Sage 9.1 Reference Manual: Miscellaneous Modular-Form-Related Modules

Release 9.1

The Sage Development Team

May 21, 2020

CONTENTS

CHAPTER

ONE

DIRICHLET CHARACTERS

A *DirichletCharacter* is the extension of a homomorphism

$$(\mathbf{Z}/N\mathbf{Z})^* \to R^*,$$

for some ring R, to the map $\mathbf{Z}/N\mathbf{Z} \to R$ obtained by sending those $x \in \mathbf{Z}/N\mathbf{Z}$ with $\gcd(N, x) > 1$ to 0.

EXAMPLES:

```
sage: G = DirichletGroup(35)
sage: x = G.gens()
sage: e = x[0]*x[1]^2; e
Dirichlet character modulo 35 of conductor 35 mapping 22 |--> zeta12^3, 31 |-->␣
↪zeta12^2 - 1
sage: e.order()
12
```

This illustrates a canonical coercion:

```
sage: e = DirichletGroup(5, QQ).0
sage: f = DirichletGroup(5,CyclotomicField(4)).0
sage: e*f
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> -zeta4
```

AUTHORS:

- William Stein (2005-09-02): Fixed bug in comparison of Dirichlet characters. It was checking that their values were the same, but not checking that they had the same level!
- William Stein (2006-01-07): added more examples
- William Stein (2006-05-21): added examples of everything; fix a *lot* of tiny bugs and design problem that became clear when creating examples.
- Craig Citro (2008-02-16): speed up __call__ method for Dirichlet characters, miscellaneous fixes
- Julian Rueth (2014-03-06): use UniqueFactory to cache DirichletGroups

class sage.modular.dirichlet.**DirichletCharacter**(*parent*, *x*, *check=True*)

Bases: sage.structure.element.MultiplicativeGroupElement

A Dirichlet character.

bar()

Return the complex conjugate of this Dirichlet character.

EXAMPLES:

```
sage: e = DirichletGroup(5).0
sage: e
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> zeta4
sage: e.bar()
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> -zeta4
```

base_ring()

Returns the base ring of this Dirichlet character.

EXAMPLES:

```
sage: G = DirichletGroup(11)
sage: G.gen(0).base_ring()
Cyclotomic Field of order 10 and degree 4
sage: G = DirichletGroup(11, RationalField())
sage: G.gen(0).base_ring()
Rational Field
```

bernoulli(*k*, *algorithm='recurrence'*, *cache=True*, ***opts*)

Returns the generalized Bernoulli number $B_{k,eps}$.

INPUT:

- `k` – a non-negative integer
- `algorithm` – either `'recurrence'` (default) or `'definition'`
- `cache` – if True, cache answers
- `**opts` – optional arguments; not used directly, but passed to the *bernoulli()* function if this is called

OUTPUT:

Let ε be a (not necessarily primitive) character of modulus N. This function returns the generalized Bernoulli number $B_{k,\varepsilon}$, as defined by the following identity of power series (see for example [DI1995], Section 2.2):

$$\sum_{a=1}^{N} \frac{\varepsilon(a)te^{at}}{e^{Nt}-1} = sum_{k=0}^{\infty}\frac{B_{k,\varepsilon}}{k!}t^k.$$

ALGORITHM:

The `'recurrence'` algorithm computes generalized Bernoulli numbers via classical Bernoulli numbers using the formula in [Coh2007], Proposition 9.4.5; this is usually optimal. The `definition` algorithm uses the definition directly.

> **Warning:** In the case of the trivial Dirichlet character modulo 1, this function returns $B_{1,\varepsilon} = 1/2$, in accordance with the above definition, but in contrast to the value $B_1 = -1/2$ for the classical Bernoulli number. Some authors use an alternative definition giving $B_{1,\varepsilon} = -1/2$; see the discussion in [Coh2007], Section 9.4.1.

EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.bernoulli(5)
```

(continues on next page)

(continued from previous page)

```
7430/13*zeta12^3 - 34750/13*zeta12^2 - 11380/13*zeta12 + 9110/13
sage: eps = DirichletGroup(9).0
sage: eps.bernoulli(3)
10*zeta6 + 4
sage: eps.bernoulli(3, algorithm="definition")
10*zeta6 + 4
```

change_ring(*R*)

Return the base extension of `self` to `R`.

INPUT:

- `R` – either a ring admitting a conversion map from the base ring of `self`, or a ring homomorphism with the base ring of `self` as its domain

EXAMPLES:

```
sage: e = DirichletGroup(7, QQ).0
sage: f = e.change_ring(QuadraticField(3, 'a'))
sage: f.parent()
Group of Dirichlet characters modulo 7 with values in Number Field in a with␣
↪defining polynomial x^2 - 3 with a = 1.732050807568878?
```

```
sage: e = DirichletGroup(13).0
sage: e.change_ring(QQ)
Traceback (most recent call last):
...
TypeError: Unable to coerce zeta12 to a rational
```

We test the case where R is a map (trac ticket #18072):

```
sage: K.<i> = QuadraticField(-1)
sage: chi = DirichletGroup(5, K)[1]
sage: chi(2)
i
sage: f = K.complex_embeddings()[0]
sage: psi = chi.change_ring(f)
sage: psi(2)
-1.83697019872103e-16 - 1.00000000000000*I
```

conductor()

Computes and returns the conductor of this character.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.conductor()
4
sage: b.conductor()
5
sage: (a*b).conductor()
20
```

decomposition()

Return the decomposition of self as a product of Dirichlet characters of prime power modulus, where the prime powers exactly divide the modulus of this character.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: c = a*b
sage: d = c.decomposition(); d
[Dirichlet character modulo 4 of conductor 4 mapping 3 |--> -1, Dirichlet␣
↪character modulo 5 of conductor 5 mapping 2 |--> zeta4]
sage: d[0].parent()
Group of Dirichlet characters modulo 4 with values in Cyclotomic Field of␣
↪order 4 and degree 2
sage: d[1].parent()
Group of Dirichlet characters modulo 5 with values in Cyclotomic Field of␣
↪order 4 and degree 2
```

We can't multiply directly, since coercion of one element into the other parent fails in both cases:

```
sage: d[0]*d[1] == c
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Group of Dirichlet␣
↪characters modulo 4 with values in Cyclotomic Field of order 4 and degree 2
↪' and 'Group of Dirichlet characters modulo 5 with values in Cyclotomic␣
↪Field of order 4 and degree 2'
```

We can multiply if we're explicit about where we want the multiplication to take place.

```
sage: G(d[0])*G(d[1]) == c
True
```

Conductors that are divisible by various powers of 2 present some problems as the multiplicative group modulo 2^k is trivial for $k = 1$ and non-cyclic for $k \geq 3$:

```
sage: (DirichletGroup(18).0).decomposition()
[Dirichlet character modulo 2 of conductor 1, Dirichlet character modulo 9 of␣
↪conductor 9 mapping 2 |--> zeta6]
sage: (DirichletGroup(36).0).decomposition()
[Dirichlet character modulo 4 of conductor 4 mapping 3 |--> -1, Dirichlet␣
↪character modulo 9 of conductor 1 mapping 2 |--> 1]
sage: (DirichletGroup(72).0).decomposition()
[Dirichlet character modulo 8 of conductor 4 mapping 7 |--> -1, 5 |--> 1,␣
↪Dirichlet character modulo 9 of conductor 1 mapping 2 |--> 1]
```

element()

Return the underlying $\mathbf{Z}/n\mathbf{Z}$-module vector of exponents.

> **Warning:** Please do not change the entries of the returned vector; this vector is mutable *only* because immutable vectors are not implemented yet.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.element()
(2, 0)
sage: b.element()
(0, 1)
```

Note: The constructor of `DirichletCharacter` sets the cache of `element()` or of `values_on_gens()`. The cache of one of these methods needs to be set for the other method to work properly, these caches have to be stored when pickling an instance of `DirichletCharacter`.

extend(*M*)

Returns the extension of this character to a Dirichlet character modulo the multiple M of the modulus.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: H.<c> = DirichletGroup(4)
sage: c.extend(20)
Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1
sage: a
Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1
sage: c.extend(20) == a
True
```

galois_orbit(*sort=True*)

Return the orbit of this character under the action of the absolute Galois group of the prime subfield of the base ring.

EXAMPLES:

```
sage: G = DirichletGroup(30); e = G.1
sage: e.galois_orbit()
[Dirichlet character modulo 30 of conductor 5 mapping 11 |--> 1, 7 |--> -
↪zeta4,
 Dirichlet character modulo 30 of conductor 5 mapping 11 |--> 1, 7 |--> zeta4]
```

Another example:

```
sage: G = DirichletGroup(13)
sage: G.galois_orbits()
[
[Dirichlet character modulo 13 of conductor 1 mapping 2 |--> 1],
...,
[Dirichlet character modulo 13 of conductor 13 mapping 2 |--> -1]
]
sage: e = G.0
sage: e
Dirichlet character modulo 13 of conductor 13 mapping 2 |--> zeta12
sage: e.galois_orbit()
[Dirichlet character modulo 13 of conductor 13 mapping 2 |--> zeta12,
 Dirichlet character modulo 13 of conductor 13 mapping 2 |--> -zeta12^3 +␣
↪zeta12,
 Dirichlet character modulo 13 of conductor 13 mapping 2 |--> zeta12^3 -␣
↪zeta12,
 Dirichlet character modulo 13 of conductor 13 mapping 2 |--> -zeta12]
sage: e = G.0^2; e
Dirichlet character modulo 13 of conductor 13 mapping 2 |--> zeta12^2
sage: e.galois_orbit()
[Dirichlet character modulo 13 of conductor 13 mapping 2 |--> zeta12^2,␣
↪Dirichlet character modulo 13 of conductor 13 mapping 2 |--> -zeta12^2 + 1]
```

A non-example:

```
sage: chi = DirichletGroup(7, Integers(9), zeta = Integers(9)(2)).0
sage: chi.galois_orbit()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: Galois orbits only defined if base ring is an integral domain
```

gauss_sum (*a=1*)

Return a Gauss sum associated to this Dirichlet character.

The Gauss sum associated to χ is

$$g_a(\chi) = \sum_{r \in \mathbf{Z}/m\mathbf{Z}} \chi(r)\,\zeta^{ar},$$

where m is the modulus of χ and ζ is a primitive m^{th} root of unity.

FACTS: If the modulus is a prime p and the character is nontrivial, then the Gauss sum has absolute value $\sqrt{p}$.

CACHING: Computed Gauss sums are *not* cached with this character.

EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G([-1])
sage: e.gauss_sum(1)
2*zeta6 - 1
sage: e.gauss_sum(2)
-2*zeta6 + 1
sage: norm(e.gauss_sum())
3
```

```
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.gauss_sum()
-zeta156^46 + zeta156^45 + zeta156^42 + zeta156^41 + 2*zeta156^40 + zeta156^
↪37 - zeta156^36 - zeta156^34 - zeta156^33 - zeta156^31 + 2*zeta156^30 +␣
↪zeta156^28 - zeta156^24 - zeta156^22 + zeta156^21 + zeta156^20 - zeta156^19␣
↪+ zeta156^18 - zeta156^16 - zeta156^15 - 2*zeta156^14 - zeta156^10 +␣
↪zeta156^8 + zeta156^7 + zeta156^6 + zeta156^5 - zeta156^4 - zeta156^2 - 1
sage: factor(norm(e.gauss_sum()))
13^24
```

See also:

- `sage.arith.misc.gauss_sum()` for general finite fields
- `sage.rings.padics.misc.gauss_sum()` for a p-adic version

gauss_sum_numerical (*prec=53, a=1*)

Return a Gauss sum associated to this Dirichlet character as an approximate complex number with prec bits of precision.

INPUT:

- `prec` – integer (default: 53), *bits* of precision
- `a` – integer, as for `gauss_sum()`.

The Gauss sum associated to χ is

$$g_a(\chi) = \sum_{r \in \mathbf{Z}/m\mathbf{Z}} \chi(r)\,\zeta^{ar},$$

where m is the modulus of χ and ζ is a primitive m^{th} root of unity.

EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G.0
sage: abs(e.gauss_sum_numerical())
1.7320508075...
sage: sqrt(3.0)
1.73205080756888
sage: e.gauss_sum_numerical(a=2)
-...e-15 - 1.7320508075...*I
sage: e.gauss_sum_numerical(a=2, prec=100)
4.7331654313260708324703713917e-30 - 1.7320508075688772935274463415*I
sage: G = DirichletGroup(13)
sage: H = DirichletGroup(13, CC)
sage: e = G.0
sage: f = H.0
sage: e.gauss_sum_numerical()
-3.07497205... + 1.8826966926...*I
sage: f.gauss_sum_numerical()
-3.07497205... + 1.8826966926...*I
sage: abs(e.gauss_sum_numerical())
3.60555127546...
sage: abs(f.gauss_sum_numerical())
3.60555127546...
sage: sqrt(13.0)
3.60555127546399
```

is_even()

Return `True` if and only if $\varepsilon(-1) = 1$.

EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.is_even()
False
sage: e(-1)
-1
sage: [e.is_even() for e in G]
[True, False, True, False, True, False, True, False, True, False, True, False]

sage: G = DirichletGroup(13, CC)
sage: e = G.0
sage: e.is_even()
False
sage: e(-1)
-1.000000...
sage: [e.is_even() for e in G]
[True, False, True, False, True, False, True, False, True, False, True, False]

sage: G = DirichletGroup(100000, CC)
```

(continues on next page)

(continued from previous page)

```
sage: G.1.is_even()
True
```

Note that `is_even` need not be the negation of is_odd, e.g., in characteristic 2:

```
sage: G.<e> = DirichletGroup(13, GF(4,'a'))
sage: e.is_even()
True
sage: e.is_odd()
True
```

is_odd()

Return `True` if and only if $\varepsilon(-1) = -1$.

EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.is_odd()
True
sage: [e.is_odd() for e in G]
[False, True, False, True, False, True, False, True, False, True, False, True]

sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.is_odd()
True
sage: [e.is_odd() for e in G]
[False, True, False, True, False, True, False, True, False, True, False, True]

sage: G = DirichletGroup(100000, CC)
sage: G.0.is_odd()
True
```

Note that `is_even` need not be the negation of is_odd, e.g., in characteristic 2:

```
sage: G.<e> = DirichletGroup(13, GF(4,'a'))
sage: e.is_even()
True
sage: e.is_odd()
True
```

is_primitive()

Return `True` if and only if this character is primitive, i.e., its conductor equals its modulus.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.is_primitive()
False
sage: b.is_primitive()
False
sage: (a*b).is_primitive()
True
sage: G.<a,b> = DirichletGroup(20, CC)
sage: a.is_primitive()
False
```

(continues on next page)

(continued from previous page)

```
sage: b.is_primitive()
False
sage: (a*b).is_primitive()
True
```

is_trivial()

Returns `True` if this is the trivial character, i.e., has order 1.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.is_trivial()
False
sage: (a^2).is_trivial()
True
```

jacobi_sum(*char*, *check=True*)

Return the Jacobi sum associated to these Dirichlet characters (i.e., J(self,char)).

This is defined as

$$J(\chi, \psi) = \sum_{a \in \mathbf{Z}/N\mathbf{Z}} \chi(a)\psi(1-a)$$

where χ and ψ are both characters modulo N.

EXAMPLES:

```
sage: D = DirichletGroup(13)
sage: e = D.0
sage: f = D[-2]
sage: e.jacobi_sum(f)
3*zeta12^2 + 2*zeta12 - 3
sage: f.jacobi_sum(e)
3*zeta12^2 + 2*zeta12 - 3
sage: p = 7
sage: DP = DirichletGroup(p)
sage: f = DP.0
sage: e.jacobi_sum(f)
Traceback (most recent call last):
...
NotImplementedError: Characters must be from the same Dirichlet Group.

sage: all_jacobi_sums = [(DP[i].values_on_gens(),DP[j].values_on_gens(),DP[i].
↪jacobi_sum(DP[j]))
....:                   for i in range(p-1) for j in range(i, p-1)]
sage: for s in all_jacobi_sums:
....:     print(s)
((1,), (1,), 5)
((1,), (zeta6,), -1)
((1,), (zeta6 - 1,), -1)
((1,), (-1,), -1)
((1,), (-zeta6,), -1)
((1,), (-zeta6 + 1,), -1)
((zeta6,), (zeta6,), -zeta6 + 3)
((zeta6,), (zeta6 - 1,), 2*zeta6 + 1)
((zeta6,), (-1,), -2*zeta6 - 1)
((zeta6,), (-zeta6,), zeta6 - 3)
```

(continues on next page)

(continued from previous page)

```
((zeta6,), (-zeta6 + 1,), 1)
((zeta6 - 1,), (zeta6 - 1,), -3*zeta6 + 2)
((zeta6 - 1,), (-1,), 2*zeta6 + 1)
((zeta6 - 1,), (-zeta6,), -1)
((zeta6 - 1,), (-zeta6 + 1,), -zeta6 - 2)
((-1,), (-1,), 1)
((-1,), (-zeta6,), -2*zeta6 + 3)
((-1,), (-zeta6 + 1,), 2*zeta6 - 3)
((-zeta6,), (-zeta6,), 3*zeta6 - 1)
((-zeta6,), (-zeta6 + 1,), -2*zeta6 + 3)
((-zeta6 + 1,), (-zeta6 + 1,), zeta6 + 2)
```

Let's check that trivial sums are being calculated correctly:

```
sage: N = 13
sage: D = DirichletGroup(N)
sage: g = D(1)
sage: g.jacobi_sum(g)
11
sage: sum([g(x)*g(1-x) for x in IntegerModRing(N)])
11
```

And sums where exactly one character is nontrivial (see trac ticket #6393):

```
sage: G = DirichletGroup(5); X=G.list(); Y=X[0]; Z=X[1]
sage: Y.jacobi_sum(Z)
-1
sage: Z.jacobi_sum(Y)
-1
```

Now let's take a look at a non-prime modulus:

```
sage: N = 9
sage: D = DirichletGroup(N)
sage: g = D(1)
sage: g.jacobi_sum(g)
3
```

We consider a sum with values in a finite field:

```
sage: g = DirichletGroup(17, GF(9,'a')).0
sage: g.jacobi_sum(g**2)
2*a
```

kernel()

Return the kernel of this character.

OUTPUT: Currently the kernel is returned as a list. This may change.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.kernel()
[1, 9, 13, 17]
sage: b.kernel()
[1, 11]
```

kloosterman_sum(*a=1*, *b=0*)

Return the "twisted" Kloosterman sum associated to this Dirichlet character.

This includes Gauss sums, classical Kloosterman sums, Salié sums, etc.

The Kloosterman sum associated to χ and the integers a,b is

$$K(a,b,\chi) = \sum_{r \in (\mathbf{Z}/m\mathbf{Z})^\times} \chi(r)\,\zeta^{ar+br^{-1}},$$

where m is the modulus of χ and ζ is a primitive m th root of unity. This reduces to the Gauss sum if $b = 0$.

This method performs an exact calculation and returns an element of a suitable cyclotomic field; see also `kloosterman_sum_numerical()`, which gives an inexact answer (but is generally much quicker).

CACHING: Computed Kloosterman sums are *not* cached with this character.

EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G([-1])
sage: e.kloosterman_sum(3,5)
-2*zeta6 + 1
sage: G = DirichletGroup(20)
sage: e = G([1 for  u in G.unit_gens()])
sage: e.kloosterman_sum(7,17)
-2*zeta20^6 + 2*zeta20^4 + 4
```

kloosterman_sum_numerical(*prec=53*, *a=1*, *b=0*)

Return the Kloosterman sum associated to this Dirichlet character as an approximate complex number with prec bits of precision. See also `kloosterman_sum()`, which calculates the sum exactly (which is generally slower).

INPUT:

- `prec` – integer (default: 53), *bits* of precision
- `a` – integer, as for `kloosterman_sum()`
- `b` – integer, as for `kloosterman_sum()`.

EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G.0
```

The real component of the numerical value of e is near zero:

```
sage: v=e.kloosterman_sum_numerical()
sage: v.real() < 1.0e15
True
sage: v.imag()
1.73205080756888
sage: G = DirichletGroup(20)
sage: e = G.1
sage: e.kloosterman_sum_numerical(53,3,11)
3.80422606518061 - 3.80422606518061*I
```

level()

Synonym for modulus.

EXAMPLES:

```
sage: e = DirichletGroup(100, QQ).0
sage: e.level()
100
```

lfunction (*prec=53*, *algorithm='pari'*)

Return the L-function of `self`.

The result is a wrapper around a PARI L-function or around the `lcalc` program.

INPUT:

- `prec` – precision (default 53)
- `algorithm` – 'pari' (default) or 'lcalc'

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: L = a.lfunction(); L
PARI L-function associated to Dirichlet character modulo 20
of conductor 4 mapping 11 |--> -1, 17 |--> 1
sage: L(4)
0.988944551741105
```

With the algorithm "lcalc":

```
sage: a = a.primitive_character()
sage: L = a.lfunction(algorithm='lcalc'); L
L-function with complex Dirichlet coefficients
sage: L.value(4)  # abs tol 1e-14
0.988944551741105 - 5.16608739123418e-18*I
```

maximize_base_ring ()

Let

$$\varepsilon : (\mathbf{Z}/N\mathbf{Z})^* \to \mathbf{Q}(\zeta_n)$$

be a Dirichlet character. This function returns an equal Dirichlet character

$$\chi : (\mathbf{Z}/N\mathbf{Z})^* \to \mathbf{Q}(\zeta_m)$$

where m is the least common multiple of n and the exponent of $(\mathbf{Z}/N\mathbf{Z})^*$.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20,QQ)
sage: b.maximize_base_ring()
Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> -1
sage: b.maximize_base_ring().base_ring()
Cyclotomic Field of order 4 and degree 2
sage: DirichletGroup(20).base_ring()
Cyclotomic Field of order 4 and degree 2
```

minimize_base_ring ()

Return a Dirichlet character that equals this one, but over as small a subfield (or subring) of the base ring as possible.

Note: This function is currently only implemented when the base ring is a number field. It's the identity function in characteristic p.

EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = DirichletGroup(13).0
sage: e.base_ring()
Cyclotomic Field of order 12 and degree 4
sage: e.minimize_base_ring().base_ring()
Cyclotomic Field of order 12 and degree 4
sage: (e^2).minimize_base_ring().base_ring()
Cyclotomic Field of order 6 and degree 2
sage: (e^3).minimize_base_ring().base_ring()
Cyclotomic Field of order 4 and degree 2
sage: (e^12).minimize_base_ring().base_ring()
Rational Field
```

modulus()

The modulus of this character.

EXAMPLES:

```
sage: e = DirichletGroup(100, QQ).0
sage: e.modulus()
100
sage: e.conductor()
4
```

multiplicative_order()

The order of this character.

EXAMPLES:

```
sage: e = DirichletGroup(100).1
sage: e.order()     # same as multiplicative_order, since group is␣
↪multiplicative
20
sage: e.multiplicative_order()
20
sage: e = DirichletGroup(100).0
sage: e.multiplicative_order()
2
```

primitive_character()

Returns the primitive character associated to self.

EXAMPLES:

```
sage: e = DirichletGroup(100).0; e
Dirichlet character modulo 100 of conductor 4 mapping 51 |--> -1, 77 |--> 1
sage: e.conductor()
4
sage: f = e.primitive_character(); f
Dirichlet character modulo 4 of conductor 4 mapping 3 |--> -1
sage: f.modulus()
4
```

restrict(*M*)

Returns the restriction of this character to a Dirichlet character modulo the divisor M of the modulus, which must also be a multiple of the conductor of this character.

EXAMPLES:

```
sage: e = DirichletGroup(100).0
sage: e.modulus()
100
sage: e.conductor()
4
sage: e.restrict(20)
Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1
sage: e.restrict(4)
Dirichlet character modulo 4 of conductor 4 mapping 3 |--> -1
sage: e.restrict(50)
Traceback (most recent call last):
...
ValueError: conductor(=4) must divide M(=50)
```

values()

Return a list of the values of this character on each integer between 0 and the modulus.

EXAMPLES:

```
sage: e = DirichletGroup(20)(1)
sage: e.values()
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1]
sage: e = DirichletGroup(20).gen(0)
sage: e.values()
[0, 1, 0, -1, 0, 0, 0, -1, 0, 1, 0, -1, 0, 1, 0, 0, 0, 1, 0, -1]
sage: e = DirichletGroup(20).gen(1)
sage: e.values()
[0, 1, 0, -zeta4, 0, 0, 0, zeta4, 0, -1, 0, 1, 0, -zeta4, 0, 0, 0, zeta4, 0, -
↪1]
sage: e = DirichletGroup(21).gen(0) ; e.values()
[0, 1, -1, 0, 1, -1, 0, 0, -1, 0, 1, -1, 0, 1, 0, 0, 1, -1, 0, 1, -1]
sage: e = DirichletGroup(21, base_ring=GF(37)).gen(0) ; e.values()
[0, 1, 36, 0, 1, 36, 0, 0, 36, 0, 1, 36, 0, 1, 0, 0, 1, 36, 0, 1, 36]
sage: e = DirichletGroup(21, base_ring=GF(3)).gen(0) ; e.values()
[0, 1, 2, 0, 1, 2, 0, 0, 2, 0, 1, 2, 0, 1, 0, 0, 1, 2, 0, 1, 2]
```

```
sage: chi = DirichletGroup(100151, CyclotomicField(10)).0
sage: ls = chi.values() ; ls[0:10]
[0,
1,
-zeta10^3,
-zeta10,
-zeta10,
1,
zeta10^3 - zeta10^2 + zeta10 - 1,
zeta10,
zeta10^3 - zeta10^2 + zeta10 - 1,
zeta10^2]
```

values_on_gens()

Return a tuple of the values of `self` on the standard generators of $(\mathbf{Z}/N\mathbf{Z})^*$, where N is the modulus.

EXAMPLES:

```
sage: e = DirichletGroup(16)([-1, 1])
sage: e.values_on_gens ()
(-1, 1)
```

Note: The constructor of `DirichletCharacter` sets the cache of `element()` or of `values_on_gens()`. The cache of one of these methods needs to be set for the other method to work properly, these caches have to be stored when pickling an instance of `DirichletCharacter`.

class `sage.modular.dirichlet.`**`DirichletGroupFactory`**

Bases: `sage.structure.factory.UniqueFactory`

Construct a group of Dirichlet characters modulo N.

INPUT:

- `N` – positive integer
- `base_ring` – commutative ring; the value ring for the characters in this group (default: the cyclotomic field $\mathbf{Q}(\zeta_n)$, where n is the exponent of $(\mathbf{Z}/N\mathbf{Z})^*$)
- `zeta` – (optional) root of unity in `base_ring`
- `zeta_order` – (optional) positive integer; this must be the order of `zeta` if both are specified
- `names` – ignored (needed so `G.<...> = DirichletGroup(...)` notation works)
- `integral` – boolean (default: `False`); whether to replace the default cyclotomic field by its rings of integers as the base ring. This is ignored if `base_ring` is not `None`.

OUTPUT:

The group of Dirichlet characters modulo N with values in a subgroup V of the multiplicative group R^* of `base_ring`. This is the group of homomorphisms $(\mathbf{Z}/N\mathbf{Z})^* \to V$ with pointwise multiplication. The group V is determined as follows:

- If both `zeta` and `zeta_order` are omitted, then V is taken to be R^*, or equivalently its n-torsion subgroup, where n is the exponent of $(\mathbf{Z}/N\mathbf{Z})^*$. Many operations, such as finding a set of generators for the group, are only implemented if V is cyclic and a generator for V can be found.
- If `zeta` is specified, then V is taken to be the cyclic subgroup of R^* generated by `zeta`. If `zeta_order` is also given, it must be the multiplicative order of `zeta`; this is useful if the base ring is not exact or if the order of `zeta` is very large.
- If `zeta` is not specified but `zeta_order` is, then V is taken to be the group of roots of unity of order dividing `zeta_order` in R. In this case, R must be a domain (so V is cyclic), and V must have order `zeta_order`. Furthermore, a generator `zeta` of V is computed, and an error is raised if such `zeta` cannot be found.

EXAMPLES:

The default base ring is a cyclotomic field of order the exponent of $(\mathbf{Z}/N\mathbf{Z})^*$:

```
sage: DirichletGroup(20)
Group of Dirichlet characters modulo 20 with values in Cyclotomic Field of order␣
↪4 and degree 2
```

We create the group of Dirichlet character mod 20 with values in the rational numbers:

```
sage: G = DirichletGroup(20, QQ); G
Group of Dirichlet characters modulo 20 with values in Rational Field
sage: G.order()
4
sage: G.base_ring()
Rational Field
```

The elements of G print as lists giving the values of the character on the generators of $(Z/NZ)^*$:

```
sage: list(G)
[Dirichlet character modulo 20 of conductor 1 mapping 11 |--> 1, 17 |--> 1,␣
↪Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1,␣
↪Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> -1,␣
↪Dirichlet character modulo 20 of conductor 20 mapping 11 |--> -1, 17 |--> -1]
```

Next we construct the group of Dirichlet character mod 20, but with values in $\mathbf{Q}(\zeta_n)$:

```
sage: G = DirichletGroup(20)
sage: G.1
Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> zeta4
```

We next compute several invariants of `G`:

```
sage: G.gens()
(Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1,␣
↪Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> zeta4)
sage: G.unit_gens()
(11, 17)
sage: G.zeta()
zeta4
sage: G.zeta_order()
4
```

In this example we create a Dirichlet group with values in a number field:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^4 + 1)
sage: DirichletGroup(5, K)
Group of Dirichlet characters modulo 5 with values in Number Field in a with␣
↪defining polynomial x^4 + 1
```

An example where we give `zeta`, but not its order:

```
sage: G = DirichletGroup(5, K, a); G
Group of Dirichlet characters modulo 5 with values in the group of order 8␣
↪generated by a in Number Field in a with defining polynomial x^4 + 1
sage: G.list()
[Dirichlet character modulo 5 of conductor 1 mapping 2 |--> 1, Dirichlet␣
↪character modulo 5 of conductor 5 mapping 2 |--> a^2, Dirichlet character␣
↪modulo 5 of conductor 5 mapping 2 |--> -1, Dirichlet character modulo 5 of␣
↪conductor 5 mapping 2 |--> -a^2]
```

We can also restrict the order of the characters, either with or without specifying a root of unity:

```
sage: DirichletGroup(5, K, zeta=-1, zeta_order=2)
Group of Dirichlet characters modulo 5 with values in the group of order 2␣
↪generated by -1 in Number Field in a with defining polynomial x^4 + 1
sage: DirichletGroup(5, K, zeta_order=2)
Group of Dirichlet characters modulo 5 with values in the group of order 2␣
↪generated by -1 in Number Field in a with defining polynomial x^4 + 1
```

```
sage: G.<e> = DirichletGroup(13)
sage: loads(G.dumps()) == G
True
```

```
sage: G = DirichletGroup(19, GF(5))
sage: loads(G.dumps()) == G
True
```

We compute a Dirichlet group over a large prime field:

```
sage: p = next_prime(10^40)
sage: g = DirichletGroup(19, GF(p)); g
Group of Dirichlet characters modulo 19 with values in Finite Field of size␣
↪10000000000000000000000000000000000000121
```

Note that the root of unity has small order, i.e., it is not the largest order root of unity in the field:

```
sage: g.zeta_order()
2
```

```
sage: r4 = CyclotomicField(4).ring_of_integers()
sage: G = DirichletGroup(60, r4)
sage: G.gens()
(Dirichlet character modulo 60 of conductor 4 mapping 31 |--> -1, 41 |--> 1, 37 |-
↪-> 1, Dirichlet character modulo 60 of conductor 3 mapping 31 |--> 1, 41 |--> -
↪1, 37 |--> 1, Dirichlet character modulo 60 of conductor 5 mapping 31 |--> 1,␣
↪41 |--> 1, 37 |--> zeta4)
sage: val = G.gens()[2].values_on_gens()[2] ; val
zeta4
sage: parent(val)
Gaussian Integers in Cyclotomic Field of order 4 and degree 2
sage: r4.residue_field(r4.ideal(29).factor()[0][0])(val)
17
sage: r4.residue_field(r4.ideal(29).factor()[0][0])(val) * GF(29)(3)
22
sage: r4.residue_field(r4.ideal(29).factor()[0][0])(G.gens()[2].values_on_
↪gens()[2]) * 3
22
sage: parent(r4.residue_field(r4.ideal(29).factor()[0][0])(G.gens()[2].values_on_
↪gens()[2]) * 3)
Residue field of Fractional ideal (-2*zeta4 + 5)
```

```
sage: DirichletGroup(60, integral=True)
Group of Dirichlet characters modulo 60 with values in Gaussian Integers in␣
↪Cyclotomic Field of order 4 and degree 2
sage: parent(DirichletGroup(60, integral=True).gens()[2].values_on_gens()[2])
Gaussian Integers in Cyclotomic Field of order 4 and degree 2
```

If the order of `zeta` cannot be determined automatically, we can specify it using `zeta_order`:

```
sage: DirichletGroup(7, CC, zeta=exp(2*pi*I/6))
Traceback (most recent call last):
...
NotImplementedError: order of element not known

sage: DirichletGroup(7, CC, zeta=exp(2*pi*I/6), zeta_order=6)
Group of Dirichlet characters modulo 7 with values in the group of order 6␣
↪generated by 0.500000000000000 + 0.866025403784439*I in Complex Field with 53␣
↪bits of precision
```

If the base ring is not a domain (in which case the group of roots of unity is not necessarily cyclic), some operations still work, such as creation of elements:

```
sage: G = DirichletGroup(5, Zmod(15)); G
Group of Dirichlet characters modulo 5 with values in Ring of integers modulo 15
sage: chi = G([13]); chi
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> 13
sage: chi^2
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> 4
sage: chi.multiplicative_order()
4
```

Other operations only work if `zeta` is specified:

```
sage: G.gens()
Traceback (most recent call last):
...
NotImplementedError: factorization of polynomials over rings with composite␣
↪characteristic is not implemented
sage: G = DirichletGroup(5, Zmod(15), zeta=2); G
Group of Dirichlet characters modulo 5 with values in the group of order 4␣
↪generated by 2 in Ring of integers modulo 15
sage: G.gens()
(Dirichlet character modulo 5 of conductor 5 mapping 2 |--> 2,)
```

create_key (*N*, *base_ring=None*, *zeta=None*, *zeta_order=None*, *names=None*, *integral=False*)
Create a key that uniquely determines a Dirichlet group.

create_object (*version*, *key*, ***extra_args*)
Create the object from the key (extra arguments are ignored). This is only called if the object was not found in the cache.

class sage.modular.dirichlet.**DirichletGroup_class** (*base_ring*, *modulus*, *zeta*, *zeta_order*)
Bases: `sage.misc.fast_methods.WithEqualityById`, `sage.structure.parent.Parent`

Group of Dirichlet characters modulo N with values in a ring R.

Element
alias of `DirichletCharacter`

base_extend (*R*)
Return the base extension of `self` to `R`.

INPUT:

- `R` – either a ring admitting a *coercion* map from the base ring of `self`, or a ring homomorphism with the base ring of `self` as its domain

EXAMPLES:

```
sage: G = DirichletGroup(7,QQ); G
Group of Dirichlet characters modulo 7 with values in Rational Field
sage: H = G.base_extend(CyclotomicField(6)); H
Group of Dirichlet characters modulo 7 with values in Cyclotomic Field of␣
↪order 6 and degree 2
```

Note that the root of unity can change:

```
sage: H.zeta()
zeta6
```

This method (in contrast to *change_ring()*) requires a coercion map to exist:

```
sage: G.base_extend(ZZ)
Traceback (most recent call last):
...
TypeError: no coercion map from Rational Field to Integer Ring is defined
```

Base-extended Dirichlet groups do not silently get roots of unity with smaller order than expected (trac ticket #6018):

```
sage: G = DirichletGroup(10, QQ).base_extend(CyclotomicField(4))
sage: H = DirichletGroup(10, CyclotomicField(4))
sage: G is H
True

sage: G3 = DirichletGroup(31, CyclotomicField(3))
sage: G5 = DirichletGroup(31, CyclotomicField(5))
sage: K30 = CyclotomicField(30)
sage: G3.gen(0).base_extend(K30) * G5.gen(0).base_extend(K30)
Dirichlet character modulo 31 of conductor 31 mapping 3 |--> -zeta30^7 +
↪zeta30^5 + zeta30^4 + zeta30^3 - zeta30 - 1
```

When a root of unity is specified, base extension still works if the new base ring is not an integral domain:

```
sage: f = DirichletGroup(17, ZZ, zeta=-1).0
sage: g = f.base_extend(Integers(15))
sage: g(3)
14
sage: g.parent().zeta()
14
```

change_ring(*R*, *zeta=None*, *zeta_order=None*)

Return the base extension of `self` to `R`.

INPUT:

- `R` – either a ring admitting a conversion map from the base ring of `self`, or a ring homomorphism with the base ring of `self` as its domain
- `zeta` – (optional) root of unity in `R`
- `zeta_order` – (optional) order of `zeta`

EXAMPLES:

```
sage: G = DirichletGroup(7,QQ); G
Group of Dirichlet characters modulo 7 with values in Rational Field
sage: G.change_ring(CyclotomicField(6))
Group of Dirichlet characters modulo 7 with values in Cyclotomic Field of
↪order 6 and degree 2
```

decomposition()

Returns the Dirichlet groups of prime power modulus corresponding to primes dividing modulus.

(Note that if the modulus is 2 mod 4, there will be a "factor" of $(\mathbf{Z}/2\mathbf{Z})^*$, which is the trivial group.)

EXAMPLES:

```
sage: DirichletGroup(20).decomposition()
[
```

(continues on next page)

(continued from previous page)

```
Group of Dirichlet characters modulo 4 with values in Cyclotomic Field of␣
↪order 4 and degree 2,
Group of Dirichlet characters modulo 5 with values in Cyclotomic Field of␣
↪order 4 and degree 2
]
sage: DirichletGroup(20,GF(5)).decomposition()
[
Group of Dirichlet characters modulo 4 with values in Finite Field of size 5,
Group of Dirichlet characters modulo 5 with values in Finite Field of size 5
]
```

exponent()

Return the exponent of this group.

EXAMPLES:

```
sage: DirichletGroup(20).exponent()
4
sage: DirichletGroup(20,GF(3)).exponent()
2
sage: DirichletGroup(20,GF(2)).exponent()
1
sage: DirichletGroup(37).exponent()
36
```

galois_orbits(*v=None*, *reps_only=False*, *sort=True*, *check=True*)

Return a list of the Galois orbits of Dirichlet characters in self, or in v if v is not None.

INPUT:

- `v` - (optional) list of elements of self
- `reps_only` - (optional: default False) if True only returns representatives for the orbits.
- `sort` - (optional: default True) whether to sort the list of orbits and the orbits themselves (slightly faster if False).
- `check` - (optional, default: True) whether or not to explicitly coerce each element of v into self.

The Galois group is the absolute Galois group of the prime subfield of Frac(R). If R is not a domain, an error will be raised.

