4.1.5])

$$\rho = \psi \otimes \mathtt{SP}(n) \otimes R,$$

with ψ and R as above, and $\mathtt{SP}(n)$ is a 'special' representation ([Tat79, 4.1.4]),

$$\mathtt{SP}(n) = \mathbb{C}e_0 + \mathbb{C}e_1 + \ldots + \mathbb{C}e_{n-1}.$$

The e_i are eigenvectors for Frob_K with eigenvalues q^{-i}, and the action of inertia is nilpotent, and represented by a matrix N (see [Tat79, 4.1.2]) that takes $e_i \mapsto e_{i+1}$ and $e_{n-1} \mapsto 0$.

This is precisely how Galois representations are stored in Magma,

$$\rho = \bigoplus_i \psi_i \otimes \mathtt{SP}(n_i) \otimes R_i,$$

except that it is convenient to

(1) assume that all R_i factor through the same Galois group $G = \mathrm{Gal}(F/K)$, and

(2) allow ψ_i to be arbitrary unramified representations, not necessarily 1-dimensional. Such a ψ is a sum of unramified characters $\chi_j : \mathrm{Frob}_K \mapsto \alpha_j$, and ψ is uniquely determined by its *Euler factor*

$$\Psi(T) = \det(1 - \mathrm{Frob}_K^{-1} T | \psi) = \prod_j (1 - \alpha_j^{-1} T) \in \mathbb{C}[T].$$

In other words, every component of ρ can be represented (though not quite uniquely) by a triple $\langle \Psi, n, c \rangle$, where $\Psi \in \mathbb{C}[T]$, $n \in \mathbb{Z}$, and c is a character of a representation R of a finite group $\mathrm{Gal}(F/K)$. Throughout the chapter we will refer to R as the 'finite part' of an indecomposable representation $\psi \otimes \mathtt{SP}(n) \otimes R$, though it depends on the choice of such a presentation.

Galois representations have Magma type `GalRep`. The base field K, field F, the Galois group F/K, and the list of components of ρ can be obtained with `BaseField`, `Field`, `Group`, and `Factorization`, respectively.

Example: Printing Galois Representations `(ex-8ff07a)`

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(Rationals());
```

First, construct a typical unramified representation specified by its Euler factor ψ.

```
> psi:=UnramifiedRepresentation(K,1+x^2); psi;
2-dim unramified Galois representation Unr(1+x^2) over Q2[20]
```

Next, construct a typical finite image representation, by picking a character of a finite Galois extension F/K, in this case an S_3-extension.

```
> S<z>:=PolynomialRing(K);
> F:=ext<K|z^3-2>;
> c:=GaloisRepresentations(F,K)[3]; c;
2-dim Galois representation (2,0,-1) with G=S3, I=C3, conductor 2^2 over Q2[20]
```

Here is a general Galois representation, the tensor product of all 3 terms, and another one — direct sum of the same three terms.

```
> psi*SP(K,2)*c;
8-dim Galois representation Unr(1+x^2)*SP(2)*(2,0,-1) with G=S3, I=C3,
   conductor 2^8 over Q2[20]
> psi+SP(K,2)+c;
6-dim Galois representation Unr(1+x^2) + (2,0,-1) + SP(2) with G=S3, I=C3,
   conductor 2^3 over Q2[20]
```

58.1.2 Conventions

There are various choices of signs in local class field theory, for which there appears to be no consensus in the literature. Our conventions follow Tate [Tat79], and are as follows. (We refer the reader to Tate's article [Tat79] for the definitions and properties of Weil-Deligne representations, and [Del79, Tat79] for their ϵ-factors.)

For a p-adic field K with ring of integers O, uniformizer π, and residue field $\mathbb{F}_q$,

- A Frobenius element Frob_K is an arithmetic Frobenius, i.e. acts as $x \mapsto x^q$ on the residue field (**not** $x \mapsto x^{1/q}$).
- The local reciprocity map $\theta : K^\times \to \mathrm{Gal}(\bar{K}/K)^{ab}$ takes π to Frob_K^{-1} (**not** Frob_K).
- The local epsilon-factors $\epsilon(\chi) = \epsilon(\chi, \psi, dx)$ rely implicitly on the choice of a measure dx on O and an additive character $\psi : K \to \mathbb{C}$. Our choices are that dx is normalized, $\int_O dx = 1$, and

$$\psi(x) = \exp\bigl(2\pi i \ \mathrm{Tr}_{K/\mathbb{Q}_p}(x)\bigr),$$

viewing $\mathrm{Tr}_{K/\mathbb{Q}_p}(x) \in \mathbb{Q}_p$ as any rational number $a/p^n \in \mathbb{Q}$ in the same class mod $\mathbb{Z}_p$. Finally, for 1-dimensional ramified χ, the formula for $\epsilon(\chi)$ is as in [Tat79, 3.2.6.2],

$$\epsilon(\chi) = \int_{c^{-1}O^\times} \chi^{-1}(\theta(x))\psi(x),$$

with

$$v_K(c) \ = \ v_K(\mathrm{Conductor}(\chi)) + v_p(\mathrm{Discriminant}(O, \mathbb{Z}_p)).$$

- Euler factors of global Artin representations A (and of Dirichlet characters) are *the same* (**not** complex conjugate) as the local ones:

`EulerFactor(A,p) = EulerFactor(GaloisRepresentation(A,p))`.

As the Artin L-function $L(A, s)$ is defined using arithmetic Frobenius and Galois representations using geometric Frobenius, this means that the Galois representation `GaloisRepresentation(A,p)` comes from the Galois action on the *dual* vector space of A.

58.1.3 Implementation Notes

Galois representations attached to Artin representations are computed using the machinery of [DD13]. Galois representations coming from elliptic curves rely partly on the theory of reconstructing representations from their Euler factors [DD15] (see §*Example: Reconstructing a Galois Representation from its Euler Factors*), and Rachel Newton's tame local reciprocity formula [New12].

58.2 Creating Galois Representations

`ZeroRepresentation(K): FldPad -> GalRep`

Galois representation 0 over a p-adic field K. It is 0-dimensional, and $0 + A = A$, $0 \otimes A = 0$ for every Galois representation A.

Example: Zero Galois Representation `(ex-153a0b)`

```
> K:=pAdicField(3,20);
> zero:=ZeroRepresentation(K);
> zero;
Galois representation 0 with G=C1, I=C1 over Q3[20]
> zero + CyclotomicCharacter(K) eq CyclotomicCharacter(K);
true
> zero*CyclotomicCharacter(K) eq zero;
true
```

PrincipalCharacter(K): FldPad -> GalRep

Principal character 1 of the absolute Galois group of K, as a Galois representation. It is a 1-dimensional unramified representation, same as `UnramifiedCharacter(K,1)`. Thus $1 \otimes A = 1$ for every Galois representation A.

Example: Principal Character of a Local Field (ex-48649a)

Take $K = \mathbb{Q}_3$, and F/K the unramified extension of degree 4, so that $G = \mathrm{Gal}(F/K) \cong C_4$. The 4 irreducible representations of G can be viewed as Galois representations, and the first one of these is the principal character (for any group).

```
> K:=pAdicField(3,20);
> one:=PrincipalCharacter(K);
> one;
1-dim trivial Galois representation 1 over Q3[20]
> F:=ext<K|4>;
> A1,A2,A3,A4:=Explode(GaloisRepresentations(F,K));
> A1 eq one;
true
```

CyclotomicCharacter(K): FldPad -> GalRep

Cyclotomic character over K. It is an unramified character (trivial on inertia) and takes the value q, the size of the residue field of K, on any Frobenius element.

Example: Cyclotomic Character (ex-4a5d9f)

```
> K:=pAdicField(3,20);
> chi:=CyclotomicCharacter(K);
> chi,EulerFactor(chi);
1-dim unramified Galois representation Unr(1/3) over Q3[20]
-1/3*x + 1
> chi^3,EulerFactor(chi^3);
1-dim unramified Galois representation Unr(1/27) over Q3[20]
-1/27*x + 1
```

UnramifiedCharacter(K, c): FldPad, Any -> GalRep

Galois representation over K given by an unramified character that sends the arithmetic Frobenius element $\mathrm{Frob}_K \mapsto c^{-1}$ (and, so, the geometric Frobenius element $\mathrm{Frob}_K^{-1} \mapsto c$.) The parameter c must be a non-zero complex number.

Example: Unramified Characters (ex-6bab13)

```
> K:=pAdicField(3,20);
> assert UnramifiedCharacter(K,1) eq PrincipalCharacter(K);
> assert UnramifiedCharacter(K,1/3) eq CyclotomicCharacter(K);
> C<i>:=ComplexField();
> UnramifiedCharacter(K,2+i);
1-dim unramified Galois representation Unr(2+i) over Q3[20]
```

UnramifiedRepresentation(K, CharPoly): FldPad, RngUPolElt -> GalRep

Unique unramified Galois representation ρ over K with Euler factor $\det(1-\mathrm{Frob}_K^{-1}|\rho) = \mathrm{CharPoly}$.

Example: Unramified Galois Representations (ex-b1231d)

```
> K:=pAdicField(37,20);
> R<x>:=PolynomialRing(Rationals());
> rho:=UnramifiedRepresentation(K,(1-37*x)*(1-3*x));
> rho;
2-dim unramified Galois representation Unr(1-40*x+111*x^2) over Q37[20]
> rho eq CyclotomicCharacter(K)^(-1)+UnramifiedCharacter(K,3);
true
```

UnramifiedRepresentation(K, dim, dimcomputed, CharPoly): FldPad, RngIntElt, RngIntElt, RngUPolElt -> GalRep

Unramified Galois representation over K of dimension *dim*, with Euler factor *CharPoly* computed up to and inclusive degree *dimcomputed*.

Example: Representations Not Fully Computed (ex-f653e8)

Consider the hyperelliptic curve $C : y^2 = x^5 + x + 1$ over the p-adic field $\mathbb{Q}_{10007}$.

```
> _<x>:=PolynomialRing(Rationals());
> p:=10007;
> K:=pAdicField(p,20);
> _<X>:=PolynomialRing(K);
> C:=HyperellipticCurve(X^5+X+1); C;
Hyperelliptic Curve defined by y^2 = x^5 + O(10007^20)*x^4 + O(10007^20)*x^3 +
   O(10007^20)*x^2 + x + 1 + O(10007^20) over pAdicField(10007, 20)
```

The Galois representation A associated to $H^1(C)$ is unramified, of dimension 4, and could be defined by

```
UnramifiedRepresentation(K,1-ap*x+bp*x^2-p*ap+p^2);
```

if we find a_p and b_p by counting points of C over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. The coefficient a_p can be computed very quickly:

```
> k:=ResidueClassField(Integers(K));
> _<X>:=PolynomialRing(k);
> Ck:=HyperellipticCurve(X^5+X+1);
> ap:=p+1-#Ck; ap;
-21
```

However, b_p would take a long time. If we are only interested in working with A up to degree 1 (e.g. to compute L-series of $C/\mathbb{Q}$ with $< 10^8$ terms), there is no reason to compute it. Instead, we can define an unramified Galois representation of degree 4, which is known to be computed only up to degree 1:

```
> A:=UnramifiedRepresentation(K,4,1,1-ap*x);
> A;
4-dim unramified Galois representation Unr(1+21*x+O(x^2)) over Q10007[20]
```

One can still take direct sums, and tensor products of such representations with (possibly ramified) Galois representations, and the Euler factors will still be correct up to degree 1:

```
> A*A;
16-dim unramified Galois representation Unr(1-441*x+O(x^2)) over Q10007[20]
> EulerFactor(A*A+1/CyclotomicCharacter(K));
-10448*x + 1
```

`SP(K, n): FldPad, RngIntElt -> GalRep`

The n-dimensional indecomposable Galois representation `SP(n)` over a p-adic field K; see §*Notation and Printing* for its description.

Example: Special Galois Representations (ex-8d3d78)

```
> K:=pAdicField(3,20);
> SP(K,1) eq PrincipalCharacter(K);
true
> rho:=SP(K,2); rho;
2-dim Galois representation SP(2) over Q3[20]
> Degree(rho);
2
> Semisimplification(rho);
2-dim unramified Galois representation Unr(1-4/3*x+1/3*x^2) over Q3[20]
> $1 eq PrincipalCharacter(K)+CyclotomicCharacter(K);
true
> InertiaInvariants(rho);
1-dim unramified Galois representation Unr(1/3) over Q3[20]
> EulerFactor(rho);
-1/3*x + 1
```

`SP(K, f, n): FldPad, RngUPolElt, RngIntElt -> GalRep`

Unramified twist $\psi \otimes \mathtt{SP}(n)$ over a p-adic field K, with ψ specified by its Euler factor f.

Example: Twisted Special Representations (ex-3bd7e7)

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(Rationals());
> SP(K,1-x^2,2);
4-dim Galois representation Unr(1-x^2)*SP(2) over Q2[20]
> $1*$1;     // Tensor product with itself
16-dim Galois representation Unr(1-1/2*x^2+1/16*x^4) + Unr(1-2*x^2+x^4)*SP(3)
   over Q2[20]
```

58.2.1 Representations from Finite Extensions

`GaloisRepresentations(F, K): FldPad, FldPad -> SeqEnum[GalRep]`

For a p-adic extension F/K, compute all irreducible Galois representations that factor through the (Galois closure of) F/K.

Example: Irreducible Characters of a Dihedral Extension (ex-314f0e)

We take F to be a degree 16 dihedral extension of $K = \mathbb{Q}_2$, and compute the irreducible characters of $\mathrm{Gal}(F/K)$, viewed as Galois representations over K.

