MCLF Documentation

Stefan Wewers

Aug 25, 2020

Contents

Contents:

CHAPTER 1

Curves

1.1 Smooth projective curves over a field.

Let k be a field and F/k a finitely generated field extension of transcendence degree one (i.e. a 'function field over k'). Then there exists a smooth projective curve X over $Spec(k)$ with function field F, unique up to unique isomorphism. The set of closed points on X are in natural bijection with the set of discrete valuations on F which are trivial on k. See

- R. Hartshorne, *Algebraic Geometry*, Theorem I.6.9.

The classes in this module provide the definition and some basic functionality for such curves.

A curve X is defined via its function field F_X. Points are represented by the corresponding valuations on F_X, and no smooth projective model of X is actually computed. However, we do compute a list of 'coordinate functions' $x_1, .., x_n$ which separate all points, meaning that the closure of the rational map from X to projective space of dimension n is injective. Then a (closed) point x on X can also be represented by the tupel $(f_1(x), .., f_n(x))$. This is useful to test for equality of points.

A function field in Sage is always realized as a simple separable extension of a rational function field. Geometrically, this means that the curve X is implicitly equipped with a finite separable morphism $\phi : X \to \mathbb{P}^1_k$ to the projective line over the base field k.

The base field k is called the *constant base field* of the curve, and it is part of the data. We do not assume that the extension F_X/k is regular (i.e. that k is algebraically closed in F_X). Geometrically this means that the curve X may not be absolutely irreducibel as a k-scheme. The *field of constants* of X is defined as the algebraic closure of k inside F_X. It is a finite extension k_c/k. If we regard X as a curve over its fields of constants then it becomes absolute irreducible.

It would be interesting to have an efficient algorithm for computing the field of constants, but it seems that this has not been implemented in Sage yet. To compute the genus of X it is necessary to know at least the degree $[k_c : k]$. If k is a finite field, it is actually easy to compute k_c. If k is a number field we use a probabilistic algorithm for computing the degree $[k_c : k]$, by reducing the curve modulo several small primes.

Currently, the function field F defining a smooth projective curve must be a simple finite extension of a rational

function field, i.e. of the form

$$F = k(x)[y]/(G)$$

where G is an irreducible polynomial over $k(x)$. If not explicitly stated otherwise, it is assumed that k is the constant base field of the curve X. If k is a finite field, then one may also declare any subfield k_0 of k to be the constant base field. Geometrically, this means that we consider X as a curve over $\mathrm{Spec}(k_0)$. In any case, the field of constants of X is a finite extension of k.

AUTHORS:

- Stefan Wewers (2016-11-11): initial version

EXAMPLES:

```
sage: from mclf import *
sage: K = GF(2)
sage: FX.<x> = FunctionField(K)
sage: R.<T> = FX[]
sage: FY.<y> = FX.extension(T^2+x^2*T+x^5+x^3)
sage: Y = SmoothProjectiveCurve(FY)
sage: Y
the smooth projective curve with Function field in y defined by y^2 + x^2*y + x^5 + x^
↪3
sage: Y.genus()
1
sage: Y.zeta_function()
(2*T^2 + T + 1)/(2*T^2 - 3*T + 1)
```

Over finite fields, we are allowed to specify the constant base field:

```
sage: K = GF(4)
sage: F.<x> = FunctionField(K)
sage: X = SmoothProjectiveCurve(F, k=GF(2))
sage: X
the smooth projective curve with Rational function field in x over Finite Field in z2␣
↪of size 2^2 and constant base field Finite Field of size 2
sage: X.field_of_constants()
Finite Field in z2 of size 2^2
```

A curve may also be defined by an irreducible bivariate polynomial:

```
sage: A.<x,y> = QQ[]
sage: X = SmoothProjectiveCurve(y^2 - x^3 - 1)
sage: X
the smooth projective curve with Function field in y defined by y^2 - x^3 - 1
```

If the curve is defined over a number field, we can find a prime of good reduction, and compute the reduction:

```
sage: v_p = X.prime_of_good_reduction()
sage: v_p
5-adic valuation
```

The result is a discrete (p-adic) valuation on the constant base field. The reduction is a smooth projective curve of the same genus:

```
sage: Xb = X.good_reduction(v_p)
sage: Xb
```

(continues on next page)

(continued from previous page)

```
the smooth projective curve with Function field in y defined by y^2 + 4*x^3 + 4
sage: Xb.genus()
1
```

Todo:

- allow to specify the constant base field in a more flexible way
- write more doctests !!
- implement a general and deterministic algorithm for computing the field of constants (and not just the degree)
- the residue field of a point should be explicitly an extension of the constant base field.
- treat the base curve X as a *curve*, not just as a function field
- realize morphisms between curves, in particular the canonical map to X

class mclf.curves.smooth_projective_curves.**PointOnSmoothProjectiveCurve**(*X*, *v*)

Bases: SageObject

A closed point on a smooth projective curve.

A point on a curve X is identified with the corresponding valuation v_x on the function field F of X.

Alternatively, a point x on X can be represented by the vector

$$[v_x(f_1), \ldots, v_x(f_n)]$$

where $f_1, \ldots, f_n$ is a list of *coordinate functions*, i.e. rational functions which define an injective map from X into $\mathbb{P}^1 \times \ldots \times \mathbb{P}^1$.

We use the latter representation to check for equality of points.

absolute_degree()

Return the absolute degree of self.

The *absolute degree* of a point x on a curve X over k is the degree of the extension $k(x)/k$.

Here k is the constant base field of the curve, which may not be equal to the field of constants.

coordinates()

Return the coordinate tupel of the point.

NOTE:

for a curve over a number field and for a point whose residue field is of high degree, this can be *very* slow. It would be better to implement this function in a lazy way, for instance as an iterator.

curve()

Return the underlying curve of the point.

degree()

Return the degree of self.

The *degree* of a point x on a curve X over k is the degree of the residue field $k(x)$ as an extension of the field of constants of X. The latter may be a proper extension of the base field k!

is_equal(*P*)

Check whether this point is equal to P.

INPUT:

- `P` – a point on the curve underlying this point

OUTPUT:

`True` is P is equal to `self`, `False` otherwise.

Currently, the check for equality is done using the 'coordinates' of the points. This may be very slow. It would probably be better to test the equality of the underlying valuations. But here we can't rely on Sage, so this would require a hack.

EXAMPLES:

```
sage: import mclf.curves.smooth_projective_curves
sage: from mclf.curves.smooth_projective_curves import SmoothProjectiveCurve
sage: F0.<x> = FunctionField(GF(3))
sage: R.<y> = F0[]
sage: F.<y> = F0.extension(y^2 - (x+1)*x^2)
sage: Y = SmoothProjectiveCurve(F)
sage: v0 = F0.valuation(x)
sage: fiber = Y.fiber(v0)
sage: fiber[0].is_equal(fiber[1])
False
```

order(*f*)
Return the order of the function in the point.

This is the same as `self.valuation()(f)`.

residue_field()
Return the residue field of the point.

valuation()
Return the valuation corresponding to the point.

value(*f*)
Return the value of the function `f` in the point.

If `f` has a pole then `Infinity` is returned.

class mclf.curves.smooth_projective_curves.**SmoothProjectiveCurve**(*F*, *k=None*, *assume_regular=False*)

Bases: `SageObject`

Return the smooth projective curve with function field F.

INPUT:

- `F` – a function field, or an irreducible bivariate polynomial over a field
- `k` – a field which has a natural embedding into the constant base field of F, such that the constant base field is a finite extension of k (or `None`).
- `assume_regular` – a boolean (default: `False`)

OUTPUT:

the smooth projective curve X with function field F. If F is an irreducible bivariate polynomial, we use the function field with two generators and relation F.

If k is given, then X is considered as a k-scheme. If k is not given then we use the field of constants of F instead.

NOTE:

At the moment, k should only be different from the constant base field of F if k is finite (because it is then easy to compute the degree of the degree of the constant base field of F over k).

compute_separable_model()
Compute a separable model of the curve (if necessary).

OUTPUT: `None`

This function only has to be called only once. It then decides whether or not the function field of the curve is given as a separable extension of the base field or not. If it is not separable then we compute a separable model, which is a tripel (Y_1, ϕ, q) where

- Y_1 is a smooth projective curve over the same constant base field k as the curve Y itself, and which is given by a separable extension,
- ϕ is a field homomorphism from the function field of Y_1 into the function field of Y corresponding to a purely inseparable extension,
- q is the degree of the extension given by ϕ, i.e. the degree of inseparability of the map $Y \to Y_1$ given by ϕ. Note that q is a power of the characteristic of k.

constant_base_field()
Return the constant base field.

coordinate_functions()
Return a list of coordinate functions.

By 'list of coordinate functions' we mean elements f_i in the function field, such that the map

$$x \mapsto (f_1(x), \ldots, f_n(x))$$

from X to $(\mathbb{P}^1)^n$ is injective.

Note that this map may not be an embedding, i.e. image of this map may not be a smooth model of the curve.

count_points(d)
Return number of points of degree less or equal to `d`.

INPUT:

- `d` – an interger ≥ 1

OUTPUT:

a list `N`, where `N[i]` is the number of points on self of *absolute* degree i, for $i = 1, .., d$.

Recall that the absolute degree of a point is the degree of the residue field of the point over the constant base field (*not* over the field of constants).

This is a very slow realization and should be improved at some point.

covering_degree()
Return the degree of this curve as a covering of the projective line.

degree(D)
Return the degree of the divisor `D`.

Note that the degree of D is defined relative to the field of constants of the curve.

degree_of_inseparability()
Return the degree of inseparability of this curve.

OUTPUT: positive integer, which is a power of the characteristic of the function field of this curve.

divisor_of_poles(*f*)
Return the divisor of poles of `f`.

divisor_of_zeroes(*f*)
Return the divisor of zeroes of `f`.

fiber(*v0*)
Return the set of points lying above a point on the projective line.

INPUT:

- `v0` – a function field valuation on the rational function field

OUTPUT:

a list containing the points on this curve Y corresponding to extensions of `v0` to the function field of Y.

field_of_constants()
Return the field of constants of this curve.

If F is the function field of the curve and k the constant base field, then the *field of constants* is the algebraic closure of k in F.

For the moment, this is implemented only if the constant base field is a finite field.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(GF(2))
sage: R.<y> = F[]
sage: G = (y+x)^4 + (y + x) + 1
sage: F1.<y> = F.extension(G)
sage: Y1 = SmoothProjectiveCurve(F1)
sage: Y1.field_of_constants()
Finite Field in z4 of size 2^4
```

field_of_constants_degree()
Return the degree of the field of constants over the constant base field.

If F is the function field of the curve and k the constant base field, then the *field of constants* k_c is the algebraic closure of k in F.

If k is a finite field then we actually compute the field of constants, and the result is provably correct. If k is a number field, then we use a heuristic method: we find at least 10 different primes of k for which the reduction of the defining equation remains irreducible, and then we apply the method for finite fields to the reduced equation. The result is very likely the true degree of the field of constants, and if the result is equal to 1 then it is provably correct.

EXAMPLES:

```
sage: from mclf import *
sage: k = GF(2^3)
sage: F.<x> = FunctionField(k)
sage: R.<y> = F[]
sage: G = (y+x)^4 + (y+x) + 1
sage: F1.<y> = F.extension(G)
sage: Y1 = SmoothProjectiveCurve(F1, GF(2))
sage: Y1.field_of_constants_degree()
12
sage: F.<x> = FunctionField(QQ)
sage: R.<y> = F[]
sage: G = y^4 + x*y + 1
```

(continues on next page)

(continued from previous page)

```
sage: F2.<y> = F.extension(G)
sage: Y2 = SmoothProjectiveCurve(F2)
sage: Y2.field_of_constants_degree()
1
sage: R.<y> = F[]
sage: G = (y+x)^3 + (y+x) + 1
sage: F3.<y> = F.extension(G)
sage: Y3 = SmoothProjectiveCurve(F3)
sage: Y3.field_of_constants_degree()
3
sage: Y3.genus()
0
```

Todo:

- implement a deterministic algorithm for number fields

function_field()
Return the function field of the curve `self`.

genus(*use_reduction=True*)
Return the genus of the curve.

INPUT:

- `use_reduction` – a boolean (default: `True`)

OUTPUT: the genus of this curve.

The genus of the curve is defined as the dimension of the cohomology group $H^1(X, \mathcal{O}_X)$, as a vector space *over the field of constants 'k_c'*.

The genus g of the curve X is computed using the Riemann-Hurwitz formula, applied to the cover $X \to \mathbb{P}^1$ corresponding to the underlying realization of the function field of X as a finite separable extension of a rational function field. See:

- Hartshorne, *Algebraic Geometry*, Corollary IV.2.4

If the constant base field is finite, we compute the degree of the 'ramification divisor'. If it is not, we assume that the characteristic is zero, and we use the 'tame' Riemann Hurwitz Formula.

If the curve is defined over a number field, and `use_reduction` is `True` (the default) then the genus of a reduction of this curve to some prime of good reduction is computed. This may be consderably faster.

EXAMPLES:

```
sage: from mclf import *
sage: F0.<x> = FunctionField(GF(2))
sage: R.<T> = F0[]
sage: G = T^2 + T + x^3 + x + 1
sage: F.<y> = F0.extension(G)
sage: Y = SmoothProjectiveCurve(F)
sage: Y.genus()
1
sage: G = T^2 + x^3 + x + 1
sage: F.<y> = F0.extension(G)
sage: Y = SmoothProjectiveCurve(F)
```

(continues on next page)

(continued from previous page)

```
sage: Y.genus()
0
```

good_reduction(*v_p*)
Return the reduction of this curve at a prime of good reduction.

INPUT:

- v_p – a discrete valuation on the constant base field K

OUTPUT: the reduction of this curve with respect to v_p, assuming that this is again a smooth projective curve of the same genus. Otherwise, an error is raised.

Note that we just reduce the plane equation for this curve with respect to v_p. This is a very naive notion of good reduction. If it works, then the curve does indeed have good reduction at v_p, and the result is correct.

is_separable()
Check whether this curve is represented as a separable cover of the projective line.

phi()
Return the natural embedding of the function field of the separable model into the function field of this curve.

OUTPUT: a field homomorphism

plane_equation()
Return the plane equation of this curve.

OUTPUT:

A polynomial in two variables over the constant base field which defines the plane model of this curve, where the first variable corresponds to the base field F0.

point(*v*)
Returns the point on the curve corresponding to v.

INPUT:

- v – a discrete valuation on the function field of the curve

OUTPUT:

The point on the curve corresponding to v. The valuation v must be trivial on the constant base field.

points_with_coordinates(*a*)
Return all points with given coordinates.

INPUT:

- a – a tupel of coordinates, of lenght n, at most the number of coordinate functions of the curve

OUTPUT:

a list containing all points on the curve whose first n coordinate values agree with a.

potential_branch_divisor()
Return list of valuations containing the branch locus.

OUTPUT:

A list of pairs $(v, d)'$, $where$ runs over a list of valuations of the base field $F_0 = K(x)$ containing all the valuations corresponding to a branch point of the cover of curves, and d is the degree of v.

prime_of_good_reduction()
Return a prime ideal where this curve has good reduction.

OUTPUT:

We assume that the constant base field K is a number field. We return a discrete valuation v on K such that the following holds:

- all the coefficients of the plane equation $G(x, y) = 0$ of this curve are v-integral
- the reduction of G to the residue field of v is irreducible and defines a plane curve with the same genus as the original curve.

Note that this implies that v is inert in the field of constants of the curve.

EXAMPLES:

```
sage: from mclf import *
sage: R.<T> = QQ[]
sage: K.<zeta> = NumberField(T^2+T+1)
sage: A.<x,y> = K[]
sage: X = SmoothProjectiveCurve(y^3 - y - x^2 + 1 + zeta)
sage: X.prime_of_good_reduction()
5-adic valuation
sage: X.good_reduction(_)
the smooth projective curve with Function field in y defined by y^3 + 4*y +␣
↪4*x^2 + u1 + 1
```

principal_divisor(f)
Return the principal divisor of `f`.

INPUT:

- `f` – a nonzero element of the function field of `self`

OUTPUT: the principal divisor $D = (f)$. This is a list of pairs (P, m), where P is a point and m is an integer.

ramification_divisor()
Return the ramification divisor of self.

OUTPUT:

The ramification divisor of the curve, if the curve is given by a separable model. Otherwise, an error is raised.

So the function field of of the curve is a finite separable extension of a rational function field. Geometrically, this means that the curve X is represented as a separable cover of the projective line. The ramification divisor of this cover is supported in the set of ramification points of this cover. Sheaf theoretically, the divisor represents the sheaf of relative differentials. See:

- Hartshorne, *Algebraic Geometry*, Definition IV.2.

random_point()
Return a random closed point on the curve.

rational_function_field()
Return the rational function field underlying the function field of X.

By definition, the function field F_X of the curve X is a finite separable extension of a rational function field $k(x)$, where k is the base field of X.

separable_model()
Return the separable model of this curve.

OUTPUT: a smooth projective curve over the same constant base field

The *separable model* of this curve Y is a curve Y_s defined over the same constant base field and whose defining equation realizes Y_s as a finite *separable* cover of the projective line. It comes equipped with a finite, purely inseparable morphism $Y \to Y_s$. In particular, Y_s has the same genus as Y.

The inclusion of function fields $\phi : F(Y_s) \to F(Y)$ can be accessed via the method `phi()`, the degree of the extension Y/Y_s via the method `degree_of_inseparability`.

singular_locus()
Return the singular locus of the affine model of the curve.

OUTPUT:

a list of discrete valuations on the base field $k(x)$ which represent the x-coordinates of the points where the affine model of the curve given by the defining equation of the function field *may* be singular.

structure_map()
Return the canonical map from this curve to the projective line.

zeta_function(*var_name='T'*)
Return the Zeta function of the curve.

For any scheme X of finite type over $\mathbb{Z}$, the **arithmetic zeta funtion** of X is defined as the product

$$\zeta(X, s) := \prod_x \frac{1}{1 - N(x)^{-s}},$$

where x runs over over all closed points of X and $N(x)$ denotes the cardinality of the residue field of x.

If X is a smooth projective curve over a field with q elements, then $\zeta(X, s) = Z(X, q^{-s})$, where $Z(X, T)$ is a rational function in T of the form

$$Z(X, T) = \frac{P(T)}{(1 - T)(1 - qT)},$$

for a polynomial P of degree $2g$, with some extra properties reflecting the Weil conjectures. See:

- Hartshorn, *Algebraic Geometry*, Appendix C, Section 1.

Note that that this makes only sense if the constant base field of self is finite, and that $Z(X, T)$ depends on the choice of the constant base field (unlike the function $\zeta(X, s)$!).

mclf.curves.smooth_projective_curves.**absolute_degree**(*K*)
Return the absolute degree of a (finite) field.

mclf.curves.smooth_projective_curves.**compute_value**(*v*, *f*)
Return the value of `f` at `v`.

INPUT:

- `v` – a function field valuation on F
- `f` – an element of F

OUTPUT: The value of `f` at the point corresponding to `v`.

This is either an element of the residue field of the valuation `v` (which is a finite extension of the base field of F), or ∞.

mclf.curves.smooth_projective_curves.**e_f_of_valuation**(*v*)
Return the ramification index of this valuation.

INPUT:

- `v` – a function field valuation on an extension of a rational function field

OUTPUT: the ramification index of v over the base field

mclf.curves.smooth_projective_curves.**extension_degree**(*K*, *L*, *check=False*)
Return the degree of the field extension.

INPUT:

- K, L – two fields, where K has a natural embedding into L
- check (default: False) – boolean

OUTPUT:

the degree $[L : K]$

At the moment, this works correctly only if K and L are finite extensions of a common base field. It is not checked whether K really maps to L.

mclf.curves.smooth_projective_curves.**extension_of_finite_field**(*K*, *n*)
Return a field extension of this finite field of degree n.

INPUT:

- K – a finite field
- n – a positive integer

OUTPUT: a field extension of K of degree n.

This function is useful if K is constructed as an explicit extension $K = K_0[x]/(f)$; then K.extension(n) is not implemented.

Note: This function should be removed once trac.sagemath.org/ticket/26103 has been merged.

---

mclf.curves.smooth_projective_curves.**field_of_constant_degree_of_polynomial**(*G*, *return_field=False*)
Return the degree of the field of constants of a polynomial.

INPUT:

- G – an irreducible monic polynomial over a rational function field
- return_field – a boolean (default:$False$)

OUTPUT: the degree of the field of constants of the function field defined by G. If return_field is True then the actual field of constants is returned. This is currently implemented for finite fields only.

This is a helper function for SmoothProjectiveCurve.field_of_constants_degree.

mclf.curves.smooth_projective_curves.**make_finite_field**(*k*)
Return the finite field isomorphic to this field.

INPUT:

- k – a finite field

OUTPUT: a triple (k_1, ϕ, ψ) where k_1 is a 'true' finite field, ϕ is an isomorphism from k to k_1 and ψ is the inverse of ϕ.

This function is useful when k is constructed as a tower of extensions with a finite field as a base field.

Note: This function should be removed once trac.sagemath.org/ticket/26103 has been merged.

---

mclf.curves.smooth_projective_curves.**separate_points**(*coordinate_functions*, *valuations*)

Add new coordinate functions to separate a given number of points.

INPUT:

- `coordinate_functions` – a list of elements of a function field F
- `valuations` – a list of function field valuations on F

OUTPUT: enlarges the list `coordinate_functions` in such a way that the lists `[value(v,x) for x in coordinate_functions]`, where `v` runs through `valuations`, are pairwise distinct.

mclf.curves.smooth_projective_curves.**separate_two_points**(*v1*, *v2*)

Return a rational function which separates two points

INPUT:

- `v1`, `v2` – discrete, nonequivalent valuations on a common function field F

OUTPUT:

An element f of F which takes distinct values at the two points corresponding to `v1` and `v2`.

mclf.curves.smooth_projective_curves.**sum_of_divisors**(*D1*, *D2*)

Return the sum of the divisors `D1` and `D2`.

INPUT:

- `D1`, `D2`: divisors on the same curve X

OUTPUT:

D_1 is replaced by the sum $D_1 + D_2$ (note that this changes D_1!).

Here a divisor D is given by a dictionary with entries `(a:(P,m))`, where `a` is a coordinate tupel, `P` is a point on X with coordinates `a` and `m` is the multiplicity of `P` in D.

1.2 Morphisms of smooth projective curves

This module defines a class `MorphismOfSmoothProjectiveCurves` which realizes finite and nonconstant morphism between smooth projective curves.

Let Y and X be smooth projective curves, with function fields F_Y and F_X, respectively. Then a nonconstant morphism

$$f : Y \to X$$

is completely determined by the induced pullback map on the function fields,

$$\phi = f^* : F_X \to F_Y.$$

It is automatic that F_Y is a finite extension of $\phi(F_X)$ and that the morphism $\phi : Y \to X$ is finite.

Note: For the time being, this module is in a very preliminary state. A morphism $\phi : Y \to X$ as above can be constructed only in the following two special cases:

- X and Y are two projective lines; then F_X and F_Y are rational function fields in one variable.
- the map $f : Y \to X$ is the *structure map* of the curve Y; by this we mean that X is the projective line f the canonical morphism realizing Y as a cover of X.

Moreover, the role of the constant base fields of the two curves still needs to be clarified.

AUTHORS:

- Stefan Wewers (2018-1-1): initial version

EXAMPLES:

```
sage: from mclf import *
sage: FX.<x> = FunctionField(QQ)
sage: R.<y> = FX[]
sage: FY.<y> = FX.extension(y^2-x^3-1)
sage: X = SmoothProjectiveCurve(FX)
sage: Y = SmoothProjectiveCurve(FY)
sage: phi = MorphismOfSmoothProjectiveCurves(Y, X)
sage: phi
morphism from the smooth projective curve with Function field in y defined by y^2 - x^
↪3 - 1
to the smooth projective curve with Rational function field in x over Rational Field,
determined by Coercion map:
  From: Rational function field in x over Rational Field
  To:   Function field in y defined by y^2 - x^3 - 1

sage: x0 = PointOnSmoothProjectiveCurve(X, FX.valuation(x-1))
sage: phi.fiber(x0)
[Point on the smooth projective curve with Function field in y defined by y^2 - x^3 -␣
↪1 with coordinates (1, u1).]
```

class mclf.curves.morphisms_of_smooth_projective_curves.**MorphismOfSmoothProjectiveCurves**(*Y*, *X*, *ph*

Bases: `SageObject`

Return the morphism between two smooth projective curves corresponding to a given morphism of function fields.

INPUT:

- `Y`, `X` – two smooth projective curves
- `phi` – a morphism from the function field of X into the function field of Y, or `None` (default: `None`)

OUTPUT: the morphism $f : Y \to X$ corresponding to the given morphism of function fields.

If no morphism of function fields is given then it is assumed that the function field of X is the canonical rational subfield of the function field of Y. This means that map $f : Y \to X$ is the structure map of Y as a cover of the projective line. If this is not the case then an error is raised.

Note: At the moment only the following two special cases are implemented:

- the map $Y \to X$ is equal to the structural morphism of Y as a cover of the projective line; in particular, X is a projective line
- X and Y are both projective lines

EXAMPLES:

We define a rational map between two projective lines and compute the fiber of a point on the target:

```
sage: from mclf import *
sage: FX.<x> = FunctionField(QQ)
sage: FY.<y> = FunctionField(QQ)
sage: X = SmoothProjectiveCurve(FX)
sage: Y = SmoothProjectiveCurve(FY)
sage: phi = FY.hom(x^2+1)
sage: psi = MorphismOfSmoothProjectiveCurves(X, Y, phi)
sage: psi
morphism from the smooth projective curve with Rational function field in x over␣
↪Rational Field
to the smooth projective curve with Rational function field in y over Rational␣
↪Field,
determined by Function Field morphism:
From: Rational function field in y over Rational Field
To:   Rational function field in x over Rational Field
Defn: y |--> x^2 + 1

sage: P = PointOnSmoothProjectiveCurve(Y, FY.valuation(y-2))
sage: psi.fiber(P)
[Point on the smooth projective curve with Rational function field in x over␣
↪Rational Field with coordinates (1,).,
 Point on the smooth projective curve with Rational function field in x over␣
↪Rational Field with coordinates (-1,).]
```

The only other map that is allowed is the structure morphism of a curve as a cover of the projective line:

```
sage: R.<x> = GF(2)[]
sage: Y = SuperellipticCurve(x^4+x+1, 3)
sage: phi = Y.structure_map()
sage: X = phi.codomain()
sage: X
the smooth projective curve with Rational function field in x over Finite Field␣
↪of size 2
sage: P = X.random_point()
sage: phi.fiber(P)  # random
[Point on superelliptic curve y^3 = x^4 + x + 1 over Finite Field of size 2 with␣
↪coordinates (0, 1).,
 Point on superelliptic curve y^3 = x^4 + x + 1 over Finite Field of size 2 with␣
↪coordinates (0, u1).]
```

codomain()
: Return the codomain of this morphism.

domain()
: Return the domain of this morphism.

fiber(P)
: Return the fiber of this map over the point P (without multiplicities).

  INPUT:

  - `P` – a point on the curve X, the codomain of this morphism

  OUTPUT: the fiber over P, as a list of points of Y (the domain of this map)

fiber_degree(P)
: Return the (absolute) degree of the fiber of this map over the point `P`.

  INPUT:

  - `P` – a point on the curve X (the codomain of this morphism)

OUTPUT: the fiber degree over P, the sum of the degrees of the points on Y (the domain of this morphism) lying above P. Here *degree* means *absolute degree*, i.e. with respect to the constant base field of Y (which may differ from the field of constants).

is_structure_map()
Return `True` if this map is the structure map of the curve Y.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = SmoothProjectiveCurve(F)
sage: phi = F.hom(x^2+1)
sage: f = MorphismOfSmoothProjectiveCurves(X, X, phi)
sage: f.is_structure_map()
False
sage: phi = F.hom(x)
sage: f = MorphismOfSmoothProjectiveCurves(X, X, phi)
sage: f.is_structure_map()
True
```

pullback(f)
Return the pullback of a function under this morphism.

pullback_map()
Return the induced inclusion of function fields.

1.3 Superelliptic curves

A *superelliptic curve* is a smooth projective curve over a field K which is given generically by an equation of the form

$$Y: \ y^n = f(x),$$

where $n \geq 2$ and $f \in K[x]$ is a polynomial over K, of degree at least 2 (and of degree at least 3 if $n = 2$). Let

$$f = c \cdot \prod_i f_i^{m_i}$$

be the prime factorization of f (where $c \in K^\times$ and the f_i are monic, irreducible and pairwise distinct). We assume that the gcd of the m_i is prime to n. This means that the defining equation is irreducible and the curve Y can be considered as a Kummer cover of $X = \mathbb{P}^1_K$ of degree n.

In this module we define a class `SuperellipticCurve` which is a subclasses of `SmoothProjectiveCurve` and whose objects represent superelliptic curves as above.

AUTHORS:

- Stefan Wewers (2018-5-18): initial version

EXAMPLES:

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: Y = SuperellipticCurve(x^4-1, 3)
sage: Y
superelliptic curve y^3 = x^4 - 1 over Rational Field
sage: Y.genus()
3
```

(continues on next page)

(continued from previous page)

```
sage: Y.kummer_gen()
y
sage: Y.polynomial()
x^4 - 1
sage: Y.covering_degree()
3
```

Todo:

- override those methods of `SmoothProjectiveCurve` where we have a faster algorithm in the superelliptic cases

class mclf.curves.superelliptic_curves.**SuperellipticCurve**(*f*, *n*, *name='y'*)

Bases: *mclf.curves.smooth_projective_curves.SmoothProjectiveCurve*

Return the superelliptic curve with equation $y^n = f(x)$.

INPUT:

- `f` – a nonconstant polynomial over a field K
- `n` – an integer ≥ 2
- `name` – a string (default 'y')

OUTPUT:

the smooth projective curve Y over K given generically by the equation

$$Y: \ y^n = f(x).$$

This means that the function field of Y is an extension of the rational function field in x generated by an element y (the Kummer generator) satisfying the above equation.

It is assumed that the gcd of the multiplicities of the irreducible factors of f is prime to n. Thus, the curve Y is a cover of $\mathbb{P}^1_K$ of degree n. If this condition is not met, an error is raised. `name` is the name given to the Kummer generator y.

covering_degree()

Return the covering degree.

If the curve is given by the equation $y^n = f(x)$ then the covering degree degree is n.

kummer_gen()

Return the Kummer generator of this superelliptic curve.

If the curve is given by the equation $y^n = f(x)$ then the Kummer generator is the element y of the the function field.

polynomial()

Return the polynomial defining this curve.

If the curve is given by the equation $y^n = f(x)$ then f is this polynomial.

CHAPTER 2

The Berkovich line

2.1 The Berkovich line over a discretely valued field

Let K be a field and v_K a discrete valuation on K. Let $F = K(x)$ be a rational function field over K. We consider F as the function field of the projective line $\mathbb{P}^1_K$ over K. Let X denote the (K, v_K)-analytic space associated to $\mathbb{P}^1_K$. We call X the *Berkovich line* with respect to v_K.

Note that we do not assume K to be complete with respect to v_K. This allows us to work with 'exact' fields, e.g. number fields. As the 'official' definition of K-analytic spaces requires K to be complete, X is really defined over the completion $\hat{K}$ with respect to v_K. We do have a continous map

$$\pi : X \to \mathbb{P}^1_K$$

whose role we discuss below.

We systematically work with additive pseudo-valuations instead of multiplicative seminorms. Thus, we identitfy a point $\xi \in X$ with a (real valued) pseudo-valuation v_ξ on F extending v_K,

$$v_\xi : F \to \mathbb{R} \cup \{\pm\infty\},$$

as follows: the subring

$$\mathcal{O}_\xi := \{f \in F \mid v_\xi(f) > -\infty\}$$

is a local subring of F, with maximal ideal

$$\mathfrak{m}_\xi := \{f \in F \mid v_\xi(f) = \infty\}.$$

Then v_ξ induces a discrete valuation on the residue field

$$K(\xi) := \mathcal{O}_\xi / \mathfrak{m}_\xi.$$

There are only two kind of points which are relevant for us and which we can represent and compute with:

- *points of type I*, which are moreover *algebraic*: these are the points $\xi \in X$ such that $\bar{\xi} := \pi(\xi)$ is a closed point on $\mathbb{P}^1_K$. Then $\mathcal{O}_\xi$ is the local ring and $K(\xi)$ the residue field of $\bar{\xi}$. Since $K(\xi)/K$ is a finite field extension, there are finitely many extensions of v_K to a discrete valuation on $K(\xi)$; the point $\xi \in \pi^{-1}(\bar{\xi})$ corresponds precisely to the valuation induces by v_ξ.
- *points of type II*: these are the points ξ such that v_ξ is a discrete valuation on F. In particular, the local ring $\mathcal{O}_\xi$ is equal to F and the image $\bar{\xi} := \pi(\xi)$ is the generic point of $\mathbb{P}^1_K$. A we see below, a point ξ of type II corresponds to a *discoid*, a certain type of affinoid subdomain of X.

Our choice of the generator x of the function field F, which we keep fixed throughout, yields certain distinguished subsets and points of X, as follows.

The *unit disk* is the subset

$$\mathbb{D} := \{\xi \in X \mid v_\xi(x) \geq 0\}.$$

Note that a point $\xi \in \mathbb{D}$ is uniquely determined by the restriction of v_ξ to the polynomial ring $K[x]$.

The *Gauss point* is the point $\xi^g \in \mathbb{D}$ of type II corresponding to the Gauss valuation on $K[x]$, with respect to v_K, i.e. by

$$v_{\xi^g}(\sum_i a_i x^i) = \min_i v_K(a_i).$$

The second distinguished is the *point at infinity*, denoted $\infty \in X$. It is the unique point of type I such that $\pi(\infty)$ is the 'usual' point at infinity on the projective line, with respect to the parameter x. It is characterized by the condition

$$v_\infty(\frac{1}{x}) = \infty.$$

By a result of Berkovich, the topological space X is a *simply connected quasi-polyhedron*. Among other things this means that for any two points $\xi_1, \xi_2 \in X$ there exists a unique closed subset

$$[\xi_1, \xi_2] \subset X$$

which is homeomorphic to the unit interval $[0, 1] \subset \mathbb{R}$ in such a way that ξ_1, ξ_2 are mapped to the endpoints $0, 1$. It follows that X has a unique partial ordering determined by the following two conditions:

- the Gauss point ξ^g is the smallest element
- we have $\xi_1 < \xi_2$ if and only if ξ_2 lies in a connected component of $X - \{\xi_1\}$ which does not contain ξ^g.

A point ξ of type II has a *discoid representation* as follows. If $\xi = \xi^g$ then $D_{\xi^g} := \mathbb{D}$ is defined as the unit disk. Otherwise, D_ξ is defined of the set of all points $\xi_1 \in X$ such that $\xi \leq \xi_1$. One can show that D_ξ is then of the form

$$D_\xi = \{\xi_1 \mid v_{\xi_1}(f) \geq s\},$$

where f is a polynomial in x, irreducible over $\hat{K}$ (or $f = 1/x$ if $\infty \in D_\xi$) and s is a rational number. The pair (f, s) determines ξ, but this representation is not unique. We call (f, s) a *discoid representation* of ξ.

Conversely, if $D \subset X$ is a *discoid*, i.e. an irreducible affinoid subdomain which becomes a union of closed disks over a finite extension of K, then there exists a unique boundary point ξ of D. We have $D = D_\xi$ if and only if D is a *standard discoid*, i.e. it is either contained in or disjoint from the unit disk.

Note that we can simply extend the discoid representation to points of type I by allowing s to take the value ∞. Then $D_\xi = \{\xi\}$ for a point ξ of type I.

AUTHORS:

- Stefan Wewers (2017-02-10): initial version

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: X
Berkovich line with function field Rational function field in x over Rational Field␣
↪with 2-adic valuation
```

We define a point of type II via its discoid.:

```
sage: xi1 = X.point_from_discoid(x^3 + 2, 3)
sage: xi1
Point of type II on Berkovich line, corresponding to v(x^3 + 2) >= 3
```

If the affinoid $v(f) \geq s$ is not irreducible, an error is raised.:

```
sage: X.point_from_discoid(x^2-1, 2)
Traceback (most recent call last):
...
AssertionError: D defined by f=x^2 - 1 and s=2 is not a discoid
```

We can similarly define points which do not lie on the unit disk.:

```
sage: xi2 = X.point_from_discoid(4*x+1, 1)
sage: xi2
Point of type II on Berkovich line, corresponding to v(4*x + 1) >= 1
```

The infimum of a point inside and a point outside the unit disk must be the Gauss point, corresponding to the unit disk.:

```
sage: xi1.infimum(xi2)
Point of type II on Berkovich line, corresponding to v(x) >= 0
sage: X.gauss_point()
Point of type II on Berkovich line, corresponding to v(x) >= 0
```

Some points of type I are *limit points*, i.e. they can only be approximated by points of type II. For instance, the zeroes of a polynomial which is irreducible over the ground field $\mathbb{Q}$, but not over its completion $\mathbb{Q}_2$.:

```
sage: f = 2*x^2 + x + 1
sage: f.factor()
(2) * (x^2 + 1/2*x + 1/2)

sage: D = X.divisor(f)
sage: D
[(Point of type I on Berkovich space approximated by v(2*x + 1) >= 1, with equation␣
↪4*x^2 + 2*x + 2 = 0,
  1),
 (Point of type I on Berkovich space approximated by v(x + 1) >= 1, with equation 4*x^
↪2 + 2*x + 2 = 0,
  1),
 (The point at infinity on the Berkovich line, -2)]
sage: xi = D[0][0]
sage: xi.equation()
4*x^2 + 2*x + 2
```

Note that the point ξ lies outside and its Galois conjugate point lies inside of the unit disk. This shows that issue #39 has been fixed.

TO DO:

- more doctests!

class mclf.berkovich.berkovich_line.**BerkovichLine**(*F*, *vK*)
Bases: SageObject

The class of a Berkovich projective line over a discretely valued field.

Let K be a field and v_K a discrete valuation on K. Let $F = K(x)$ be a rational function field over K. We consider F a the function field of the projective line X over K. Let X denote the (K, v_K)-analytic space associated to X. Then a point ξ on X may be identified with a (real valued) pseudo-valuation v_ξ on F extending v_K.

INPUT:

- F – a rational function field over a base field K
- vK – a discrete valuation on the base field K

find_zero(*xi1*, *xi2*, *f*)
Return the point between xi1 and xi2 where f has valuation 0.

INPUT:

- xi1, xi2 – points on the Berkovich line such that $\xi_1 < \xi 2$
- **f – a nonconstant rational function; it is assumed that the signs** of the valuations of f at $\xi 1$ and $\xi 2$ are different

OUTPUT:

The smallest point between $\xi 1$ and $\xi 2$ where the valuation of f is zero.

NOTE:

We are assuming for the moment that the function

$$v \mapsto v(f)$$

is affine (i.e. has no kinks) on the interval $[\xi_1, \xi_2]$.

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: xi1 = X.gauss_point()
sage: xi2 = X.point_from_discoid(x^4+2*x^2+2, 10)
sage: X.find_zero(xi1, xi2, (x^4+2*x^2+2)/4)
Point of type II on Berkovich line, corresponding to v(x^4 + 2*x^2 + 2) >= 2

sage: xi3 = X.point_from_discoid(4*x^4+2*x^2+1, 10)
sage: f = 2*x^3
sage: xi1.v(f), xi3.v(f)
(1, -1/2)
sage: X.find_zero(xi1, xi3, f)
Point of type II on Berkovich line, corresponding to v(1/x) >= 1/3
```

Todo: Remove the assumption on the kinks.

gauss_point()
Return the Gauss point of self.

The Gauss point is the type-II-point corresponding to the Gauss valuation on $K[x]$. Its discoid is the unit disk.

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: X.gauss_point()
Point of type II on Berkovich line, corresponding to v(x) >= 0
```

infty()
Return the point ∞.

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: X.infty()
The point at infinity on the Berkovich line
```

point_from_pseudovaluation(*v*)
Return the point on the Berkovich line corresponding to the pseudovaluation v.

INPUT:

- **v – a discrete pseudovaluation on the function field of self,** extending the base valuation v_K

OUTPUT:

The point ξ on the Berkovich line X =‘‘self‘‘ corresponding to the pseudo valuation v on the function field of X.

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: v = F.valuation(x)
sage: X.point_from_pseudovaluation(v)
Traceback (most recent call last):
...
AssertionError: v is not an extension of the base valuation

sage: v = F.valuation(GaussValuation(F._ring, v_2))
sage: X.point_from_pseudovaluation(v)
Point of type II on Berkovich line, corresponding to v(x) >= 0
```

point_from_pseudovaluation_on_polynomial_ring(*v0*, *parameter=None*)
Return the point corresponding to a pseudo-valuation on a polynomial ring.

INPUT:

- v0 – a discrete pseudo-valuation on a polynomial ring over the base field K, extending the base valuation v_K

- `parameter` – a parameter for the function field (default: `None`)

OUTPUT:

The point on this Berkovich line corresponding to `v0`, with respect to `parameter`. If `parameter` is not given, we assume that it is the standard parameter x.

EXAMPLES:

```
sage: from mclf import *
sage: from sage.all import GaussValuation
sage: F.<x> = FunctionField(QQ)
sage: v2 = QQ.valuation(2)
sage: X = BerkovichLine(F, v2)
sage: v0 = GaussValuation(F._ring, v2)
sage: X.point_from_pseudovaluation_on_polynomial_ring(v0, 2*x)
Point of type II on Berkovich line, corresponding to v(x) >= 1
```

point_from_valuation(*v*)

Return the point corresponding to a discrete valuation.