EXAMPLES:

```
sage: DirichletGroup(20).galois_orbits()
[
[Dirichlet character modulo 20 of conductor 20 mapping 11 |--> -1, 17 |--> -
↪1],
...,
[Dirichlet character modulo 20 of conductor 1 mapping 11 |--> 1, 17 |--> 1]
]
sage: DirichletGroup(17, Integers(6), zeta=Integers(6)(5)).galois_orbits()
Traceback (most recent call last):
...
TypeError: Galois orbits only defined if base ring is an integral domain
sage: DirichletGroup(17, Integers(9), zeta=Integers(9)(2)).galois_orbits()
Traceback (most recent call last):
...
TypeError: Galois orbits only defined if base ring is an integral domain
```

gen (*n=0*)

Return the n-th generator of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.gen(0)
Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1
sage: G.gen(1)
Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> zeta4
sage: G.gen(2)
Traceback (most recent call last):
...
IndexError: n(=2) must be between 0 and 1
```

```
sage: G.gen(-1)
Traceback (most recent call last):
...
IndexError: n(=-1) must be between 0 and 1
```

gens ()

Returns generators of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.gens()
(Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1, Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> zeta4)
```

integers_mod ()

Returns the group of integers $\mathbf{Z}/N\mathbf{Z}$ where N is the modulus of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.integers_mod()
Ring of integers modulo 20
```

list ()

Return a list of the Dirichlet characters in this group.

EXAMPLES:

```
sage: DirichletGroup(5).list()
[Dirichlet character modulo 5 of conductor 1 mapping 2 |--> 1,
 Dirichlet character modulo 5 of conductor 5 mapping 2 |--> zeta4,
 Dirichlet character modulo 5 of conductor 5 mapping 2 |--> -1,
 Dirichlet character modulo 5 of conductor 5 mapping 2 |--> -zeta4]
```

modulus ()

Returns the modulus of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.modulus()
20
```

ngens()

Returns the number of generators of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.ngens()
2
```

order()

Return the number of elements of self. This is the same as len(self).

EXAMPLES:

```
sage: DirichletGroup(20).order()
8
sage: DirichletGroup(37).order()
36
```

random_element()

Return a random element of self.

The element is computed by multiplying a random power of each generator together, where the power is between 0 and the order of the generator minus 1, inclusive.

EXAMPLES:

```
sage: DirichletGroup(37).random_element()
Dirichlet character modulo 37 of conductor 37 mapping 2 |--> zeta36^4
sage: DirichletGroup(20).random_element()
Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1
sage: DirichletGroup(60).random_element()
Dirichlet character modulo 60 of conductor 3 mapping 31 |--> 1, 41 |--> -1,␣
↪37 |--> 1
```

unit_gens()

Returns the minimal generators for the units of $(\mathbf{Z}/N\mathbf{Z})^*$, where N is the modulus of self.

EXAMPLES:

```
sage: DirichletGroup(37).unit_gens()
(2,)
sage: DirichletGroup(20).unit_gens()
(11, 17)
sage: DirichletGroup(60).unit_gens()
(31, 41, 37)
sage: DirichletGroup(20,QQ).unit_gens()
(11, 17)
```

zeta()

Return the chosen root of unity in the base ring.

EXAMPLES:

```
sage: DirichletGroup(37).zeta()
zeta36
sage: DirichletGroup(20).zeta()
zeta4
sage: DirichletGroup(60).zeta()
```

(continues on next page)

(continued from previous page)

```
zeta4
sage: DirichletGroup(60,QQ).zeta()
-1
sage: DirichletGroup(60, GF(25,'a')).zeta()
2
```

zeta_order()

Return the order of the chosen root of unity in the base ring.

EXAMPLES:

```
sage: DirichletGroup(20).zeta_order()
4
sage: DirichletGroup(60).zeta_order()
4
sage: DirichletGroup(60, GF(25,'a')).zeta_order()
4
sage: DirichletGroup(19).zeta_order()
18
```

sage.modular.dirichlet.**TrivialCharacter**(*N*, *base_ring=Rational Field*)

Return the trivial character of the given modulus, with values in the given base ring.

EXAMPLES:

```
sage: t = trivial_character(7)
sage: [t(x) for x in [0..20]]
[0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
sage: t(1).parent()
Rational Field
sage: trivial_character(7, Integers(3))(1).parent()
Ring of integers modulo 3
```

sage.modular.dirichlet.**is_DirichletCharacter**(*x*)

Return True if x is of type DirichletCharacter.

EXAMPLES:

```
sage: from sage.modular.dirichlet import is_DirichletCharacter
sage: is_DirichletCharacter(trivial_character(3))
True
sage: is_DirichletCharacter([1])
False
```

sage.modular.dirichlet.**is_DirichletGroup**(*x*)

Returns True if x is a Dirichlet group.

EXAMPLES:

```
sage: from sage.modular.dirichlet import is_DirichletGroup
sage: is_DirichletGroup(DirichletGroup(11))
True
sage: is_DirichletGroup(11)
False
sage: is_DirichletGroup(DirichletGroup(11).0)
False
```

sage.modular.dirichlet.**kronecker_character**(*d*)

Return the quadratic Dirichlet character (d/.) of minimal conductor.

EXAMPLES:

```
sage: kronecker_character(97*389*997^2)
Dirichlet character modulo 37733 of conductor 37733 mapping 1557 |--> -1, 37346 |-
↪-> -1
```

```
sage: a = kronecker_character(1)
sage: b = DirichletGroup(2401,QQ)(a)     # NOTE -- over QQ!
sage: b.modulus()
2401
```

AUTHORS:

- Jon Hanke (2006-08-06)

sage.modular.dirichlet.**kronecker_character_upside_down**(*d*)

Return the quadratic Dirichlet character (./d) of conductor d, for d0.

EXAMPLES:

```
sage: kronecker_character_upside_down(97*389*997^2)
Dirichlet character modulo 37506941597 of conductor 37733 mapping 13533432536 |-->
↪ -1, 22369178537 |--> -1, 14266017175 |--> 1
```

AUTHORS:

- Jon Hanke (2006-08-06)

sage.modular.dirichlet.**trivial_character**(*N*, *base_ring=Rational Field*)

Return the trivial character of the given modulus, with values in the given base ring.

EXAMPLES:

```
sage: t = trivial_character(7)
sage: [t(x) for x in [0..20]]
[0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
sage: t(1).parent()
Rational Field
sage: trivial_character(7, Integers(3))(1).parent()
Ring of integers modulo 3
```

CHAPTER

TWO

THE SET $\mathbb{P}^1(\mathbf{Q})$ OF CUSPS

EXAMPLES:

```
sage: Cusps
Set P^1(QQ) of all cusps
```

```
sage: Cusp(oo)
Infinity
```

class sage.modular.cusps.**Cusp**(*a*, *b=None*, *parent=None*, *check=True*)

Bases: sage.structure.element.Element

A cusp.

A cusp is either a rational number or infinity, i.e., an element of the projective line over Q. A Cusp is stored as a pair (a,b), where gcd(a,b)=1 and a,b are of type Integer.

EXAMPLES:

```
sage: a = Cusp(2/3); b = Cusp(oo)
sage: a.parent()
Set P^1(QQ) of all cusps
sage: a.parent() is b.parent()
True
```

apply(*g*)

Return g(self), where g=[a,b,c,d] is a list of length 4, which we view as a linear fractional transformation.

EXAMPLES: Apply the identity matrix:

```
sage: Cusp(0).apply([1,0,0,1])
0
sage: Cusp(0).apply([0,-1,1,0])
Infinity
sage: Cusp(0).apply([1,-3,0,1])
-3
```

denominator()

Return the denominator of the cusp a/b.

EXAMPLES:

```
sage: x=Cusp(6,9); x
2/3
sage: x.denominator()
3
```

(continues on next page)

(continued from previous page)

```
sage: Cusp(oo).denominator()
0
sage: Cusp(-5/10).denominator()
2
```

galois_action(t, N)

Suppose this cusp is α, G a congruence subgroup of level N and σ is the automorphism in the Galois group of $\mathbf{Q}(\zeta_N)/\mathbf{Q}$ that sends ζ_N to ζ_N^t. Then this function computes a cusp β such that $\sigma([\alpha]) = [\beta]$, where $[\alpha]$ is the equivalence class of α modulo G.

This code only needs as input the level and not the group since the action of Galois for a congruence group G of level N is compatible with the action of the full congruence group $\Gamma(N)$.

INPUT:

- t – integer that is coprime to N
- N – positive integer (level)

OUTPUT:

- a cusp

Warning: In some cases N must fit in a long long, i.e., there are cases where this algorithm isn't fully implemented.

Note: Modular curves can have multiple non-isomorphic models over $\mathbf{Q}$. The action of Galois depends on such a model. The model over $\mathbf{Q}$ of $X(G)$ used here is the model where the function field $\mathbf{Q}(X(G))$ is given by the functions whose Fourier expansion at ∞ have their coefficients in $\mathbf{Q}$. For $X(N) := X(\Gamma(N))$ the corresponding moduli interpretation over $\mathbf{Z}[1/N]$ is that $X(N)$ parametrizes pairs (E, a) where E is a (generalized) elliptic curve and $a : \mathbf{Z}/N\mathbf{Z} \times \mu_N \to E$ is a closed immersion such that the Weil pairing of $a(1, 1)$ and $a(0, \zeta_N)$ is ζ_N. In this parameterisation the point $z \in H$ corresponds to the pair (E_z, a_z) with $E_z = \mathbf{C}/(z\mathbf{Z} + \mathbf{Z})$ and $a_z : \mathbf{Z}/N\mathbf{Z} \times \mu_N \to E$ given by $a_z(1, 1) = z/N$ and $a_z(0, \zeta_N) = 1/N$. Similarly $X_1(N) := X(\Gamma_1(N))$ parametrizes pairs (E, a) where $a : \mu_N \to E$ is a closed immersion.

EXAMPLES:

```
sage: Cusp(1/10).galois_action(3, 50)
1/170
sage: Cusp(oo).galois_action(3, 50)
Infinity
sage: c=Cusp(0).galois_action(3, 50); c
50/67
sage: Gamma0(50).reduce_cusp(c)
0
```

Here we compute the permutations of the action for t=3 on cusps for Gamma0(50).

```
sage: N = 50; t=3; G = Gamma0(N); C = G.cusps()
sage: cl = lambda z: exists(C, lambda y:y.is_gamma0_equiv(z, N))[1]
sage: for i in range(5):
....:     print((i, t^i))
....:     print([cl(alpha.galois_action(t^i,N)) for alpha in C])
(0, 1)
```

(continues on next page)

(continued from previous page)

```
[0, 1/25, 1/10, 1/5, 3/10, 2/5, 1/2, 3/5, 7/10, 4/5, 9/10, Infinity]
(1, 3)
[0, 1/25, 7/10, 2/5, 1/10, 4/5, 1/2, 1/5, 9/10, 3/5, 3/10, Infinity]
(2, 9)
[0, 1/25, 9/10, 4/5, 7/10, 3/5, 1/2, 2/5, 3/10, 1/5, 1/10, Infinity]
(3, 27)
[0, 1/25, 3/10, 3/5, 9/10, 1/5, 1/2, 4/5, 1/10, 2/5, 7/10, Infinity]
(4, 81)
[0, 1/25, 1/10, 1/5, 3/10, 2/5, 1/2, 3/5, 7/10, 4/5, 9/10, Infinity]
```

REFERENCES:

- Section 1.3 of Glenn Stevens, "Arithmetic on Modular Curves"
- There is a long comment about our algorithm in the source code for this function.

AUTHORS:

- William Stein, 2009-04-18

is_gamma0_equiv (*other*, *N*, *transformation=None*)

Return whether self and other are equivalent modulo the action of $\Gamma_0(N)$ via linear fractional transformations.

INPUT:

- `other` - Cusp
- `N` - an integer (specifies the group Gamma_0(N))
- `transformation` - None (default) or either the string 'matrix' or 'corner'. If 'matrix', it also returns a matrix in Gamma_0(N) that sends self to other. The matrix is chosen such that the lower left entry is as small as possible in absolute value. If 'corner' (or True for backwards compatibility), it returns only the upper left entry of such a matrix.

OUTPUT:

- a boolean - True if self and other are equivalent
- a matrix or an integer- returned only if transformation is 'matrix' or 'corner', respectively.

EXAMPLES:

```
sage: x = Cusp(2,3)
sage: y = Cusp(4,5)
sage: x.is_gamma0_equiv(y, 2)
True
sage: _, ga = x.is_gamma0_equiv(y, 2, 'matrix'); ga
[-1  2]
[-2  3]
sage: x.is_gamma0_equiv(y, 3)
False
sage: x.is_gamma0_equiv(y, 3, 'matrix')
(False, None)
sage: Cusp(1/2).is_gamma0_equiv(1/3,11,'corner')
(True, 19)

sage: Cusp(1,0)
Infinity
sage: z = Cusp(1,0)
sage: x.is_gamma0_equiv(z, 3, 'matrix')
```

(continues on next page)

(continued from previous page)

```
(
      [-1  1]
True, [-3  2]
)
```

ALGORITHM: See Proposition 2.2.3 of Cremona's book 'Algorithms for Modular Elliptic Curves', or Prop 2.27 of Stein's Ph.D. thesis.

is_gamma1_equiv(*other*, *N*)

Return whether self and other are equivalent modulo the action of Gamma_1(N) via linear fractional transformations.

INPUT:

- `other` - Cusp
- `N` - an integer (specifies the group Gamma_1(N))

OUTPUT:

- `bool` - True if self and other are equivalent
- `int` - 0, 1 or -1, gives further information about the equivalence: If the two cusps are u1/v1 and u2/v2, then they are equivalent if and only if v1 = v2 (mod N) and u1 = u2 (mod gcd(v1,N)) or v1 = -v2 (mod N) and u1 = -u2 (mod gcd(v1,N)) The sign is +1 for the first and -1 for the second. If the two cusps are not equivalent then 0 is returned.

EXAMPLES:

```
sage: x = Cusp(2,3)
sage: y = Cusp(4,5)
sage: x.is_gamma1_equiv(y,2)
(True, 1)
sage: x.is_gamma1_equiv(y,3)
(False, 0)
sage: z = Cusp(QQ(x) + 10)
sage: x.is_gamma1_equiv(z,10)
(True, 1)
sage: z = Cusp(1,0)
sage: x.is_gamma1_equiv(z, 3)
(True, -1)
sage: Cusp(0).is_gamma1_equiv(oo, 1)
(True, 1)
sage: Cusp(0).is_gamma1_equiv(oo, 3)
(False, 0)
```

is_gamma_h_equiv(*other*, *G*)

Return a pair (b, t), where b is True or False as self and other are equivalent under the action of G, and t is 1 or -1, as described below.

Two cusps $u1/v1$ and $u2/v2$ are equivalent modulo Gamma_H(N) if and only if $v1 = h * v2(\mathrm{mod} N)$ and $u1 = h^{(-1)} * u2(\mathrm{mod} gcd(v1, N))$ or $v1 = -h * v2(mod N)$ and $u1 = -h^{(-1)} * u2(\mathrm{mod} gcd(v1, N))$ for some $h \in H$. Then t is 1 or -1 as c and c' fall into the first or second case, respectively.

INPUT:

- `other` - Cusp
- `G` - a congruence subgroup Gamma_H(N)

OUTPUT:

- `bool` - True if self and other are equivalent
- `int` - -1, 0, 1; extra info

EXAMPLES:

```
sage: x = Cusp(2,3)
sage: y = Cusp(4,5)
sage: x.is_gamma_h_equiv(y,GammaH(13,[2]))
(True, 1)
sage: x.is_gamma_h_equiv(y,GammaH(13,[5]))
(False, 0)
sage: x.is_gamma_h_equiv(y,GammaH(5,[]))
(False, 0)
sage: x.is_gamma_h_equiv(y,GammaH(23,[4]))
(True, -1)
```

Enumerating the cusps for a space of modular symbols uses this function.