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> F:=ext<K|x^8+2>;
> list:=GaloisRepresentations(F,K);
> list;
[
1-dim trivial Galois representation 1 over Q2[20],
1-dim Galois representation (1,1,-1,-1,1,1,1) with G=D8, I=D8, conductor 2^2
   over Q2[20],
1-dim Galois representation (1,1,-1,1,1,-1,-1) with G=D8, I=D8, conductor 2^3
   over Q2[20],
1-dim Galois representation (1,1,1,-1,1,-1,-1) with G=D8, I=D8, conductor 2^3
```

(continues on next page)

(continued from previous page)

```
   over Q2[20],
2-dim Galois representation (2,2,0,0,-2,0,0) with G=D8, I=D8, conductor 2^8
   over Q2[20],
2-dim Galois representation (2,-2,0,0,0,-zeta(8)_8^3+zeta(8)_8,
   zeta(8)_8^3-zeta(8)_8) with G=D8, I=D8, conductor 2^10 over Q2[20],
2-dim Galois representation (2,-2,0,0,0,zeta(8)_8^3-zeta(8)_8,
   -zeta(8)_8^3+zeta(8)_8) with G=D8, I=D8, conductor 2^10 over Q2[20]
]
```

The first 5 characters are not faithful, and we can descend them to smaller quotients of $\mathrm{Gal}(F/K)$.

```
> min:=[Minimize(rho): rho in list | not IsFaithful(Character(rho))];
> min;
[
1-dim trivial Galois representation 1 over Q2[20],
1-dim Galois representation (1,-1) with G=C2, I=C2, conductor 2^2 over Q2[20],
1-dim Galois representation (1,-1) with G=C2, I=C2, conductor 2^3 over Q2[20],
1-dim Galois representation (1,-1) with G=C2, I=C2, conductor 2^3 over Q2[20],
2-dim Galois representation (2,-2,0,0,0) with G=D4, I=D4, conductor 2^8
   over Q2[20]
]
```

`GaloisRepresentations(f): RngUPolEltFldPad -> SeqEnum[GalRep]`

For a polynomial f over a p-adic field K and splitting field F, returns irreducible representations of $\mathrm{Gal}(F/K)$.

Example: One-Dimensional Characters of a Cyclotomic Extension (ex-da3085)

We construct 4 one-dimensional characters of $\mathbb{Q}_2(\zeta_8)/\mathbb{Q}_2$.

```
> K:=pAdicField(2,20);
> R:=PolynomialRing(K);
> GaloisRepresentations(R!CyclotomicPolynomial(8));
[
1-dim trivial Galois representation 1 over Q2[20],
1-dim Galois representation (1,-1,1,-1) with G=C2^2, I=C2^2, conductor 2^3
   over Q2[20],
1-dim Galois representation (1,1,-1,-1) with G=C2^2, I=C2^2, conductor 2^2
   over Q2[20],
1-dim Galois representation (1,-1,-1,1) with G=C2^2, I=C2^2, conductor 2^3
   over Q2[20]
]
```

`PermutationCharacter(F, K): FldPad, FldPad -> GalRep`

For a p-adic extension F/K, compute $\mathbb{C}[\mathrm{Gal}(\bar{K}/K)/\mathrm{Gal}(\bar{K}/F)]$ as a Galois representation over K of degree $[F:K]$.

Example: Permutation Character of a Cubic Extension (ex-ba722b)

Take $K=\mathbb{Q}_2$ and $F=\mathbb{Q}_2(\sqrt[3]{2})$. Then `PermutationCharacter(F,K)` is a 3-dimensional representation which is the trivial representation plus a 2-dimensional irreducible one.

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> F:=ext<K|x^3-2>;
> PermutationCharacter(F,K);
3-dim Galois representation (3,1,0) with G=S3, I=C3, conductor 2^2 over Q2[20]
> $1 - PrincipalCharacter(K);
2-dim Galois representation (2,0,-1) with G=S3, I=C3, conductor 2^2 over Q2[20]
```

`A !! chi: GalRep, AlgChtrElt -> GalRep`

`A !! chi: GalRep, SeqEnum -> GalRep`

Change a Galois representation by a finite representation with character χ, which must be a character of `Group(A)`, or a list of values that determine such a character.

Example: Replacing the Character of a Representation (ex-b9793f)

Take $K = \mathbb{Q}_2$ and $F = \mathbb{Q}_2(\sqrt[3]{2})$, so that $G = \mathrm{Gal}(F/K) \cong S_3$. Using `!!` we can start with any Galois representation whose finite part comes from this Galois group, and replace it by any other character of G.

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> F:=ext<K|x^3-2>;
> rho:=PermutationCharacter(F,K);
> rho!![1,1,1];
1-dim trivial Galois representation 1 over Q2[20]
> rho!![6,0,0];
6-dim Galois representation (6,0,0) with G=S3, I=C3, conductor 2^4 over Q2[20]
> rho!![0,0,0];   // but not [-1,0,0] - may not be virtual
Galois representation 0 with G=S3, I=C3 over Q2[20]
```

58.2.2 Local Representations of Global Objects

`GaloisRepresentation(chi, p): GrpDrchElt, RngIntElt -> GalRep`

```
Precision: RngIntElt                    Default: 40
```

Local Galois representation at p of a Dirichlet character χ.

Example: Local Components of a Dirichlet Character (ex-a9f044)

Local components of a Dirichet character χ of order 6 at $p = 2, 3, 7$.

```
> G<chi>:=FullDirichletGroup(7);
> GaloisRepresentation(chi,2);
1-dim unramified Galois representation (1,-zeta(3)_3-1,zeta(3)_3)
   with G=C3, I=C1 over Q2[40]
> GaloisRepresentation(chi,3);
1-dim unramified Galois representation (1,-1,-zeta(3)_3-1,zeta(3)_3,-zeta(3)_3,
   zeta(3)_3+1) with G=C6, I=C1 over Q3[40]
> GaloisRepresentation(chi,7);
1-dim Galois representation (1,-1,-zeta(3)_3-1,zeta(3)_3,-zeta(3)_3,zeta(3)_3+1)
   with G=C6, I=C6, conductor 7^1 over Q7[40]
```

By our convention, the character χ, the associated Artin representation, and the Galois representations associated to them all have the same Euler factors.

```
> loc1:=EulerFactor(chi,2);
> loc2:=EulerFactor(ArtinRepresentation(chi),2);
> loc3:=EulerFactor(GaloisRepresentation(chi,2));
> loc4:=EulerFactor(GaloisRepresentation(ArtinRepresentation(chi),2));
> [PolynomialRing(ComplexField(5))| loc1,loc2,loc3,loc4];
[
(0.50000 - 0.86603*$.1)*$.1 + 1.0000,
(0.50000 - 0.86603*$.1)*$.1 + 1.0000,
(0.50000 - 0.86603*$.1)*$.1 + 1.0000,
(0.50000 - 0.86603*$.1)*$.1 + 1.0000
]
```

`GaloisRepresentation(A, p): ArtRep, RngIntElt -> GalRep`

Precision: RngIntElt	Default: 40
Minimize : BoolElt	Default: false

Local Galois representation at p of an Artin representation A. This is the representation of a decomposition group at p of $\mathrm{Gal}(\bar{\mathbb{Q}}/\mathbb{Q})$ on the *dual* vector space of A. (The reason for the dual is that, by our convention, the global and the local Euler factors agree; see §*Conventions* and Example *Local Components of a Dirichlet Character*.) If `Minimize` is `true`, choose the field through which it factors to be as small as possible (automatic for faithful representations).

Example: Local Components of an Artin Representation (ex-e01700)

Local components of an Artin representation. We take the Trinks' polynomial $x^7 - 7x - 3$ with Galois group $\mathrm{PSL}(2,7)$ over $\mathbb{Q}$, one of its 7-dimensional representations A of conductor $3^8 7^8$, and compute its local components over $\mathbb{Q}_2$, $\mathbb{Q}_3$, $\mathbb{Q}_5$, and $\mathbb{Q}_7$.

```
> R<x>:=PolynomialRing(Rationals());
> K:=NumberField(x^7-7*x-3);
> GroupName(GaloisGroup(K));
PSL(2,7)
> A:=ArtinRepresentations(K)[5];
> GaloisRepresentation(A,2);
7-dim unramified Galois representation (7,0,0,0,0,0,0) with G=C7, I=C1
   over Q2[40]
> GaloisRepresentation(A,3);
7-dim Galois representation (7,-1,1) with G=S3, I=S3, conductor 3^8 over Q3[40]
> GaloisRepresentation(A,5);
7-dim unramified Galois representation (7,0,0,0,0,0,0) with G=C7, I=C1
   over Q5[40]
> GaloisRepresentation(A,7);
7-dim Galois representation (7,1,1,0,0) with G=C7:C3, I=C7:C3, conductor 7^8
   over Q7[40]
> Conductor(A) eq 3^8*7^8;
true
```

`GaloisRepresentation(E): CrvEll -> GalRep`

Minimize: BoolElt	Default: true

Local Galois representation of (the first l-adic étale cohomology group of) an elliptic curve over a p-adic field. If `Minimize` is `true` (default), choose the field through which it factors to be as small as possible.

Example: Galois Representation of a Local Elliptic Curve (ex-29c470)

Take an elliptic curve $E/\mathbb{Q}_5$, with additive (potentially good) reduction of type II.

```
> K:=pAdicField(5,20);
> E:=EllipticCurve([K|0,5]);
> E;
Elliptic Curve defined by y^2 = x^3 + O(5^20)*x + (5 + O(5^21))
   over pAdicField(5, 20)
> loc:=LocalInformation(E); loc;
<5 + O(5^21), 2, 2, 1, II, true>
```

Its Galois representation is an unramified twist of a representation with finite image that factors through the dihedral extension $\mathbb{Q}_5(\zeta_6, \sqrt[6]{5})$ of $\mathbb{Q}_5$.

```
> A:=GaloisRepresentation(E); A;
2-dim Galois representation Unr(sqrt(5)*i)*(2,-2,0,0,-1,1) with G=D6, I=C6,
   conductor 5^2 over Q5[20]
> Field(A);
Totally ramified extension defined by the polynomial x^6 - 5
 over Unramified extension defined by the polynomial x^2 + 4*x + 2
 over 5-adic field mod 5^20
```

`GaloisRepresentation(E, p): CrvEll, RngIntElt -> GalRep`

```
Precision: RngIntElt                    Default: 40
Minimize : BoolElt                      Default: true
```

Local Galois representation of (the first l-adic étale cohomology group of) an elliptic curve over $\mathbb{Q}$ at p. If `Minimize` is `true` (default), choose the field through which it factors to be as small as possible.

Example: Elliptic Curve 20a1 at Several Primes (ex-94d575)

We take the elliptic curve 20a1 over $\mathbb{Q}$ and compute its local Galois representation at a prime $p = 3$ of good reduction, $p = 5$ of (non-split) multiplicative reduction and $p = 2$ of additive reduction.

```
> E:=EllipticCurve("20a1");
> GaloisRepresentation(E,3);
2-dim unramified Galois representation Unr(1+2*x+3*x^2) over Q3[40]
> GaloisRepresentation(E,5);
2-dim Galois representation Unr(-5)*SP(2) over Q5[40]
> GaloisRepresentation(E,2);
2-dim Galois representation Unr(sqrt(2)*i)*(2,0,-1) with G=S3, I=C3, conductor
   2^2 over Q2[40]
> EulerFactor($3),EulerFactor($2),EulerFactor($1);
3*x^2 + 2*x + 1
x + 1
1
```

`GaloisRepresentation(E, P): CrvEll, RngOrdIdl -> GalRep`

```
Precision: RngIntElt                    Default: 40
Minimize : BoolElt                      Default: true
```

Local Galois representation of (the first l-adic étale cohomology group of) an elliptic curve E over a number field F at a given prime ideal P. If `Minimize` is `true` (default), choose the field through which it factors to be as small as possible.

Example: Elliptic Curve over a Cyclotomic Field (ex-f7aec3)

```
> K:=CyclotomicField(5);
> E:=BaseChange(EllipticCurve("75a1"),K);
> P:=Ideal(Decomposition(K,5)[1,1]);
> GaloisRepresentation(E,P);
2-dim Galois representation Unr(sqrt(5)*i)*(2,0,-1) with G=S3, I=C3, conductor
   pi^2 over ext<Q5[10]|x^4-15*x^3-40*x^2-90*x-45>
```

`GaloisRepresentation(C): CrvHyp[FldPad] -> GalRep`

```
Degree  : RngIntElt                     Default: Infinity()
Minimize: BoolElt                       Default: false
```

Galois representation associated to (H^1 of) a hyperelliptic curve C over a p-adic field. `Degree` specifies that Euler factors of unramified pieces should only be computed up to that degree. (See

Example *Representations Not Fully Computed*.) Setting `Minimize:=true` forces the representation to be minimized. (See *Minimize*.)

Example: Hyperelliptic Curve over a p-adic Field (ex-1fdb67)

We take a curve C over $K = \mathbb{Q}_{23}$ of conductor 23^2 and compute its Galois representation.