INPUT:

- `v` – a discrete valuation on the function field of this Berkovich line

OUPUT:

The point corresponding to v.

If the restriction of v to the constant base field is trivial, then we obtain a point of type I. Otherwise, the restriction of v to K must be equivalent to the base valuation v_K, and in this case we obtain a point of type II.

EXAMPLES:

```
sage: from mclf.berkovich.berkovich_line import BerkovichLine
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: v = F.valuation(x^2+1/2)
sage: X.point_from_valuation(v)
Point of type I on Berkovich line given by x^2 + 1/2 = 0
sage: xi = X.point_from_discoid(x, 1/2)
sage: xi1 = X.point_from_valuation(xi.valuation())
sage: xi.is_equal(xi1)
True
```

class mclf.berkovich.berkovich_line.**PointOnBerkovichLine**

Bases: `SageObject`

A point on a Berkovich projective line.

We only allow two different types of points:

- **Type I, algebraic: these are the points that come from a closed point** on the (algebraic) projective line over the completed base field.
- **Type II: these are the points which correspond to discrete valuations** on the function field whose residue field is a function field over the residue base field

In particular, the Gauss valuation on $F = K(x)$ with respect to the parameter x corresponds t a point ξ^g of type II on X which we call the *Gauss point*.

The set X has a canonical partial ordering in which the Gauss point is the smallest elements. All point of type I are maximal elements.

base_field()
Return the base field of this Berkovich line.

base_valuation()
Return the valuation on the base field of this Berkovich line.

berkovich_line()
Return the Berkovich line of which this point lies.

function_field()
Return the function field of this Berkovich line.

inverse_parameter()
Return the inverse parameter of the polynomial ring on which `self` is defined.

Let $\phi : F \to F$ be the automorphism of the function field $F = K(x)$ such that $Y := \phi(x)$ is the *parameter* used to define `self`. Then the inverse parameter is $z := \phi^{-1}(x)$.

EXAMPLES:

```
sage: from mclf import *
sage: from sage.all import GaussValuation
sage: F.<x> = FunctionField(QQ)
sage: v_2 = QQ.valuation(2)
sage: X = BerkovichLine(F, v_2)
sage: v0 = GaussValuation(F._ring, v_2)
sage: xi = X.point_from_pseudovaluation_on_polynomial_ring(v0, x/2)
sage: xi
Point of type II on Berkovich line, corresponding to v(1/x) >= 1
sage: xi.parameter()
1/x
sage: xi.inverse_parameter()
1/x
```

At the moment, only parameters y of the form c*x or 1/x are allowed.

```
sage: y = (2*x-1)/(x+2)
sage: xi = X.point_from_pseudovaluation_on_polynomial_ring(v0, y)
Traceback (most recent call last):
...
AssertionError: y must be c*x or 1/x
```

parameter()
Return the parameter of the polynomial ring on which `self` is defined.

The point `self` corresponds to a discrete pseudo-valuation v which is the extension of a pseudo-valuation v_0 on $K[y]$, where y is the *parameter* in question.

class mclf.berkovich.berkovich_line.**TypeIIPointOnBerkovichLine**(X, v)
Bases: *mclf.berkovich.berkovich_line.PointOnBerkovichLine*

A point of type II on a Berkovich line.

INPUT:

- `X` – a Berkovich line over a valued field K
- `v` – a discrete valuation on the function field of X extending the base valuation

OUTPUT:

The type-II-point ξ on X corresponding to v.

It is also possible to replace `v` by a pair `(v0, y)`, where `v0` is a discrete valuation on a polynomial ring $K[x]$, and `y` is a parameter for the function field of the Berkovich line. Then ξ is the point corresponding to the valuation v on the function field $F = K(x)$ which pulls back to v_0 via the inclusion $K[x] \to F$ that sends x to y.

NOTE:

At the moment, we only allow the generators y of the form cx or $1/x$.

approximation()
Return an approximation of `self`. For a point of type II, `self` is already an approximation of itself.

discoid(*certified_point=None*)
Return a representation of the discoid of which this type II point is the unique boundary point.

INPUT:

- `certified_point` (default=None) – this argument is not used for type-II-points

OUTPUT:

A pair (f, s), where f is a polynomial in the generator x of the function field of X which is irreducible over $\hat{K}$, or $1/x$, and where s is a nonnegative rational number. This data represents a discoid D via the condition $v_\xi(f) \geq s$.

Then `self` is the unique boundary point of D, and if, moreover, `self` is not the Gauss point then D contains precisely the points ξ which are greater or equal to `self`.

The representation (f, s) is normalized as follows:

- if this point lies in the closed unit disk then f is monic and integral, and $s \geq 0$. We either have $(f, s) = (x, 0)$ (if this point is the Gauss point and D the closed unit disk) or $s > 0$.
- if this point is not in the closed unit disk then $s > 0$, and f is either integral with constant term 1 and only strictly positive slopes, or $f = 1/x$. In the first case, D does not contain the point at infinity, in the second case it does. In both cases, D is disjoint from the closed unit disk.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: v_2 = QQ.valuation(2)
sage: X = BerkovichLine(F, v_2)
sage: X.gauss_point().discoid()
(x, 0)
sage: X.infty().discoid()
(1/x, +Infinity)
```

improved_approximation()
Return an improved approximation of `self`.

This is meaningless for type-II-points, so self is returned.

is_gauss_point()
Return True if self is the Gauss point.

is_in_unit_disk()
True is self is contained in the unit disk.

is_inductive()
True if self corresponds to an inductive pseud-valuation. This is always true for points of type II.

is_infinity()
Check whether self is the point at infinity.

is_limit_point()
True is self corresponds to a limit valuation. This is never true for points of type II.

parameter()
Return the parameter with respect to which this point is defined.

This is either x (if the point lies in the unit disk) or $1/x$ otherwise.

pseudovaluation()
Return the pseudovaluation corresponding to this point.

OUTPUT:

Since self is a point of type II, the output is a discrete valuation on the function field of the underlying Berkovich line.

pseudovaluation_on_polynomial_ring()
Return the pseudo-valuation on the polynomial ring 'K[y]' corresponding to self, where y is either x or $1/x$ depending on whether self lies in the standard closed unit disk or not.

type()
Return the type of self.

valuation()
Return the valuation corresponding to this point.

OUTPUT:

The discrete valuation on the function field of the underlying Berkovich line corresponding to this point.

class mclf.berkovich.berkovich_line.**TypeIPointOnBerkovichLine**(*X*, *v*)
Bases: *mclf.berkovich.berkovich_line.PointOnBerkovichLine*

An algebraic point of type I.

INPUT:

- X – a Berkovich projective line over a valued field K
- v – an infinite discrete pseudovaluation on the function field $F = K(x)$

OUTPUT: a point of type I on X

Here the point ξ on X corresponds to the discrete pseudo-valuation v on the function field $F = K(x)$.

Alternatively, v can be a pair (v_0, y), where v_0 is an infinite discrete pseudo-valuation on a polynomial ring over K and y is a generator of the function field F. Then v is the infinite discrete pseudo-valuation of F whose restriction to the subring ring $K[y]$ is equal to v_0.

approximation(*certified_point=None*, *require_maximal_degree=False*)
Return an approximation of this point.

INPUT:

- certified point (default=None) – a point on the Berkovich line
- require_maximal_degree (default=False) – boolean

OUTPUT:

A a point which is inductive and approximates `self`, in such a way that we can distinguish `self` from `certified point`.

If `self` is an inductive point, then `self` is returned. Otherwise, `self` is a limit point, and the output is a point of type II greater or equal to `self` (i.e. corresponding to a discoid containing `self`). If `certified_point` is not None and distinct from `self`, then the output is not greater or equal to `certified_point`.

If `require_maximal_degree` is `True` then any approximation will have the same degree as the limit point. Here the *degree* of an inductive point means the degree of the last key polynomial describing it, and the degree of a type-I-point is the degree of its minimal polynomial.

discoid(*certified_point=None*)

Return a representation of a discoid approximating `self`.

INPUT:

- `certified_point` (default=None) – a point of type II

OUTPUT:

A pair (f, s), where f is a polynomial in the generator x of the function field of X which is irreducible over $\hat{K}$, or $f = 1/x$, and where s is a nonrational number, or is equal to ∞. This data represents a (possibly degenerate) discoid D via the condition $v_\xi(f) \geq s$.

D as above is either the degenerate discoid with $s = \infty$ which has `self` as the unique point, or D is an approximation of `self` (this simply means that `self` is contained in D). If `certified_point` is given and is not equal to `self` then it is guaranteed that it is not contained in D.

We further demand that the discoid D is either contained in the closed unit disk, or is disjoint from it. Such discoids correspond one-to-one to points of type II.

equation()

Return an equation for the Galois orbit of this point.

OUTPUT:

An element f of the function field of X which is either an irreducible polynomial in the standard generator x, or is equal to $1/x$, and such that $v_\xi(f) = \infty$.

function_field_valuation()

Return the function field valuation corresponding to this point

OUTPUT:

the discrete valuation on the function field $F = K(x)$ which corresponds to the image of this point on $X = \mathbb{P}^1_K$ (which is, by hypothesis, a closed point).

improved_approximation()

Return an improved approximation of `self`.

is_gauss_point()

Return True if self is the Gauss point.

is_in_unit_disk()

True is self is contained in the unit disk.

is_inductive()

Check whether this points corresponds to an inductive valuation.

is_limit_point()

Check whether this point corresponds to a limit valuation.

pseudovaluation()
Return the pseudovaluation corresponding to this point.

OUTPUT:

a discrete pseudovaluation on the rational function field of the Berkovich line of which `self` is a point.

pseudovaluation_on_polynomial_ring()
Return the pseudovaluation representing `self`.

OUTPUT:

A discrete pseudovaluation on a polynomial subringring $K[y]$ from which `self` is induced. It is either inductive or a limit valuation.

type()
Return the type of self

valuation()
Return the function field valuation corresponding to this point.

OUTPUT:

the normalized discrete valuation on the function field of the Berkovich line corresponding to this point of type I.

This should not be confused with the *pseudovaluation* usually associated with a type-I-point.

EXAMPLES:

```
sage: from mclf.berkovich.berkovich_line import BerkovichLine
sage: F.<x> = FunctionField(QQ)
sage: v_2 = QQ.valuation(2)
sage: X = BerkovichLine(F, v_2)
sage: X.gauss_point().valuation()
2-adic valuation
sage: xi = X.point_from_discoid(x+1, Infinity)
sage: xi.valuation()
(x + 1)-adic valuation
```

mclf.berkovich.berkovich_line.**inverse_generator**(*y*)
Return the inverse generator of a given generator of a rational function field.

INPUT:

- `y` - a generator of a rational function field $F = K(x)$

OUTPUT:

The inverse generator for y. So if $\phi : F \to F$ is the automorphism of F such that $\phi(x) = y$ then $z := \phi^{-1}(x)$ is the inverse generator.

mclf.berkovich.berkovich_line.**is_generator**(*y*)
Test whether an element is a generator of a rational function field.

INPUT:

- `y` - an element of a rational function field $F = K(x)$

OUTPUT:

`True` if $F = K(y)$, `False` otherwise.

mclf.berkovich.berkovich_line.**valuation_from_discoid**(*vK*, *f*, *s*)
Return the inductive valuation corresponding to a discoid.

INPUT:

- vK – a discrete valuation on a field K
- f – a nonconstant monic integral polynomial over K
- s – a nonnegative rational number, or ∞

OUTPUT:

an inductive valuation v on the domain of f, extending vK, corresponding to the discoid D defined by $w(f) \geq s$. In other words, this means that D defined above is irreducible (and hence a discoid), and v is its unique boundary point.

If D is not irreducible, an error is raised.

EXAMPLES:

An example that created an error in a previous version:

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: v_2 = QQ.valuation(2)
sage: f =  x^6 - 8030/3241*x^5 + 24468979*x^4 + 14420644*x^3 + 24136511*x^2 +␣
↪5386/1505*x + 3981/5297
sage: valuation_from_discoid(v_2, f, 76/15)
[ Gauss valuation induced by 2-adic valuation, v(x + 1) = 2/3, v(x^3 + 3*x^2 +␣
↪3*x - 3) = 38/15 ]
sage: _(f)
76/15
```

mclf.berkovich.berkovich_line.**valuations_from_inequality**(*vK*, *f*, *s*, *v0=None*)

Return the list of inductive valuations corresponding to the given inequlities.

INPUT:

- vK – a discrete valuation on a field K
- f – a nonconstant monic integral polynomial over K
- s – a nonnegative rational number, or ∞
- v0 – an inductive valuation on the parent of f (default: None)

OUTPUT:

a list of inductive valuations on the domain of f, extending vK, corresponding to the boundary points of the irreducible components of the affinoid defined by the condition $v(f) \geq s$. Note that these components are all discoids.

If v_0 is given then the output only includes the valuations greater or equal to v_0.

2.2 Finite subtrees of the Berkovich line

Let K be a field and v_K a discrete valuation on K. Let $X = \mathbb{P}^1_K$ be the projective line over K. Let X^{an} denote the (K, v_K)-analytic space associated to X. We call X^{an} the *Berkovich line* over K.

Let ξ^g be the *Gauss point* on X^{an}, corresponding to the Gauss valuation on the function field of X with respect to the canonical parameter x. Then X^{an} has a natural partial ordering for which ξ^g is the smallest element. With respect to this partial ordering, any two elements have a unique infimum.

A *Berkovich tree* is a finite (nonempty) subset T with the property that for any two elements in T the infimum is also contained in T. In particular, a T has a least element, called the *root* of the tree.

Given any finite subset S of X^{an}, there is now a unique minimal subtree T contaning S. We call T the tree *spanned* by S.

This module realizes finite subtrees of X^{an} as combinatorial objects, more precisely as *finite rooted combinatorial trees*. So a tree consists of a root, and a list of children. If the tree is a subtree of another tree, then there is a link to its parent.

AUTHORS:

- Stefan Wewers (2017-02-13): initial version

EXAMPLES:

```
<Lots and lots of examples>
```

class mclf.berkovich.berkovich_trees.**BerkovichTree**(*X*, *root=None*, *children=None*, *parent=None*)

Bases: SageObject

Create a new Berkovich tree T.

INPUT:

- X – a Berkovich line
- root – a point of X (default: None)
- children – a list of Berkovich trees on X (default = None)
- parent – a Berkovich tree or None (default: None)

OUTPUT:

A Berkovich tree T on X with root root, children children and parent parent. T may be empty (no root and no children), but if there are children then there must be root.

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: T = BerkovichTree(X); T
Berkovich tree with 0 vertices
sage: xi = X.gauss_point()
sage: T.find_point(xi)
```

adapt_to_function(*f*)

Add all zeroes and poles of f as leaves of the tree.

INPUT:

- f – a rational function on X

OUTPUT:

the new tree obtained by adding all zeroes and poles of f as vertices to the old tree.

add_point(*xi*)

Return the tree spanned by self and the point xi.

INPUT:

- `xi` – A point of type I or II on X

OUTPUT: T_1, T_2, where

- T_1 is the tree obtained from T_0 as a vertex.
- T_2 is the subtree of T_1 with root ξ

If T_0 has a parent, then the root of T_0 must be less than ξ. Therefore, the parent of T_1 will be the original parent of T_0.

Note that this command may change the tree T_0! For instance, ξ may become the root of T_1 and then T_0 has T_1 as new parent.

adjacent_vertices(*xi0*)
List all vertices of the tree adjacent to a given vertex.

berkovich_line()
Return the Berkovich line underlying this tree.

children()
Return the list of all children.

This is a deep copy of the internal list of children! Therefore, it cannot be used to change the tree.

copy()
Return a copy of self.

direction_from_parent()
Return the direction from the parent.

OUTPUT: the type V point η representing the direction from the root of the parent of `self` to the root of `self`.

If `self` has no parent, an error is raised.

direction_to_parent()
Return the direction to the parent.

OUTPUT: the type V point η representing the direction from the root of `self` to the root of its parent.

If `self` has no parent, an error is raised.

The direction is not well defined if the root of `self` is a point of type I. Therefore, an error is raised in this case.

find_point(*xi*)
Find subtree with root `xi`.

INPUT:

- `xi` – a point on the Berkovich line underlying `self`

OUTPUT:

The subtree T of `self` with root `xi`, or `None` if `xi` is not a vertex of `self`.

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: T = BerkovichTree(X); T
Berkovich tree with 0 vertices
```

Searching in the empty tree does not give an error anymore.:

```
sage: xi = X.gauss_point()
sage: T.find_point(xi)

sage: T.add_point(xi)
(Berkovich tree with 1 vertices, Berkovich tree with 1 vertices)
sage: T.find_point(xi)
Berkovich tree with 1 vertices
```

graph()
Return a graphical representation of self.

OUTPUT:

G, vert_dict,

where G is a graph object and vert_dict is a dictionary associating to a vertex of G the corresponding vertex of self.

has_parent()
Return True if self has a parent.

is_leaf()
Return True if self is a leaf.

leaves(*subtrees=False*)
Return the list of all leaves.

If `subtrees` is `True`, then we return the list of subtrees corresponding to the leaves.

make_child(*new_child*, *check=True*)
Make `new_child` a child of self.

INPUT:

- `new_child` – a Berkovich tree
- `check` – a boolean (default `False`)

We make the tree `new_child` a child of `self`. For this to make sense, two conditions have to be satisfied:

- the root of `new_child` has to be strictly greater than the root of `self`
- the root of `new_child` has to be incomparable to the roots of the already existing children of `self`

These conditions are checked only if `check` ist `True`.

Note:

This changes both trees `self` and `new_child`.

make_parent(*parent*)
add `parent` as parent of self.

parent()
Return the parent of self.

paths()
Return the list of all directed paths of the tree.

OUTPUT:

the list of all directed paths of the tree, as a list of pairs (ξ_1, ξ_2), where ξ_2 is a child of ξ_1.

permanent_completion()

Return the permanent completion of `self`.

OUTPUT:

A Berkovich tree T_1 which is the permanent completion of `self`.

A Berkovich tree tree T on a Berkovich line X over (K, v_K) is called *permanently complete* if for all finite extensions (L, v_L) of (K, v_K), the inverse image of the set of vertices of T in X_L is again the set of vertices of a Berkovich tree. It is easy to see that for any Berkovich tree T there exists a minimal refinement T_1 which is permanently complete. It is called the *permanent completion* of T.

ALGORITHM:

Let ξ_0 be the root and $\xi_1, \ldots, \xi_n$ the leaves of T. To compute T_1 we consider, for $i = 1, \ldots, n$, the path $\gamma = [\xi_0, \xi_n]$ and the function on γ which maps a point ξ to the number of geometric components of the discoid D_ξ. We add the jumps of this function to T. Having done this for all i we obtain the permant completion T_1 of T.

EXAMPLES:

```
sage: from mclf import *
sage: FX.<x> = FunctionField(QQ)
sage: v_2 = QQ.valuation(2)
sage: X = BerkovichLine(FX, v_2)
sage: xi0 = X.point_from_discoid(x^4+2, 5)
sage: T = BerkovichTree(X, xi0)
sage: T.permanent_completion()
Berkovich tree with 3 vertices
```

position(*xi*)

Find the position of `xi` in the tree T=self.

INPUT:

- `xi` – a point on the Berkovich line underlying T

OUTPUT:

xi1, T1, T2, is_vertex,

where

- xi1 is the image of xi under the retraction map onto the total space of T
- T1 is the smallest subtree of T whose total space contains xi1
- T2 is the child of T1 such that xi1 lies on the edge connecting T1 and T2 (or is equal to T1 if xi1 is the root of T1)
- is_vertex is True if xi1 is a vertex of T (which is then the root of T1) or False otherwise

print_tree(*depth=0*)

Print the vertices of the tree, with identation corresponding to depth.

It would be nicer to plot the graph and then list the points corresponding to the vertices.

remove_child(*child*)

Remove child from list of children of `self`.

INPUT:

- `child` – a Berkovich tree

We remove `child` from the list of children of `self`. If `child` is not in this list, an error is raised.

Note:

This function changes both `self` and `child`.

remove_point(*xi*, *test_inequality=True*)
Remove a point from the tree, if possible.

INPUT:

- `xi` – a point of type I or II on the Berkovich line
- `test_inequality -- a boolean (default: ``True)`

OUTPUT:

the tree obtained from `self` by removing, if possible, the vertex with root ξ.

Removing the vertex with root ξ is possible if a vertex with root ξ exists, and if it has at most one child. Otherwise, nothing is changed.

Note that the vertex to be removed may be the root of this tree. If this is the case and there is no child, then an empty tree is return *and* the tree is remove as a child from its parent.

If `test_inequality` is `False` then it is assumed that ξ is greater or equal to the root of `self`. This saves us a test for inequality.