```
sage: G = GammaH(25,[6]) ; M = G.modular_symbols() ; M
Modular Symbols space of dimension 11 for Congruence Subgroup Gamma_H(25)␣
↪with H generated by [6] of weight 2 with sign 0 and over Rational Field
sage: M.cusps()
[33/100, 1/3, 31/125, 1/4, 1/15, -7/15, 7/15, 4/15, 1/20, 3/20, 7/20, 9/20]
sage: len(M.cusps())
12
```

This is always one more than the associated space of weight 2 Eisenstein series.

```
sage: G.dimension_eis(2)
11
sage: M.cuspidal_subspace()
Modular Symbols subspace of dimension 0 of Modular Symbols space of dimension␣
↪11 for Congruence Subgroup Gamma_H(25) with H generated by [6] of weight 2␣
↪with sign 0 and over Rational Field
sage: G.dimension_cusp_forms(2)
0
```

is_infinity()

Returns True if this is the cusp infinity.

EXAMPLES:

```
sage: Cusp(3/5).is_infinity()
False
sage: Cusp(1,0).is_infinity()
True
sage: Cusp(0,1).is_infinity()
False
```

numerator()

Return the numerator of the cusp a/b.

EXAMPLES:

```
sage: x=Cusp(6,9); x
2/3
sage: x.numerator()
2
```

(continues on next page)

(continued from previous page)

```
sage: Cusp(oo).numerator()
1
sage: Cusp(-5/10).numerator()
-1
```

sage.modular.cusps.**Cusps = Set P^1(QQ) of all cusps**

class sage.modular.cusps.**Cusps_class**

Bases: sage.misc.fast_methods.Singleton, sage.structure.parent.Parent

The set of cusps.

EXAMPLES:

```
sage: C = Cusps; C
Set P^1(QQ) of all cusps
sage: loads(C.dumps()) == C
True
```

Element

alias of *Cusp*

CHAPTER

THREE

DIMENSIONS OF SPACES OF MODULAR FORMS

AUTHORS:

- William Stein
- Jordi Quer

ACKNOWLEDGEMENT: The dimension formulas and implementations in this module grew out of a program that Bruce Kaskel wrote (around 1996) in PARI, which Kevin Buzzard subsequently extended. I (William Stein) then implemented it in C++ for Hecke. I also implemented it in Magma. Also, the functions for dimensions of spaces with nontrivial character are based on a paper (that has no proofs) by Cohen and Oesterlé [CO1977]. The formulas for $\Gamma_H(N)$ were found and implemented by Jordi Quer.

The formulas here are more complete than in Hecke or Magma.

Currently the input to each function below is an integer and either a Dirichlet character ε or a finite index subgroup of $\mathrm{SL}_2(\mathbf{Z})$. If the input is a Dirichlet character ε, the dimensions are for subspaces of $M_k(\Gamma_1(N), \varepsilon)$, where N is the modulus of ε.

These functions mostly call the methods `dimension_cusp_forms`, `dimension_modular_forms` and so on of the corresponding congruence subgroup classes.

REFERENCES:

sage.modular.dims.**CO_delta**(*r*, *p*, *N*, *eps*)

This is used as an intermediate value in computations related to the paper of Cohen-Oesterlé.

INPUT:

- `r` – positive integer
- `p` – a prime
- `N` – positive integer
- `eps` – character

OUTPUT: element of the base ring of the character

EXAMPLES:

```
sage: G.<eps> = DirichletGroup(7)
sage: sage.modular.dims.CO_delta(1,5,7,eps^3)
2
```

sage.modular.dims.**CO_nu**(*r*, *p*, *N*, *eps*)

This is used as an intermediate value in computations related to the paper of Cohen-Oesterlé.

INPUT:

- `r` – positive integer

- `p` – a prime
- `N` – positive integer
- `eps` – character

OUTPUT: element of the base ring of the character

EXAMPLES:

```
sage: G.<eps> = DirichletGroup(7)
sage: G.<eps> = DirichletGroup(7)
sage: sage.modular.dims.CO_nu(1,7,7,eps)
-1
```

`sage.modular.dims.`**`CohenOesterle`**(*eps*, *k*)

Compute the Cohen-Oesterlé function associate to eps, k.

This is a summand in the formula for the dimension of the space of cusp forms of weight 2 with character ε.

INPUT:

- `eps` – Dirichlet character
- `k` – integer

OUTPUT: element of the base ring of eps.

EXAMPLES:

```
sage: G.<eps> = DirichletGroup(7)
sage: sage.modular.dims.CohenOesterle(eps, 2)
-2/3
sage: sage.modular.dims.CohenOesterle(eps, 4)
-1
```

`sage.modular.dims.`**`dimension_cusp_forms`**(*X*, *k=2*)

The dimension of the space of cusp forms for the given congruence subgroup or Dirichlet character.

INPUT:

- `X` – congruence subgroup or Dirichlet character or integer
- `k` – weight (integer)

EXAMPLES:

```
sage: dimension_cusp_forms(5,4)
1
```

```
sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma1(13),2)
2
```

```
sage: dimension_cusp_forms(DirichletGroup(13).0^2,2)
1
sage: dimension_cusp_forms(DirichletGroup(13).0,3)
1
```

```
sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma0(11),0)
0
sage: dimension_cusp_forms(Gamma0(1),12)
1
sage: dimension_cusp_forms(Gamma0(1),2)
0
sage: dimension_cusp_forms(Gamma0(1),4)
0
```

```
sage: dimension_cusp_forms(Gamma0(389),2)
32
sage: dimension_cusp_forms(Gamma0(389),4)
97
sage: dimension_cusp_forms(Gamma0(2005),2)
199
sage: dimension_cusp_forms(Gamma0(11),1)
0
```

```
sage: dimension_cusp_forms(Gamma1(11),2)
1
sage: dimension_cusp_forms(Gamma1(1),12)
1
sage: dimension_cusp_forms(Gamma1(1),2)
0
sage: dimension_cusp_forms(Gamma1(1),4)
0
```

```
sage: dimension_cusp_forms(Gamma1(389),2)
6112
sage: dimension_cusp_forms(Gamma1(389),4)
18721
sage: dimension_cusp_forms(Gamma1(2005),2)
159201
```

```
sage: dimension_cusp_forms(Gamma1(11),1)
0
```

```
sage: e = DirichletGroup(13).0
sage: e.order()
12
sage: dimension_cusp_forms(e,2)
0
sage: dimension_cusp_forms(e^2,2)
1
```

Check that trac ticket #12640 is fixed:

```
sage: dimension_cusp_forms(DirichletGroup(1)(1), 12)
1
sage: dimension_cusp_forms(DirichletGroup(2)(1), 24)
5
```

sage.modular.dims.**dimension_eis**(*X*, *k=2*)

The dimension of the space of Eisenstein series for the given congruence subgroup.

INPUT:

- X – congruence subgroup or Dirichlet character or integer
- k – weight (integer)

EXAMPLES:

```
sage: dimension_eis(5,4)
2
```

```
sage: dimension_eis(Gamma0(11),2)
1
sage: dimension_eis(Gamma1(13),2)
11
sage: dimension_eis(Gamma1(2006),2)
3711
```

```
sage: e = DirichletGroup(13).0
sage: e.order()
12
sage: dimension_eis(e,2)
0
sage: dimension_eis(e^2,2)
2
```

```
sage: e = DirichletGroup(13).0
sage: e.order()
12
sage: dimension_eis(e,2)
0
sage: dimension_eis(e^2,2)
2
sage: dimension_eis(e,13)
2
```

```
sage: G = DirichletGroup(20)
sage: dimension_eis(G.0,3)
4
sage: dimension_eis(G.1,3)
6
sage: dimension_eis(G.1^2,2)
6
```

```
sage: G = DirichletGroup(200)
sage: e = prod(G.gens(), G(1))
sage: e.conductor()
200
sage: dimension_eis(e,2)
4
```

```
sage: dimension_modular_forms(Gamma1(4), 11)
6
```

sage.modular.dims.**dimension_modular_forms**(*X*, *k=2*)

The dimension of the space of cusp forms for the given congruence subgroup (either $\Gamma_0(N)$, $\Gamma_1(N)$, or $\Gamma_H(N)$) or Dirichlet character.

INPUT:

- `X` – congruence subgroup or Dirichlet character
- `k` – weight (integer)

EXAMPLES:

```
sage: dimension_modular_forms(Gamma0(11),2)
2
sage: dimension_modular_forms(Gamma0(11),0)
1
sage: dimension_modular_forms(Gamma1(13),2)
13
sage: dimension_modular_forms(GammaH(11, [10]), 2)
10
sage: dimension_modular_forms(GammaH(11, [10]))
10
sage: dimension_modular_forms(GammaH(11, [10]), 4)
20
sage: e = DirichletGroup(20).1
sage: dimension_modular_forms(e,3)
9
sage: dimension_cusp_forms(e,3)
3
sage: dimension_eis(e,3)
6
sage: dimension_modular_forms(11,2)
2
```

`sage.modular.dims.`**`dimension_new_cusp_forms`**(*X*, *k=2*, *p=0*)

Return the dimension of the new (or p-new) subspace of cusp forms for the character or group X.

INPUT:

- `X` – integer, congruence subgroup or Dirichlet character
- `k` – weight (integer)
- `p` – 0 or a prime

EXAMPLES:

```
sage: dimension_new_cusp_forms(100,2)
1
```

```
sage: dimension_new_cusp_forms(Gamma0(100),2)
1
sage: dimension_new_cusp_forms(Gamma0(100),4)
5
```

```
sage: dimension_new_cusp_forms(Gamma1(100),2)
141
sage: dimension_new_cusp_forms(Gamma1(100),4)
463
```

```
sage: dimension_new_cusp_forms(DirichletGroup(100).1^2,2)
2
sage: dimension_new_cusp_forms(DirichletGroup(100).1^2,4)
8
```

```
sage: sum(dimension_new_cusp_forms(e,3) for e in DirichletGroup(30))
12
sage: dimension_new_cusp_forms(Gamma1(30),3)
12
```

Check that trac ticket #12640 is fixed:

```
sage: dimension_new_cusp_forms(DirichletGroup(1)(1), 12)
1
sage: dimension_new_cusp_forms(DirichletGroup(2)(1), 24)
1
```

`sage.modular.dims.`**`eisen`**`(`*p*`)`

Return the Eisenstein number n which is the numerator of $(p-1)/12$.

INPUT:

- `p` – a prime

OUTPUT: Integer

EXAMPLES:

```
sage: [(p, sage.modular.dims.eisen(p)) for p in prime_range(24)]
[(2, 1), (3, 1), (5, 1), (7, 1), (11, 5), (13, 1), (17, 4),
 (19, 3), (23, 11)]
```

`sage.modular.dims.`**`sturm_bound`**`(`*level*, *weight=2*`)`

Return the Sturm bound for modular forms with given level and weight.

For more details, see the documentation for the `sturm_bound` method of `sage.modular.arithgroup.CongruenceSubgroup` objects.

INPUT:

- **`level` – an integer (interpreted as a level for Gamma0) or a** congruence subgroup
- `weight` – an integer ≥ 2 (default: 2)

EXAMPLES:

```
sage: sturm_bound(11,2)
2
sage: sturm_bound(389,2)
65
sage: sturm_bound(1,12)
1
sage: sturm_bound(100,2)
30
sage: sturm_bound(1,36)
3
sage: sturm_bound(11)
2
```

CHAPTER

FOUR

CONJECTURAL SLOPES OF HECKE POLYNOMIALS

Interface to Kevin Buzzard's PARI program for computing conjectural slopes of characteristic polynomials of Hecke operators.

AUTHORS:

- William Stein (2006-03-05): Sage interface
- Kevin Buzzard: PARI program that implements underlying functionality

sage.modular.buzzard.**buzzard_tpslopes**(*p*, *N*, *kmax*)

Return a vector of length kmax, whose k'th entry ($0 \leq k \leq k_{max}$) is the conjectural sequence of valuations of eigenvalues of T_p on forms of level N, weight k, and trivial character.

This conjecture is due to Kevin Buzzard, and is only made assuming that p does not divide N and if p is $\Gamma_0(N)$-regular.

EXAMPLES:

```
sage: from sage.modular.buzzard import buzzard_tpslopes
sage: c = buzzard_tpslopes(2,1,50)
sage: c[50]
[4, 8, 13]
```

Hence Buzzard would conjecture that the 2-adic valuations of the eigenvalues of T_2 on cusp forms of level 1 and weight 50 are $[4, 8, 13]$, which indeed they are, as one can verify by an explicit computation using, e.g., modular symbols:

```
sage: M = ModularSymbols(1,50, sign=1).cuspidal_submodule()
sage: T = M.hecke_operator(2)
sage: f = T.charpoly('x')
sage: f.newton_slopes(2)
[13, 8, 4]
```

AUTHORS:

- Kevin Buzzard: several PARI/GP scripts
- William Stein (2006-03-17): small Sage wrapper of Buzzard's scripts

sage.modular.buzzard.**gp**()

Return a copy of the GP interpreter with the appropriate files loaded.

EXAMPLES:

```
sage: import sage.modular.buzzard
sage: sage.modular.buzzard.gp()
PARI/GP interpreter
```

CHAPTER

FIVE

LOCAL COMPONENTS OF MODULAR FORMS

If f is a (new, cuspidal, normalised) modular eigenform, then one can associate to f an *automorphic representation* π_f of the group $\mathrm{GL}_2(\mathbf{A})$ (where $\mathbf{A}$ is the adele ring of $\mathbf{Q}$). This object factors as a restricted tensor product of components $\pi_{f,v}$ for each place of $\mathbf{Q}$. These are infinite-dimensional representations, but they are specified by a finite amount of data, and this module provides functions which determine a description of the local factor $\pi_{f,p}$ at a finite prime p.

The functions in this module are based on the algorithms described in [LW2012].

AUTHORS:

- David Loeffler
- Jared Weinstein

`sage.modular.local_comp.local_comp.`**`LocalComponent`**(*f*, *p*, *twist_factor=None*)

Calculate the local component at the prime p of the automorphic representation attached to the newform f.

INPUT:

- `f` (`Newform`) a newform of weight $k \geq 2$
- `p` (integer) a prime
- `twist_factor` (integer) an integer congruent to k modulo 2 (default: $k - 2$)

Note: The argument `twist_factor` determines the choice of normalisation: if it is set to $j \in \mathbf{Z}$, then the central character of $\pi_{f,\ell}$ maps ℓ to $\ell^j \varepsilon(\ell)$ for almost all ℓ, where ε is the Nebentypus character of f.

In the analytic theory it is conventional to take $j = 0$ (the "Langlands normalisation"), so the representation π_f is unitary; however, this is inconvenient for k odd, since in this case one needs to choose a square root of p and thus the map $f \to \pi_f$ is not Galois-equivariant. Hence we use, by default, the "Hecke normalisation" given by $j = k - 2$. This is also the most natural normalisation from the perspective of modular symbols.

We also adopt a slightly unusual definition of the principal series: we define $\pi(\chi_1, \chi_2)$ to be the induction from the Borel subgroup of the character of the maximal torus $\begin{pmatrix} x & \\ & y \end{pmatrix} \mapsto \chi_1(a)\chi_2(b)|b|$, so its central character is $z \mapsto \chi_1(z)\chi_2(z)|z|$. Thus $\chi_1\chi_2$ is the restriction to $\mathbf{Q}_p^\times$ of the unique character of the id'ele class group mapping ℓ to $\ell^{k-1}\varepsilon(\ell)$ for almost all ℓ. This has the property that the *set* $\{\chi_1, \chi_2\}$ also depends Galois-equivariantly on f.

EXAMPLES:

```
sage: Pi = LocalComponent(Newform('49a'), 7); Pi
Smooth representation of GL_2(Q_7) with conductor 7^2
sage: Pi.central_character()
Character of Q_7*, of level 0, mapping 7 |--> 1
```

(continues on next page)

(continued from previous page)

```
sage: Pi.species()
'Supercuspidal'
sage: Pi.characters()
[
Character of unramified extension Q_7(s)* (s^2 + 6*s + 3 = 0), of level 1,␣
↪mapping s |--> d, 7 |--> 1,
Character of unramified extension Q_7(s)* (s^2 + 6*s + 3 = 0), of level 1,␣
↪mapping s |--> -d, 7 |--> 1
]
```

class sage.modular.local_comp.local_comp.**LocalComponentBase**(*newform*, *prime*, *twist_factor*)

Bases: sage.structure.sage_object.SageObject

Base class for local components of newforms. Not to be directly instantiated; use the *LocalComponent()* constructor function.

central_character()

Return the central character of this representation. This is the restriction to $\mathbf{Q}_p^\times$ of the unique smooth character ω of $\mathbf{A}^\times/\mathbf{Q}^\times$ such that $\omega(\varpi_\ell) = \ell^j \varepsilon(\ell)$ for all primes $\ell \nmid Np$, where ϖ_ℓ is a uniformiser at ℓ, ε is the Nebentypus character of the newform f, and j is the twist factor (see the documentation for *LocalComponent()*).

EXAMPLES:

```
sage: LocalComponent(Newform('27a'), 3).central_character()
Character of Q_3*, of level 0, mapping 3 |--> 1

sage: LocalComponent(Newforms(Gamma1(5), 5, names='c')[0], 5).central_
↪character()
Character of Q_5*, of level 1, mapping 2 |--> c0 + 1, 5 |--> 125

sage: LocalComponent(Newforms(DirichletGroup(24)([1, -1,-1]), 3, names='a
↪')[0], 2).central_character()
Character of Q_2*, of level 3, mapping 7 |--> 1, 5 |--> -1, 2 |--> -2
```

check_tempered()

Check that this representation is quasi-tempered, i.e. $\pi \otimes |\det|^{j/2}$ is tempered. It is well known that local components of modular forms are *always* tempered, so this serves as a useful check on our computations.

EXAMPLES:

```
sage: from sage.modular.local_comp.local_comp import LocalComponentBase
sage: LocalComponentBase(Newform('50a'), 3, 0).check_tempered()
Traceback (most recent call last):
...
NotImplementedError: <abstract method check_tempered at ...>
```

coefficient_field()

The field K over which this representation is defined. This is the field generated by the Hecke eigenvalues of the corresponding newform (over whatever base ring the newform is created).

EXAMPLES:

```
sage: LocalComponent(Newforms(50)[0], 3).coefficient_field()
Rational Field
sage: LocalComponent(Newforms(Gamma1(10), 3, base_ring=QQbar)[0], 5).
↪coefficient_field()
```

(continues on next page)

(continued from previous page)

```
Algebraic Field
sage: LocalComponent(Newforms(DirichletGroup(5).0, 7,names='c')[0], 5).
↪coefficient_field()
Number Field in c0 with defining polynomial x^2 + (5*zeta4 + 5)*x - 88*zeta4␣
↪over its base field
```

conductor()

The smallest r such that this representation has a nonzero vector fixed by the subgroup $\begin{pmatrix} * & * \\ 0 & 1 \end{pmatrix} \pmod{p^r}$. This is equal to the power of p dividing the level of the corresponding newform.

EXAMPLES:

```
sage: LocalComponent(Newform('50a'), 5).conductor()
2
```

newform()

The newform of which this is a local component.

EXAMPLES:

```
sage: LocalComponent(Newform('50a'), 5).newform()
q - q^2 + q^3 + q^4 + O(q^6)
```

prime()

The prime at which this is a local component.

EXAMPLES:

```
sage: LocalComponent(Newform('50a'), 5).prime()
5
```

species()

The species of this local component, which is either 'Principal Series', 'Special' or 'Supercuspidal'.

EXAMPLES:

```
sage: from sage.modular.local_comp.local_comp import LocalComponentBase
sage: LocalComponentBase(Newform('50a'), 3, 0).species()
Traceback (most recent call last):
...
NotImplementedError: <abstract method species at ...>
```

twist_factor()

The unique j such that $\begin{pmatrix} p & 0 \\ 0 & p \end{pmatrix}$ acts as multiplication by p^j times a root of unity.

There are various conventions for this; see the documentation of the *LocalComponent()* constructor function for more information.

The twist factor should have the same parity as the weight of the form, since otherwise the map sending f to its local component won't be Galois equivariant.

EXAMPLES:

```
sage: LocalComponent(Newforms(50)[0], 3).twist_factor()
0
sage: LocalComponent(Newforms(50)[0], 3, twist_factor=173).twist_factor()
173
```

class sage.modular.local_comp.local_comp.**PrimitivePrincipalSeries**(*newform*, *prime*, *twist_factor*)

Bases: *sage.modular.local_comp.local_comp.PrincipalSeries*

A ramified principal series of the form $\pi(\chi_1, \chi_2)$ where χ_1 is unramified but χ_2 is not.

EXAMPLES:

```
sage: Pi = LocalComponent(Newforms(Gamma1(13), 2, names='a')[0], 13)
sage: type(Pi)
<class 'sage.modular.local_comp.local_comp.PrimitivePrincipalSeries'>
sage: TestSuite(Pi).run()
```

characters()

Return the two characters (χ_1, χ_2) such that the local component $\pi_{f,p}$ is the induction of the character $\chi_1 \times \chi_2$ of the Borel subgroup.

EXAMPLES:

```
sage: LocalComponent(Newforms(Gamma1(13), 2, names='a')[0], 13).characters()
[
Character of Q_13*, of level 0, mapping 13 |--> 3*a0 + 2,
Character of Q_13*, of level 1, mapping 2 |--> a0 + 2, 13 |--> -3*a0 - 7
]
```

class sage.modular.local_comp.local_comp.**PrimitiveSpecial**(*newform*, *prime*, *twist_factor*)

Bases: *sage.modular.local_comp.local_comp.LocalComponentBase*

A primitive special representation: that is, the Steinberg representation twisted by an unramified character. All such representations have conductor 1.

EXAMPLES:

```
sage: Pi = LocalComponent(Newform('37a'), 37)
sage: Pi.species()
'Special'
sage: Pi.conductor()
1
sage: type(Pi)
<class 'sage.modular.local_comp.local_comp.PrimitiveSpecial'>
sage: TestSuite(Pi).run()
```

characters()

Return the defining characters of this representation. In this case, it will return the unique unramified character χ of $\mathbf{Q}_p^\times$ such that this representation is equal to $\mathrm{St} \otimes \chi$, where St is the Steinberg representation (defined as the quotient of the parabolic induction of the trivial character by its trivial subrepresentation).

EXAMPLES:

Our first example is the newform corresponding to an elliptic curve of conductor 37. This is the nontrivial quadratic twist of Steinberg, corresponding to the fact that the elliptic curve has non-split multiplicative reduction at 37:

```
sage: LocalComponent(Newform('37a'), 37).characters()
[Character of Q_37*, of level 0, mapping 37 |--> -1]
```

We try an example in odd weight, where the central character isn't trivial:

```
sage: Pi = LocalComponent(Newforms(DirichletGroup(21)([-1, 1]), 3, names='j
↪')[0], 7); Pi.characters()
[Character of Q_7*, of level 0, mapping 7 |--> -1/2*j0^2 - 7/2]
sage: Pi.characters()[0] ^2 == Pi.central_character()
True
```

An example using a non-standard twist factor:

```
sage: Pi = LocalComponent(Newforms(DirichletGroup(21)([-1, 1]), 3, names='j
↪')[0], 7, twist_factor=3); Pi.characters()
[Character of Q_7*, of level 0, mapping 7 |--> -7/2*j0^2 - 49/2]
sage: Pi.characters()[0]^2 == Pi.central_character()
True
```

check_tempered()

Check that this representation is tempered (after twisting by $|\det|^{j/2}$ where j is the twist factor). Since local components of modular forms are always tempered, this is a useful check on our calculations.

EXAMPLES:

```
sage: Pi = LocalComponent(Newforms(DirichletGroup(21)([-1, 1]), 3, names='j
↪')[0], 7)
sage: Pi.check_tempered()
```

species()

The species of this local component, which is either 'Principal Series', 'Special' or 'Supercuspidal'.

EXAMPLES:

```
sage: LocalComponent(Newform('37a'), 37).species()
'Special'
```

class sage.modular.local_comp.local_comp.**PrimitiveSupercuspidal**(*newform*, *prime*, *twist_factor*)

Bases: *sage.modular.local_comp.local_comp.LocalComponentBase*

A primitive supercuspidal representation.

Except for some exceptional cases when $p = 2$ which we do not implement here, such representations are parametrized by smooth characters of tamely ramified quadratic extensions of $\mathbf{Q}_p$.

EXAMPLES:

```
sage: f = Newform("50a")
sage: Pi = LocalComponent(f, 5)
sage: type(Pi)
<class 'sage.modular.local_comp.local_comp.PrimitiveSupercuspidal'>
sage: Pi.species()
'Supercuspidal'
sage: TestSuite(Pi).run()
```

characters()

Return the two conjugate characters of $K^\times$, where K is some quadratic extension of $\mathbf{Q}_p$, defining this representation. This is fully implemented only in the case where the power of p dividing the level of the form is even, in which case K is the unique unramified quadratic extension of $\mathbf{Q}_p$.

EXAMPLES:

The first example from [LW2012]:

```
sage: f = Newform('50a')
sage: Pi = LocalComponent(f, 5)
sage: chars = Pi.characters(); chars
[
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1,␣
↪mapping s |--> d, 5 |--> 1,
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1,␣
↪mapping s |--> -d - 1, 5 |--> 1
]
sage: chars[0].base_ring()
Number Field in d with defining polynomial x^2 + x + 1
```

These characters are interchanged by the Frobenius automorphism of $\mathbf{F}_{25}$:

```
sage: chars[0] == chars[1]**5
True
```

A more complicated example (higher weight and nontrivial central character):

```
sage: f = Newforms(GammaH(25, [6]), 3, names='j')[0]; f
q + j0*q^2 + 1/3*j0^3*q^3 - 1/3*j0^2*q^4 + O(q^6)
sage: Pi = LocalComponent(f, 5)
sage: Pi.characters()
[
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1,␣
↪mapping s |--> d, 5 |--> 5,
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1,␣
↪mapping s |--> -d - 1/3*j0^3, 5 |--> 5
]
sage: Pi.characters()[0].base_ring()
Number Field in d with defining polynomial x^2 + 1/3*j0^3*x - 1/3*j0^2 over␣
↪its base field
```

> **Warning:** The above output isn't actually the same as in Example 2 of [LW2012], due to an error in the published paper (correction pending) – the published paper has the inverses of the above characters.

A higher level example:

```
sage: f = Newform('81a', names='j'); f
q + j0*q^2 + q^4 - j0*q^5 + O(q^6)
sage: LocalComponent(f, 3).characters()  # long time (12s on sage.math, 2012)
[
Character of unramified extension Q_3(s)* (s^2 + 2*s + 2 = 0), of level 2,␣
↪mapping -2*s |--> -2*d - j0, 4 |--> 1, 3*s + 1 |--> -j0*d - 2, 3 |--> 1,
Character of unramified extension Q_3(s)* (s^2 + 2*s + 2 = 0), of level 2,␣
↪mapping -2*s |--> 2*d + j0, 4 |--> 1, 3*s + 1 |--> j0*d + 1, 3 |--> 1
]
```

In the ramified case, it's not fully implemented, and just returns a string indicating which ramified extension is being considered:

```
sage: Pi = LocalComponent(Newform('27a'), 3)
sage: Pi.characters()
'Character of Q_3(sqrt(-3))'
sage: Pi = LocalComponent(Newform('54a'), 3)
```

(continues on next page)

(continued from previous page)

```
sage: Pi.characters()
'Character of Q_3(sqrt(3))'
```

check_tempered()

Check that this representation is tempered (after twisting by $|\det|^{j/2}$ where j is the twist factor). Since local components of modular forms are always tempered, this is a useful check on our calculations.

Since the computation of the characters attached to this representation is not implemented in the odd-conductor case, a NotImplementedError will be raised for such representations.

EXAMPLES:

```
sage: LocalComponent(Newform("50a"), 5).check_tempered()
sage: LocalComponent(Newform("27a"), 3).check_tempered() # not tested
```

species()

The species of this local component, which is either 'Principal Series', 'Special' or 'Supercuspidal'.

EXAMPLES:

```
sage: LocalComponent(Newform('49a'), 7).species()
'Supercuspidal'
```

type_space()

Return a *TypeSpace* object describing the (homological) type space of this newform, which we know is dual to the type space of the local component.

EXAMPLES:

```
sage: LocalComponent(Newform('49a'), 7).type_space()
6-dimensional type space at prime 7 of form q + q^2 - q^4 + O(q^6)
```

class sage.modular.local_comp.local_comp.**PrincipalSeries**(*newform*, *prime*, *twist_factor*)

Bases: *sage.modular.local_comp.local_comp.LocalComponentBase*

A principal series representation. This is an abstract base class, not to be instantiated directly; see the subclasses *UnramifiedPrincipalSeries* and *PrimitivePrincipalSeries*.

characters()

Return the two characters (χ_1, χ_2) such this representation $\pi_{f,p}$ is equal to the principal series $\pi(\chi_1, \chi_2)$.

EXAMPLES:

```
sage: from sage.modular.local_comp.local_comp import PrincipalSeries
sage: PrincipalSeries(Newform('50a'), 3, 0).characters()
Traceback (most recent call last):
...
NotImplementedError: <abstract method characters at ...>
```

check_tempered()

Check that this representation is tempered (after twisting by $|\det|^{j/2}$), i.e. that $|\chi_1(p)| = |\chi_2(p)| = p^{(j+1)/2}$. This follows from the Ramanujan–Petersson conjecture, as proved by Deligne.

EXAMPLES:

```
sage: LocalComponent(Newform('49a'), 3).check_tempered()
```

species()

The species of this local component, which is either 'Principal Series', 'Special' or 'Supercuspidal'.

EXAMPLES:

```
sage: LocalComponent(Newform('50a'), 3).species()
'Principal Series'
```

class sage.modular.local_comp.local_comp.**UnramifiedPrincipalSeries**(*newform*, *prime*, *twist_factor*)

Bases: *sage.modular.local_comp.local_comp.PrincipalSeries*

An unramified principal series representation of $\mathrm{GL}_2(\mathbf{Q}_p)$ (corresponding to a form whose level is not divisible by p).

EXAMPLES:

```
sage: Pi = LocalComponent(Newform('50a'), 3)
sage: Pi.conductor()
0
sage: type(Pi)
<class 'sage.modular.local_comp.local_comp.UnramifiedPrincipalSeries'>
sage: TestSuite(Pi).run()
```

characters()

Return the two characters (χ_1, χ_2) such this representation $\pi_{f,p}$ is equal to the principal series $\pi(\chi_1, \chi_2)$. These are the unramified characters mapping p to the roots of the Satake polynomial, so in most cases (but not always) they will be defined over an extension of the coefficient field of self.

EXAMPLES:

```
sage: LocalComponent(Newform('11a'), 17).characters()
[
Character of Q_17*, of level 0, mapping 17 |--> d,
Character of Q_17*, of level 0, mapping 17 |--> -d - 2
]
sage: LocalComponent(Newforms(Gamma1(5), 6, names='a')[1], 3).characters()
[
Character of Q_3*, of level 0, mapping 3 |--> -3/2*a1 + 12,
Character of Q_3*, of level 0, mapping 3 |--> -3/2*a1 - 12
]
```

satake_polynomial()

Return the Satake polynomial of this representation, i.e.~the polynomial whose roots are $\chi_1(p), \chi_2(p)$ where this representation is $\pi(\chi_1, \chi_2)$. Concretely, this is the polynomial

$$X^2 - p^{(j-k+2)/2} a_p(f) X + p^{j+1} \varepsilon(p)`.$$

An error will be raised if $j \ne k \bmod 2$.

EXAMPLES:

```
sage: LocalComponent(Newform('11a'), 17).satake_polynomial()
X^2 + 2*X + 17
sage: LocalComponent(Newform('11a'), 17, twist_factor = -2).satake_
↪polynomial()
X^2 + 2/17*X + 1/17
```

CHAPTER

SIX

SMOOTH CHARACTERS OF P-ADIC FIELDS

Let F be a finite extension of $\mathbf{Q}_p$. Then we may consider the group of smooth (i.e. locally constant) group homomorphisms $F^\times \to L^\times$, for L any field. Such characters are important since they can be used to parametrise smooth representations of $\mathrm{GL}_2(\mathbf{Q}_p)$, which arise as the local components of modular forms.

This module contains classes to represent such characters when F is $\mathbf{Q}_p$ or a quadratic extension. In the latter case, we choose a quadratic extension K of $\mathbf{Q}$ whose completion at p is F, and use Sage's wrappers of the Pari `idealstar` and `ideallog` methods to work in the finite group $\mathcal{O}_K/p^c$ for $c \geq 0$.

An example with characters of $\mathbf{Q}_7$:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: K.<z> = CyclotomicField(42)
sage: G = SmoothCharacterGroupQp(7, K)
sage: G.unit_gens(2), G.exponents(2)
([3, 7], [42, 0])
```

The output of the last line means that the group $\mathbf{Q}_7^\times/(1 + 7^2\mathbf{Z}_7)$ is isomorphic to $C_{42} \times \mathbf{Z}$, with the two factors being generated by 3 and 7 respectively. We create a character by specifying the images of these generators:

```
sage: chi = G.character(2, [z^5, 11 + z]); chi
Character of Q_7*, of level 2, mapping 3 |--> z^5, 7 |--> z + 11
sage: chi(4)
z^8
sage: chi(42)
z^10 + 11*z^9
```

Characters are themselves group elements, and basic arithmetic on them works:

```
sage: chi**3
Character of Q_7*, of level 2, mapping 3 |--> z^8 - z, 7 |--> z^3 + 33*z^2 + 363*z +
↪1331
sage: chi.multiplicative_order()
+Infinity
```

class sage.modular.local_comp.smoothchar.**SmoothCharacterGeneric**(*parent*, *c*, *values_on_gens*)

Bases: `sage.structure.element.MultiplicativeGroupElement`

A smooth (i.e. locally constant) character of $F^\times$, for F some finite extension of $\mathbf{Q}_p$.

galois_conjugate()

Return the composite of this character with the order 2 automorphism of $K/\mathbf{Q}_p$ (assuming K is quadratic).

Note that this is the Galois operation on the *domain*, not on the *codomain*.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupUnramifiedQuadratic
sage: K.<w> = CyclotomicField(3)
sage: G = SmoothCharacterGroupUnramifiedQuadratic(2, K)
sage: chi = G.character(2, [w, -1,-1, 3*w])
sage: chi2 = chi.galois_conjugate(); chi2
Character of unramified extension Q_2(s)* (s^2 + s + 1 = 0), of level 2,␣
↪mapping s |--> -w - 1, 2*s + 1 |--> 1, -1 |--> -1, 2 |--> 3*w

sage: chi.restrict_to_Qp() == chi2.restrict_to_Qp()
True
sage: chi * chi2 == chi.parent().compose_with_norm(chi.restrict_to_Qp())
True
```

level()

Return the level of this character, i.e. the smallest integer $c \geq 0$ such that it is trivial on $1 + \mathfrak{p}^c$.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: SmoothCharacterGroupQp(7, QQ).character(2, [-1, 1]).level()
1
```

multiplicative_order()

Return the order of this character as an element of the character group.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: K.<z> = CyclotomicField(42)
sage: G = SmoothCharacterGroupQp(7, K)
sage: G.character(3, [z^10 - z^3, 11]).multiplicative_order()
+Infinity
sage: G.character(3, [z^10 - z^3, 1]).multiplicative_order()
42
sage: G.character(1, [z^7, z^14]).multiplicative_order()
6
sage: G.character(0, [1]).multiplicative_order()
1
```

restrict_to_Qp()

Return the restriction of this character to $\mathbf{Q}_p^\times$, embedded as a subfield of $F^\times$.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupRamifiedQuadratic
sage: SmoothCharacterGroupRamifiedQuadratic(3, 0, QQ).character(0, [2]).
↪restrict_to_Qp()
Character of Q_3*, of level 0, mapping 3 |--> 4
```

class sage.modular.local_comp.smoothchar.**SmoothCharacterGroupGeneric**(*p*, *base_ring*)

Bases: `sage.structure.parent_base.ParentWithBase`

The group of smooth (i.e. locally constant) characters of a p-adic field, with values in some ring R. This is an abstract base class and should not be instantiated directly.

Element
alias of `SmoothCharacterGeneric`

base_extend (*ring*)
Return the character group of the same field, but with values in a new coefficient ring into which the old coefficient ring coerces. An error will be raised if there is no coercion map from the old coefficient ring to the new one.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: G = SmoothCharacterGroupQp(3, QQ)
sage: G.base_extend(QQbar)
Group of smooth characters of Q_3* with values in Algebraic Field
sage: G.base_extend(Zmod(3))
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Ring of integers␣
↪modulo 3
```

change_ring (*ring*)
Return the character group of the same field, but with values in a different coefficient ring. To be implemented by all derived classes (since the generic base class can't know the parameters).

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupGeneric
sage: SmoothCharacterGroupGeneric(3, QQ).change_ring(ZZ)
Traceback (most recent call last):
...
NotImplementedError: <abstract method change_ring at ...>
```

character (*level*, *values_on_gens*)
Return the unique character of the given level whose values on the generators returned by `self.unit_gens(level)` are `values_on_gens`.

INPUT:

- `level` (integer) an integer ≥ 0
- `values_on_gens` (sequence) a sequence of elements of length equal to the length of `self.unit_gens(level)`. The values should be convertible (that is, possibly noncanonically) into the base ring of self; they should all be units, and all but the last must be roots of unity (of the orders given by `self.exponents(level)`.

Note: The character returned may have level less than `level` in general.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: K.<z> = CyclotomicField(42)
sage: G = SmoothCharacterGroupQp(7, K)
sage: G.character(2, [z^6, 8])
Character of Q_7*, of level 2, mapping 3 |--> z^6, 7 |--> 8
sage: G.character(2, [z^7, 8])
Character of Q_7*, of level 1, mapping 3 |--> z^7, 7 |--> 8
```

Non-examples:

```
sage: G.character(1, [z, 1])
Traceback (most recent call last):
...
ValueError: value on generator 3 (=z) should be a root of unity of order 6
sage: G.character(1, [1, 0])
Traceback (most recent call last):
...
ValueError: value on uniformiser 7 (=0) should be a unit
```

An example with a funky coefficient ring:

```
sage: G = SmoothCharacterGroupQp(7, Zmod(9))
sage: G.character(1, [2, 2])
Character of Q_7*, of level 1, mapping 3 |--> 2, 7 |--> 2
sage: G.character(1, [2, 3])
Traceback (most recent call last):
...
ValueError: value on uniformiser 7 (=3) should be a unit
```

compose_with_norm(*chi*)

Calculate the character of $K^\times$ given by $\chi \circ \mathrm{Norm}_{K/\mathbf{Q}_p}$. Here K should be a quadratic extension and χ a character of $\mathbf{Q}_p^\times$.

EXAMPLES:

When K is the unramified quadratic extension, the level of the new character is the same as the old:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp,
↪SmoothCharacterGroupRamifiedQuadratic,
↪SmoothCharacterGroupUnramifiedQuadratic
sage: K.<w> = CyclotomicField(6)
sage: G = SmoothCharacterGroupQp(3, K)
sage: chi = G.character(2, [w, 5])
sage: H = SmoothCharacterGroupUnramifiedQuadratic(3, K)
sage: H.compose_with_norm(chi)
Character of unramified extension Q_3(s)* (s^2 + 2*s + 2 = 0), of level 2,
↪mapping -2*s |--> -1, 4 |--> -w, 3*s + 1 |--> w - 1, 3 |--> 25
```

In ramified cases, the level of the new character may be larger:

```
sage: H = SmoothCharacterGroupRamifiedQuadratic(3, 0, K)
sage: H.compose_with_norm(chi)
Character of ramified extension Q_3(s)* (s^2 - 3 = 0), of level 3, mapping 2
↪|--> w - 1, s + 1 |--> -w, s |--> -5
```

On the other hand, since norm is not surjective, the result can even be trivial:

```
sage: chi = G.character(1, [-1, -1]); chi
Character of Q_3*, of level 1, mapping 2 |--> -1, 3 |--> -1
sage: H.compose_with_norm(chi)
Character of ramified extension Q_3(s)* (s^2 - 3 = 0), of level 0, mapping s
↪|--> 1
```

discrete_log(*level*)

Given an element $x \in F^\times$ (lying in the number field K of which F is a completion, see module docstring), express the class of x in terms of the generators of $F^\times/(1+\mathfrak{p}^c)^\times$ returned by `unit_gens()`.