```
> K:=pAdicField(23,20);
> R<x>:=PolynomialRing(K);
> C:=HyperellipticCurve(-x,x^3+x^2+1);    // genus 2, conductor 23^2
> A:=GaloisRepresentation(C); A;
4-dim Galois representation Unr(1-46*x+529*x^2)*SP(2) over Q23[20]
```

If F/K is a finite extension, then the base change of A to F is the same as the Galois representation of C/F:

```
> F:=ext<K|2>;
> BaseChange(A,F);
4-dim Galois representation Unr(1-1058*x+279841*x^2)*SP(2) over ext<Q23[20]|2>
> GaloisRepresentation(BaseChange(C,F));
4-dim Galois representation Unr(1-1058*x+279841*x^2)*SP(2) over ext<Q23[20]|2>
```

`GaloisRepresentation(C, p): CrvHyp[FldRat], RngIntElt -> GalRep`

```
Degree  : RngIntElt                    Default: Infinity()
Minimize: BoolElt                      Default: false
```

Galois representation associated to (H^1 of) a hyperelliptic curve $C/\mathbb{Q}$ at p. `Degree` specifies that Euler factors of unramified pieces should only be computed up to that degree. (See Example *Representations Not Fully Computed*.) Setting `Minimize:=true` forces the representation to be minimized. (See *Minimize*.)

Example: Hyperelliptic Curve over the Rationals (ex-1f2ef0)

```
> R<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve((x^2+5)*(x+1)*(x+2)*(x+3));
> GaloisRepresentation(C,5);   // bad reduction
4-dim Galois representation Unr(1+2*x+5*x^2) + Unr(5)*SP(2) over Q5[20]
> GaloisRepresentation(C,11);  // good reduction
4-dim unramified Galois representation Unr(1-2*x+6*x^2-22*x^3+121*x^4)
   over Q11[5]
> GaloisRepresentation(C,997: Degree:=1);  // don't count pts over GF(997^2)
4-dim unramified Galois representation Unr(1+26*x+O(x^2)) over Q997[5]
```

`GaloisRepresentation(C, P): CrvHyp[FldNum], RngOrdIdl -> GalRep`

```
Degree  : RngIntElt                    Default: Infinity()
Minimize: BoolElt                      Default: false
```

Galois representation associated to (H^1 of) a hyperelliptic curve C over a number field at a prime ideal P. `Degree` specifies that Euler factors of unramified pieces should only be computed up to that degree. (See Example *Representations Not Fully Computed*.) Setting `Minimize:=true` forces the representation to be minimized. (See *Minimize*.)

Example: Hyperelliptic Curve over a Cyclotomic Field (ex-e4e4fd)

We take a curve of genus 4 over $\mathbb{Q}(\zeta_{11})$ and compute its Galois representation at a unique prime P above 11.

```
> K<zeta>:=CyclotomicField(11);
> R<x>:=PolynomialRing(K);
> C:=HyperellipticCurve(x^9+x^2+(zeta-1));
> P:=Ideal(Decomposition(K,11)[1,1]);
> GaloisRepresentation(C,P);
8-dim Galois representation Unr(1-44*x^3+1331*x^6) + Unr(11)*SP(2)
   over ext<Q11[2]|x^10+22*x^9+55*x^8+44*x^7-33*x^6-22*x^5-22*x^4-33*x^3+44*x^2+
   55*x+11>
```

GaloisRepresentation(f, p): ModFrmElt, RngIntElt -> GalRep

```
Precision: RngIntElt                    Default: 40
```

Local Galois representation at p of a modular form f. Currently only implemented when p^2 does not divide the level.

Example: Local Representations of a Modular Form (ex-e7791f)

We take a rational modular form of weight 4 and level 5, and compute its Galois representations at $p = 3$ (unramified principal series) and $p = 5$ (Steinberg).

```
> f:=Newforms("5k4")[1,1];
> GaloisRepresentation(f,3);
2-dim unramified Galois representation Unr(1-2*x+27*x^2) over Q3[40]
> GaloisRepresentation(f,5);
2-dim Galois representation Unr(-25)*SP(2) over Q5[40]
```

58.3 Basic Invariants

BaseField(A): GalRep -> FldPad

The base field K of the Galois representation $A : \mathrm{Gal}(\bar{K}/K) \to \mathrm{GL}_m(\mathbb{C})$.

Degree(A): GalRep -> RngIntElt

Dimension(A): GalRep -> RngIntElt

Degree (=dimension) m of a Galois representation $A : \mathrm{Gal}(\bar{K}/K) \to \mathrm{GL}_m(\mathbb{C})$

Example: Degree and Base Field of Representations (ex-890f18)

```
> K:=pAdicField(3,20);
> R<x>:=PolynomialRing(K);
> F:=ext<K|x^3-3>;
> list:=GaloisRepresentations(F,K);
> forall{A: A in list | BaseField(A) eq K};
true
> [Degree(A): A in list];
[ 1, 1, 2 ]
```

Group(A): GalRep -> GrpPerm

GaloisGroup(A): GalRep -> GrpPerm

Finite Galois group $\mathrm{Gal}(F/K)$ that computes the finite part of a Galois representation, where F is `Field(A)` and K is `BaseField(A)`.

Example: Galois and Inertia Groups of Representations (ex-baedd9)

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> list1:=[PrincipalCharacter(K),CyclotomicCharacter(K),SP(K,3)];
```

(continues on next page)

(continued from previous page)

```
> [Degree(A): A in list1];
[ 1, 1, 3 ]
> [GroupName(Group(A)): A in list1];
[ C1, C1, C1 ]
> list2:=GaloisRepresentations(x^4-2);
> [Degree(A): A in list2];
[ 1, 1, 1, 1, 2 ]
> [GroupName(Group(A)): A in list2];
[ D4, D4, D4, D4, D4 ]
> list1[1] eq list2[1];
true
```

`FrobeniusElement(A): GalRep -> GrpPermElt`

An arithmetic Frobenius element of `Group(A)` for a Galois representation A.

Example: Frobenius Element of an Unramified Extension (ex-abc80d)

Take $K = \mathbb{Q}_2$ and $F = \mathbb{Q}_2(\zeta_5)$, a degree 4 unramified extension of K. A Frobenius element $\mathrm{Frob} \in \mathrm{Gal}(F/K)$ is characterized by the property that $\mathrm{Frob}(x) \equiv x^q \mod m_F$. In this example, $q = 2$ (size of the residue field of K).

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> A:=GaloisRepresentations(x^4+x^3+x^2+x+1)[4]; A;
1-dim unramified Galois representation (1,-1,-zeta(4)_4,zeta(4)_4)
   with G=C4, I=C1 over Q2[20]
> frob:=FrobeniusElement(A); frob;
(1, 2, 4, 3)
> F<u>:=Field(A);
> Valuation(Automorphism(A,frob)(u)-u^2) gt 0;
true
```

`Character(A): GalRep -> AlgChtrElt`

Character of the finite part of a Galois representation A. For this to be well-defined, A must have only one component (and not several components with different characters).

Example: Character of a Galois Representation (ex-ce7d20)

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> A1:=PrincipalCharacter(K);
> Character(A1);
( 1 )
> A2:=CyclotomicCharacter(K);
> Character(A2);
( 1 )
> A3:=PermutationCharacter(ext<K|3>,K);
> Character(A3);
( 3, 0, 0 )
> A1+A2+A3;
5-dim unramified Galois representation Unr(1-3/2*x+1/2*x^2) + (3,0,0)
   with G=C3, I=C1 over Q2[20]
```

`Field(A): GalRep -> FldPad`

Given a Galois representation A, return the p-adic field F such that the finite part of A factors through $\mathrm{Gal}(F/K)$, $K =$ `BaseField(A)`.

`DefiningPolynomial(A): GalRep -> RngUPolElt`

For a Galois representation A over K, returns a polynomial over K whose splitting field is F =`Field(A)`. The Galois group $\mathrm{Gal}(F/K)$ is represented as a permutation group `Group(A)` on the roots of this polynomial.

Example: Defining Polynomial of a Representation (ex-216918)

In this example $K = \mathbb{Q}_5$ and F/K is a dihedral extension of degree 12, represented as a splitting field of a degree 6 polynomial.

```
> K:=pAdicField(5,20);
> E:=EllipticCurve([K|0,5]);
> A:=GaloisRepresentation(E); A;
2-dim Galois representation Unr(sqrt(5)*i)*(2,-2,0,0,-1,1) with G=D6, I=C6,
   conductor 5^2 over Q5[20]
> Field(A);
Totally ramified extension defined by the polynomial x^6 - 5
 over Unramified extension defined by the polynomial x^2 + 4*x + 2
 over 5-adic field mod 5^20
> DefiningPolynomial(A);
x^6 + O(5^20)*x^5 + O(5^20)*x^4 + O(5^20)*x^3 + O(5^20)*x^2 + O(5^20)*x - 5 +
   O(5^20)
```

`Automorphism(A, g): GalRep, GrpPermElt -> Map`

The automorphism of `Field(A)`/`BaseField(A)` given by g.

Example: Field and Automorphisms of a Representation (ex-3bde84)

In this example $K = \mathbb{Q}_5$ and F/K is a dihedral extension of degree 12, represented as a splitting field of the polynomial $x^6 - 5$.

```
> R<x>:=PolynomialRing(Rationals());
> K:=NumberField(x^6-5);
> a:=ArtinRepresentations(K)[6];
> A:=GaloisRepresentation(a,5); A;
2-dim Galois representation (2,-2,0,0,-1,1) with G=D6, I=C6, conductor 5^2
   over Q5[40]
> F:=Field(A); F;
Totally ramified extension defined by the polynomial x^6 - 5
 over Unramified extension defined by the polynomial x^2 + 4*x + 2
 over 5-adic field mod 5^40
> DefiningPolynomial(A);
x^6 - 5
```

The 12 elements $\sigma \in \mathrm{Gal}(F/K) \cong D_6$ act on $\pi = \sqrt[6]{5}$ by multiplying it by 6th roots of unity, with $\sigma(\pi) = \pi$ for 2 of them, and $v(\sigma(\pi) - \pi) = 1$ for the other 10.

```
> pi:=UniformizingElement(F);
> autF:=[* Automorphism(A,g): g in Group(A) *];
> [Valuation(sigma(pi)-pi): sigma in autF];
[ 241, 1, 1, 1, 1, 1, 240, 1, 1, 1, 1, 1 ]
```

`EulerFactor(A): GalRep -> RngUPolElt`

```
R: Rng                          Default:
```

Euler factor (=local polynomial) of a Galois representation A over a p-adic field K. It is defined by

$$P(T) = \det(1 - \mathrm{Frob}_K^{-1}\, T | A^{I_K}),$$

and has degree `Dimension(A)` if and only if A is unramified. The coefficient ring of P (rational/complex/cyclotomic field) may be specified with the optional parameter R.

Example: Euler Factors of Galois Representations (ex-641625)

```
> G<chi>:=DirichletGroup(5);
> A:=GaloisRepresentation(chi,2);
> EulerFactor(A);
x + 1
> A:=GaloisRepresentation(chi,5);
> EulerFactor(A);
1
```

`IsZero(A): GalRep -> BoolElt`

Return `true` if A is the Galois representation 0.

`IsOne(A): GalRep -> BoolElt`

Return `true` if A is the trivial 1-dimensional Galois representation.

`Factorization(A): GalRep -> List, GalRep`

Returns the list of tuples $\langle \chi_i, n_i, \rho_i \rangle$, where A is the direct sum over i of twists by $\mathtt{SP}(n_i)$ by unramified representations with Euler factor χ_i, and a finite Weil representation given by a character ρ_i of `Group(A)`.

Example: Factorization of a Galois Representation (ex-d8fc15)

```
> R<x>:=PolynomialRing(ComplexField());    // prettier print for complex polys
> K:=pAdicField(2,20);
> S:=SP(K,2);
> S; Factorization(S);
2-dim Galois representation SP(2) over Q2[20]
[*
<-x + 1, 2, ( 1 )>
*]
> A:=Semisimplification(S);
> A; Factorization(A);
2-dim unramified Galois representation Unr(1-3/2*x+1/2*x^2) over Q2[20]
[*
<1/2*x^2 - 3/2*x + 1, 1, ( 1 )>
*]
> [Factorization(I)[1]: I in Decomposition(A)];
[ <-x + 1, 1, ( 1 )>, <-1/2*x + 1, 1, ( 1 )> ]
```

58.3.1 Ramification

`InertiaGroup(A): GalRep -> GrpPerm`

For a Galois representation A over a p-adic field K this is the image of inertia $I_K \subset \mathrm{Gal}(\bar{K}/K)$ under the semisimplification of A. Equivalently, if

$$A = \psi \otimes \mathtt{SP}(n) \otimes R,$$

as in §*Notation and Printing*, with ψ unramified and R a representation of a finite Galois group $\mathrm{Gal}(F/K)$, this is the image of the inertia subgroup of $\mathrm{Gal}(F/K)$ under R. If F is chosen to be minimal possible (so that R is faithful), then `InertiaGroup(A)` simply *is* the inertia subgroup of $\mathrm{Gal}(F/K)$.

Example: Inertia Groups of Irreducible Representations (ex-e156fa)

Take $K = \mathbb{Q}_3$ and $F = K(\zeta_6, \sqrt[6]{3})$, a D_6-extension of K. For each of the 6 irreducible representations of $\mathrm{Gal}(F/K)$ we compute their inertia (=ramification) groups:

```
> K:=pAdicField(3,20);
> R<x>:=PolynomialRing(K);
> list:=GaloisRepresentations(x^6-3);
> [GroupName(InertiaGroup(A)): A in list];
[ C1, C2, C1, C2, S3, S3 ]
```

`InertiaGroup(A, n): GalRep, RngIntElt -> GrpPerm`

The nth (lower) ramification subgroup of `InertiaGroup(A)`.

Example: Higher Ramification Groups (ex-92ac63)

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> list:=GaloisRepresentations(x^8-2);
> a:=list[#list]; a;
2-dim Galois representation (2,-2,0,0,0,zeta(8)_8^3+zeta(8)_8,
   -zeta(8)_8^3-zeta(8)_8) with G=SD16, I=SD16, conductor 2^10 over Q2[20]
> [GroupName(InertiaGroup(a,n)): n in [1..17]];
[ SD16, SD16, C8, C8, C4, C4, C4, C4, C2, C2, C2, C2, C2, C2, C2, C2, C1 ]
```

`IsUnramified(A): GalRep -> BoolElt`

Return `true` if a Galois representation is unramified.

Example: Testing Whether a Representation Is Unramified (ex-57fb0d)

```
> K:=pAdicField(2,20);
> IsUnramified(CyclotomicCharacter(K));
true
> IsUnramified(SP(K,2));
false
> IsUnramified(Semisimplification(SP(K,2)));
true
```

`IsRamified(A): GalRep -> BoolElt`

Return `true` if a Galois representation is ramified.

Example: Ramification of a Galois Representation (ex-27c5ab)

```
> K:=pAdicField(2,20);
> IsRamified(CyclotomicCharacter(K));
false
> IsRamified(SP(K,2));
true
> IsRamified(Semisimplification(SP(K,2)));
false
```

`IsTamelyRamified(A): GalRep -> BoolElt`

Return `true` if a Galois representation A over a p-adic field K is tamely ramified. Equivalently, `InertiaGroup(A)` has order prime to p (and is then automatically cyclic).

`IsWildlyRamified(A): GalRep -> BoolElt`

Return `true` if a Galois representation A over a p-adic field K is wildly ramified, i.e. not tamely ramified. Equivalently, `InertiaGroup(A)` has non-trivial p-Sylow.

Example: Wild Ramification of Elliptic Curves (ex-1ea277)

Galois representations attached to elliptic curves are always tamely ramified when $p \geq 5$, but may be wildly ramified when $p = 2$ or 3.