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, v_2)
sage: T = BerkovichTree(X)
sage: T, _ = T.add_point(X.gauss_point())
sage: T = T.remove_point(X.gauss_point()); T
Berkovich tree with 0 vertices
sage: xi_list = [xi for xi, m in X.divisor(x*(x^2+2))]
sage: for xi in xi_list: T, _ = T.add_point(xi)
sage: T
Berkovich tree with 5 vertices
sage: T.remove_point(xi_list[0])
Berkovich tree with 4 vertices
```

root()
Return the root of the tree.

subtrees()
Return the list of all subtrees.

vertices()
Return the list of all vertices.

mclf.berkovich.berkovich_trees.**component_jumps**(*xi0*, *xi1*)
Helper function for `permanent_completion`.

mclf.berkovich.berkovich_trees.**create_graph_recursive**(*T*, *G*, *vertex_dict*, *root_index*)
Create recursively a graph from a Berkovich tree.

mclf.berkovich.berkovich_trees.**replace_subtree**(*T1*, *T2*)
Replace a subtree of a Berkovich tree by another tree.

INPUT:

- `T1`, `T2` - Berkovich trees with the same root

It is assumed that T_1 has a parent, so it is a proper subtree of an affinoid tree T_0. We replace the subtree T_1 with T_2.

NOTE:

```
This changes the tree `T_0`; therefore this function must be
used carefully.
```

2.3 Points of type V on the Berkovich line.

Let X^{an} be a Berkovich line over a discretely valued field K. A "point" η of type V on X^{an} corresponds to a pair $(v, \bar{v})$, where v is a type-II-valuation and $\bar{v}$ is a function field valuation on the residue field of v. We call v the "major valuation" and $\bar{v}$ the "minor valuation" associated to η.

Note that η is not, properly speaking, a point on the analytic space X^{an}, but rather a point on the adic space X^{ad}.

Equivalent ways to describe η are:

- the rank-2-valuation given as the composition of v and $\bar{v}$
- a "residue class" on X^{an}; more precisely, η corresponds to a connected component of $X^{an} - \{\xi\}$, where ξ is the type-II-point corresponding to v (and then ξ is the unique boundary point of the residue class)
- an "open discoid": more precise, a pair (ϕ, s), where ϕ is a rational function such that the open discoid

$$D = \{v \mid v(\phi) > s\}$$

  is the residue class corresponding to η. Moreover, either ϕ of $1/\phi$ is a monic, integral and irreducible polynomial in x or in $1/x$.
- a "tangent vector" on X^{an}; more precisely a group homomorphism

$$\partial : K(x)^* \to \mathbb{Z}$$

with the following properties: let (ϕ, s) be the discoid representation of η. We define, for $t \geq s$, the valuation v_t as the valuation corresponding to the boundary point of the open discoid $v(\phi) > t$. Then $\partial(f)$ is the right derivative at $t = s$ of the function

$$t \mapsto v_t(f).$$

The most convenient way to determine a point of type V is as follows. Let ξ_1 be a point of type II and ξ_2 be of type I or II, distinct from ξ_1. Then

$$\eta = \eta(\xi_1, \xi_2)$$

is the point of type V corresponding to the connected component of $X - \{\xi_1\}$ containing ξ_2. We call η the *direction* from ξ_1 towards ξ_2.

class mclf.berkovich.type_V_points.**TypeVPointOnBerkovichLine**(*xi0*, *xi1*)

Bases: `SageObject`

A point of type V on the Berkovich line.

Let ξ_1 be a point of type II, and ξ_2 a point of type I or II. Then we can define the point of type V $\eta := \eta(\xi_1, \xi_2)$ as the unique residue class with boundary point ξ_1 containing ξ_2.

INPUT:

- `xi0` – point of type II
- `xi1` – arbitrary point of X, distinct from xi0

OUTPUT:

The type-V-point corresponding to the connected component of $X^{an} - \xi 0$ which contains $\xi 1$.

EXAMPLES:

```
sage: from mclf import *
sage: K = QQ
sage: vK = K.valuation(2)
sage: F.<x> = FunctionField(K)
sage: X = BerkovichLine(F, vK)
sage: xi1 = X.point_from_discoid(x,1)
sage: xi2 = X.point_from_discoid(x^2+4,3)
sage: eta = TypeVPointOnBerkovichLine(xi1, xi2)
```

We see that `eta` represents an open disk inside the closed unit disk.

```
sage: eta
Point of type V given by residue class v(x + 2) > 1
```

Here is an example of a type-V-point representing an open disk in the complement of the closed unit disk:

```
sage: xi0 = X.gauss_point()
sage: xi3 = X.point_from_discoid(2*x+1, 2)
sage: eta = TypeVPointOnBerkovichLine(xi0, xi3)
sage: eta
Point of type V given by residue class v(1/x) > 0
```

We check that `xi0` lies outside the open disk and `xi3` inside:

```
sage: eta.is_in_residue_class(xi0)
False
sage: eta.is_in_residue_class(xi3)
True

sage: xi4 = X.point_from_discoid(2*x+1, 4)
sage: TypeVPointOnBerkovichLine(xi3, xi4)
Point of type V given by residue class v((2*x + 1)/x) > 3
sage: TypeVPointOnBerkovichLine(xi4, xi3)
Point of type V given by residue class v(1/2*x/(x + 1/2)) > -5
```

The following example shows that the minor valuation is computed correctly

```
sage: xi5 = X.point_from_discoid(1/x,1)
sage: eta = TypeVPointOnBerkovichLine(xi0,xi5)
sage: eta
Point of type V given by residue class v(1/x) > 0
sage: eta.minor_valuation()
Valuation at the infinite place
```

berkovich_line()
Return the Berkovich line underlying the point.

boundary_point()
Return the boundary point of the type-V-point.

minor_valuation()
Return the minor valuation of this type V point.

open_discoid()
Return the representation of self as an open discoid.

INPUT:

- self: a point of type V on a Berkovich line

OUTPUT:

a pair (ϕ, s), where ϕ is a rational function and s a rational number is such that

$$D = \{v \in X \mid v(\phi) > s\}$$

is the open discoid representing the type-V-point `self`.

Either ϕ of $1/\phi$ is a monic, integral and strongly irreducible polynomial in x or in $1/x$.

point_inside_discoid(*t*)
Return the point inside the residue class at the value t.

The type-V-point corresponds to an open discoid defined by

$$v(\phi) > s.$$

For for a rational number $t > s$ we can define the type-II-point ξ_t corresponding to the closed discoid defined by

$$v(\phi) >= t.$$

If $t = \infty$ we obtain the type-I-point corresponding to $\phi = 0$.

INPUT:

- `t` – a rational number or Infinity

OUTPUT:

The point ξ_t inside the residue class corresponding to the closed discoid defined by $v(\phi) >= t$.

If $t <= s$ then an error is raised.

2.4 Affinoid subdomains of the Berkovich line.

Let K be a field, v_K a discrete valuation on K and X the Berkovich line over K, with respect to v_K.

In this file we realize a Sage class which allows us to create and work with strictly affinoid subdomains of X.

Let T be a Berkovich tree in X and let $r : X \to T$ denote the canonical retraction map. Let S be a nonempty proper subset of $V(T)$. We define $\bar{S}$ as the union of S and of all edges connecting two points of S. Then the inverse image $U := r^{-1}(\bar{S})$ is an affinoid subdomain of X. We use the notation $U = U(T, S)$.

We say that a Berkovich tree T *supports* an affinoid domain U if there exists a nonempty proper subset S of $V(T)$ with $U = U(T, S)$. If this is the case then S consists exactly of the vertices of T which lie in U.

Given any affinoid domain U, there exists a unique minimal tree T which supports U. Moreover, every tree T' which contracts to T' supports U as well.

AUTHORS:

- Stefan Wewers (2017-07-29): initial version

EXAMPLES:

```
<Lots and lots of examples>
```

TO DO:

- more doctests
- add missing functions: intersection, ..
- see if we can remove some obsolete functions
- improve `point_close_to_boundary`

class mclf.berkovich.affinoid_domain.**AffinoidDomainOnBerkovichLine**(*T*)

Bases: `SageObject`

Return the affinoid domain corresponding to the affinoid tree `T`.

Objects of this class represent (generic) affinoid domains on the Berkovich line.

INPUT:

- T – an affinoid tree

OUTPUT:

The affinoid corresponding to `T`.

EXAMPLES:

```
sage: from mclf import *
sage: K = QQ
sage: vK = K.valuation(2)
sage: F.<x> = FunctionField(K)
sage: X = BerkovichLine(F, vK)
```

TO DO:

-

affinoid_subtree(*xi0*, *is_in=None*)

Return the affinoid subtree with given root.

This function is used inductively to construct a tree representing the affinoid U, if such a tree is not explicitly given, and the affinoid is defined in some other way (as a rational domain, or as a union of other affinoid domains,..).

INPUT:

- `xi0` - a point of type II
- `is_in` – a boolean, or `None` (default `None`)

OUTPUT: an affinoid tree with root $\xi 0$ which represents the intersection of U with D_{ξ_0}, the set of points $\geq \xi_0$ (a closed discoid with boundary point ξ_0, or the full Berkovich line if xi_0 is the Gauss point).

If `is_in` is given, we assume it is equal to the truth value of "ξ_0 lies in this affinoid". This is useful to avoid an extra test for membership.

affinoid_subtree_in_hole(*eta*)

Return the affinoid subtree with given root.

This is a helper function for `affinoid_subtree`.

INPUT:

- `eta` - a point of type V

OUTPUT:

We assume that η represents a downward hole of this affinoid U. This means that the boundary point of η lies in U but η does not. We return an affinoid tree T whose root is the boundary point of η, representing the affinoid

$$(X \backslash D_\eta) \cup (D_\eta \cap U)`.$$

berkovich_line()
Return the Berkovich line underlying this affinoid.

boundary()
Return the Shilov boundary of the affinoid.

The Shilov boundary is a finite set of type-II-points contained in the affinoid with the property that the valuative function of every rational function which is regular on the affinoid takes a minimum on this set.

The Shilov boundary is simply the union of the boundaries of the connected components.

components()
Return the list of connected components of this affinoid.

connected_component_tree(*xi0*)
Return the tree of the connected component of this affinoid with given root.

INPUT:

- `xi0` – a point type II or V

OUTPUT:

If ξ_0 is a point of type II, then we return an affinoid tree underlying the connected component of this affinoid U with minimal point ξ_0.

It is assumed but not checked that ξ_0 lies in this affinoid.

If ξ_0 is of type V then we return the branch of this tree in the direction of ξ_0. This has the effect of "filling in all holes" which do not lie in the open discoid D_{ξ_0}. It does *not* correspond to the intersection with D_{ξ_0}.

Note:

This is the generic algorithm for the parent class `AffinoidDomainOnBerkovichLine`. It is assumed that the underlying affinoid tree has already been computed. Otherwise we run into an infinite loop.

intersection(*V*)
Return the affinoid which is the intersection of `self` with `V`.

Not yet implemented.

intersection_with_unit_disk()
Return the intersection of this affinoid with the unit disk.

is_empty()
Return wether this affinoid is the empty set.

is_full_berkovich_line()
Return whether this affinoid is equal to the full Berkovich line.

is_in(*xi*)
Return whether `xi` lies on the affinoid.

INPUT:

- `xi` – a point of type I, II or V

OUTPUT:

`True` if `xi` lies on the affinoid, `False` otherwise.

minimal_points (*xi0=None*)

Return the minimal points of this affinoid greater than a given point.

INPUT:

- `xi0` – a point of type II, or `None` (default `None`)

OUTPUT:

The list of all minimal points of this affinoid which are $\geq \xi_0$.

number_of_components ()

Return the number of connected components of ths affinoid.

point_close_to_boundary (*xi0*)

Return a type-I-point inside the affinoid, close to `xi0`.

INPUT:

- `xi0` – a boundary point of the affinoid `self`

OUTPUT:

A type-I-point `xi1` on the affinoid $U :=$ `self` which is "close" to the boundary point `xi0`.

The latter means that `xi1` maps onto the irreducible components of the canonical reduction of U corresponding to `xi0`.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
sage: U = rational_domain(X, 2/x/(x+1))
sage: U
Elementary affinoid defined by
v(1/x) >= -1
v(1/(x + 1)) >= -1
<BLANKLINE>

sage: xi0 = U.boundary()[0]
sage: U.point_close_to_boundary(xi0)
Point of type I on Berkovich line given by x + 2 = 0
```

At the moment, our choice of point close to the boundary is not optimal, as the following example shows:

```
sage: U = rational_domain(X, 2/(x^2+x+1))
sage: U
Elementary affinoid defined by
v(1/(x^2 + x + 1)) >= -1

sage: xi0 = U.boundary()[0]
sage: U.point_close_to_boundary(xi0)
Point of type I on Berkovich line given by x^2 + 3*x + 1 = 0
```

The point at infinity is also inside U and 'close to the boundary', and has smaller degree than the point produced above.

The following raised an error in an earlier version

```
sage: f = (-2/25*x^6 - 4*x^5 - 1351/225*x^4 - 52/225*x^3 - 173/225*x^2 - 2/
↪9*x + 2/3)/(x^2 + 2/3*x)
sage: h = valuative_function(X, f)
sage: U = h.affinoid_domain()
sage: U
Affinoid with 2 components:
Elementary affinoid defined by
v(x) >= 0
v(1/x) >= -1/2
Elementary affinoid defined by
v((2*x^2 + 1)/x^2) >= 2
<BLANKLINE>

sage: U.point_close_to_boundary(U.boundary()[1])
Point of type I on Berkovich line given by x^2 + 2 = 0
```

Todo:

- Use a better strategie to find a point of minimal degree.

simplify()
Simplify this affinoid.

This only changes the internal representation by an "affinoid tree". Very likely, this is unnecessary because the simplification has already occured when the affinoid was first constructed.

tree()
Return the Berkovich tree representing this affinoid.

union(*V*)
Return the affinoid which is the union of `self` with `V`.

Need new implementation.

class mclf.berkovich.affinoid_domain.**AffinoidTree**(*X*, *root=None*, *children=None*, *parent=None*, *is_in=False*)
Bases: *mclf.berkovich.berkovich_trees.BerkovichTree*

A marked Berkovich tree representing an affinoid subdomain.

An AffinoidTree is a Berkovich tree T in which every vertex has an additional flag "is_in" with value `True` or `False`. It represents an affinoid subdomain U in the way explained above.

INPUT:

- `X` – a Berkovich line
- `root` – a point on `X` or None (default = None)
- `children` – a list of affinoid trees or None (default = None)
- `parent` – an affinoid tree or none (default = None)
- `is_in` – a boolean or None (default = None)

OUTPUT:

An affinoid tree on `X`. It is either empty (if only `X` is given) or it has root, parent, children and the flag `is_in` as given by the extra parameters.

EXAMPLES:

```
sage: from mclf import *
sage: K = QQ
sage: vK = K.valuation(2)
sage: F.<x> = FunctionField(K)
sage: X = BerkovichLine(F, vK)
```

compute_connected_components (*comp_list*, *new_comp*)

Compute the connected components of the represented affinoid.

INPUT:

- `comp_list` – a list (of lists of lists)
- `new_comp` – a list (of lists)

OUTPUT:

- all connected components whose root is a vertex of T=‘‘self‘‘ are added to the list `comp_list`.
- all boundary_lists which belong to T and to the connected component which contains the root of T are added to `new_comp` (in particular, if the root of T does not lie in the affinoid then the list is unchanged).

Here a *boundary list* is a list of type-V-points which represent holes of the affinoid with a common boundary point. A *connected component* is a list of boundary lists.

EXAMPLES:

```
sage: from mclf import *
sage: K = QQ
sage: vK = K.valuation(2)
sage: F.<x> = FunctionField(K)
sage: X = BerkovichLine(F, vK)
```

connected_components (*xi0=None*)

Return a list of affinoid trees representing the connected components below a given point.

INPUT:

- `xi0` – a point of type II or V

OUTPUT:

A list of affinoid trees representing the connected components of the affinoid corresponding to this tree, which are $\geq$ to the given point ξ_0. If it is not given, then we ignore this condition.

Note that ξ_0 may be of type V.

copy (*parent=None*)

Return a copy of self, force `parent` as parent.

WARNING! something is wrong with this function!!

holes (*upward_hole=True*)

Return the holes of this affinoid tree.

OUTPUT:

A list of triples (T_1, T_2, η), where T_1, T_2 are subtrees of `self` and η is a point of type V, satisfying the following conditions:

- T_2 is a child of T_1, or vice versa
- the root of T_1 is a boundary point of the affinoid underlying `self`

- the root of T_2 does not lie in the affinoid
- η is the direction from the root of T_1 to the root of T_2

This implies that η is a *hole* of the affinoid represented by `self`.

intersection_with_unit_disk()
Return the tree representing the intersection with the unit disk.

is_in(*xi*)
Return True if `xi` lies in the affinoid U represented by `self`.

INPUT:

- `xi` – a point on the Berekovich space underlying this affinoid tree

Note that ξ may also be a point of type V.

To test this, we compute the image of xi under the retraction map onto the total space of T=self and check whether it lies on a vertex in U or on an edge connecting two vertices in U.

minimal_points(*xi0=None*)
Return the minimal points of the affinoid corresponding to this tree.

INPUT:

- `xi0` – a point of type II or V, or `None` (default `None`)

OUTPUT: the list of all minimal points of the affinoid corresponding to this tree, which are $\geq \xi_0$. If ξ_0 is not given, this condition is ignored.

Note that ξ_0 may be of type V.

root_is_in()
Return whether the root of `self` lies in the affinoid.

show()
Display a graphical representation of self.

simplify()
Simplify this tree without changing the represented affinoid.

class mclf.berkovich.affinoid_domain.**ClosedUnitDisk**(*X*)
Bases: *mclf.berkovich.affinoid_domain.AffinoidDomainOnBerkovichLine*

Return the closed unit disk.

The **closed unit disk** is the affinoid on the Berkovich line with function field $F = K(x)$ defined by the inequality

$$v(x) >= 0.$$

INPUT:

- `X` – a Berkovich line

OUTPUT:

The closed unit disk inside `X`.

EXAMPLES:

```
sage: from mclf import *
sage: K = QQ
sage: vK = K.valuation(3)
sage: F.<x> = FunctionField(K)
```

(continues on next page)

(continued from previous page)

```
sage: X = BerkovichLine(F, vK)
sage: ClosedUnitDisk(X)
Elementary affinoid defined by
v(x) >= 0
<BLANKLINE>
```

class mclf.berkovich.affinoid_domain.**ElementaryAffinoidOnBerkovichLine**(T)
Bases: *mclf.berkovich.affinoid_domain.AffinoidDomainOnBerkovichLine*

Return the elementary affinoid corresponding to a boundary list.

An "elementary affinoid" is a connected affinoid subdomain of a Berkovich line X which is the complement of a finite set of disjoint residue classes in X. It can be represented by a "boundary list" as follows.

A "boundary list" is a list of lists, whose entries at the lowest level are type-V-points on X. Each sublist contains the type-V-points with a common boundary point. The elementary affinoid corresponding to a "boundary list" is the complement of the residue classes corresponding to the type-V-points contained in the sublists. The set of boundary points of the type-V-points is exactly the Shilov boundary of the affinoid.

INPUT:

- `boundary_list` – a list of lists containing type-V-points.

OUTPUT:

The elementare affinoid corresponding to `comp_list`.

TO DO:

- we need a function which produces an (algebraic) type-I-point inside the affinoid.

inequalities()
Return the inequalities defining the elementary affinoid, as a string.

is_empty()
Return whether this affinoid is the empty set.

is_full_berkovich_line()
Return whether this affinoid is the full Beerkovich line.

class mclf.berkovich.affinoid_domain.**UnionOfDomains**(*affinoid_list*)
Bases: *mclf.berkovich.affinoid_domain.AffinoidDomainOnBerkovichLine*

Return the union of a list of affinoid domains.

INPUT:

- `affinoid_list` - a nonempty list of affinoid domains

OUTPUT:

The union of the affinoid domains in `affinoid_list`

connected_component_tree(*xi0*)
Return the tree of the connected component of this affinoid with given root.

INPUT:

- `xi0` – a point type II or V

OUTPUT:

If ξ_0 is a point of type II, then we return an affinoid tree underlying the connected component of this affinoid U with minimal point ξ_0.

It is assumed but not checked that ξ_0 is a minimal point of a connected component of U.

If ξ_0 is of type V then we return the branch of this tree in the direction of ξ_0. This has the effect of 'filling in all holes' which do not lie in the open discoid D_{ξ_0}. It does *not* correspond to the intersection with D_{ξ_0}.

is_in(*xi*)

Return whether `xi` lies in this affinoid.

INPUT:

- `xi` – a point on the Berkovich line (type V points are allowed)

OUTPUT: `True` if ξ lies on this affinoid.

minimal_points(*xi0=None*)

Return the minimal points of this affinoid greater than a given point.