```
> E:=EllipticCurve("75a1");
> A5:=GaloisRepresentation(E,5); A5;
2-dim Galois representation Unr(sqrt(5)*i)*(2,0,-1) with G=S3, I=C3, conductor
   5^2 over Q5[40]
> IsWildlyRamified(A5);
false
> E:=EllipticCurve("256a1");
> A2:=GaloisRepresentation(E,2); A2;
2-dim Galois representation Unr(sqrt(2))*(2,-2,0,0,0) with G=D4, I=C4,
   conductor 2^8 over Q2[40]
> IsWildlyRamified(A2);
true
```

`InertiaInvariants(A): GalRep -> GalRep`

Inertia invariants of a Galois representation A. This is an unramified Galois representation.

Example: Inertia Invariants of a Representation (ex-04fb22)

```
> K:=pAdicField(5,20);
> E:=BaseChange(EllipticCurve("15a1"),K);
> A:=GaloisRepresentation(E); A;
2-dim Galois representation Unr(5)*SP(2) over Q5[20]
> I:=InertiaInvariants(A); I;
1-dim trivial Galois representation 1 over Q5[20]
> Dimension(A),Dimension(I);
2 1
```

`ConductorExponent(A): GalRep -> RngIntElt`

Conductor exponent of a Galois representation.

`Conductor(A): GalRep -> FldPadElt`

Conductor of a Galois representation.

Example: Conductor of a Galois Representation (ex-7491ac)

```
> K:=pAdicField(2,40);
> E:=BaseChange(EllipticCurve("256a1"),K);
> A:=GaloisRepresentation(E); A;
2-dim Galois representation Unr(sqrt(2))*(2,-2,0,0,0) with G=D4, I=C4,
   conductor 2^8 over Q2[40]
> ConductorExponent(A);
8
> Conductor(A);
2^8 + O(2^48)
> Conductor(E);    // same, by definition
2^8 + O(2^48)
```

`EpsilonFactor(A): GalRep -> FldComElt`

Epsilon-factor $\epsilon(A)$ of a Galois representation over a p-adic field. Currently only implemented in a few basic cases, and raises an error otherwise. See also Example *Local and Global Epsilon Factors* in §*Example: Local and Global Epsilon Factors for Dirichlet Characters*.

`RootNumber(A): GalRep -> FldComElt`

Root number $\epsilon(A)/|\epsilon(A)|$ of a Galois representation. Currently only implemented in a few basic cases, and raises an error otherwise. See also Example *Local and Global Epsilon Factors* in §*Example: Local and Global Epsilon Factors for Dirichlet Characters*.

Example: Epsilon Factors and Root Numbers (ex-3d8d04)

```
> E:=EllipticCurve("98a1");
> A:=GaloisRepresentation(E,7); A;
2-dim Galois representation Unr(7)*SP(2)*(1,-1) with G=C2, I=C2, conductor 7^2
   over Q7[40]
> RootNumber(A);
-1
> RootNumber(E,7);  // same
-1
```

58.3.2 Semisimplicity and Irreducibles

`IsIrreducible(A): GalRep -> BoolElt`

Return `true` if a Galois representation A is irreducible.

Example: Irreducibility of Local Components (ex-f6575f)

We take the polynomial $x^8 - 6$ with Galois group $C_8 : C_2^2$ over $\mathbb{Q}$, its irreducible 4-dimensional Artin representation A, and compute whether its local components over $\mathbb{Q}_2$, $\mathbb{Q}_3$, $\mathbb{Q}_5$, and $\mathbb{Q}_7$ are irreducible.

```
> R<x>:=PolynomialRing(Rationals());
> K:=NumberField(x^8-6);
> GroupName(GaloisGroup(K));
C8:C2^2
> assert exists(A){A: A in ArtinRepresentations(K) | Degree(A) eq 4};
> A;
Artin representation C8:C2^2: (4,-4,0,0,0,0,0,0,0,0,0,0) of ext<Q|x^8-6>
> [IsIrreducible(GaloisRepresentation(A,p)): p in PrimesUpTo(10)];
[ true, false, false, false ]
```

`IsIndecomposable(A): GalRep -> BoolElt`

Return `true` if a Galois representation A is indecomposable (for semisimple representations, i.e. Weil representations, same as irreducible).

`IsSemisimple(A): GalRep -> BoolElt`

Return `true` if a Galois representation A is semisimple, i.e. a Weil representation.

`Semisimplification(A): GalRep -> GalRep`

Semisimplification of a Galois representation A.

`Decomposition(A): GalRep -> SeqEnum[GalRep]`

Decompose A into indecomposable (for semisimple representations same as irreducible) consituents and return them as a sequence, possibly with repetitions.

Example: Decomposition of a Galois Representation (ex-2ab1fc)

```
> K:=pAdicField(2,20);
> S:=SP(K,2);
> IsIndecomposable(S);
true
> IsIrreducible(S);
false
> IsSemisimple(S);
false
> Decomposition(S);
[ 2-dim Galois representation SP(2) over Q2[20] ]
> Decomposition(Semisimplification(S));
[
```

(continues on next page)

(continued from previous page)

```
1-dim trivial Galois representation 1 over Q2[20],
1-dim unramified Galois representation Unr(1/2) over Q2[20]
]
```

58.4 Arithmetic

`A1 + A2: GalRep, GalRep -> GalRep`

Direct sum of two Galois representations, both defined over the same p-adic field K.

Example: Sums of Galois Representations (ex-1e67b8)

```
> K:=pAdicField(2,20);
> SP(K,1)+SP(K,2)+SP(K,3);
6-dim Galois representation 1 + SP(2) + SP(3) over Q2[20]
```

`A1 - A2: GalRep, GalRep -> BoolElt`

Assuming $A2$ is a Galois subrepresentation of $A1$, compute $A1 - A2$.

Example: Differences of Galois Representations (ex-2242b4)

We take $K = \mathbb{Q}_2$, F its unique degree 3 unramified extension, and compute the regular representation $\mathrm{Gal}(F/K)$ minus the trivial representation, as a Galois representation over K.

```
> K:=pAdicField(2,20);
> F:=ext<K|3>;
> PermutationCharacter(F,K)-PrincipalCharacter(K);
2-dim unramified Galois representation (2,-1,-1) with G=C3, I=C1 over Q2[20]
```

`A1 * A2: GalRep, GalRep -> GalRep`

Tensor product of two Galois representations, both defined over the same p-adic field K.

Example: Tensor Products of Galois Representations (ex-eb562c)

We take $K = \mathbb{Q}_2$ and compute `SP(K,3)`$\otimes$`SP(K,3)`; this is basically the Clebsch-Gordan decomposition.

```
> K:=pAdicField(2,20);
> SP(K,3)*SP(K,3);
9-dim Galois representation Unr(1/4) + Unr(1/2)*SP(3) + SP(5) over Q2[20]
```

`A1 / A2: GalRep, GalRep -> GalRep`

`A1 / A2: RngIntElt, GalRep -> GalRep`

Tensor A_1 with A_2^{-1}, for 1-dimensional A_2. (A_1 may also be the integer 0 or the integer 1.)

`A ^ n: GalRep, Any -> GalRep`

Tensor power of a Galois representation. The power n should be a non-negative integer for a general representation, but may be negative for 1-dimensional representations, and an arbitrary complex number for (powers of) the cyclotomic character.

Example: Powers of Galois Representations (ex-173850)

```
> K:=pAdicField(2,20);
> CyclotomicCharacter(K)^(1/2);
1-dim unramified Galois representation Unr(1/2*sqrt(2)) over Q2[20]
> SP(K,2)^2 / CyclotomicCharacter(K);
4-dim Galois representation 1 + Unr(2)*SP(3) over Q2[20]
```

`A1 eq A2: GalRep, GalRep -> BoolElt`

Return `true` if the two Galois representations are equal.

Example: Equality of Galois Representations (ex-e6a517)

```
> K:=pAdicField(2,20);
> w:=CyclotomicCharacter(K);
> A:=Semisimplification(SP(K,2));
> A eq w^0+w;
true
```

`Determinant(A): GalRep -> GalRep`

Determinant of a Galois representation (a 1-dimensional Galois representation).

Example: Determinant of a Galois Representation (ex-e59efc)

```
> K:=pAdicField(5,20);
> E:=EllipticCurve([K|0,5]);
> A:=GaloisRepresentation(E); A;
2-dim Galois representation Unr(sqrt(5)*i)*(2,-2,0,0,-1,1) with G=D6, I=C6,
   conductor 5^2 over Q5[20]
> Determinant(A) eq CyclotomicCharacter(K)^(-1);
true
```

`TateTwist(A, n): GalRep, RngIntElt -> GalRep`

Tate twist $A(n)$ of a Galois representation. So $A(n) = A \otimes w^{\otimes n}$ where w is the cyclotomic character.

Example: Tate Twists of Galois Representations (ex-a9cca5)

```
> K:=pAdicField(5,20);
> R<x>:=PolynomialRing(K);
> A:=GaloisRepresentations(x^2-2)[2];
> A;
1-dim unramified Galois representation (1,-1) with G=C2, I=C1 over Q5[20]
> TateTwist(A,1);
1-dim unramified Galois representation Unr(1/5)*(1,-1) with G=C2, I=C1
   over Q5[20]
> [EulerFactor(TateTwist(A,n)): n in [-2..2]];
[ 25*x + 1, 5*x + 1, x + 1, 1/5*x + 1, 1/25*x + 1 ]
```

58.5 Changing Precision

`Precision(A): GalRep -> RngIntElt`

Get p-adic precision of the base field of a Galois representation.

`ChangePrecision(~A, Prec): GalRep, RngIntElt -> {}`

Destructively change p-adic precision of the base field of a Galois representation.

`ChangePrecision(A, Prec): GalRep, RngIntElt -> GalRep`

Change p-adic precision of the base field of a Galois representation.

Example: Changing the Precision of a Representation (ex-8bdd36)

```
> K:=pAdicField(5,20);
> R<x>:=PolynomialRing(K);
> A:=GaloisRepresentations(x^4-5)[3];
> A;
```

(continues on next page)

(continued from previous page)

```
1-dim Galois representation (1,1,-1,-1) with G=C4, I=C4, conductor 5^1
   over Q5[20]
> Precision(A);
20
> ChangePrecision(~A,40);
> Precision(A);
40
> ChangePrecision(A,20);
1-dim Galois representation (1,1,-1,-1) with G=C4, I=C4, conductor 5^1
   over Q5[20]
```

58.6 Changing Fields

`Minimize(A): GalRep -> GalRep`

To: GalRep	Default:

Replace `Group(A)` by its smallest possible quotient through which all components of A factor. If `To` is specified, instead replace `Group(A)` by `Group(To)`, assuming the A factors through it.

Example: Minimizing a Galois Representation (ex-b8e822)

We take an extension F of $K = \mathbb{Q}_3$ with Galois group $F_5 = C_5 : C_4$ of order 20. It has five irreducible representations. Four of them are 1-dimensional, and so they actually factor through a smaller Galois group (C_1, C_2, or C_4). Minimize descends them to these Galois groups, although they are of course still the same as representations of the absolute Galois group.

```
> K:=pAdicField(3,20);
> R<x>:=PolynomialRing(K);
> list:=GaloisRepresentations(x^5-3);
> list;
[
1-dim trivial Galois representation 1 over Q3[20],
1-dim unramified Galois representation (1,1,-1,-1,1) with G=F5, I=C5
   over Q3[20],
1-dim unramified Galois representation (1,-1,-zeta(4)_4,zeta(4)_4,1)
   with G=F5, I=C5 over Q3[20],
1-dim unramified Galois representation (1,-1,zeta(4)_4,-zeta(4)_4,1)
   with G=F5, I=C5 over Q3[20],
4-dim Galois representation (4,0,0,0,-1) with G=F5, I=C5, conductor 3^4
   over Q3[20]
]
> [Minimize(A): A in list];
[
1-dim trivial Galois representation 1 over Q3[20],
1-dim unramified Galois representation (1,-1) with G=C2, I=C1 over Q3[20],
1-dim unramified Galois representation (1,-1,-zeta(4)_4,zeta(4)_4)
   with G=C4, I=C1 over Q3[20],
1-dim unramified Galois representation (1,-1,zeta(4)_4,-zeta(4)_4)
   with G=C4, I=C1 over Q3[20],
4-dim Galois representation (4,0,0,0,-1) with G=F5, I=C5, conductor 3^4
   over Q3[20]
]
> forall{A: A in list | A eq Minimize(A)};
true
```

Finally, we illustrate how the parameter To may be used to descend a Galois representation to a specific Galois group, in this case the Galois group $\mathrm{Gal}(F/K) \cong C_4$ of the degree 4 unramified extension of K.

```
> F:=ext<K|4>;                    // Take F = degree 4 unr. ext. of K, and
> B:=PermutationCharacter(F,K);   // any B with BaseField(B)=K, Field(B)=F
> list[2];
1-dim unramified Galois representation (1,1,-1,-1,1) with G=F5, I=C5 over Q3[20]
> Minimize(list[2]: To:=B);
1-dim unramified Galois representation (1,1,-1,-1) with G=C4, I=C1 over Q3[20]
> Minimize(list[2]);
1-dim unramified Galois representation (1,-1) with G=C2, I=C1 over Q3[20]
```

`Restriction(A, L): GalRep, FldPad -> GalRep`

`BaseChange(A, L): GalRep, FldPad -> GalRep`

Base change (restriction) of a Galois representation A over K over a finite extension L/K.

Example: Base Change of a Galois Representation (ex-3dc7d9)

We take a 2-dimensional irreducible representation of $\mathrm{Gal}(\mathbb{Q}_2(\zeta_3, \sqrt[3]{2})) \cong S_3$ and check that its base change to $\mathbb{Q}_2(\zeta_3)$ is reducible.

```
> K:=pAdicField(2,20);
> R<x>:=PolynomialRing(K);
> A:=GaloisRepresentations(x^3-2)[3]; A;
2-dim Galois representation (2,0,-1) with G=S3, I=C3, conductor 2^2 over Q2[20]
> L:=ext<K|2>;
> R:=Restriction(A,L); R;
2-dim Galois representation (2,-1,-1) with G=C3, I=C3, conductor 2^2
   over ext<Q2[20]|2>
> IsIrreducible(A),IsIrreducible(R);
true false
```

`Induction(A, K0): GalRep, FldPad -> GalRep`

Induction of a Galois representation A over K to a subfield $K_0 \subset K$.

Example: Induction of a Galois Representation (ex-aae099)

```
> K0:=pAdicField(2,20);     // K0=Q2
> K:=ext<K0|2>;             // K=Q2(zeta_3)
> R<x>:=PolynomialRing(K);
> A:=GaloisRepresentations(x^3-102)[3];
> A;                        // 1-dim character over K
1-dim Galois representation (1,-zeta(3)_3-1,zeta(3)_3) with G=C3, I=C3,
   conductor 2^1 over ext<Q2[20]|2>
> Induction(A,K0);          // Induced to K0
2-dim Galois representation (2,0,-1) with G=S3, I=C3, conductor 2^2 over Q2[20]
```

58.7 Advanced Examples

58.7.1 Example: Local and Global Epsilon Factors for Dirichlet Characters

Example: Local and Global Epsilon Factors (ex-938a3a)

We illustrate the relation between local and global epsilon-factors for Dirichlet characters (or Artin representations). Pick any Dirichlet character χ; in this example we take the unique odd one of conductor $5 \cdot 19$, order 6, and $\mathrm{Im}\,\chi(2) > 0$.

```
> D:=FullDirichletGroup(5*19);
> L:=[chi: chi in Elements(D) | (Order(chi) eq 6) and
>   (Conductor(chi) eq 5*19) and IsOdd(chi) and (Imaginary(chi(2)) ge 0)];
> assert #L eq 1;
> chi:=L[1];
```

We compute its local Galois representations at all the bad places: 5, 19, and ∞. (See §Hodge Structure for Hodge structures at ∞.)

```
> G5:=GaloisRepresentation(chi,5); G5;
1-dim Galois representation (1,-1,-zeta(3)_3-1,zeta(3)_3,zeta(3)_3+1,-zeta(3)_3)
   with G=C6, I=C2, conductor 5^1 over Q5[40]
> G19:=GaloisRepresentation(chi,19); G19;
1-dim Galois representation (1,-1,zeta(3)_3,-zeta(3)_3-1,zeta(3)_3+1,-zeta(3)_3)
   with G=C6, I=C6, conductor 19^1 over Q19[40]
> Ginfty:=HodgeStructure(ArtinRepresentation(chi)); Ginfty;
Hodge structure of weight 0 given by <0,0,1>
```

Now compute the corresponding local root numbers - the way the conventions are set up for Artin representations, at ∞ it is the root number of the Hodge structure that enters the functional equation.

```
> localrootno:=[ComplexField()| RootNumber(G5),RootNumber(G19),
>               RootNumber(Ginfty)];
> globalrootno:=&*localrootno;
```

The global root number is the sign of the L-series of χ, and we check that it agrees with the one determined numerically from the functional equation:

```
> L:=LSeries(chi);
> ok:=CheckFunctionalEquation(L);
> Sign(L);
0.91074721581647173872399680097 + 0.41296429492456735877069994366 4*I
> globalrootno;
0.91074721581647173872399680108 + 0.41296429492456735877069994364 7*I
```

58.7.2 Example: Reconstructing a Galois Representation from its Euler Factors

Example: Reconstruction from Euler Factors (ex-6f8047)

Every semisimple Galois representation A over a p-adic field K can be uniquely recovered from its Euler factors over the extensions of K ([DD15, Thm. 1]). We illustrate this with a Galois representation attached to an elliptic curve

$$E/K : y^2 = x^3 - 26x, \qquad K = \mathbb{Q}_{13}.$$

```
> K:=pAdicField(13,20);
> E:=EllipticCurve([K|-26,0]);
> A:=GaloisRepresentation(E);
> Degree(A),IsSemisimple(A);
2 true
```

Without looking at A, let us reconstruct it from its Euler factors over extensions of K. First we determine the inertia group `InertiaGroup(A)` by looking for a field over which A is unramified (i.e. E has good reduction). As the residue characteristic is > 3, one of the fields $\mathbb{Q}_{13}(\sqrt[d]{13})$ will do, for $d = 1, 2, 3, 4$ or 6.

```
> R<x>:=PolynomialRing(K);
> [EulerFactor(BaseChange(A,ext<K|x^d-13>)): d in [1,2,3,4,6]];
[ 1, 1, 1, 13*x^2 + 4*x + 1, 1 ]
```

We see that the representation becomes unramified over $L = \mathbb{Q}_{13}(\sqrt[4]{13})$, a cyclic extension of degree 4, but not over its subfields. So the inertia group must be C_4,

$$\text{InertiaGroup}(A) \;\cong\; I_{L/K} \;=\; \text{Gal}(L/K) \;\cong\; C_4.$$

Over L the representation $U = \text{Res}_L\, A$ is unramified, and there it is determined by its Euler factor. It is a sum of two unramified characters, $\text{Frob}_L^{-1} \mapsto -2 \pm 3i$.

```
> L:=ext<K|x^4-13>;
> f1:=EulerFactor(BaseChange(A,L));
> U:=UnramifiedRepresentation(L,f1); U;
2-dim unramified Galois representation Unr(1+4*x+13*x^2) over
   ext<Q13[20]|x^4-13>
> Decomposition(U);
[
1-dim unramified Galois representation Unr(-2+3*i) over ext<Q13[20]|x^4-13>,
1-dim unramified Galois representation Unr(-2-3*i) over ext<Q13[20]|x^4-13>
]
```

We are ready to reconstruct A. Its restriction to inertia is a faithful 2-dimensional representation of C_4, of determinant 1 (as it comes from an elliptic curve), so it must be $\sigma \oplus \sigma^{-1}$ where σ is one of the faithful 1-dimensional characters of C_4.

```
> GroupName(InertiaGroup(A)),IsUnramified(Determinant(A));  // just checking
C4 true
> list:=GaloisRepresentations(x^4-13);
> sigma:=[g: g in list | Order(Character(g)) eq 4][1];
> sigma;
1-dim Galois representation (1,-1,zeta(4)_4,-zeta(4)_4) with G=C4, I=C4,
   conductor 13^1 over Q13[20]
```

Since $\text{Frob}_K = \text{Frob}_L$ commutes with inertia in the Galois group

$$\text{Gal}(K^{nr}(\sqrt[4]{13})/K) \cong \text{Gal}(K^{nr}/K) \times \text{Gal}(L/K) \cong \hat{\mathbb{Z}} \times C_4$$

through which A factors, $A(\text{Frob}_K)$ and $A(\sigma)$ are simultaneously diagonalizable in $\text{GL}_2(\mathbb{C})$, so A must be one of the following two representations A_1 and A_2:

```
> Qi<i>:=CyclotomicField(4);
> A1 := sigma * UnramifiedCharacter(K,-2-3*i) +
>       sigma^(-1)*UnramifiedCharacter(K,-2+3*i);
> A2 := sigma * UnramifiedCharacter(K,-2+3*i) +
>       sigma^(-1)*UnramifiedCharacter(K,-2-3*i);
```

Finally, to determine which one it is, we pick *another* extension of K where A becomes unramified and compare the Euler factors. Then we see that A must be A_2.

```
> L2:=ext<K|x^4-26>;
> EulerFactor(BaseChange(A,L2));
13*x^2 - 6*x + 1
> EulerFactor(BaseChange(A1,L2));
13*x^2 + 6*x + 1
> EulerFactor(BaseChange(A2,L2));
13*x^2 - 6*x + 1
```

(continues on next page)

(continued from previous page)