INPUT:

- `xi0` – a point of type II, or `None` (default `None`)

OUTPUT:

The list of all minimal points of this affinoid which are $\geq \xi_0$.

mclf.berkovich.affinoid_domain.**all_polynomials**(*F*, *x*, *d*)

List all polynomials in x over F of degree d.

INPUT:

- F: a finite field F
- x: generator of a polynomial ring over F

OUTPUT:

an iterator which list all elements of F[x] of degree d.

mclf.berkovich.affinoid_domain.**rational_domain**(*X*, *f*)

Return the rational domain defined by the function `f`.

INPUT:

- `X` – a Berkovich line
- `f` – a nonconstant rational function on X

OUTPUT:

The rational domain

$$U := \{\xi \in X \mid v_\xi(f) \geq 0\}.$$

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
sage: rational_domain(X, (x^2+2)/x*(x+1)/2)
Affinoid with 2 components:
Elementary affinoid defined by
v(x^2 + 2) >= 3/2
Elementary affinoid defined by
v(x + 1) >= 1
<BLANKLINE>
```

f may be nonconstant:

```
sage: rational_domain(X, F(1/2))
The empty set
sage: rational_domain(X, F(1))
the full berkovich line
```

mclf.berkovich.affinoid_domain.**simplify_tree_at_vertex**(*T*, *T1*)

Simplify the affinoid tree at a given vertex.

This is now obsolete.

INPUT:

- `T` – an affinoid tree
- `T1` – a subtree of T

OUTPUT: the affinoid tree T is simplified, starting at the subtree T_1.

We check whether the root of T_1 (which is a vertex of T) may be contracted, or whether T_1 has a unique child which may be omitted. In the first case, we try to iterate this, if possible.

This may not simplify T as much as possible. However, if T has been obtained from a simplified

mclf.berkovich.affinoid_domain.**union_of_affinoid_trees**(*T1*, *T2*)

Return the tree representing the union of the affinoids with given trees.

This is now obsolete.

INPUT:

- `T1`, `T2` – affinoid trees

OUTPUT: the tree representing the union of the affinoids represented by T_1 and T_2.

2.5 Piecewise affine functions on the Berkovich projective line.

Let K be a field, v_K a discrete valuation on K and $X = \mathbb{P}^1_K$ the *Berkovich line* over K.

A continuous function

$$h : X \to \mathbb{R} \cup \{\pm\infty\}$$

is called *piecewise affine* if it factors over the retraction map

$$r_T : X \to T$$

onto a Berkovich subtree $T \subset X$, and the restriction of h to the edges of T are affine (with respect to the natural affine structure of a path on a Berkovich line).

The most important examples of piecewise linear functions are *valuative functions*. For instance, to any nonzero rational function $f \in F = K(x)$ we associate the function

$$h_f : X \to \mathbb{R} \cup \{\pm\infty\}, \xi \mapsto v_\xi(f).$$

A general valuative function on X is simply a rational multiple of a function of the form h_f. Any such function h can be written uniquely in the form

$$h = a_0 + \sum_i a_i \cdot h_{f_i}$$

where the f_i are irreducible polynomials in the parameter x, and the a_i are rational numbers, with $a_i \neq 0$ for $i > 0$.

Let h be a nonconstant piecewise affine function on X. Then the subset

$$U := \{\xi \in X \mid h(\xi) \geq 0\}$$

is an affinoid subdomain (unless it is empty, or the full Berkovich line X). If $h = h_f$ is the valuative function associated to a rational function f, then U is actually a rational domain.

AUTHORS:

- Stefan Wewers (2017-2019)

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
```

We can define a valuative function by a rational function on X:

```
sage: f = (x^2+2*x-2)/(2*x-1)
sage: h = valuative_function(X, f)
```

We check that the value of h is the valuation of f, at several points:

```
sage: xi = X.gauss_point()
sage: h(xi), xi.v(f)
(0, 0)
sage: xi = X.infty()
sage: h(xi), xi.v(f)
(-Infinity, -Infinity)
sage: xi = X.point_from_discoid(x, 3)
sage: h(xi), xi.v(f)
(1, 1)
```

We can also define a valuative function by a pair (L, a_0):

```
sage: L = [(x - 1, 2/3), (x + 1, 3/2)]
sage: a_0 = -3
sage: h = valuative_function(X, (L, a_0))
sage: xi = X.point_from_discoid(x + 1, 2)
sage: h(xi)
2/3
```

We can compute the affinoid domain given by the inequality $h(\xi) \geq 0$:

```
sage: h.affinoid_domain()
Affinoid with 2 components:
Elementary affinoid defined by
v(x - 1) >= 9/4
Elementary affinoid defined by
v(x + 1) >= 14/9
```

class mclf.berkovich.piecewise_affine_functions.**AffineFunction**(*gamma*, *a*, *b*)

Bases: SageObject

An affine function on a closed annuloid or a closed discoid.

INPUT:

- `xi1`, `xi2` – points on the Berkovich line X such that $\xi_1 <= \xi_2$
- `a`, `b` – rational numbers

OUTPUT:

the affine function h on the domain D of the path $\gamma = [\xi_1, \xi]$ defined by a and b. If $t : D \to \mathbb{R}$ is the standard parametrization, then

$$h(\xi) := a \cdot r_\gamma(\xi) + b.$$

berkovich_line()
Return the Berkovich line underlying this affine function.

domain()
Return the domain of definition of this affine function.

initial_point()
Return the initial point of the path underlying this affine function.

initial_value()
Return the initial value of this affine function.

is_constant()
Return whether this affine function is constant.

is_in_domain(*xi*)
Return whether a point is in the domain of definition of this affine function.

is_increasing()
Return whether this affine function is strictly increasing.

path()
Return the path underlying this affine function.

terminal_point()
Return the terminal point of the path underlying this affine function.

terminal_value()
Return the terminal value of this affine function.

class mclf.berkovich.piecewise_affine_functions.**DirectedPath**(*xi1*, *xi2*)
Bases: SageObject

A directed path on the Berkovic path.

INPUT:

- `xi1`, `xi2` – two points on the Berkovich line such that $\xi 1 \leq \xi_2$

OUTPUT:

the directed path $\gamma = [\xi_1, \xi_2]$.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
```

We can define a path by specifying the initial and the terminal point:

```
sage: xi1 = X.point_from_discoid(x, 1)
sage: xi2 = X.point_from_discoid(x^2 + 4, 5)
sage: gamma = DirectedPath(xi1, xi2)
sage: gamma
path from Point of type II on Berkovich line, corresponding to v(x) >= 1 to Point␣
↪of type II on Berkovich line, corresponding to v(x^2 + 4) >= 5
```

We use the *standard parametrization* for a path; it depends on the discoid representation of the terminal point:

```
sage: gamma.point(3)
Point of type II on Berkovich line, corresponding to v(x + 2) >= 3/2
```

Given a path $\gamma = [\xi_1, \xi_2]$, we define its *tube* D as follows. If ξ_2 is of type II, then D is the open annuloid with boundary point points ξ_1 and ξ_2. If ξ_1 is of type I, then $D := D_{\xi_1}$ is the discoid with boundary point ξ_1.:

```
sage: gamma.tube()
domain defined by
v(x + 2) > 1
v(1/(x^2 + 4)) > -5

sage: gamma.is_in_tube(X.gauss_point())
False

sage: gamma.is_in_tube(xi2)
False
```

berkovich_line()
Return the Berkovich line on which this path lives.

initial_parameter()
Return the initial parameter of this path.

OUTPUT:

a rational number s_0 such that $\gamma(s_0)$ is the initial point of this path γ.

initial_point()
Return the initial point of this path.

initial_slope()
Return the slope of this path at the initial point.

is_limit_path()
Return whether the terminal point of this path is a limit point.

terminal_parameter()
Return the terminal parameter of this path.

OUTPUT:

a rational number s_1 (or ∞)‘ such that $\gamma(s_1)$ is the terminal point of this path γ.

terminal_point()
Return the terminal point of this path.

tube()
Return the tube around this path.

Let ξ_1 be the initial and ξ_2 be the terminal point of this path. Then the *tube* is either the open annuloid with boundary points ξ_1 and $\xi 2$ (if ξ_2 is of type II) or the closed discoid with boundary point xi_1 (if ξ_2 is of type I).

class mclf.berkovich.piecewise_affine_functions.**Discoid**(*xi0*, *xi1=None*)
Bases: *mclf.berkovich.piecewise_affine_functions.Domain*

Return a closed discoid of the Berkovich line.

INPUT:

- `xi0` - a point on the Berkovich line X
- `xi1` (default: `None`) – another point on X

OUTPUT:

the closed discoid D consisting of all point on the Berkovich line which are greater or equal to ξ_0. If ξ_1 is given, then it is checked whether $\xi 1$ lies in D.

If ξ_0 is the Gauss point, then D is taken to be the closed discoid with minimal point ξ_0, containing ξ_1. If ξ_1 is not given, then D is taken to be the closed unit disk.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
sage: Discoid(X.gauss_point())
the closed discoid defined by v(x) >= 0
```

minimal_point()
Return the minimal point of this closed discoid.

class mclf.berkovich.piecewise_affine_functions.**Domain**(*X*, *inequalities*, *strict_inequalities*)
Bases: `SageObject`

A domain in the Berkovich line, defined by inequalities.

Objects of this class are used as domians of definition of affine and piecewise affine functions. Although they may be affinoid domains, this class has no relation to the class `AffinoidDomainOnBerkovichLine`.

INPUT:

- `X` – a Berkovich line
- `inequalities` – a list of pairs (f, a)
- `strict_inequalities` – a list of pairs (g, b)

OUTPUT: the domain of X which is the intersection of the domains defined by the inequalities

$$v(f) \leq a, \quad v(g) < b.$$

Here f, g are assumed to be nonconstant rational functions on X, and a, b rational numbers.

If the `inequalities` and `strict_inequalities` are both empty then the full Berkovich line is returned.

berkovich_line()
Return the Berkovich line underlying this domain.

contains_infty()
Return whether this domain contains the point at infinity.

inequalities()
Return the list of non-strict inequalities which are part of the definition of this domain.

is_full_berkovich_line()
Return whether this domain is the full Berkovich line.

minimal_point()
Return the minimal point of this domain.

Since an arbitrary domain has no minimal point, this method raises an error, unless this domain is the full Berkovich line.

strict_inequalities()
Return the list of strict inequalities which are part of the definition of this domain.

class mclf.berkovich.piecewise_affine_functions.**PiecewiseAffineFunction**(*D*, *a0*, *restrictions*)

Bases: SageObject

A piecewise affine function on a domain in the Berkovich line.

INPUT:

- D – a domain in the Berkovich line
- a0 – a rational number
- restrictions – a list of pairs (h_1, h_2)

OUTPUT:

a piecewise affine function h on D.

We assume that D is either a closed discoid, or the full Berkovich line X. Let ξ_0 be the *initial point* of the function, which is either the boundary point of D or, if $D = X$, the Gauss point on X.

The *restrictions* are the restrictions of h to proper open subdiscoids D_1 of D. It is assumed that h is constant, with value a_0, on the complement of these subdiscoids. The restriction of h to D_1 is given by a pair (h_1, h_2), where h_2 is the restriction of h to a proper closed subdiscoid $D_2 \subset D_1$ and h_1 is the restriction of h to the open annuloid $D_1 \backslash D_2$. It is assumed that h_1 is an *affine* functions, whereas h_2 is piecewise affine.

We allow D_2 to be empty, in which case h_2 is None and the domain of h_1 is an open discoid.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
```

We can define the "valuative function" of a rational function:

```
sage: f = (x^2 - 2) / x
sage: h = valuative_function(X, f)
sage: h
piecewise affine function on the Berkovich line, with initial value 0
```

A piecewise affine function can be evaluated at any point.

```
sage: xi = X.point_from_discoid(x, 2)
sage: h(xi)
-1
sage: xi.v(f)
-1
```

A piecewise affine function defines an affininoid subdomain (the point where the function takes nonnegative values).

sage: h.affinoid_domain() Elementary affinoid defined by v(x) >= 0 v(1/x) >= -1 <BLANKLINE>

affinoid_domain()
Return the affinoid domain defined by this function.

OUTPUT:

the affinoid subdomain of the domain of this function h, defind by the inequality

$$h(xi) \geq 0.$$

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
sage: h1 = valuative_function(X, 2*x)
sage: h1.affinoid_domain()
Elementary affinoid defined by
v(x) >= -1
<BLANKLINE>
sage: h2 = valuative_function(X, x*(x-1)/2)
sage: h2.affinoid_domain()
Affinoid with 2 components:
Elementary affinoid defined by
v(x - 1) >= 1
Elementary affinoid defined by
v(x) >= 1
```

berkovich_line()
Return the Berkovich line on which this function is defined.

domain()
Return the domain of this function.

find_next_points_with_value(*a*, *xi0=None*)
Return the next point where this function takes a given value, after a given point.

INPUT:

- `a` – a rational number
- `xi0` (default: `None`) – a point in the domain of this function

OUTPUT: The list of all points on the nerf of this function

- at which this function takes the value a,
- at which the function is not constant,
- which are strictly greater than ξ_0, and
- which are minimal with this property.

If `xi0` is `None` then the second condition is ignored.

NOTE:

```
In this form, the problem is not well defined. Note that the function
may be constant on pathes of the nerf. If this constant value is equal
to a, and xi0 lies on this path and ist not the terminal point, then
there is no minimal next point with value a.
```

find_next_zeroes(*xi0=None*)
Return the next zeroes of this function after a given point.

INPUT:

- `xi0` (default: `None`) – a point in the domain of this function

OUTPUT: The list of all points in the domain of this function which

- are zeroes of this function,
- are not in the constant locus of the function
- are greater or equal to ξ_0, and
- are minimal with this property.

If `xi0` is `None` then the third condition is ignored.

initial_point()
Return the initial point of this function.

This is the minimal point of the domain of this function.

initial_value()
Return the value of this function at the initial point.

is_constant()
Return whether this function is constant.

is_in_domain(*xi*)
Return whether a given point is in the domain of this function.

restrictions()
Return the restrictions of this piecewise affine functions.

OUTPUT:

a list of pairs (h_1, h_2), where h_1 is an *affine* function which is the restriction of this function h to an open subannuloid of the domain of h, and h_2 is the restriction of h to the closed discoid which the unique hole of the domain of h_1.

If the domain of h_1 is an open discoid (so there is no hole), then h_2 is `None`.

Together with the initial value, these restrictions define h, because h is constant on the complement of the domains of definitios of these restrictions.

mclf.berkovich.piecewise_affine_functions.**open_annuloid**(*xi0*, *xi1*)
Return an open annuloid.

INPUT:

- `xi0`, `xi1` – points of type II on the Berkovich line X

OUTPUT:

the open annuloid A with boundary points xi_0 and ξ_1. Note that $\xi 0$ and ξ_1 are *not* contained in D.

It is assumed that $\xi_0 < \xi_1$. This means that A cannot contain the Gauss point.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
sage: xi0 = X.point_from_discoid(x-1, 1)
sage: xi1 = X.point_from_discoid(x+1, 2)
sage: open_annuloid(xi0, xi1)
domain defined by
v(x + 1) > 1
v(1/(x + 1)) > -2
```

mclf.berkovich.piecewise_affine_functions.**open_discoid**(*xi0*, *xi1*)

Return an open discoid.

INPUT:

- `xi0`, `xi1` – points on the Berkovich line X

OUTPUT:

the open discoid D with boundary point xi_0 which contains ξ_1. Note that $\xi 0$ is *not* contained in D.

It is assumed that $\xi_0 < \xi_1$. This means that D cannot contain the Gauss point.

EXAMPLES:

```
sage: from mclf import *
sage: F.<x> = FunctionField(QQ)
sage: X = BerkovichLine(F, QQ.valuation(2))
sage: xi0 = X.point_from_discoid(x-1, 1)
sage: xi1 = X.point_from_discoid(x+1, 2)
sage: open_discoid(xi0, xi1)
domain defined by
v(x + 1) > 1
```

mclf.berkovich.piecewise_affine_functions.**valuative_function**(*D*, *f*, *T=None*, *is_factored=False*)

A valuative function on a domain in the Berkovich line.

INPUT:

- `D` – a domain in the Berkovich line, or the Berkovich line itself
- **`f` – a nonconstant rational function on X, or** a pair (L, a_0), where L is a list of pairs (g, a) consisting of
  - a polynomial g (element of the function field on X)
  - a nonzero rational number a

  and where a_0 is a rational number
- `T` (default: `None`) – a Berkovich tree
- `is_factored` – a boolean (default: `False`)

OUTPUT:

If f is a rational function, then we create the valuative function h on D defined as

$$h(\xi) := v_K(f(\xi))$$

If `f` is a pair (L, a_0) as above, where L is a list of pairs (f_i, a_i), then the corresponding valuative function is defined as

$$h(\xi) := a_0 + \sum_i a_i v(f(\xi)).$$

The domain D must be either a standard closed discoid, or the the full Berkovich line.

If the Berkovich tree T is given, then it is assumed that T is the *dendrite* of the function h on D. This means that the root of T is the minimal point of D, and that the restriction of h to the edges of T are affine.

If `is_factored` is `True` then we assume that L is actually a list of triples $(f, a, [\xi])$, where the f are *irreducible* polynomials, and $[\xi]$ is the list of all points of type I which are zeroes of f.

CHAPTER 3

p-adic extensions

3.1 Fake p-adic completions

p-adic fields as completions of absolute number fields

This module realizes a class `FakepAdicCompletion` which represents a p-adic number field K. Internally, the p-adic number field is represented by a number field K_0 together with a discrete valuation v_K on K_0 which extends the p-adic valuation v_p on $\mathbb{Q}$, such that K is the completion of K_0 at v_K.

Our main goal is to be able to compute *weak p-adic Galois extensions* of p-adic number fields of large degree. We want to:

- compute efficiently with general extensions of $\mathbb{Q}_p$ of high ramification index (up to several hundreds)
- obtain provably correct results.

Both objectives seem difficult to reach with the existing functionality of `Sage`. Therefore, we decided to represent a p-adic number field K by a pair (K_0, v_K), where K_0 is an *absolute* number field and v_K is a discrete valuation on K_0 extending the'p'-adic valuation v_p on $\mathbb{Q}$, for some prime p. A more systematic realization of this idea, Julian Rueth's Sage package `completion`, should soon be part of Sage.

The first advantage of our approach (which adresses the first point above) is that arithmetic in absolute number fields is reasonably fast in `Sage`. Moreover, since there are many pairs (K_0, v_K) which have the same completion, we can choose K_0 in such a way that certain operations we use heavily (like evaluation of v_K) can be done very efficiently. At the moment, we choose (K_0, v_K) such that

- v_K is the unique extension of v_p to K_0,
- $K_0/\mathbb{Q}$ is generated by a uniformizer π_K of v_K.

For instance, these assumptions allow us to easily write down a p-integral basis for $K_0/\mathbb{Q}$ which is also an integral basis for $K/\mathbb{Q}_p$. We can also choose K_0 such that the defining polynomial has small coefficients.

The second advantage is that, since for most of our needs we do not really need the p-adic number field K explicitly, we can instead work with the *henselization* K^h of (K_0, v_K) which is an *exact* field: an element of K^h is uniquely determined by a finite amount of data, namely its minimal polynomial over $\mathbb{Q}$ and a sufficiently precise approximation

by an element of K_0. Moreover, K^h only depends on the p-adic number field K but not on our particular choice of K_0.

The main drawback of our approach is that morphisms between two p-adic number fields K and L are somewhat difficult to realize: the problem is that for our particular choice of the underlying number fields K_0 and L_0, there may not exist a nonzero morphism $K_0 \to L_0$ even if K is a subfield of L.

AUTHORS:

- Stefan Wewers (2017-08-24): initial version

EXAMPLES:

TO DO:

- one should try to always find a totally ramified extension (but it is not so clear how to do that)

class mclf.padic_extensions.fake_padic_completions.**FakepAdicCompletion**(*K0*, *vK*)

Bases: SageObject

Return the completion of a number field at a p-adic valuation.

INPUT:

- K0 – an absolute number field,
- vK – a p-adic valuation on K0.

We assume that v_K is the unique extension of the p-adic extension v_p on $\mathbb{Q}$ to K_0, and that the standard generator π_K of $K_0/\mathbb{Q}$ is a uniformizer for v_K. If this is not the case, an error is raised.

absolute_degree()
Return the degree of this p-adic field as an extension of $\mathbb{Q}_p$.

absolute_inertia_degree()
Return the absolute inertia degree of this p-adic extension.

absolute_ramification_degree()
Return the absolute ramification degree of this p-adic extension.

approximate_factorization(*f*, *only_ramified_factors=False*)
Return a list of approximate irreducible factors of f.

INPUT:

- f – a squarefree univariate polynomial over K_0, the number field underlying this p-adic field K
- only_ramified_factors – boolean (default: False)

OUTPUT: a list of pairs (g, e), where g is an approximate irreducible factor of f over K and $e >= 1$. The pairs (g, e) are in bijection with the irreducible factors of f over K. If (g, e) corresponds to the irreducible factor $f_i' then$ and f_i generate the same extension of K. In particular, f_i and g have the same degree. Moreover e is the ramification degree of the extension of K generated by g (or f_i).

If only_ramified_factors is True then all pairs (g, e) with $e = 1$ are omitted.

At the moment we have to assume that f is monic and integral with respect to v_K (and hence all roots have nonnegative valuation).

approximate_irreducible_factor(*f*)
Return one approximate irreducible factor of a given polynomial.

INPUT:

- f – a squarefree and integral polynomial over the number field K_0 underlying this p-adic number field K.

Output:

Either a monic poynomial g over K_0 which is an approximate irreducible factor of f, or a positive integer e.

The polynomial g is then irreducible over K and the extension of K generated by adjoining a root of g is contained in the splitting field of f.

If an integer e is retuned, then any tame extension of K of ramification degree e is a weak splitting field of f over K.

TODO:

One should also allow a list of polynomials as input.

base_change_matrix(*integral_basis='standard'*, *precision=20*)
Return the base change matrix to an integral basis.

INPUT:

- `integral_basis` – a string (default: "standard")
- `precision` – a positive integer

OUTPUT:

An invertible (n, n)-matrix S over $\mathbb{Q}$ with the following property: for an element a of K_0, the vector `S*a.vector()` gives the representation of a as a linear combination of `integral_basis`.

TODO:

- clarify the role of `precision` in this function

base_valuation()
Return the p-adic valuation.

characteristic_polynomial(*f*, *g*)
Return the characteristic polynomial of an element of a simple extension of K.

INPUT:

- `f`, `g` – univariate monic polynomials over K

OUTPUT:

the characteristic polynomial of the element $\beta = g(\alpha)$ with respect to the relative extension $L = K[\alpha]/K$, where α is a formal root of f. Of course, L is a field only if f i irreducible, but this is not checked, and no error would occur within this function if f is reducible.

characteristic_polynomial_mod(*f*, *N*)
Return the absolute characteristic polynomial of the root of a given polynomial, modulo p^N.

INPUT:

- `f` – a monic p-integral and irreducible polynomial over the underlying number field K
- `N` – a positive integer

OUTPUT:

the absolute characteristic polynomial of a root of f, modulo p^N.

degree()
Return the degree of this p-adic field as an extension of $\mathbb{Q}_p$.

element_from_vector(*v*, *integral_basis='standard'*)
Return the element corresponding to a given vector.

INPUT:

- `v` – a vector of rational numbers with length equal to the degree of K
- `integral_basis` – a string (default: "standard")

OUTPUT:

the linear combination of the canonical integral basis of K corresponding to v.

extension(*f*, *embedding=False*)
Return a p-adic extension such that `f` has a root in an unramified extension of it.

INPUT:

- `f` – a monic univariate polynomial over the number field K_0 underlying this p-adic extension K, which is irreducible over K.
- `embedding` – a boolean (default: `False`)

OUTPUT:

A p-adic extension L of K such that `f` has a root in an unramified extension of L, or (if `embedding` is `True`) the pair (L, ϕ), where ϕ is the canonical embedding of K into L.

If K_0 is the underlying number field, then $f \in K_0[x]$ is irreducible over the completion K; the resulting finite extension L/K is a subextension of the extension of K

$$K[x]/(f).$$

However, the number field L_0 underlying L is in general not equal to $K_0[x]/(f)$, and there may not exist any embedding of K_0 into L_0.

generator()
Return the standard generator of this p-adic extension.

inertia_degree()
Return the absolute inertia degree of this p-adic extension.

integral_basis_of_unramified_subfield(*precision=5*)
Return an (approximate) integral basis of the maximal unramified subfield.

INPUT:

- `precison` – a positive integer

OUTPUT:

A list $\alpha = (\alpha_0, \ldots, \alpha_{m-1})$ of elements of K_0 which approximate (with precision p^N an integral basis of the maximal unramified subfield.

is_Qp()
Return `True` if this is the padic-completion of the field of rational numbers.

is_approximate_irreducible_factor(*g*, *f*, *v=None*)
Check whether `g` is an approximate irreducible factor of `f`.

INPUT:

- `g`: univariate polynomial over the underlying number field K_0
- `f`: univariate polynomial over K_0

- v: a MacLane valuation on $K_0[x]$ approximating g, or None; here *approximating* means that LimitValuation(v, g) is well-defined.

Output: True if g is an approximate irreducible factor of f, i.e. if g is irreducible over K and Krasner's condition is satified, If true, the stem field of g over K is a subfield of the splitting field of f over K.

Her we say that *Krasner's Condition* holds if for some root α of g there exists a root β of f such that α is p-adically closer to β than to any other root of g.

Note that if $\deg(g) = 1$ then the condition is nontrivial, even though the conclusion from Krasner's Lemma is trivial.

matrix(*a*, *integral_basis='standard'*)

Return the matrix representing the element a.

INPUT:

- a – an element of the underlying number field of this p-adic extension
- integral_basis – a string (default: "standard")

OUTPUT:

The matrix representing the element a of the underlying number field K, with respect to the canonical p-integral basis.

minpoly_over_unramified_subextension(*N*)

Return the minimal polynomial of the standard uniformizer of this p-adic number field K, relative to the maximal unramified subfield, as a polynomial over K itself.

INPUT:

- N – a positive integer

OUTPUT:

A polynomial P over the number field K_0 underlying K. P is an approximation (with precision N) of the minimal polynomial of the standard uniformizer π of K, relative to the maximal unramified subfield K_{nr} of K. Moreover, $P(\pi) = 0$ holds exactly.

If the approximation is sufficient (note: this still has to be made precise) then P has a root π_1 in K which is also a uniformizer of K, and $K = K_{nr}[\pi_1]$.

NOTE:

To check that P is ok, it should suffice to see that P is "Eisenstein over K_nr" and has a root in K. Unfortunately, the coefficient will likely not lie in K_{nr} exactly, and so this criterion probably makes no sense.

normalized_valuation()

Return the normalized valuation on this p-adic field.

Here *normalized* means that the valuation takes the value 1 on a uniformizer.

number_field()

Return the number field representing this p-adic extension.

p()

Return the prime p.

polynomial()

Return the minimal polynomial of the standard generator of K.

ramification_degree()

Return the absolute ramification degree of this p-adic extension.

ramified_extension(*n*, *embedding=False*)
Return a purely ramified extension with given ramification degree.

INPUT:

- `n` – a positive integer
- `embedding` – a boolean (default: `False`)

OUTPUT:

A finite extension L of this p-adic field K such that L/K is purely ramified, of degree `n`. If `embedding` is `True` then the pair (L, ϕ) is returned, where $\phi : K \to L$ is the canonical embedding.

reduce(*a*, *N*)
Return an approximation of `a` which is reduced modulo p^N.

INPUT:

- `a` – an element of the underlying number field K
- `N` – a positive Integer

OUTPUT: an element $\tilde{a}$ of K which is congruent to a modulo p^N, and whose representation in terms of the canonical integral basis of K has coefficents of the form c/p^m, with $0 \leq c < p^N$ and $m \geq 0$.

reduce_rational_number(*a*, *N*)
Return an approximation of `a` which is reduced modulo p^N.

INPUT:

- `a` – a rational number
- `N` – a positive Integer

OUTPUT: an element $\tilde{a}$ of $\mathbb{Q}$ which is congruent to a modulo p^N, of the form c/p^m, with $0 \leq c < p^N$ and $m \geq 0$.

simplify_irreducible_polynomial(*f*)
Return a simpler polynomial generating the same extension.

INPUT:

- `f` – an univariate polynomial over the underlying number field K which is integral and irreducible over $\hat{K}$

OUTPUT:

A polynomial g over K which is irreducible over $\hat{K}$, and which generates the same extension of $\hat{K}$ as f.

subfield(*alpha*, *e*)
Return a subfield approximated by a given element.

INPUT:

- `alpha` – an element of the number field K_0 underlying K
- `e` – a divisor of the absolute degree of K_0

OUTPUT:

A p-adic number field L with ramification index e which has an embedding into K, or `None` if no such field can be found.

If L is a subfield of K with ramification index e and α_i is a sequence of element of K_0 converging to a generator of L, then calling `K.subfield(alpha_i,e)` will find the subfield L for i sufficiently large.

TODO:

One easy improvement could be to not try to embed L into K after every MacLane step, but only for the last step before the degree of v jumps.

uniformizer()
Return the standard unifomizer of this p-adic extension.

valuation()
Return the valuation on the underlying number field of this p-adic extension.

vector(*a*, *integral_basis='standard'*)
Return the vector corresponding to an element of the underlying number field.

INPUT:

- `a` – an element of the number field K_0 underlying this p-adic number field
- `integral_basis` – a string (default: "standard")

OUTPUT:

the vector of coefficients corresponding to the representation of a as a linear combination of an integral basis of K.

If `integral_basis` is "standard" then we use the integral basis

$$p^{\lceil i/e \rceil}\pi^i, i = 0, \ldots, n,$$

where π is the standard uniformizer of L, e is the absolute ramification degree and n the absolute degree of K.

If it is `mixed` then we use the integral basis

$$\pi^i\alpha_j, i = 0, \ldots, e-1, j = 0, \ldots, n/e-1,$$

where α_j is an approximation of an integral basis of the maximal unramified subfield.

weak_splitting_field(*F*)
Return the weak splitting field of a list of polynomials.

INPUT:

- `F` – a polynomial over the underlying number field K_0, or a list of polynomials

OUTPUT:

A weak splitting field L/K of `F`.

This means that F splits over an unramified extension of L.

Note:

This function works at the moment only for the base field $\mathbb{Q}_p$.

TODO:

The following trick should give a massive improvement for large examples: Instead of calling `K.approximate_factorization(F)` one should calling a new version of `K.approximate_irreducible_factor(F, *options*)`.

The function `approximate_irreducible_factor(..)` would do a MacLane approximation with `require_maximal_degree=True`. From the resulting v's one can then select (according to the options) one which, after enough MacLane steps, gives an approximate irreducible factor.

The point is that the test for being an approximate factor seems to be the most time consuming for large examples. A further speedup could be obtained by

- either finding a more clever way to test,
- or by simply doing several MacLane steps before testing.

The options should be set such that factors are chosen accoring to the following rules:

1. one prefers purely ramified factors over factors with inertia
2. one prefers purely wild factors over mixed factors
3. one prefers mixed factors over tame factors

Note that for tame factors the inertia may be ignored, hence they may be considered as purely ramified. However, it is better to leave them until the end, because doing a tame extension is almost trivial. It is not clear to me whether a purely wild factor with inertia is better than a mixed unramified factor.

EXAMPLES:

The following example created an error in a previous version

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: Q2 = FakepAdicCompletion(QQ, v_2)
sage: R.<x> = QQ[]
sage: f = x^12 + 192*x^9 - 5352*x^6 + 33344*x^3 - 293568
sage: L = Q2.weak_splitting_field(f)
sage: L.ramification_degree()
4
```

Check that non-integral polynomials are allowed as well

```
sage: Q2.weak_splitting_field(2*x^2 + 1)
2-adic completion of Number Field in pi2 with defining polynomial x^2 + 2
```

3.2 Fake p-adic embeddings

Let K and L be p-adic number fields. In this module we define a class `FakepAdicEmbedding` whose objects represent embeddings $\phi : K \to L$ over $\mathbb{Q}_p$.

Here the p-adic number fields K and L are objects of the class `FakepAdicCompletion`. This means that K and L represented as pairs (K_0, v_K) and (L_0, v_L), where e.g. K_0 is a number field and v_K a p-adic valuation on K_0 such that K is the completion of K_0 at v_K. In fact, we do not work with actual p-adic numbers.

Given an embedding $\phi : K \to L$, there need not exist any embedding $K_0 \to L_0$ of the underlying number fields. Therefore, the embedding ϕ has to be construced in a rather indirect way. Recall that K_0 and L_0 are absolute number fields generated by prime elements π_K and π_L over $\mathbb{Q}$ (with respect to v_K and v_L). So an embedding $\phi : K \to L$ is uniquely determined by a root of the absolute minimal polynomial P_K of π_K over $\mathbb{Q}$ in L. Such a root may be represented by a limit pseudo valuation v on the polynomial ring $L_0[x]$ with $v(P_K) = \infty$.

AUTHORS:

- Stefan Wewers (2017-08-30): initial version

EXAMPLES:

TO DO:

class mclf.padic_extensions.fake_padic_embeddings.**FakepAdicEmbedding**(*K*, *L*, *approximation=None*)

Bases: SageObject

Return an embedding of two p-adic number fields.

INPUT:

- K, L – two p-adic number fields, given as objects of FakepAdicCompletion
- approximation - an approximation of the desired emdding, or None (default: None)

OUTPUT: an embedding of K into L which is approximated by approximation, or None if no such embedding exists.

WARNING: to return None doesn't make sense, because __init__ returns an instance of FakepAdicEmbedding.

Internally, the embedding ϕ is represented by a limit pseudo valuation v on $L_0[x]$ such that $v(P_K) = \infty$. Here K_0 and L_0 are the algebraic number fields underlying K and L and P_K is the minimal valuation of the canonical generator of K_0 over $\mathbb{Q}$.

An *approximation* of ϕ is any discrete valuation v_0 on $L_0[x]$ which approximates v. This may actually be v itself.

Note that the resulting embedding may not be unique, in which case an arbitrary embedding is chosen.

eval(*alpha*, *precision=2*)

Evaluate this embedding on an element of this domain, or on a polynomial.

INPUT:

- alpha – an element of the domain of this embedding, or a polynomial over the underlying number field of the domain
- precision – a positive integer, or None (default: None)

OUTPUT:

the image of alpha under this embedding $\phi : K \to L$, with the guaranteed precision precision.

The element α of K may be given as an element of the number field K_0 underlying K. In this case the image $\phi(\alpha)$ will be given as an element of the number field L_0 underlying L, which is an approximation of the true value of $\phi(\alpha)$ modulo p^N, where N is the guaranteed precision. If precision is given then N is larger or equal to precision. Otherwise the internal precision of ϕ is used (which only guarantees that ϕ is uniquely determined).

The element α in K may also be given by a ...

improve_approximation(*N=None*)

Improve the underlying approximation of this embedding.

INPUT:

- N – a positive integer, or None (default: None)

The effect of this method is that the underlying approximation of the limit valuation representing this embedding is improved. If N is given then this improvement will guarantee that for any integral element α of the number field K_0 underlying the domain K of this embedding, the value of self.eval(alpha) will agree with the true value $\phi(\alpha)$ modulo p^N.

3.3 Fake p-adic extensions

Let K be a p-adic number field. For our project we need to be able to compute with Galois extensions L/K of large degree.

At the moment, computations with general extensions of p-adic fields of large degree are still problematic. In particular, it seems difficult to obtain results which are provably correct. For this reason we do not work which p-adic numbers at all. Instead, we use our own class `FakepAdicCompletion`, in which a p-adic number field is approximated by a pair (K_0, v_K), where K_0 is a suitable number field and v_K is a p-adic valuation on K_0 such that K is the completion of K_0 at v_K.

In this module we define a class `FakepAdicExtension`, which realizes a finite extension L/K of p-adic number fields. Both fields K and L are realized as objects in the class `FakepAdicCompletion`, the embedding $K \to L$ as an object of `FakepAdicEmbedding`.

AUTHORS:

- Stefan Wewers (2017-08-30): initial version

EXAMPLES:

TO DO:

- the method `polynomial` should give an *exact* result, namely a object of some class `FakepAdicPolynomial`

class mclf.padic_extensions.fake_padic_extensions.**FakepAdicExtension**(*phi*)
Bases: `SageObject`

Return the extension of p-adic number fields corresponding to an embedding.

INPUT:

- `phi` – an embedding of p-adic number fields

OUTPUT: the extension L/K where K is the domain and L the target of phi

base_field()
Return the base field.

degree()
Return the degree of the extension.

extension_field()
Return the extension field.

inertia_degree()
Return the inertia degree of the extension.

normalized_valuation()
Return the normalized valuation.

p()
Return the prime p.

polynomial()
Return the minimal polynomial of a generator of this extension.

OUTPUT:

A monic, integral and irreducible polynomial P over the number field underlying the base field of this extension.

Let L/K be our extension of p-adic number fields. Then P is a monic polynomial over the number field K_0 underlying K, of degree $[L : K]$, with the following properties:

- P is integral and irreducible over K
- P is the minimal polynomial of a prime element π of L such that $L = K[\pi]$.

Note, however, that π is in general not equal to the canonical absolute generator π_L of $L/\mathbb{Q}_p$. Moreover, in general no root of P is contained in the number field L_0 underlying L.

TODO:

P should be naturally equipped with

ramification_degree()
Return the ramification degree of the extension.

subextension(*alpha*, *d*)
Return a subextension of given degree, containing (approximately) a given element.

INPUT:

- `alpha` – an (approximate) element of this extension field
- `d` – a positive integer

OUTPUT:

a subextension of degree d which (approximately) contains `alpha`, or `None` if no such subextension exists.

Let L/K be the given extension of p-adic number fields. Then we are looking for a subextension $K \subset M \subset L$ such that $[M : K] = d$. If α is given exactly, then we demand that $\alpha \in M$. If α is given by an approximation (α_0, N) then we demand that there exists an element $\alpha_1 \in M$ such that

$$v_M(\alpha_1 - \alpha_0) \geq N`.$$

Here the valuation v_M is normalized such that $v_M(p) = 1$.

valuation()
Return the valuation of the extension.

3.4 Weak p-adic Galois extensions

Let K be a p-adic number field. For our project we need to be able to compute with Galois extensions L/K of large ramification degree. For instance, we need to be able to compute the breaks of the ramification filtration of the Galois group of L/K, as well as the corresponding subfields.

At the moment, computations with large Galois extensions of p-adic fields are still problematic. In particular, it seems difficult to obtain results which are provably correct. For this reason we do not work which p-adic numbers at all. Instead, we use our own class `FakepAdicCompletion`, in which a p-adic number field is approximated by a pair (K_0, v_K), where K_0 is a suitable number field and v_K is a p-adic valuation on K_0 such that K is the completion of K_0 at v_K.

Let L/K be a finite field extension. We say that L/K is a **weak Galois extension** if the induced extension L^{nr}/K^{nr} is Galois. Given a polynomial f in $K[x]$, we say that L/K is a **weak splitting field** for f if f splits over L^{nr}.

Given a weak Galois extension L/K, we have canonical isomorphisms between the following groups:

- the Galois group of L^{nr}/K^{nr},
- the inertia subgroup of the Galois closure of L/K,

Moreover, this isomorphism respects the filtrations by higher ramification groups.

If L/K is totally ramified then the degree of L/K is equal to the degree of L^{nr}/K^{nr}, which is equal to the order of the inertia subgroup of the Galois closure of L/K. Therefore, our approach allows us to fully understand the inertia group of a Galois extension of p-adic fields, while keeping the degree of the field extensions with which one works as small as possible.

Our method can also be used to work with approximations of the subfields of a p-adic Galois extension corresponding to the higher ramification subgroups.

For $u \geq 0$ we let $L^{sh,u}$ denote the subfield of L^{sh}/K^{sh} corresponding to the u th filtration step of the Galois group of L^{sh}/K^{sh}. Then the completion of $L^{sh,u}$ agrees with the maximal unramified extension of the subextension $\hat{L}^u$ of the Galois closure $\hat{L}/\hat{K}$ corresponding to the u th ramification step. Moreover, there exists a finite extensions L^u/K, together with an extension v_{L^u} of v_K to L^u such that

- the strict henselization of (L^u, v_{L^u}) is isomorphic to $L^{sh,u}$,
- the completion of (L^u, v_{L^u}) agrees with $\hat{L}^u$, up to a finite unramified extension.

Note that L^u will in general not be a subfield of L (and not even of the Galois closure of L/K).

In this module we define a class `WeakPadicGaloisExtension`, which realizes an approximation of a p-adic Galois extension, up to unramified extensions.

AUTHORS:

- Stefan Wewers (2017-08-06): initial version

EXAMPLES:

This example is from the "Database of Local Fields":

```
sage: from mclf import *
sage: v_3 = QQ.valuation(3)
sage: Q_3 = FakepAdicCompletion(QQ, v_3)
sage: R.<x> = QQ[]
sage: f = x^6+6*x^4+6*x^3+18
sage: L = WeakPadicGaloisExtension(Q_3, f)
sage: L.upper_jumps()
[0, 1/2]
```

TO DO:

class mclf.padic_extensions.weak_padic_galois_extensions.**WeakPadicGaloisExtension**(*K*, *F*, *minimal_ramifica*

Bases: *mclf.padic_extensions.fake_padic_extensions.FakepAdicExtension*

Return the weak p-adic splitting field of a polynomial.