```
> A eq A2;
true
```

58.8 Bibliography

- [Del79] P. Deligne. Valeurs de fonctions L et périodes d'intégrales. *Proc. Symposia Pure Math.*, 33/2:313–346, 1979.
- [DD13] T. Dokchitser and V. Dokchitser. Identifying Frobenius elements in Galois groups. *Algebra Number Theory*, 7(6):1325–1352, 2013.
- [DD15] T. Dokchitser and V. Dokchitser. Euler factors determine local Weil representations. *J. reine angew. Math.*, 2015. doi:http://dx.doi.org/10.1515/crelle-2014-0013.
- [New12] Rachel Newton. Explicit local reciprocity for tame extensions. *Math. Proc. Cambridge Philos. Soc.*, 152(3):425–454, 2012.
- [Tat79] J. Tate. Number Theoretic Background. *Proc. Symp. Pure Math.*, 33, part 2:3–26, 1979.

References

[Bec07] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007. http://www.ricam.oeaw.ac.at/publications/reports/.

[CL12] X. Caurso and D. Lubicz. Linear algebra over $Z_p[[u]]$ and related rings. *LMS Journal of Computation and Mathematics*, 2012.

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

[Del79] P. Deligne. Valeurs de fonctions L et périodes d'intégrales. *Proc. Symposia Pure Math.*, 33/2:313–346, 1979.

[DD13] T. Dokchitser and V. Dokchitser. Identifying Frobenius elements in Galois groups. *Algebra Number Theory*, 7(6):1325–1352, 2013.

[DD15] T. Dokchitser and V. Dokchitser. Euler factors determine local Weil representations. *J. reine angew. Math.*, 2015. doi:http://dx.doi.org/10.1515/crelle-2014-0013.

[Dor21] Christopher Doris. Exact p-adic computation in Magma. *Journal of Symbolic Computation*, 104:476493, May 2021. URL: http://dx.doi.org/10.1016/j.jsc.2020.08.005, doi:10.1016/j.jsc.2020.08.005.

[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.

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

[JR06] J. W. Jones and D. P. Roberts. A database of local fields. *J. Symb. Comput.*, 41(1):80–97, 2006.

[New12] Rachel Newton. Explicit local reciprocity for tame extensions. *Math. Proc. Cambridge Philos. Soc.*, 152(3):425–454, 2012.

[PR01] S. Pauli and X.-F. Roblot. On the computation of all extensions of a p-adic field of a given degree. *Math. Comp.*, 70(236):1641–1659, 2001.

[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.

[Tat79] J. Tate. Number Theoretic Background. *Proc. Symp. Pure Math.*, 33, part 2:3–26, 1979.

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

Index

Symbols

+ a (*operation*), 116
+ b (*operation*), 75
+ v (*operation*), 110
# L (*operation*), 15
# R (*operation*), 115
- a (*operation*), 66, 116
- b (*operation*), 75
- s (*operation*), 93
- v (*operation*), 110
- x (*operation*), 21, 87
- y (*operation*), 52, 56

A

A !! chi (*operation*), 158
a * b (*operation*), 66, 75, 116
a *:= b (*operation*), 116
a + b (*operation*), 66, 75, 116
a +:= b (*operation*), 116
a / b (*operation*), 66, 75
a - b (*operation*), 66, 75, 116
a -:= b (*operation*), 116
a ^ k (*operation*), 75, 116
A ^ n (*operation*), 170
a ^ n (*operation*), 66
a div b (*operation*), 75, 116
a eq b (*operation*), 66, 75, 116
a in R (*operation*), 75, 116
a mod b (*operation*), 116
a ne b (*operation*), 75, 116
a notin R (*operation*), 75, 116
A1 * A2 (*operation*), 170
A1 + A2 (*operation*), 170
A1 / A2 (*operation*), 170
A1 - A2 (*operation*), 170
A1 eq A2 (*operation*), 170
AbsoluteDegree(F) (*function*), 49
AbsoluteDegree(L) (*function*), 10
AbsoluteGenerator(R) (*function*), 48
AbsoluteInertiaDegree(L) (*function*), 9, 49
AbsoluteInertiaIndex(L) (*function*), 9, 49
AbsolutePrecision(e) (*function*), 87
AbsolutePrecision(f) (*function*), 76
AbsolutePrecision(x) (*function*), 24, 51
AbsoluteRamificationDegree(L) (*function*), 10, 49
AbsoluteRamificationIndex(L) (*function*), 10, 49
AbsoluteRootNumber(K) (*function*), 13
AbsoluteTotallyRamifiedExtension(R) (*function*), 8
Accessing an Algebraic Power Series (*example*), 105
AdditiveGroup(R) (*function*), 12
AlgComb(c,ss) (*function*), 105
AlgebraicPowerSeries(dp, ip) (*function*), 101
AlgebraicPowerSeries(dp, ip, e) (*function*), 101
AlgebraicPowerSeries(dp, ip, L, e) (*function*), 101
All Extensions of Degree 9 over Q3 (*example*), 44
AllExtensions(R, n) (*function*), 43
AllFaces(N) (*function*), 122
AllSlopes(N) (*function*), 124
AllVertices(N) (*function*), 122
Ambient(S) (*function*), 148
Arccos(f) (*function*), 80
Arcsin(f) (*function*), 80
Arctan(f) (*function*), 80
Argcosh(f) (*function*), 81
Argsinh(f) (*function*), 81
Argtanh(f) (*function*), 81
Arithmetic in Series Ring Extensions (*example*), 87
Arithmetic with Exact p-adic Elements (*example*), 53
Arithmetic with Exact p-adic Polynomials (*example*), 57
Arithmetic with Lazy Series (*example*), 94
Arithmetic-Geometric Mean Point Counting (*example*), 28
AssertAttribute(S, "DefaultPrecision", n) (*procedure*), 72
AssignNames(~E, S) (*procedure*), 86
AssignNames(~L, S) (*procedure*), 14, 63
AssignNames(~R, S) (*procedure*), 48, 55, 91
AssignNames(~R, [f]) (*procedure*), 113
AssignNames(~S, ["x"]) (*procedure*), 72
Automorphism(A,g) (*function*), 165
AutomorphismGroup(K, k) (*function*), 39
AutomorphismGroup(L) (*function*), 39, 65
Automorphisms and Inertia Groups (*example*), 65
Automorphisms of a Local Ring (*example*), 39
Automorphisms(K, k) (*function*), 38
Automorphisms(L) (*function*), 38

B

Base Change of a Galois Representation (*example*), 173
BaseChange(A,L) (*function*), 173
BaseField(A) (*function*), 163
BaseField(F) (*function*), 46
BaseField(L) (*function*), 11
BaseRing(E) (*function*), 85
BaseRing(f) (*function*), 55
BaseRing(L) (*function*), 11, 62
BaseRing(R) (*function*), 46, 55, 73, 91
Basic Operations in a Slope Ring (*example*), 142
Basis(S) (*function*), 148
BasisMatrix(S) (*function*), 148
Bernoulli Numbers from Generating Functions (*example*), 79
BigO(f) (*function*), 73
BigO(x) (*function*), 18

C

c * s (*operation*), 94
CanChangeRing(f, R) (*function*), 55
CanonicalElement(S, v) (*function*), 144
Category(a) (*function*), 116
Category(R) (*function*), 73, 115
Category(r) (*function*), 75
Category(V) (*function*), 110
Category(v) (*function*), 110
Center(V) (*function*), 110
Centre(R) (*function*), 115
ChangePrecision(~A,Prec) (*function*), 171
ChangePrecision(~E, r) (*procedure*), 84
ChangePrecision(~f, r) (*procedure*), 76
ChangePrecision(~L, k) (*function*), 15
ChangePrecision(~R, r) (*function*), 74
ChangePrecision(~x, k) (*function*), 24
ChangePrecision(A,Prec) (*function*), 171
ChangePrecision(E, r) (*function*), 84
ChangePrecision(f, r) (*function*), 76
ChangePrecision(L, k) (*function*), 15
ChangePrecision(R, r) (*function*), 74
ChangePrecision(x, k) (*function*), 24
ChangeRing(f, R) (*function*), 55
ChangeRing(L, C) (*function*), 90
ChangeRing(R, C) (*function*), 74
ChangeRing(s,R) (*function*), 107
Changing Precision of Local Structures (*example*), 15
Changing the Precision of a Representation (*example*), 171
Character of a Galois Representation (*example*), 164
Character(A) (*function*), 164
Characteristic(L) (*function*), 14
Characteristic(R) (*function*), 74, 115
Characteristic(V) (*function*), 110
CharacteristicPolynomial(x) (*function*), 27
CharacteristicPolynomial(x, R) (*function*), 27
Checking That a Partial Root Can Be Expanded (*example*), 129
ClassField(m, G) (*function*), 43
Coefficient Ring and Defining Polynomial (*example*), 63
Coefficient(f, i) (*function*), 31, 56, 76
Coefficient(s, i) (*function*), 94
Coefficient(s, T) (*function*), 94
Coefficient(x, i) (*function*), 20
CoefficientField(F) (*function*), 46
CoefficientField(L) (*function*), 11
CoefficientRing(E) (*function*), 85
CoefficientRing(f) (*function*), 55
CoefficientRing(L) (*function*), 11, 62
CoefficientRing(R) (*function*), 46, 55, 73, 91
CoefficientRing(S) (*function*), 141
Coefficients by the Non-Spiral Algorithm (*example*), 96
Coefficients by the Spiral Algorithm (*example*), 95
Coefficients(e) (*function*), 87
Coefficients(f) (*function*), 56, 76
Coefficients(s, l, n) (*function*), 94
Coefficients(s, n) (*function*), 94
Coefficients(x) (*function*), 19
CoefficientsNonSpiral(s, n) (*function*), 96
CoerceAndLift(S, x) (*function*), 52, 56
Completion of a Number Field (*example*), 40
Completion(K, P) (*function*), 40
Completion(O, P) (*function*), 40
Composite(R, S) (*function*), 8
Composition and Reversion of Series (*example*), 78
Composition(f, g) (*function*), 77
Conductor of a Galois Representation (*example*), 168
Conductor(A) (*function*), 168
ConductorExponent(A) (*function*), 168
Constructing an Unramified Extension (*example*), 23
Constructing Exact p-adic Fields (*example*), 46
Constructors for Algebraic Power Series (*example*), 103
Continuations(m, L) (*function*), 39
Convolution(f, g) (*function*), 78
Cos(f) (*function*), 80
Cosh(f) (*function*), 81
Creating Finite Lazy Series (*example*), 92
Creating Lazy Power Series Rings (*example*), 91
Creating Newton Polygons (*example*), 121
Creating p-adic Rings (*example*), 3
Creating Totally Ramified Extensions (*example*), 6
Creating Unramified Extensions (*example*), 5
Cyclotomic Character (*example*), 155

CyclotomicCharacter(K) (*function*), 155
CyclotomicUnramifiedExtension(R, f) (*function*), 5

D

Decomposition of a Galois Representation (*example*), 169
Decomposition(A) (*function*), 169
DecompositionGroup(L) (*function*), 65
Defining a Galois Ring (*example*), 114
Defining Polynomial of a Representation (*example*), 165
DefiningMap(L) (*function*), 11
DefiningPoints(N) (*function*), 120
DefiningPolynomial(A) (*function*), 164
DefiningPolynomial(E) (*function*), 85
DefiningPolynomial(K, L) (*function*), 11
DefiningPolynomial(L) (*function*), 11, 62
DefiningPolynomial(R) (*function*), 49
DefiningPolynomial(s) (*function*), 105
Degree and Base Field of Representations (*example*), 163
Degree(A) (*function*), 163
Degree(f) (*function*), 55, 77
Degree(K, L) (*function*), 10
Degree(L) (*function*), 10, 48, 62
Degree(L, K) (*function*), 48
Degree(L, R) (*function*), 62
Degree(R) (*function*), 115
Degree(S) (*function*), 148
Derivative(f) (*function*), 31, 56, 77
Derivative(f, m) (*function*), 56
Derivative(f, n) (*function*), 77
Derivative(s) (*function*), 97
Derivative(s, v) (*function*), 97
Derivative(s, v, n) (*function*), 97
Determinant of a Galois Representation (*example*), 171
Determinant(A) (*function*), 171
Differences of Galois Representations (*example*), 170
Dimension(A) (*function*), 163
Dimension(S) (*function*), 148
DirectSum(M1, M2) (*function*), 149
Discriminant(f) (*function*), 56
Discriminant(K, k) (*function*), 12
Discriminant(L) (*function*), 64
Discriminant(R) (*function*), 12
DiscriminantValuation(L) (*function*), 49
DiscriminantValuation(L, K) (*function*), 49
Distance(f, g) (*function*), 36
Division of Non-units (*example*), 22
Domain(s) (*function*), 104
Duval's Algorithm for Puiseux Expansions (*example*), 132
DuvalPuiseuxExpansion(f, n) (*function*), 131

E

E . i (*operation*), 85
E1 eq E2 (*operation*), 85
EchelonForm(M) (*function*), 146
Element Constructors and Imprecise Zeros (*example*), 18
Elementary Invariants of Local Rings (*example*), 12
ElementToSequence(a) (*function*), 114
ElementToSequence(e) (*function*), 87
ElementToSequence(f) (*function*), 76
ElementToSequence(x) (*function*), 19
Elliptic Curve 20a1 at Several Primes (*example*), 161
Elliptic Curve over a Cyclotomic Field (*example*), 161
elt< R | v, [a_1, ..., a_d], p > (*constructor*), 73
elt<K | x> (*constructor*), 51
elt<L | u, r> (*constructor*), 18
elt<L | u> (*constructor*), 17
elt<L | v, u, r> (*constructor*), 18
elt<R | m> (*constructor*), 92
elt<R | x> (*constructor*), 51
Eltseq(a) (*function*), 66, 114
Eltseq(e) (*function*), 87
Eltseq(f) (*function*), 76
Eltseq(x) (*function*), 19
End Vertices and Containing Faces (*example*), 123
EndVertices(F) (*function*), 123
Epsilon Factors and Root Numbers (*example*), 168
EpsilonFactor(A) (*function*), 168
Equality of Galois Representations (*example*), 171
Euclidean Operations on Series (*example*), 144
EuclideanNorm(v) (*function*), 111
EuclideanNorm(x) (*function*), 28
Euler Factors of Galois Representations (*example*), 166
EulerFactor(A) (*function*), 165
Evaluate(f, s) (*function*), 77
Evaluate(f, x) (*function*), 31, 56
Evaluate(s, T) (*function*), 97
Evaluate(s, t) (*function*), 97
Evaluating a Lazy Series (*example*), 97
EvaluationPowerSeries(s, nu, v) (*function*), 102
Exact p-adic Polynomial Rings (*example*), 55
ExactPolynomial(f) (*function*), 56
Exp(f) (*function*), 79
Exp(x) (*function*), 26
Expand(s,ord) (*function*), 105
Expand(x) (*function*), 25
ExpandToPrecision(f, c, n) (*function*), 127
ExponentDenominator(f) (*function*), 76, 77

ExponentLattice(s) (*function*), 105