INPUT:

- `K` – a p-adic number field
- `F` – a polynomial over the number field underlying K, or a list of such polynomials
- `minimal_ramification` – a positive integer (default: `1`)

OUTPUT: the extension L/K, where L is a weak splitting field of `F` whose ramification index over K is a multiple of `minimal_ramification`.

NOTE:

For the time being, `F` has to be defined oover $\mathbb{Q}$, and `minimal ramification` has to be prime to p.

factors_of_ramification_polynomial (*precision=10*)
Return the factorization of the ramification polynomial into factors with fixed slope.

OUTPUT: a dictionary `factors` such that $g =$ `factors[s]` is the maximal factor of the ramification polynomial G whose Newton polygon has a single slope s. We omit the factor with slope $s = -1$.

lower_jumps ()
Return the upper jumps of the ramification filtration of this extension.

ramification_filtration (*upper_numbering=False*)
Return the list of ramification jumps.

INPUT:

- `upper_numbering` – a boolean (default: `False`)

OUTPUT: an ordered list of pairs (u, m_u), where u is a jump in the filtration of higher ramification groups and m_u is the order of the corresponding subgroup. The ordering is by increasing jumps.

If `upper_numbering` is `False`, then the filtration is defined as follows. Let L/K be a Galois extension of p-adic number fields, with Galois group G. Let π be a prime element of L, and let v_L denote the normalized valuation on L (such that $v_L(\pi) = 1$). For $u \geq 0$, the ramification subgroup G_u consists of all element σ of the inertia subgroup I of G such that

$$v_L(\sigma(\pi) - \pi) \geq i + 1.$$

In particular, $I = G_0$. An integer $u \geq 0$ is called a "jump" if G_{u+1} is strictly contained in G_u. Note that this is equivalent to the condition that there exists an element $\sigma \in G$ such that

$$v_L(\sigma(\pi) - \pi) = u + 1.$$

It follows that the ramification filtration can easily be read off from the Newton polygon (with respect to v_L) of the polynomial

$$G := P(x + \pi)/x,$$

where P is a minimal polynomial of π over K. The polynomial G is called the *ramification polynomial* of the Galois extension L/K.

ramification_polygon ()
Return the ramification polygon of this extension.

The *ramification polygon* of a weak Galois extension L/K of p-adic number fields is the Newton polygon of the ramification polynomial, i.e. the polynomial

$$G := P(x + \pi)/x$$

where π is a prime element of L which generates the extension L/K and P is the minimal polynomial of π over K^{nr}, the maximal unramified subextension of .

The (strictly negative) slopes of the ramification polygon (with respect to the valuation v_L on L, normalized such that $v_L(\pi_L) = 1$) correspond to the jumps in the filtration of higher ramification groups, and the abscissae of the vertices of the corresponding vertices are equal to the order of the ramification subgroups that occur in the filtration.

NOTE:

- For the time being, we have to assume that $K = \mathbb{Q}_p$. In this case we can choose for π the canonical generator of the absolute number field L_0 underlying L.

ramification_polynomial (*precision=20*)
Return the ramification polynomial of this weak Galois extension.

The *ramification polynomial* is the polynomial

$$G := P(x+\pi)/x$$

where π is a prime element of L which generates the extension L/K and P is the minimal polynomial of π over K^{nr}, the maximal unramified subextension of .

NOTE:

- For the time being, we have to assume that $K = \mathbb{Q}_p$. In this case we can choose for π the canonical generator of the absolute number field L_0 underlying L.

ramification_subfield (*u*)
Return the ramification subfield corresponding to a given lower jump.

Here a nonnegative integer $u \geq 0$ is called a *lower jump* for the weak p-adic Galois extension L/K if u is a jump in the filtration $(G_u)_{u\geq 0}$ of the Galois group $G = Gal(L^{nr}/K^{nr})$ of the induced extension L^{nr}/K^{nr}. This is equivalent to the following condition: there exists an element $g \in G$, such that

$$v_L(g(\pi_L) - \pi_L) = u + 1.$$

Here v_L is the valuation of the extension field L and π_L is a prime element of L. We normalize v_L such that $v_L(\pi_L) = 1$.

ramification_subfields (*precision=1*)
Return the ramification subfields of this weak Galois extension.

The set of all subfields is returned as dictionary, in which the keys are the lower jumps and the values are the corresponding subfields, given as extension of the base field.

upper_jumps ()
Return the lower jumps of the ramification filtration of this extension.

CHAPTER 4

Semistable reduction of curves

4.1 Semistable reduction of a smooth projective curve over a local field

Let K be a field and v_K a discrete valuation on K. We let $\mathcal{O}_K$ denote the valuation ring of v_K and $\mathbb{F}_K$ the residue field.

We consider a smooth projective curve Y over K. Our goal is to compute the *semistable reduction* of Y at v_K and to extract nontrivial arithmetic information on Y from this.

Let us define what we mean by 'semistable reduction' and by 'computing'. By the famous result of Deligne and Mumford there exists a finite, separable field extension L/K, an extension v_L of v_K to L (whose valuation ring we call $\mathcal{O}_L$) and an $\mathcal{O}_L$-model $\mathcal{Y}$ of Y_L whose special fiber $\bar{Y} := \mathcal{Y}_s$ is reduced and has at most ordinary double points as singularities. We call $\mathcal{Y}$ a semistabel model and $\bar{Y}$ a semistable reduction of Y.

Let us assume, for simplicity, that K is complete with respect to v_K. Then the extension v_L to L is unique and L is complete with respect to v_L. Then we we may moreover assume that L/K is a Galois extension and that the tautological action of the Galois group of L/K extends to the semistable model $\mathcal{Y}$. By restriction we obtain an action of $\mathrm{Gal}(L/K)$ on $\bar{Y}$. (In practise we mostly work with fields K which are not complete. To make sense of the above definitions, one simply has to replace K by its completion.)

When we say *the semistable reduction* of Y we actually mean the extension L/K, the $\mathbb{F}_L$-curve $\bar{Y}$ and the action of the former on the latter.

Note that neither L/K nor $\bar{Y}$ are unique, but their nonuniqueness is in a sense inessential. For instance, one may replace L by a larger Galois extension L'/K; as a consequence the curve $\bar{Y}$ gets replaced by its base extension to the residue field of L'. Also, certain blowups of the semistable model $\mathcal{Y}$ may result in a new semistable model $\mathcal{Y}$ with special fiber $\bar{Y}'$. The only difference between $\bar{Y}$ and $\bar{Y}'$ are some new irreducible components, which are 'contractible', i.e. they are smooth of genus 0 and meet the rest of $\bar{Y}'$ in at most two points.

At the moment, we do not have an effective method at our disposal to compute the semistable reduction of an arbitrary curve Y, but only a set of methods which can be applied for certain classes of curves. We always assume that the curve Y is given as a finite separable cover

$$\phi : Y \to X,$$

where $X = \mathbb{P}^1_K$ is the projective line over K. There are two main cases that we can handle:

- the order of the monodromy group of ϕ (i.e. the Galois group of its Galois closure) is prime to the residue characteristic of the valuation v_K.
- ϕ is a Kummer cover of degree p, where p is the (positive) residue characteristic of v_K

In the first case, the method of *admissible reduction* is available. In the second case, the results of [We17] tell us what to do. In both cases, there exists a normal $\mathcal{O}_K$-model $\mathcal{X}_0$ of $X = \mathbb{P}^1_K$ (the *inertial model*) whose normalization in the function field of Y_L is a semistable model, for a sufficiently large finite extension L/K. Once the right inertial model is defined, the method for computing the semistable model $\mathcal{Y}$ and its special fiber $\bar{Y}$ are independent of the particular case (these computations are done within the Sage class `ReductionTree`).

In this module we define a base class `SemistableModel`. An object in this class is initialized by a pair (Y, v_K) , where Y is a smooth projective curve over a field K and v_K is a discrete valuation on K. The class provides access to functions which compute and extract information from the semistable reduction of Y with respect to v_K.

Note: For the time being, we have to assume that K is a number field. Then v_K is the valuation associated to a prime ideal of K (i.e. a maximal ideal of its ring of integers).

AUTHORS:

- Stefan Wewers (2018-5-16): initial version

EXAMPLES:

We compute the stable reduction and the conductor exponent of the Picard curve

$$Y : \ y^3 = x^4 - 1.$$

at the primes $p = 2, 3$:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: R.<x> = QQ[]
sage: Y = SuperellipticCurve(x^4-1, 3)
sage: Y
superelliptic curve y^3 = x^4 - 1 over Rational Field
sage: Y2 = SemistableModel(Y, v_2)
sage: Y2.is_semistable()
True
```

The stable reduction of Y at $p = 2$ has four components, one of genus 0 and three of genus 1.

```
sage: [Z.genus() for Z in Y2.components()]
[0, 1, 1, 1]
sage: Y2.components_of_positive_genus()
[the smooth projective curve with Function field in u2 defined by u2^3 + x^4 + x^2,
 the smooth projective curve with Function field in u2 defined by u2^3 + x^2 + x,
 the smooth projective curve with Function field in u2 defined by u2^3 + x^2 + x + 1]
sage: Y2.conductor_exponent()
6
sage: v_3 = QQ.valuation(3)
sage: Y3 = SemistableModel(Y, v_3)
sage: Y3.is_semistable()
True
sage: Y3.components_of_positive_genus()
[the smooth projective curve with Function field in u2 defined by u2^3 + u2 + 2*x^4]
```

(continues on next page)

(continued from previous page)

```
sage: Y3.conductor_exponent()
6
```

class mclf.semistable_reduction.semistable_models.**SemistableModel**(*Y*, *vK*, *check=True*)

Bases: SageObject

This is a base class for various classes of curves and methods for computing the semistable reduction. Objects of this class are determined by a smooth projective curve Y over a field K and a discrete valuation v_K on K.

INPUT:

- Y – a smooth projective curve
- vK – a discrete valuation on the base field K of Y
- check – a boolean (default: True)

Instantiation of this class actually creates an instant of a suitable subclass, which represents the kind of curve for which an algorithm for computing the semistable reduction has been implemented. At the moment, there are two such subclasses:

- If the degree of Y as a cover of the projective line is prime to the residue characteristic of v_K then we invoke the subclass AdmissibleModel. Note that this may not work: we can only guarantee that Y has admissible reduction if the order of the Galois group of the cover $Y \to X = \mathbb{P}^1_K$ is prime to the residue characteristic.
- If Y is a superelliptic curve of degree p, where p is the residue characteristic of v_K and K has characteristic 0 then the subclass SuperpModel is invoked.
- if none of the above holds, then we either raise a NotImplementedError (if check=True) or we create an AdmissibleModel (if check=False)

EXAMPLES:

```
sage: from mclf import *
sage: v_5 = QQ.valuation(5)
sage: FX.<x> = FunctionField(QQ)
sage: R.<y> = FX[]
sage: FY.<y> = FX.extension(y^3 - y^2 + x^4 + x + 1)
sage: Y = SmoothProjectiveCurve(FY)
sage: YY = SemistableModel(Y, v_5)
sage: YY
semistable model of the smooth projective curve with Function field in y defined␣
↪by y^3 - y^2 + x^4 + x + 1, with respect to 5-adic valuation
```

The degree of Y as a cover of the projective line is 4, which is strictly less than $p = 5$. Hence Y has admissible reduction and we have created an instance of the class AdmissibleModel:

```
sage: isinstance(YY, AdmissibleModel)
True
```

Actually, Y has good reduction at $p = 5$:

```
sage: YY.is_semistable()
True
sage: YY.components_of_positive_genus()
[the smooth projective curve with Function field in u1 defined by u1^3 + 4*u1^2 +␣
↪x^4 + x + 1]
```

base_valuation()
Return the valuation on the constant base field of the curve.

components()
Return the list of all components of the admissible reduction of the curve.

components_of_positive_genus()
Return the list of all components of of the admissible reduction of the curve which have positive genus.

compute_semistable_reduction(*verbosity=1*)
Compute the semistable reduction of this curve (and report on the ongoing computation).

INPUT:

– `verbosity` - a nonnegative integer (default: 1)

OUTPUT:

Calling this function initiates the creation of a `ReductionTree` which essentially encodes a semistable model of the curve. Depending on the verbosity level, messages will be printed which report on the ongoing computation. If `verbosity` is set to 0, no message will be printed.

This method must be implemented by the subclasses of SemistableModel (which are characterized by a particular method for computing a semistable model). At the moment, these subclasses are - `AdmissibleModel` - `Superell` - `Superp`

conductor_exponent()
Return the conductor exponent at p of this curve.

EXAMPLES

In this example the conductor exponent was computed wrongly in a previous version:

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: f = x^4+2*x^3+2*x^2+x
sage: Y = SuperellipticCurve(f, 3)
sage: Y3 = SemistableModel(Y, QQ.valuation(3))
sage: Y3.conductor_exponent()
11
```

constant_base_field()
Return the constant base field of this curve.

curve()
Return the curve.

is_semistable()
Check whether the model is really (potentially) semistable.

reduction_tree()
Return the reduction tree underlying this semistable model.

semistable_reduction()
Return the special fiber of this semistable model.

Note: not yet implemented

stable_reduction()
Return the special fiber of the stable model of this curve.

The stable model is obtained from this semistable model by contracting all 'unstable' components.

Note: not yet implemented

4.2 Admissible reduction of curves

Let K be a field equipped with a discrete valuation v_K. For the time being, we assume that K is a number field. Then v_K is the $\mathfrak{p}$-adic valuation corresponding to a prime ideal $\mathfrak{p}$ of K.

We consider a smooth projective curve Y over K. Our goal is to compute the semistable reduction of Y at v_K and to extract nontrivial arithmetic information on Y from this.

In this module we realize a class `AdmissibleModel` which computes the semistable reduction of a given curve Y at v_K provided that it has *admissible reduction* at v_K. This is always the case if the residue characteristic of v_K is zero or strictly larger than the degree of Y (as a cover of the projective line).

AUTHORS:

- Stefan Wewers (2018-7-03): initial version

EXAMPLES:

```
<Lots and lots of examples>
```

TO DO:

- more doctests

class mclf.semistable_reduction.admissible_reduction.**AdmissibleModel**(*Y*, *vK*)

Bases: *mclf.semistable_reduction.semistable_models.SemistableModel*

A class representing a curve Y over a field K with a discrete valuation v_K. Assuming that Y has (potentially) admissible reduction at v_K, we are able to compute the semistable reduction of Y at v_K.

INPUT:

- `Y` – a smooth projective curve over a field K
- `vK` – a discrete valuation on K

OUTPUT: the object representing the curve Y and the valuation v_K. This object has various functionalities to compute the semistable reduction of Y relative to v_K, and some arithmetic invariants associated to it (for instance the "exponent of conductor" of Y with respect to v_K).

EXAMPLES:

original_model_of_curve()

Return the original model of the curves.

4.3 Semistable models of superelliptic curves of degree p

Let K be a field of characteritic zero and v_K a discrete valuation on K whose residue field is finite of characteristic $p > 0$.

Let $f \in K[x]$ be a polynomial over K which is not a p-th power and whose radical has degree at least three. We consider the smooth projective curve Y over K defined generically by the equation

$$Y : y^p = f(x).$$

So Y is a *superelliptic curve* of degree p.

In this module we define a class `SuperpModel` which represents a semistable model of a superelliptic curve Y of degree p, with respect to a p-adic valuation on the base field K of Y.

The method to define and compute a semistable model in this particular case is taken from

- [We17] S. Wewers, *Semistable reduction of superelliptic curves of degree p*, preprint, 2017.

The key notion is the **etale locus**.

The superelliptic curve Y is, by definition, a cyclic cover

$$\phi : Y \to X$$

of degree p of the projective line X over the base field K. We consider X and Y as analytic spaces over the completion of K at the base valuation v_K. Let

$$\bar{\phi} : \bar{Y} \to \bar{X}$$

denote the *semistable reduction* of the cover $Y \to X$. The **etale locus** is an affinoid subdomain X^{et} of X consisting of those points which specialize to a point on $\bar{X}$ above which the map $\bar{\phi}$ is etale.

While the affinoid X^{et} is determined by the semistable reduction of the cover ϕ, conversely X^{et} contains a lot of information on the semistable reduction. The main result of [We17] gives an explicit description of the affinoid X^{et} as a union of rational domains defined by rational functions which can be easily computed in terms of the polynomial f defining Y.

Once the etale locus X^{et} is computed, we can define an *inertial model* $\mathcal{X}_0$ of X. A semistable model $\mathcal{Y}$ of Y can then be obtained as the normalization of $\mathcal{X}_0$ inside Y_L, for a sufficiently large finite extension L/K.

The class `SuperpModel` is a subclass of the class `SemistableModel` and can be instantiated via its parent. All methods to extract information about the semistable reduction of Y are simply inherited from `SemistableModel`. The subclass itself only defines the methods to compute the etale locus and to create the corresponding inertail model.

AUTHORS:

- Stefan Wewers (2017-07-29): initial version

EXAMPLES:

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: f = x^4 + x^2 + 1
sage: Y = SuperellipticCurve(f, 3)
sage: Y
superelliptic curve y^3 = x^4 + x^2 + 1 over Rational Field
sage: v_3 = QQ.valuation(3)
sage: YY = SuperpModel(Y, v_3)
sage: YY
semistable model of superelliptic curve Y: y^3 = x^4 + x^2 + 1 over Rational Field,␣
↪with respect to 3-adic valuation
sage: YY.etale_locus()
Affinoid with 3 components:
Elementary affinoid defined by
v(x) >= 3/4
Elementary affinoid defined by
v(x - 2) >= 5/4
Elementary affinoid defined by
v(x + 2) >= 5/4

sage: YY.is_semistable()
True
sage: YY.components()
[the smooth projective curve with Function field in u1 defined by u1^3 + 2*x^4 + 2*x^
↪2 + 2,
 the smooth projective curve with Function field in u2 defined by u2^3 + u2 + 2*x^2,
```

(continues on next page)

(continued from previous page)

```
 the smooth projective curve with Function field in u2 defined by u2^3 + u2 + 2*x^2,
 the smooth projective curve with Function field in u2 defined by u2^3 + u2 + 2*x^2]
sage: YY.conductor_exponent()
12
```

We check that issues #39 and #40 have been fixed:

```
sage: v_2 = QQ.valuation(2)
sage: f =  x^5 - 5*x^4 + 3*x^3 - 3*x^2 + 4*x - 1
sage: Y = SuperellipticCurve(f, 2)
sage: Y2 = SemistableModel(Y, v_2)
sage: Y2.etale_locus()
Affinoid with 2 components:
Elementary affinoid defined by
v(x + 1) >= 2/3
Elementary affinoid defined by
v(x^4 +  4*x + 4) >= 8/3
sage: Y2.is_semistable()
True
```

TO DO:

- more doctests

class mclf.semistable_reduction.superp_models.**SuperpModel**(*Y*, *vK*)

Bases: *mclf.semistable_reduction.semistable_models.SemistableModel*

Return a semistable model of a superelliptic curve of degree p.

INPUT:

- `Y` – a superelliptic curve over a field K
- `vK` – a discrete valuation on K

The field K must be of characteristic 0 and the residue characteristic of `vK` must be a prime p which is equal to the covering degree of Y.

OUTPUT: the object representing a semistable model of Y.

Note: For the time being, we need to make the following additional assumptions on the curve Y:

- the polynomial f which is part of the defining equation $y^p = f(x)$ is of degree prime to p.

This restrictions is preliminary and will be removed in a future version. Note that a superelliptic curve of degree p can be written in the required form if and only if the map $Y \to X$ has a K-rational branch point.

EXAMPLES:

compute_semistable_reduction()

Compute the semistable reduction of this curve, and report on the computation and the result.

etale_locus()

Return the etale locus of the cover $Y \to X$.

OUTPUT: the etal locus, an affinoid subdomain of the Berkovich line X (the analytic space associated to the projektive line over the completion of the base field K with respect to the valuation v_K).

EXAMPLES:

```
sage: from mclf import *
sage: v_2 = QQ.valuation(2)
sage: R.<x> = QQ[]
sage: f = x^3 + x^2 + 1
sage: Y = SuperellipticCurve(f, 2)
sage: Y
superelliptic curve y^2 = x^3 + x^2 + 1 over Rational Field
sage: YY = SuperpModel(Y, v_2)
sage: YY.etale_locus()
Elementary affinoid defined by
v(x^4 + 4/3*x^3 + 4*x + 4/3) >= 8/3
```

We check Example 4.14 from [BouWe16]. The original equation is $y^2 = f(x) = 2x^3 + x^2 + 32$, and f is not monic, as required. To fix this, we substitute $x/2$ and multiply with 4. Then the new equation is $y^2 = x^3 + x^2 + 128$:

```
sage: f = x^3 + x^2 + 128
sage: Y = SuperellipticCurve(f, 2)
sage: YY = SuperpModel(Y, v_2)
sage: YY.etale_locus()
Elementary affinoid defined by
v(x) >= 2
v(1/x) >= -5/2
<BLANKLINE>
```

Note: At the moment, the construction of the superelliptic curve Y requires that the polynomial f defining Y is monic, integral with respect to v_K and of degree prime to p. The motivation for this restriction, and its result is that the etale locus is contained in the closed unit disk.

reduction_tree()
Return the reduction tree which determines the semistabel model.

mclf.semistable_reduction.superp_models.**p_approximation**(*f*, *p*)
Return the p-approximation of `f`.

INPUT:

- **f – a polynomial of degree n over a field K, with nonvanishing** constant coefficient
- p – a prime number

OUTPUT:

Two polynomials h and g in $K[x]$, such that

- $f = a_0(h^p + g)$, where a_0 is the constant coefficient of f
- $r := deg(h) \leq n/p$, and
- $x^{(r+1)}$ divides g

Note that h, g are uniquely determined by these conditions.

mclf.semistable_reduction.superp_models.**p_approximation_generic**(*f*, *p*)
Return the generic p-approximation of `f`.

INPUT:

- f – a polynomial of degree n over a field K, with nonvanishing constant coefficient
- p – a prime number

OUTPUT:

Two polynomials H and G in $K(x)[t]$ which are the p-approximation of the polynomial $F := f(x+t)$, considered as polynomial in t.

4.4 Reduction trees: a data structure for semistable reduction of covers of the projective line.

Let K be a field with a discrete valuation v_K. We let $X := \mathbb{P}^1_K$ denote the projective line over K. We are also given a finite cover

$$\phi : Y \to X,$$

where Y is a smooth projective and absolutely irreducible curve. We assume that Y has positive genus.

Let $\mathcal{X}_0$ be a (normal) v_K-model of X. Then for every finite field extension L/K and every extension v_L of v_K to L, we obtain v_L-models $\mathcal{X}$ of X_L and $\mathcal{Y}$ of Y_L and a finite map $\mathcal{Y} \to \mathcal{X}$ extending ϕ by normalizing $\mathcal{X}_0$. Restricting this map to the special fiber yields a finite map

$$\bar{\phi} : \bar{Y} \to \bar{X}$$

between projective curves over the residue field of v_L. We call this the *reduction* of ϕ *over* (L, v_L) with respect to the *inertial model* $\mathcal{X}_0$.

If we fix ϕ and $\mathcal{X}_0$ then there exists (L, v_L) such that the curves $\bar{Y}$ and $\bar{X}$ are reduced. If this is the case, any further extension of (L, v_L) will not change $\bar{Y}$ and $\bar{X}$ in an essential way (more precisely, it will only replace $\bar{Y}$ and $\bar{X}$ by their base extensions to the larger residue field). Therefore we call the models $\mathcal{Y}$ and $\mathcal{X}$ *permanent*.

We say that $\mathcal{X}_0$ is a *semistable inertial model* of ϕ if the permanent models $\mathcal{Y}$ and $\mathcal{X}$ are semistable, and all irreducible components of the (semistable) curves $\bar{Y}$ and $\bar{X}$ are smooth (i.e. they do not intersect themselves).

The class `ReductionTree` defined in this module is a datastructure which encodes a cover $\phi : Y \to X$ and an inertial model $\mathcal{X}_0$ as above, and provides functionality for computing the reduction $\bar{\phi} : \bar{Y} \to \bar{X}$ with respect to extensions (L, v_L). In particular, it allows us to check whether the given inertial model $\mathcal{X}_0$ is semistable and, if this is the case, to compute the semistable reduction of the curve Y.

The inertial model

The inertial model $\mathcal{X}_0$ of X is determined by a *Berkovich tree* T on the analytic space X^{an} (the *Berkovich line* over $\hat{K}$, the completion of K with respect to v_K). Thus,

- the irreducible components of the special fiber of $\mathcal{X}_0$ (called the *inertial components*) correspond to the vertices of T which are points of type II on X^{an}.
- the vertices of T which are points of type I (i.e. closed points on X) are considered *marked points* on X
- an edge of T connecting two points of type II correspond to the point of intersection of the two corresponding inertial components
- an edge of T connecting a point of type II and a point of type II corresponds to the specialization of a marked point to an inertial component

In particular, the inertial model $\mathcal{X}_0$ is a *marked* model. As a result, the models $\mathcal{X}$ and $\mathcal{Y}$ induced by $\mathcal{X}_0$ and an extension (L, v_L) are marked models, too. The condition that $\mathcal{X}_0$ is a semistable inertial model therefore implies that $\mathcal{X}$ and $\mathcal{Y}$ are *marked* semistable models, for L/K sufficiently large. Recall that this means that the marked points specialize to the smooth points on the special fiber.

Reduction components

Let us fix an inertial component Z_0. The *interior* of Z_0 is the affinoid subdomain of X^{an} consisting of all points which specialize to a point on Z_0 which is neither the point of intersection with another inertial component nor the specialization of a marked point (exception: if there is a unique inertial component and no marking then this is all of X^{an} and not an affinoid). We have to choose a *basepoint* for Z_0, which is a closed point on X lying inside the interior. This choice is made in a heuristic manner; the degree of the base point should be as small as possible. Then a *splitting field* for Z_0 is a finite extension (L, v_L) of (K, v_K) with the property that the base point and all points on Y above it are $\hat{L}$-rational (where $\hat{L}$ denotes the completion of L with respect to v_L).

AUTHORS:

- Stefan Wewers (2017-8-10): initial version

EXAMPLES:

```
sage: from mclf import *
sage: FX.<x> = FunctionField(QQ)
sage: v_2 = QQ.valuation(2)
sage: X = BerkovichLine(FX, v_2)
sage: T = BerkovichTree(X, X.gauss_point())
sage: T, _ = T.add_point(X.infty())
sage: R.<y> = FX[]
sage: FY.<y> = FX.extension(y^2-x^3-1)
sage: Y = SmoothProjectiveCurve(FY)
sage: RT = ReductionTree(Y, v_2, T)
sage: RT
A reduction tree for  the smooth projective curve with Function field in y defined by␣
↪y^2 - x^3 - 1, relative to 2-adic valuation
sage: RT.inertial_components()
[inertial component of reduction tree with interior Elementary affinoid defined by
 v(x) >= 0
 ]
```

TODO:

- better documentation
- more doctests

class mclf.semistable_reduction.reduction_trees.**InertialComponent**(*R*, *xi*, *is_separable=True*)

Bases: SageObject

Return the inertial component corresponding to a type-II-point which is a vertex of T.

INPUT:

- R – a reduction tree
- xi – a point of type II on the Berkovich line X underlying R; it is assumed that ξ is a vertex of the Berkovich tree T underlying R
- is_separable – boolean (default: True)

OUTPUT: The base component corresponding to ξ.

It is assumed that ξ is a vertex of the given Berkovich tree. It thus corresponds to an irreducible component of the special fiber of the inertial model $\mathcal{X}_0$. If this is not the case, an error is raised.

The *inertial component* which is generated by this command is an object for hosting methods which compute information about the irreducible components of models Y of X lying above ξ, over various extensions of the base field.

basepoint()
Return the base point.

The *basepoint* is a type-I-point on the underlying Berkovich line which specializes to the interior of this component of the special fiber of $\mathcal{X}_0$. If no base point is given when the base component was created, then such a point is computed now.

berkovich_line()
Return the underlying Berkovich line X.

component()
Return the smooth projective curve underlying this inertial component.

Note that the constant base field of this curve is, by definition, the residue field of the base valuation. This may differ from the field of constants of its function field.

component_degree(u=∞)
Return the sum of the degrees of the upper components above this inertial component.

Here the *degree* of an upper component ist the degree of its field of constants, as extension of the constant base field.

function_field()
Return the function field of this inertial component (which is the residue field of the valuation corresponding to it).

interior()
Return the interior of this inertial component.

OUTPUT:

an elementary affinoid subdomain of the underlying Berkovich line.

The **interior** of a base component is the elementary affinoid subdomain of the underlying Berkovich line whose canonical reduction is an open affine subset of this inertial component of the inertial model $\mathcal{X}_0$.

It is chosen such that the canonical reduction does not contain the points of intersection with the other components of $\mathcal{X}_{0,s}$ and is disjoint from the residue classes of the marked points.

is_separable()
Return `True` is this inertial component is separable.

lower_components(u=∞)
Return the lower components relative to a given extension of the base field.

INPUT:

- `u` – an integer, or `Infinity` (default: `Infinity`)

OUTPUT: the list of lower components of the model of the reduction tree lying over this base component. If $u = \infty$ then these components are computed over the splitting field of the base component. Otherwise, u is assumed to be a break in the ramification filtration of the splitting field, and then we use the corresponding subfield.

The entries of the list correspond to the irreducible components of the special fiber of the v_L-model $\mathcal{X}$ (the normalization of $\mathcal{X}_0$) lying over the given inertial component. By definition, the constant base field of these components is the residue field of v_L (and it may differ from its field of constants).

outdegree(u=∞)
Return the outdegree of this inertial component.

INPUT:

- `u` – an integer, or Infinity (default: `Infinity`)

OUTPUT: the sum of the degrees of all edges emanating from components of the curve $\bar{Y}^u$ which lie above this inertial component.

Here u is a break in the ramification filtration of splitting field of this inertial component, and the curve $\bar{Y}^u$ is the special fiber of the reduction of Y over the corresponding subfield L^u (with respect to the given inertial model). By *edge* we mean an edge of the component graph of the curve $\bar{Y}^u$; it corresponds to a point in which two components intersect. We call an edge *outgoing* (with respect to this inertail component) if it lies above an edge of the component graph of the special fiber of the inertial model which is directed away from this inertial component. The *degree* of an (upper) edge is the degree of the corresponding point of $\bar{Y}^u$, with respect to the residue field of L^u.

outgoing_edges()

Return the list of outgoing edges from this inertial component.

Here an *edge* is a point on this inertial component where it intersects another component; so it corresponds to an edge on the Berkovich tree underlying the chosen inertial model. *Outgoing* is defined with respect to the natural orientation of the Berkovich tree.

reduce(f)

Return the reduction of a rational function to this component.

INPUT:

- `f` – an element of the function field of the Berkovich line

It is assumed that f a unit of the valuation ring corresponding to this component.

OUTPUT: the image of f in the function field of this component.

EXAMPLES:

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: Y = SuperellipticCurve(x^3-1, 2)
sage: Y3 = SemistableModel(Y, QQ.valuation(3))
sage: Z = Y3.reduction_tree().inertial_components()[1]
sage: f = Z.valuation().element_with_valuation(1/2)^2/3
sage: Z.reduce(f)
x
```

reduction_conductor()

Return the contribution of this inertial component to the conductor exponent.

OUTPUT: an integer f_Z (where Z is this inertial component).

The conductor exponent f_Y of the curve Y can be written in the form

$$f_Y = 1 + \sum_Z f_Z$$

where Z runs over all inertial components of the reduction tree and f_Z is an integer, called the *contribution* of Z to the conductor exponent.

TODO: Write better documentation.

reduction_genus(u=∞)

Return the sum of the genera of the upper components.

INPUT:

- `u` – an integer, or Infinity (default: `Infinity`)

OUTPUT: a nonnegative integer, the sum of the genera of the upper components for this base component, computed with respect to the splitting field. If $u \neq \infty$ then it is assumed that u is a break in the ramification filtration of the splitting field, and then the corresponding subfield is used instead.

reduction_tree()
Return the reduction tree of this component.

splitting_field(*check=False*)
Return a splitting field for this inertial component.

INPUT:

- `check` – a boolean (default: `False`)

OUTPUT: a weak Galois extension (L, v_L) of the base field.

At the moment, the *splitting field* of a inertial component is a weak Galois extension (L, v_L) of the base field with the following properties:

- the basepoint becomes rational over the strict henselization of (L, v_L)
- all lower components have multiplicities one over (L, v_L)
- if the inertial component is marked as *separable* then the fiber of the cover $\phi : Y \to X$ over the base point splits over the strict henselization of (L, v_L)

Warning: For the moment, this only works if the basepoint is contained inside the closed unit disk.

type_II_point()
Return the type-II-point ξ corresponding to this base component.

upper_components(*u=∞*)
Return the upper components relative to a given extension of the base field.

INPUT:

- `u` – an integer, or `Infinity` (default: `Infinity`)

OUTPUT: the list of upper components of the model of the reduction tree over this inertial component. If $u =$ `Infinity` then the splitting field of this inertial component is used to compute the upper components. Otherwise, u must be step in the ramification filtration of the splitting field, and then the corresponding subfield is used.

class mclf.semistable_reduction.reduction_trees.**LowerComponent**(*Z0*, *vL*, *v*, *phi*)
Bases: *mclf.semistable_reduction.reduction_trees.ReductionComponent*

Return the lower component corresponding to a given valuation.

A *lower component* is a reduction component Z on the base change X_L of the Berkovich line X to some finite extension L/K. It is by construction the inverse image of a given inertial component Z_0 on X, which is part of a *reduction tree*.

INPUT:

- `Z0` – an inertial component of a reduction tree Y
- `vL` – a discrete valuation on a finite extension L of the base field of Y, extending the base valuation on Y
- `v` – a discrete valuation on the base extension to L of the function field F_X, extending v_L
- **phi – the natural morphism from the function field of `Z0` into** the residue field of `v`

OUTPUT: The lower component above Z corresponding to v.

fiber_degree_in_upper_components(*P*)
Return the sum of the absolute degrees of the points above P on all upper components.

map_to_inertial_component()
Return the natural map from this lower component to its inertial component.

EXAMPLES:

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: Y = SuperellipticCurve(x^3-x, 2)
sage: Y3 = SemistableModel(Y,QQ.valuation(3))
sage: Z = Y3.reduction_tree().inertial_components()[0]
sage: W = Z.lower_components()[0]
sage: f = W.map_to_inertial_component()
sage: f.domain()
the smooth projective curve with Rational function field in x over Finite␣
↪Field of size 3
sage: f.codomain()
the smooth projective curve with Rational function field in x over Finite␣
↪Field of size 3
```

upper_components()
Return the list of all upper components lying above this lower component.

This lower component corresponds to a discrete valuation v on a rational function field $L(x)$ extending the valuation v_L, where L/K is some finite extension of the base field K. The upper components correspond to the extensions of v to the function field of Y_L (which is a finite extension of $L(x)$).

Since the computation of all extensions of a nonstandard valuation on a function field to a finite extension is not yet part of Sage, we have to appeal to the MacLane algorithm ourselves.

EXAMPLES:

This example shows that extending valuations also works if the equation is not integral wrt the valuation v

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: Y = SuperellipticCurve(5*x^3 + 1, 2)
sage: Y2 = SemistableModel(Y, QQ.valuation(5))
sage: Y2.is_semistable()  # indirect doctest
True
```

class mclf.semistable_reduction.reduction_trees.**ReductionComponent**
Bases: SageObject

The superclass for the classes LowerComponent and UpperComponent.

base_field()
Return the base field of this reduction component.

base_valuation()
Return the base valuation of this reduction component.

component()
Return the normalization of this reduction component.

constant_base_field()
Return the constant base field of this reduction component.

Note that this field is isomorphic but not equal to the residue field of the base valuation. The isomorphism can be obtained via the restriction of isomorphism between the residue field of the valuation

corresponding to the component and the function field of the component (can be obtained via `self.from_residue_field()`).

from_residue_field()
Return the isomorphism from the residue field of the valuation corresponding to this reduction component to its function field.

function_field()
Return the function field of this reduction component.

Note that the *function field* of this reduction component is the residue field of the corresponding valuation v. It must not be confused with the domain of v, which is the function field of the generic fiber.

function_field_of_generic_fiber()
Return the function field of the generic fiber of the model underlying this reduction component.

inertial_component()
Return the inertial component underlying this reduction component.

multiplicity()
Return the multiplicity of this reduction component.

By definition, this is equal to the ramification index of the valuation corresponding to this component over the base valuation.

reduce(*f*)
Return the image of a function on the generic fiber to this component.

reduction_tree()
Return the reduction tree underlying the reduction component.

valuation()
Return the valuation corresponding to this reduction component.

class mclf.semistable_reduction.reduction_trees.**ReductionTree**(*Y*, *vK*, *T*, *separable_components=None*)

Bases: `SageObject`

Initialize and return a reduction tree associated to a curve and a valuation.

INPUT:

- `Y` – a curve over a basefield K, given as `SmoothProjectiveCurve`
- `vK` – a discrete valuation on K
- `T` – a Berkovich tree on the Berkovich line X^{an} underlying (Y, v_K)
- `separable_components` – a list of type-II-points on X^{an} which are vertices of T (or `None`)

OUTPUT: a reduction tree for `Y` relative to `vK`; the inertial model $\mathcal{X}_0$ is the marked model of X induced by the Berkovich tree T.

Note that the tree T will be modified by the creation of the reduction tree.

Note: In the present release, the base field K must be the field of rational numbers.

---

add_inertial_component(*xi*)
Add a new inertial component to the list of such.

INPUT:

- `xi` – a point of type II on the underlying Berkovich line; it is assumed that `xi` is a vertex of the Berkovich tree T

OUTPUT: a new inertial component Z is created and appended to the list of all inertial components. Moreover, Z is assigned to the new attribute `inertial_component` of the subtree of T with root ξ.

base_field()
Return the base field.

base_valuation()
Return the specified valuation of the base field.

berkovich_line()
Return the Berkovich line X of which the curve Y is a cover.

berkovich_tree()
Return the Berkovich tree underlying this reduction tree.

curve()
Return the curve Y.

inertial_components()
Return the list of inertial components.

is_semistable()
Check wether the reduction specified by this object is semistable.

reduction_conductor()
Return the conductor of the curve.

OUTPUT: a nonnegative integer, which is the conductor of the local Galois representation associated to the reduction which is specified in this `ReductionTree`. If the reduction is semistable, then the result is the conductor of Y.

TODO: Write better documentation.

EXAMPLES:

We check that the conductor exponent takes the component graph into account as well:

```
sage: from mclf import *
sage: R.<x> = QQ[]
sage: Y = SuperellipticCurve(x^3 + x^2 + 3, 2)
sage: Y3 = SemistableModel(Y, QQ.valuation(3))
sage: Y3.is_semistable()
True
sage: Y3.conductor_exponent() # indirect doctest
1
```

reduction_genus()
Return the genus of the reduction.

OUTPUT: a nonnegative integer, which is the arithmetic genus of the reduction of the curve Y specified by the data in this `ReductionTree`, provided this reduction is semistable.

In fact, the number we compute is the sum of the genera of the upper components (i.e. the normalizations of the irreducible components of $\bar{Y}$) and the number of loops of the component graph of $\bar{Y}$, which is (number of double points) - (number of components) + 1.

EXAMPLES:

We test that the arithmetic genus of a totally degenerate curve is computed correctly:

```
sage: from mclf import *
sage: R.<x> = QQ[]
```

(continues on next page)

(continued from previous page)

```
sage: v_3 = QQ.valuation(3)
sage: f = (x^2 - 3)*(x^2 + 3)*(x^3 - 3)
sage: Y = SuperellipticCurve(f, 2)
sage: Y.genus()
3
sage: Y3 = SemistableModel(Y, v_3)
sage: Y3.reduction_tree().reduction_genus()
3
```

class mclf.semistable_reduction.reduction_trees.**UpperComponent**(*Z*, *v*)

Bases: *mclf.semistable_reduction.reduction_trees.ReductionComponent*

Return the upper component above this lower component, corresponding to a given valuation.

INPUT:

- `Z` – a lower component of a reduction tree Y
- `v` – a discrete valuation on the base extension to L of the function field F_Y, extending the valuation corresponding to Z

OUTPUT: The upper component above Z corresponding to v.

Note that the constant base fields of the upper and the lower components are equal, by definition, and isomorphic to the residue field of L.

field_of_constants_degree()

Return the degree of the field of constants over the constant base field of this upper reduction component.

genus()

Return the genus of this upper reduction component.

lower_component()

Return the lower component underneath this upper component.

map_to_lower_component()

Return the natural map from this upper component to the lower component beneath.

mclf.semistable_reduction.reduction_trees.**make_function_field**(*K*)

Return the function field isomorphic to this field, an isomorphism, and its inverse.

INPUT:

- `K` – a field

OUTPUT: A triple (F, ϕ, ψ), where F is a rational function field, $\phi : K \to F$ is a field isomorphism and ψ the inverse of ϕ.

It is assumed that K is either the fraction field of a polynomial ring over a finite field k, or a finite simple extension of such a field.

In the first case, $F = k_1(x)$ is a rational function field over a finite field k_1, where k_1 as an *absolute* finite field isomorphic to k. In the second case, F is a finite simple extension of a rational function field as in the first case.

Note: this command seems to be partly superflous by now, because the residue of a valuation is already of type "function field" whenever this makes sense. However, even if K is a function field over a finite field, it is not guaranteed that the constant base field is a 'true' finite field, and then it is important to change that.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[We17] S. Wewers, *Semistable reduction of superelliptic curves of degree p*, preprint, 2017.

Python Module Index

m

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S