Exponents of Puiseux Expansions (*example*), 130
ext<K | d> (*constructor*), 46
ext<K | f> (*constructor*), 46
ext<L | f> (*constructor*), 4, 6
ext<L | m> (*constructor*), 7
ext<L | n> (*constructor*), 4
ExtendedGcd(A, B) (*function*), 143, 144
ExtendedGreatestCommonDivisor(v, w) (*function*), 111
ExtendedGreatestCommonDivisor(x, y) (*function*), 53

F

f div g (*operation*), 31
f mod g (*operation*), 31
FaceFunction(F) (*function*), 125
Faces of a Newton Polygon (*example*), 122
Faces(N) (*function*), 121
FacesContaining(N,p) (*function*), 123
Factorisation(f) (*function*), 58
Factorisation(f, R) (*function*), 58
Factorization of a Galois Representation (*example*), 166
Factorization over a Local Field (*example*), 67
Factorization over Exact p-adic Fields (*example*), 59
Factorization over Series Rings (*example*), 82
Factorization(A) (*function*), 166
Factorization(f) (*function*), 36, 58, 67, 81
Factorization(f, R) (*function*), 58
Field and Automorphisms of a Representation (*example*), 165
Field(A) (*function*), 164
FieldOfFractions(E) (*function*), 84
FieldOfFractions(L) (*function*), 46
FieldOfFractions(R) (*function*), 8, 74, 115
FieldOfFractions(V) (*function*), 110
FixedField(L, G) (*function*), 65
Frobenius Element of an Unramified Extension (*example*), 164
FrobeniusAutomorphism(L) (*function*), 65
FrobeniusElement(A) (*function*), 164

G

Galois and Inertia Groups of Representations (*example*), 163
Galois Descent of a Polynomial (*example*), 20
Galois Group of a Local Polynomial (*example*), 40
Galois Representation of a Local Elliptic Curve (*example*), 160
GaloisGroup(A) (*function*), 163
GaloisGroup(f) (*function*), 39
GaloisImage(x, i) (*function*), 28
GaloisRepresentation(A,p) (*function*), 159
GaloisRepresentation(C) (*function*), 161
GaloisRepresentation(C,P) (*function*), 162
GaloisRepresentation(C,p) (*function*), 162
GaloisRepresentation(chi,p) (*function*), 159
GaloisRepresentation(E) (*function*), 160
GaloisRepresentation(E,P) (*function*), 161
GaloisRepresentation(E,p) (*function*), 161
GaloisRepresentation(f,p) (*function*), 163
GaloisRepresentations(f) (*function*), 158
GaloisRepresentations(F,K) (*function*), 157
GaloisRing(p, a, D) (*function*), 113
GaloisRing(p, a, d) (*function*), 112
GaloisRing(q, D) (*function*), 113
GaloisRing(q, d) (*function*), 112
GaussValuation(f) (*function*), 141
GaussValuations(M) (*function*), 146
GaussValuations(v) (*function*), 148
GCD(a, b) (*function*), 116
GCD(f, g) (*function*), 31
Gcd(f, g) (*function*), 31
GCD(v, w) (*function*), 111
Gcd(v, w) (*function*), 111
GCD(x, y) (*function*), 52
Gcd(x, y) (*function*), 52
GCDs of p-adic Polynomials (*example*), 31
Generating Elements of Exact p-adic Rings (*example*), 48
Generator(R) (*function*), 48, 55, 113
GetExactpAdicsPrintPrecision() (*function*), 51
GetPrecision(E) (*function*), 85
GetPrecision(R) (*function*), 74
Global and Non-Global Series Rings (*example*), 72
GR(p, a, D) (*function*), 113
GR(p, a, d) (*function*), 112
GR(q, D) (*function*), 113
GR(q, d) (*function*), 112
Gradient Vector and Weight of a Face (*example*), 124
GradientVector(F) (*function*), 124
GradientVectors(N) (*function*), 124
GreatestCommonDivisor(f, g) (*function*), 31
GreatestCommonDivisor(v, w) (*function*), 111
GreatestCommonDivisor(x, y) (*function*), 52
Groebner Bases over Galois Rings (*example*), 117
Group(A) (*function*), 163

H

HasAttribute(S, "DefaultPrecision") (*function*), 72
HasDefiningMap(L) (*function*), 11
HasGNB(L, n, t) (*function*), 5
HasGNB(R, n, t) (*function*), 5
HasPolynomial(N) (*function*), 125
HasPRoot(R) (*function*), 12
HasRoot(f) (*function*), 34, 58, 137

HasRoot(f, R) (*function*), 58
HasRootOfUnity(L, n) (*function*), 12
Hensel Factorization of Polynomials (*example*), 35
HenselLift(f, L) (*function*), 81
HenselLift(f, s) (*function*), 35
HenselLift(f, x) (*function*), 33
HenselLift(f, x, k) (*function*), 33
HermiteForm(M) (*function*), 146
Higher Ramification Groups (*example*), 167
hom< L -> R | a > (*constructor*), 65
hom< L -> R | cfm, a > (*constructor*), 65
Hyperelliptic Curve over a Cyclotomic Field (*example*), 162
Hyperelliptic Curve over a p-adic Field (*example*), 162
Hyperelliptic Curve over the Rationals (*example*), 162
HypergeometricSeries(a,b,c, z) (*function*), 81

I

Identity(Q) (*function*), 73
Identity(R) (*function*), 113
IdentityMatrix(S) (*function*), 145
IdentityMatrix(S, n) (*function*), 145
Image(M) (*function*), 146
ImplicitFunction(dp) (*function*), 102
ImplicitFunction(f, d, n) (*function*), 127
Index(s, i, n) (*function*), 96
Induction of a Galois Representation (*example*), 173
Induction(A,K0) (*function*), 173
Inertia Groups of Irreducible Representations (*example*), 166
Inertia Invariants of a Representation (*example*), 168
InertiaDegree(E) (*function*), 85
InertiaDegree(K, L) (*function*), 9
InertiaDegree(L) (*function*), 9, 49, 62
InertiaDegree(L, K) (*function*), 49
InertiaGroup(A) (*function*), 166
InertiaGroup(A,n) (*function*), 167
InertiaGroup(L) (*function*), 65
InertiaInvariants(A) (*function*), 168
InertialElement(L) (*function*), 66
InfinitePrecisionApproximation(K) (*function*), 47
InnerFaces(N) (*function*), 121
InnerSlopes(N) (*function*), 124
InnerVertices(N) (*function*), 122
IntegerRing(E) (*function*), 85
IntegerRing(F) (*function*), 7
IntegerRing(L) (*function*), 46
IntegerRing(R) (*function*), 74
Integers(E) (*function*), 85
Integers(F) (*function*), 7
Integers(L) (*function*), 46
Integers(R) (*function*), 74
Integral Basis of a Local Field (*example*), 64
Integral(f) (*function*), 77
Integral(s) (*function*), 97
Integral(s, v) (*function*), 97
IntegralBasis(L) (*function*), 64
Invariants of Exact p-adic Fields (*example*), 50
InverseRoot(x, n) (*function*), 30
InverseRoot(x, y, n) (*function*), 30
InverseSqrt(x) (*function*), 29
InverseSqrt(x, y) (*function*), 30
InverseSquareRoot(x) (*function*), 29
InverseSquareRoot(x, y) (*function*), 30
Irreducibility of Local Components (*example*), 169
Irreducible Characters of a Dihedral Extension (*example*), 157
IsAbelian(K, k) (*function*), 39
IsBoundary(N, p) (*function*), 125
IsCommutative(Q) (*function*), 74
IsCommutative(R) (*function*), 115
IsConsistent(M, e) (*function*), 149
IsConsistent(M, v) (*function*), 149
IsConsistent(M, W) (*function*), 149
IsDefinitelyEqual(f, g) (*function*), 56
IsDefinitelyEqual(x, y) (*function*), 52
IsDefinitelyZero(f) (*function*), 56
IsDefinitelyZero(x) (*function*), 52
IsDegenerate(F) (*function*), 125
IsDegenerate(N) (*function*), 125
IsDistinguished(f) (*function*), 141
IsDivisionRing(Q) (*function*), 74
IsDivisionRing(R) (*function*), 115
IsDomain(Q) (*function*), 74
IsDomain(R) (*function*), 115
IsEisenstein(f) (*function*), 6, 56
IsEuclideanDomain(Q) (*function*), 74
IsEuclideanDomain(R) (*function*), 115
IsEuclideanRing(Q) (*function*), 74
IsEuclideanRing(R) (*function*), 115
IsExactlyDivisible(x, y) (*function*), 22
IsExactpAdic(x) (*function*), 45
IsFace(N, F) (*function*), 125
IsField(Q) (*function*), 74
IsField(R) (*function*), 115
IsFinite(Q) (*function*), 74
IsFinite(R) (*function*), 115
IsHenselLiftable(f, x) (*function*), 59
IsIdempotent(a) (*function*), 117
IsIdempotent(n) (*function*), 111
IsIdempotent(x) (*function*), 75
IsIdentical(f, g) (*function*), 75
IsIndecomposable(A) (*function*), 169
IsInertial(f) (*function*), 5, 56
IsIntegral(a) (*function*), 64
IsIntegral(x) (*function*), 24, 52
IsInterior(N,p) (*function*), 125
IsIrreducible(A) (*function*), 169

IsIrreducible(f) (*function*), 35, 59
IsIrreducible(x) (*function*), 75
IsIsomorphic(E, K) (*function*), 39
IsIsomorphic(f, g) (*function*), 36
IsMinusOne(a) (*function*), 66, 75, 116
IsMinusOne(e) (*function*), 87
IsMinusOne(n) (*function*), 111
IsMinusOne(s) (*function*), 97
IsMinusOne(x) (*function*), 23
IsNewtonPolygonOf(N, f) (*function*), 125
IsNilpotent(a) (*function*), 116
IsNilpotent(n) (*function*), 111
IsNilpotent(x) (*function*), 75
IsNondegenerate(F) (*function*), 126
IsNondegenerate(N) (*function*), 126
IsNormal(K) (*function*), 39
IsNormal(K, k) (*function*), 39
IsOne(A) (*function*), 166
IsOne(a) (*function*), 66, 75, 116
IsOne(e) (*function*), 87
IsOne(n) (*function*), 111
IsOne(s) (*function*), 97
IsOne(x) (*function*), 23
IsOrdered(Q) (*function*), 74
IsOrdered(R) (*function*), 115
IsPartialRoot(f, c) (*function*), 129
IsPID(Q) (*function*), 74
IsPID(R) (*function*), 115
IsPoint(N,p) (*function*), 125
IsPolynomial(s) (*function*), 106
IsPower(x, n) (*function*), 30
IsPrime(x) (*function*), 75
IsPrincipalIdealRing(Q) (*function*), 74
IsPrincipalIdealRing(R) (*function*), 115
IsRamified(A) (*function*), 167
IsRamified(L) (*function*), 64
IsRamified(R) (*function*), 16
IsRegular(n) (*function*), 111
IsRegular(x) (*function*), 75
IsSemisimple(A) (*function*), 169
IsSquare(s) (*function*), 98
IsSquare(x) (*function*), 29
IsSubspace(A, B) (*function*), 149
IsTamelyRamified(A) (*function*), 167
IsTamelyRamified(L) (*function*), 64
IsTamelyRamified(R) (*function*), 16
IsTotallyRamified(L) (*function*), 64
IsTotallyRamified(R) (*function*), 16
IsUFD(Q) (*function*), 74
IsUFD(R) (*function*), 115
IsUniquePartialRoot(f, c) (*function*), 129
IsUnit(a) (*function*), 75, 117
IsUnit(e) (*function*), 87
IsUnit(n) (*function*), 111
IsUnit(s) (*function*), 97
IsUnit(x) (*function*), 24, 52
IsUnitary(Q) (*function*), 74
IsUnitary(R) (*function*), 115
IsUnramified(A) (*function*), 167
IsUnramified(L) (*function*), 64
IsUnramified(R) (*function*), 16
IsVertex(N, p) (*function*), 125
IsWeaklyEqual(f, g) (*function*), 56, 75
IsWeaklyEqual(s, t, n) (*function*), 97
IsWeaklyEqual(x, y) (*function*), 52
IsWeaklyZero(a) (*function*), 66
IsWeaklyZero(f) (*function*), 56, 75, 141
IsWeaklyZero(M) (*function*), 146
IsWeaklyZero(s, n) (*function*), 97
IsWeaklyZero(v) (*function*), 149
IsWeaklyZero(x) (*function*), 52
IsWildlyRamified(A) (*function*), 167
IsWildlyRamified(L) (*function*), 64
IsWildlyRamified(R) (*function*), 16
IsZero(A) (*function*), 166
IsZero(a) (*function*), 66, 75, 116
IsZero(e) (*function*), 87
IsZero(n) (*function*), 111
IsZero(s) (*function*), 97, 106
IsZero(x) (*function*), 23
IsZeroDivisor(a) (*function*), 117
IsZeroDivisor(n) (*function*), 111
IsZeroDivisor(x) (*function*), 75

K

K ! x (*operation*), 51
Kernel(M) (*function*), 146

L

L ! r (*operation*), 65
L ! u (*operation*), 17
L . 1 (*operation*), 12
L . i (*operation*), 65
L eq K (*operation*), 15
L ne K (*operation*), 15
L`DefaultPrecision (*attribute*), 8
L`SeriesPrinting (*attribute*), 8
Laplace(f) (*function*), 77
LaurentSeriesRing(R) (*function*), 71
LaurentSeriesRing(R, p) (*function*), 71
LazyPowerSeriesRing(C, n) (*function*), 90
LazySeries(R, f) (*function*), 92
LCM(a, b) (*function*), 116
LeadingCoefficient(f) (*function*), 31, 76
LeadingCoefficient(s) (*function*), 95
LeadingTerm(f) (*function*), 76, 141
LeadingTerm(s) (*function*), 95
LeadingTerms(M) (*function*), 146
LeadingTerms(v) (*function*), 148
LeastCommonMultiple(f, g) (*function*), 31
Local and Global Epsilon Factors (*example*), 173
Local Components of a Dirichlet Character (*example*), 159
Local Components of an Artin Representation (*example*), 160

Local Factorizations and Galois Groups (*example*), 38
Local Representations of a Modular Form (*example*), 163
LocalFactorization(f) (*function*), 36
LocalField(L, f) (*function*), 61
LocalRing(P, k) (*function*), 40
Log(f) (*function*), 79
Log(x) (*function*), 26
LowerFaces(N) (*function*), 122
LowerSlopes(N) (*function*), 124
LowerVertices(N) (*function*), 122

M

M1 + M2 (*operation*), 149
M1 meet M2 (*operation*), 149
Matrices over Sp Rings (*example*), 146
Matrices over Su Rings (*example*), 147
MinimalPolynomial(a) (*function*), 66
MinimalPolynomial(a, F) (*function*), 66
MinimalPolynomial(x) (*function*), 27
MinimalPolynomial(x, R) (*function*), 27
Minimize(A) (*function*), 172
Minimizing a Galois Representation (*example*), 172
Module with a Nontrivial Kernel (*example*), 150
Modules over Sp Rings (*example*), 149
Modules over Su Rings (*example*), 150
Multivariate Lazy Series from a Map (*example*), 93

N

Name(L, i) (*function*), 64
Name(L, k) (*function*), 15
Name(R, 1) (*function*), 113
Name(R, i) (*function*), 47, 55
Name(S, 1) (*function*), 72
Names and Properties of a Lazy Series Ring (*example*), 91
Newton Polygons and Root Valuations (*example*), 32
NewtonPolygon(C) (*function*), 120
NewtonPolygon(f) (*function*), 32, 58, 119, 120
NewtonPolygon(f, p) (*function*), 120
NewtonPolygon(V) (*function*), 120
Norm(a) (*function*), 66
Norm(a, F) (*function*), 66
Norm(m1, m2, G) (*function*), 42
Norm(x) (*function*), 27
Norm(x, R) (*function*), 27
NormEquation(m1, m2, G) (*function*), 42
NormEquation(R, m, b) (*function*), 42
NormGroup(R, m) (*function*), 42
NormGroupDiscriminant(m, G) (*function*), 43
NormKernel(m1, m2) (*function*), 43
NumberOfExtensions(R, n) (*function*), 44

O

O(f) (*function*), 73
O(x) (*function*), 18, 142
One(L) (*function*), 17
One(Q) (*function*), 73
One(R) (*function*), 113
One-Dimensional Characters of a Cyclotomic Extension (*example*), 158
One-Step Extension from a Two-Step Field (*example*), 61
Operations on Series Ring Extensions (*example*), 86
OptimisedRepresentation(E) (*function*), 88
Optimized Representation of an Extension (*example*), 88
OptimizedRepresentation(E) (*function*), 88
Order(s) (*function*), 105
OreConditions(R, n, j) (*function*), 44
OuterFaces(N) (*function*), 122
OuterVertices(N) (*function*), 122

P

pAdicField(L) (*function*), 11
pAdicField(p : parameters) (*function*), 45
pAdicField(p) (*function*), 3
pAdicField(p, k) (*function*), 3
pAdicQuotientRing(L, k) (*function*), 47
pAdicQuotientRing(p, k) (*function*), 3
pAdicRing(L) (*function*), 11
pAdicRing(p : parameters) (*function*), 45
pAdicRing(p) (*function*), 3
pAdicRing(p, k) (*function*), 3
ParametrizationToPuiseux(T) (*function*), 132
Parent(a) (*function*), 116
Parent(f) (*function*), 141
Parent(R) (*function*), 73, 115
Parent(r) (*function*), 75
Parent(V) (*function*), 110
Parent(v) (*function*), 110, 148
Parent(x) (*function*), 24
ParentRing(N) (*function*), 125
Permutation Character of a Cubic Extension (*example*), 158
PermutationCharacter(F,K) (*function*), 158
Pitfalls of Unbounded Precision (*example*), 25
Polynomial(N) (*function*), 125
PolynomialCoefficient(s, i) (*function*), 98
PolynomialRing(R : parameters) (*function*), 54
PolyToSeries(s) (*function*), 101
PowerRelation(x,n) (*function*), 28
Powers of Galois Representations (*example*), 170
PowerSeriesRing(R) (*function*), 71
PowerSeriesRing(R, p) (*function*), 71
Precision When Coercing Sequences (*example*), 19
Precision(A) (*function*), 171

Precision(E) (*function*), 85
Precision(L) (*function*), 12, 62
Precision(R) (*function*), 74
Precision(S) (*function*), 141
Precision(x) (*function*), 24
Prime(L) (*function*), 9, 48, 62
PrimeField(L) (*function*), 11
PrimeField(R) (*function*), 115
PrimeRing(L) (*function*), 11
PrimeRing(R) (*function*), 115
PrimeRing(V) (*function*), 110
Principal Character of a Local Field (*example*), 155
PrincipalCharacter(K) (*function*), 155
PrincipalUnitGroup(R) (*function*), 42
PrincipalUnitGroupGenerators(R) (*function*), 41
Printing Elements as Series (*example*), 8
Printing Galois Representations (*example*), 153
PrintTermsOfDegree(s, l, n) (*procedure*), 95
PrintToPrecision(s, n) (*procedure*), 94
pSelmerGroup(p,F) (*function*), 42
Puiseux Expansions to Higher Precision (*example*), 128
PuiseuxExpansion(f, n) (*function*), 126
PuiseuxExponents(p) (*function*), 130
PuiseuxExponentsCommon(p, q) (*function*), 130
PuiseuxSeriesRing(R) (*function*), 71
PuiseuxSeriesRing(R, p) (*function*), 71
PuiseuxToParametrization(S) (*function*), 132

Q

quo<L | x> (*constructor*), 3, 47
QuotientRepresentation(L) (*function*), 63
Quotrem(A, B) (*function*), 143
Quotrem(a, b) (*function*), 116
Quotrem(v, w) (*function*), 111
Quotrem(x, y) (*function*), 52, 57

R

R ! a (*operation*), 113
R ! c (*operation*), 92
R ! f (*operation*), 55
R ! S (*operation*), 92
R ! s (*operation*), 73, 92
R ! x (*operation*), 51
r + s (*operation*), 93
R . 1 (*operation*), 73, 113
R . i (*operation*), 47, 55, 91
R eq G (*operation*), 115
R eq S (*operation*), 74, 140
R eq T (*operation*), 47, 55
R ne G (*operation*), 115
R ne S (*operation*), 74, 140
R1 eq R2 (*operation*), 91
Ramification of a Galois Representation (*example*), 167
RamificationDegree(E) (*function*), 85
RamificationDegree(K, L) (*function*), 10
RamificationDegree(L) (*function*), 10, 49, 62
RamificationDegree(L, K) (*function*), 49
RamificationGroup(L, i) (*function*), 65
RamificationIndex(E) (*function*), 85
RamificationIndex(K, L) (*function*), 10
RamificationIndex(L) (*function*), 10, 49, 62
RamificationIndex(L, K) (*function*), 49
RamificationPolygon(f) (*function*), 49
RamificationPolygon(L) (*function*), 49
RamificationPolygon(L, K) (*function*), 49
RamificationResidualPolynomial(f, face) (*function*), 59
RamificationResidualPolynomials(f) (*function*), 59
Ramified Quadratic Extensions of Z2 (*example*), 34
RamifiedRepresentation(L) (*function*), 63
Random(L) (*function*), 17
Random(R) (*function*), 113
Rank(R) (*function*), 91
RationalPuiseux(p) (*function*), 102
Reconstruction from Euler Factors (*example*), 174
Relative and Absolute Precision (*example*), 25
RelativeField(L, m) (*function*), 64
RelativePrecision(a) (*function*), 66
RelativePrecision(e) (*function*), 87
RelativePrecision(f) (*function*), 76
RelativePrecision(x) (*function*), 24, 51
Replacing the Character of a Representation (*example*), 159
RepresentationMatrix(a) (*function*), 66
Representations Not Fully Computed (*example*), 156
Representative(L) (*function*), 17
Representative(Q) (*function*), 73
Representative(R) (*function*), 113
ResidueClassField(E) (*function*), 85
ResidueClassField(L) (*function*), 11, 47, 64
ResidueClassField(R) (*function*), 74
ResidueField(R) (*function*), 115
ResidueGenerator(R) (*function*), 48
ResidueSystem(R) (*function*), 12
Restriction(A,L) (*function*), 173
Resultant(f, g) (*function*), 56
Reverse(f) (*function*), 77
Reversion(f) (*function*), 77
RingOfIntegers(E) (*function*), 85
RingOfIntegers(F) (*function*), 7
RingOfIntegers(L) (*function*), 46
RingOfIntegers(R) (*function*), 8, 74
Root Numbers of p-adic Extensions (*example*), 14
Root(x, n) (*function*), 30
RootNumber(A) (*function*), 168

RootNumber(K) (*function*), 13
Roots of Polynomials over Puiseux Series (*example*), 137
Roots(f) (*function*), 34, 58, 67, 136
Roots(f, n) (*function*), 136
Roots(f, R) (*function*), 34, 58, 67

S

s * c (*operation*), 94
S * M (*operation*), 149
s * n (*operation*), 94
s * t (*operation*), 93, 106
s + r (*operation*), 93
s + t (*operation*), 93, 106
S . 1 (*operation*), 72
s - t (*operation*), 93, 106
s ^ n (*operation*), 94
s eq t (*operation*), 97, 106
ScaleGenerators(s,ls) (*function*), 107
Scaling the Generators of a Series (*example*), 107
Semisimplification(A) (*function*), 169
SetExactpAdicsPrintPrecision(k) (*procedure*), 51
SetVerbose("Newton", v) (*procedure*), 126
ShiftValuation(f, n) (*function*), 31
ShiftValuation(x, n) (*function*), 51
SimplifyRep(s) (*function*), 107
Sin(f) (*function*), 80
Sincos(f) (*function*), 80
Sinh(f) (*function*), 81
Slope(S) (*function*), 141
Slopes(N) (*function*), 124
SmithForm(M) (*function*), 146
SnuRing(F) (*function*), 140
SnuRing(F, nu) (*function*), 140
SnuRing(p) (*function*), 140
SnuRing(p, e) (*function*), 140
SnuRing(S) (*function*), 140
SnuRing(S, nu) (*function*), 140
SP(K,f,n) (*function*), 157
SP(K,n) (*function*), 156
Special Galois Representations (*example*), 157
Splitting Field over a p-adic Field (*example*), 38
SplittingField(f) (*function*), 38
SplittingField(f, R) (*function*), 8
SpMatrix(A) (*function*), 145
SpMatrix(r, c, A) (*function*), 145
SpMatrix(v) (*function*), 145
SpMatrixSpace(S, r, c) (*function*), 145
SpRing(F) (*function*), 140
SpRing(F, nu) (*function*), 140
SpRing(p) (*function*), 140
SpRing(p, e) (*function*), 140
SpRing(S) (*function*), 140
SpRing(S, nu) (*function*), 140
SpSpace(M) (*function*), 148
SpSpace(R, n) (*function*), 148
SpSpace(v) (*function*), 148
SpVector(e) (*function*), 148
Sqrt(f) (*function*), 77
Sqrt(s) (*function*), 98
Sqrt(x) (*function*), 29
Square Roots by Hensel Lifting (*example*), 33
SquareFreeFactorization(f) (*function*), 35
SquareRoot(f) (*function*), 77
SquareRoot(s) (*function*), 98
SquareRoot(x) (*function*), 29
Structures Related to Exact p-adic Rings (*example*), 47
sub< L | a_1, ..., a_n > (*constructor*), 62
sub< L | S> (*constructor*), 62
Subfield Generated by an Element (*example*), 62
Substituting Series into Polynomials (*example*), 106
Suggested Precision for Factorization (*example*), 36
SuggestedPrecision(f) (*function*), 36, 67
SuMatrix(r, c, A) (*function*), 145
SuMatrix(v) (*function*), 145
SuMatrixSpace(S, r, c) (*function*), 145
Sums of Galois Representations (*example*), 170
SuRing(F) (*function*), 140
SuRing(F, nu) (*function*), 140
SuRing(p) (*function*), 140
SuRing(p, e) (*function*), 140
SuRing(S) (*function*), 140
SuRing(S, nu) (*function*), 140
SuSpace(M) (*function*), 148
SuSpace(R, n) (*function*), 148
SuSpace(v) (*function*), 148
SuVector(e) (*function*), 148

T

Tan(f) (*function*), 80
Tanh(f) (*function*), 81
Tate Twists of Galois Representations (*example*), 171
TateTwist(A,n) (*function*), 171
TeichmuellerLift(u, R) (*function*), 29
Tensor Products of Galois Representations (*example*), 170
Testing Whether a Representation Is Unramified (*example*), 167
Testing Whether a Series Is Polynomial (*example*), 106
Timing Logarithms and Exponentials (*example*), 26
TotallyRamifiedExtension(L, f) (*function*), 6
TotallyRamifiedExtension(R, f) (*function*), 84
Trace(a) (*function*), 66
Trace(a, F) (*function*), 66

Trace(x) (*function*), 27
Trace(x, R) (*function*), 27
Truncate(f) (*function*), 76
Twisted Special Representations (*example*), 157
Two-Step Extension of a Series Ring (*example*), 84
Two-Step Representation of a Local Field (*example*), 63

U

Unbounded Precision Extensions from Maps (*example*), 7
Uniformizing and Inertial Elements (*example*), 67
UniformizingElement(E) (*function*), 85
UniformizingElement(L) (*function*), 12, 18, 66
UniformizingElement(R) (*function*), 48, 73
UnitGroup(F) (*function*), 42
UnitGroup(R) (*function*), 42
UnitGroupGenerators(F) (*function*), 42
UnitGroupGenerators(R) (*function*), 42
Univariate Lazy Series from a Map (*example*), 93
Unramified Characters (*example*), 155
Unramified Galois Representations (*example*), 156
UnramifiedCharacter(K,c) (*function*), 155
UnramifiedExtension(L, f) (*function*), 4
UnramifiedExtension(L, n) (*function*), 4
UnramifiedExtension(R, f) (*function*), 84
UnramifiedQuotientRing(K, k) (*function*), 4
UnramifiedRepresentation(K,CharPoly) (*function*), 155
UnramifiedRepresentation(K,dim, dimcomputed,CharPoly) (*function*), 156

V

V ! r (*operation*), 109
v * w (*operation*), 110
v *:= w (*operation*), 110
v + w (*operation*), 110
v +:= w (*operation*), 110
v / w (*operation*), 110
v - w (*operation*), 110
v -:= w (*operation*), 110
v ^ k (*operation*), 110
v div w (*operation*), 110
v eq w (*operation*), 110
v in M (*operation*), 149
v in V (*operation*), 110
v ne w (*operation*), 110
v notin V (*operation*), 110
Valuation(a) (*function*), 66
Valuation(e) (*function*), 87
Valuation(f) (*function*), 77
Valuation(s) (*function*), 94
Valuation(v) (*function*), 111
Valuation(x) (*function*), 25, 51
ValuationEq(x, n) (*function*), 51
ValuationGe(x, n) (*function*), 51
ValuationGt(x, n) (*function*), 51
ValuationLe(x, n) (*function*), 51
ValuationLt(x, n) (*function*), 51
ValuationNe(x, n) (*function*), 51
ValuationRing(F) (*function*), 109
ValuationRing(F, f) (*function*), 109
ValuationRing(K, p) (*function*), 109
ValuationRing(Q, p) (*function*), 109
ValuationsOfRoots(f) (*function*), 32, 126
ValuationsOfRoots(f, p) (*function*), 126
Vertices of a Newton Polygon (*example*), 122
Vertices(N) (*function*), 122

W

WeakDegree(f) (*function*), 56
WeakValuation(x) (*function*), 51
WeierstrassDegree(f) (*function*), 141
WeierstrassDegrees(M) (*function*), 146
WeierstrassDegrees(v) (*function*), 149
WeierstrassPreparation(f) (*function*), 143
WeierstrassTerm(f) (*function*), 142
WeierstrassTerms(M) (*function*), 146
WeierstrassTerms(v) (*function*), 148
Weight(F) (*function*), 124
Wild Ramification of Elliptic Curves (*example*), 167

X

x * y (*operation*), 21, 52, 56, 87
x + y (*operation*), 21, 52, 56, 87
x / c (*operation*), 57
x / y (*operation*), 21, 52, 87
x - y (*operation*), 21, 52, 57, 87
x ^ k (*operation*), 21
x ^ n (*operation*), 52, 57, 87
x div y (*operation*), 21, 52, 57, 87
x div:= y (*operation*), 21
x eq y (*operation*), 22, 87
x in L (*operation*), 22
x mod y (*operation*), 57
x ne y (*operation*), 22
x notin L (*operation*), 22
XGCD(a, b) (*function*), 116
XGCD(v, w) (*function*), 111
Xgcd(v, w) (*function*), 111
XGCD(x, y) (*function*), 53
Xgcd(x, y) (*function*), 53

Z

Zero Galois Representation (*example*), 154
Zero(L) (*function*), 17
Zero(Q) (*function*), 73
Zero(R) (*function*), 113
ZeroMatrix(S) (*function*), 145
ZeroMatrix(S, n) (*function*), 145

`ZeroMatrix(S, r, c)` (*function*), 145
`ZeroRepresentation(K)` (*function*), 154
`ZeroVector(S)` (*function*), 148