This should be overridden by all derived classes. The method should first attempt to canonically coerce x into `self.number_field()`, and check that the result is not zero.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupGeneric
sage: SmoothCharacterGroupGeneric(3, QQ).discrete_log(3)
Traceback (most recent call last):
...
NotImplementedError: <abstract method discrete_log at ...>
```

exponents (*level*)

The orders $n_1, \ldots, n_d$ of the generators x_i of $F^\times/(1+\mathfrak{p}^c)^\times$ returned by `unit_gens()`.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupGeneric
sage: SmoothCharacterGroupGeneric(3, QQ).exponents(3)
Traceback (most recent call last):
...
NotImplementedError: <abstract method exponents at ...>
```

ideal (*level*)

Return the `level`-th power of the maximal ideal of the ring of integers of the p-adic field. Since we approximate by using number field arithmetic, what is actually returned is an ideal in a number field.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupGeneric
sage: SmoothCharacterGroupGeneric(3, QQ).ideal(3)
Traceback (most recent call last):
...
NotImplementedError: <abstract method ideal at ...>
```

prime ()

The residue characteristic of the underlying field.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupGeneric
sage: SmoothCharacterGroupGeneric(3, QQ).prime()
3
```

subgroup_gens (*level*)

A set of elements of $(\mathcal{O}_F/\mathfrak{p}^c)^\times$ generating the kernel of the reduction map to $(\mathcal{O}_F/\mathfrak{p}^{c-1})^\times$.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupGeneric
sage: SmoothCharacterGroupGeneric(3, QQ).subgroup_gens(3)
Traceback (most recent call last):
...
NotImplementedError: <abstract method subgroup_gens at ...>
```

unit_gens(*level*)

A list of generators $x_1, \ldots, x_d$ of the abelian group $F^\times/(1+\mathfrak{p}^c)^\times$, where c is the given level, satisfying no relations other than $x_i^{n_i} = 1$ for each i (where the integers n_i are returned by `exponents()`). We adopt the convention that the final generator x_d is a uniformiser (and $n_d = 0$).

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupGeneric
sage: SmoothCharacterGroupGeneric(3, QQ).unit_gens(3)
Traceback (most recent call last):
...
NotImplementedError: <abstract method unit_gens at ...>
```

class sage.modular.local_comp.smoothchar.**SmoothCharacterGroupQp**(*p*, *base_ring*)

Bases: *sage.modular.local_comp.smoothchar.SmoothCharacterGroupGeneric*

The group of smooth characters of $\mathbf{Q}_p^\times$, with values in some fixed base ring.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: G = SmoothCharacterGroupQp(7, QQ); G
Group of smooth characters of Q_7* with values in Rational Field
sage: TestSuite(G).run()
sage: G == loads(dumps(G))
True
```

change_ring(*ring*)

Return the group of characters of the same field but with values in a different ring. This need not have anything to do with the original base ring, and in particular there won't generally be a coercion map from self to the new group – use `base_extend()` if you want this.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: SmoothCharacterGroupQp(7, Zmod(3)).change_ring(CC)
Group of smooth characters of Q_7* with values in Complex Field with 53 bits␣
↪of precision
```

discrete_log(*level*, *x*)

Express the class of x in $\mathbf{Q}_p^\times/(1+p^c)^\times$ in terms of the generators returned by `unit_gens()`.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: G = SmoothCharacterGroupQp(7, QQ)
sage: G.discrete_log(0, 14)
[1]
sage: G.discrete_log(1, 14)
[2, 1]
sage: G.discrete_log(5, 14)
[9308, 1]
```

exponents(*level*)

Return the exponents of the generators returned by `unit_gens()`.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: SmoothCharacterGroupQp(7, QQ).exponents(3)
[294, 0]
sage: SmoothCharacterGroupQp(2, QQ).exponents(4)
[2, 4, 0]
```

ideal (*level*)

Return the level-th power of the maximal ideal. Since we approximate by using rational arithmetic, what is actually returned is an ideal of $\mathbf{Z}$.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: SmoothCharacterGroupQp(7, Zmod(3)).ideal(2)
Principal ideal (49) of Integer Ring
```

number_field ()

Return the number field used for calculations (a dense subfield of the local field of which this is the character group). In this case, this is always the rational field.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: SmoothCharacterGroupQp(7, Zmod(3)).number_field()
Rational Field
```

subgroup_gens (*level*)

Return a list of generators for the kernel of the map $(\mathbf{Z}_p/p^c)^\times \to (\mathbf{Z}_p/p^{c-1})^\times$.

INPUT:

- c (integer) an integer ≥ 1

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: G = SmoothCharacterGroupQp(7, QQ)
sage: G.subgroup_gens(1)
[3]
sage: G.subgroup_gens(2)
[8]

sage: G = SmoothCharacterGroupQp(2, QQ)
sage: G.subgroup_gens(1)
[]
sage: G.subgroup_gens(2)
[3]
sage: G.subgroup_gens(3)
[5]
```

unit_gens (*level*)

Return a set of generators $x_1, \dots, x_d$ for $\mathbf{Q}_p^\times/(1 + p^c\mathbf{Z}_p)^\times$. These must be independent in the sense that there are no relations between them other than relations of the form $x_i^{n_i} = 1$. They need not, however, be in Smith normal form.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp
sage: SmoothCharacterGroupQp(7, QQ).unit_gens(3)
```

(continues on next page)

(continued from previous page)

```
[3, 7]
sage: SmoothCharacterGroupQp(2, QQ).unit_gens(4)
[15, 5, 2]
```

class sage.modular.local_comp.smoothchar.**SmoothCharacterGroupRamifiedQuadratic**(*prime*, *flag*, *base_ring*, *names='s'*)

Bases: *sage.modular.local_comp.smoothchar.SmoothCharacterGroupGeneric*

The group of smooth characters of $K^\times$, where K is a ramified quadratic extension of $\mathbf{Q}_p$, and $p \neq 2$.

change_ring(*ring*)

Return the character group of the same field, but with values in a different coefficient ring. This need not have anything to do with the original base ring, and in particular there won't generally be a coercion map from self to the new group – use *base_extend()* if you want this.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupRamifiedQuadratic
sage: SmoothCharacterGroupRamifiedQuadratic(7, 1, Zmod(3), names='foo').change_ring(CC)
Group of smooth characters of ramified extension Q_7(foo)* (foo^2 + 7 = 0) with values in Complex Field with 53 bits of precision
```

discrete_log(*level*, *x*)

Solve the discrete log problem in the unit group.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupRamifiedQuadratic
sage: G = SmoothCharacterGroupRamifiedQuadratic(3, 1, QQ)
sage: s = G.number_field().gen()
sage: G.discrete_log(4, 3 + 2*s)
[5, 1, 1, 1]
sage: gs = G.unit_gens(4); gs[0]^5 * gs[1] * gs[2] * gs[3] - (3 + 2*s) in G.ideal(4)
True
```

exponents(*c*)

Return the orders of the independent generators of the unit group returned by *unit_gens()*.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupRamifiedQuadratic
sage: G = SmoothCharacterGroupRamifiedQuadratic(5, 0, QQ)
sage: G.exponents(0)
(0,)
sage: G.exponents(1)
(4, 0)
sage: G.exponents(8)
(500, 625, 0)
```

ideal(*c*)

Return the ideal p^c of `self.number_field()`. The result is cached, since we use the methods idealstar() and ideallog() which cache a Pari bid structure.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupRamifiedQuadratic
sage: G = SmoothCharacterGroupRamifiedQuadratic(5, 1, QQ, 'a'); I = G.
↪ideal(3); I
Fractional ideal (25, 5*a)
sage: I is G.ideal(3)
True
```

number_field()

Return a number field of which this is the completion at p.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupRamifiedQuadratic
sage: SmoothCharacterGroupRamifiedQuadratic(7, 0, QQ, 'a').number_field()
Number Field in a with defining polynomial x^2 - 7
sage: SmoothCharacterGroupRamifiedQuadratic(5, 1, QQ, 'b').number_field()
Number Field in b with defining polynomial x^2 - 10
sage: SmoothCharacterGroupRamifiedQuadratic(7, 1, Zmod(6), 'c').number_field()
Number Field in c with defining polynomial x^2 + 7
```

subgroup_gens(*level*)

A set of elements of $(\mathcal{O}_F/\mathfrak{p}^c)^\times$ generating the kernel of the reduction map to $(\mathcal{O}_F/\mathfrak{p}^{c-1})^\times$.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupRamifiedQuadratic
sage: G = SmoothCharacterGroupRamifiedQuadratic(3, 1, QQ)
sage: G.subgroup_gens(2)
[s + 1]
```

unit_gens(*c*)

A list of generators $x_1, \dots, x_d$ of the abelian group $F^\times/(1+\mathfrak{p}^c)^\times$, where c is the given level, satisfying no relations other than $x_i^{n_i} = 1$ for each i (where the integers n_i are returned by *exponents()*). We adopt the convention that the final generator x_d is a uniformiser.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupRamifiedQuadratic
sage: G = SmoothCharacterGroupRamifiedQuadratic(5, 0, QQ)
sage: G.unit_gens(0)
[s]
sage: G.unit_gens(1)
[2, s]
sage: G.unit_gens(8)
[2, s + 1, s]
```

class sage.modular.local_comp.smoothchar.**SmoothCharacterGroupUnramifiedQuadratic**(*prime*, *base_ring*, *names='s'*)

Bases: *sage.modular.local_comp.smoothchar.SmoothCharacterGroupGeneric*

The group of smooth characters of $\mathbf{Q}_{p^2}^\times$, where $\mathbf{Q}_{p^2}$ is the unique unramified quadratic extension of $\mathbf{Q}_p$. We represent $\mathbf{Q}_{p^2}^\times$ internally as the completion at the prime above p of a quadratic number field, defined by (the obvious lift to $\mathbf{Z}$ of) the Conway polynomial modulo p of degree 2.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupUnramifiedQuadratic
sage: G = SmoothCharacterGroupUnramifiedQuadratic(3, QQ); G
Group of smooth characters of unramified extension Q_3(s)* (s^2 + 2*s + 2 = 0) with values in Rational Field
sage: G.unit_gens(3)
[-11*s, 4, 3*s + 1, 3]
sage: TestSuite(G).run()
sage: TestSuite(SmoothCharacterGroupUnramifiedQuadratic(2, QQ)).run()
```

change_ring(*ring*)

Return the character group of the same field, but with values in a different coefficient ring. This need not have anything to do with the original base ring, and in particular there won't generally be a coercion map from self to the new group – use `base_extend()` if you want this.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupUnramifiedQuadratic
sage: SmoothCharacterGroupUnramifiedQuadratic(7, Zmod(3), names='foo').change_ring(CC)
Group of smooth characters of unramified extension Q_7(foo)* (foo^2 + 6*foo + 3 = 0) with values in Complex Field with 53 bits of precision
```

discrete_log(*level, x*)

Express the class of x in $F^\times/(1+\mathfrak{p}^c)^\times$ in terms of the generators returned by `self.unit_gens(level)`.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupUnramifiedQuadratic
sage: G = SmoothCharacterGroupUnramifiedQuadratic(2, QQ)
sage: G.discrete_log(0, 12)
[2]
sage: G.discrete_log(1, 12)
[0, 2]
sage: v = G.discrete_log(5, 12); v
[0, 2, 0, 1, 2]
sage: g = G.unit_gens(5); prod([g[i]**v[i] for i in [0..4]])/12 - 1 in G.ideal(5)
True
sage: G.discrete_log(3,G.number_field()([1,1]))
[2, 0, 0, 1, 0]
sage: H = SmoothCharacterGroupUnramifiedQuadratic(5, QQ)
sage: x = H.number_field()([1,1]); x
s + 1
sage: v = H.discrete_log(5, x); v
[22, 263, 379, 0]
sage: h = H.unit_gens(5); prod([h[i]**v[i] for i in [0..3]])/x - 1 in H.ideal(5)
True
```

exponents(c)

The orders $n_1, \ldots, n_d$ of the generators x_i of $F^\times/(1+\mathfrak{p}^c)^\times$ returned by `unit_gens()`.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupUnramifiedQuadratic
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ).exponents(2)
[48, 7, 7, 0]
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ).exponents(3)
[3, 4, 2, 2, 0]
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ).exponents(2)
[3, 2, 2, 0]
```

extend_character(*level*, *chi*, *x*, *check=True*)

Return the unique character of $F^\times$ which coincides with χ on $\mathbf{Q}_p^\times$ and maps the generator α returned by `quotient_gen()` to x.

INPUT:

- `chi`: a smooth character of $\mathbf{Q}_p$, where p is the residue characteristic of F, with values in the base ring of self (or some other ring coercible to it)
- `level`: the level of the new character (which should be at least the level of `chi`)
- `x`: an element of the base ring of self (or some other ring coercible to it).

A `ValueError` will be raised if $x^t \neq \chi(\alpha^t)$, where t is the smallest integer such that α^t is congruent modulo p^{level} to an element of $\mathbf{Q}_p$.

EXAMPLES:

We extend an unramified character of $\mathbf{Q}_3^\times$ to the unramified quadratic extension in various ways.

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupQp, SmoothCharacterGroupUnramifiedQuadratic
sage: chi = SmoothCharacterGroupQp(5, QQ).character(0, [7]); chi
Character of Q_5*, of level 0, mapping 5 |--> 7
sage: G = SmoothCharacterGroupUnramifiedQuadratic(5, QQ)
sage: G.extend_character(1, chi, -1)
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1, mapping s |--> -1, 5 |--> 7
sage: G.extend_character(2, chi, -1)
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1, mapping s |--> -1, 5 |--> 7
sage: G.extend_character(3, chi, 1)
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 0, mapping 5 |--> 7
sage: K.<z> = CyclotomicField(6); G.base_extend(K).extend_character(1, chi, z)
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1, mapping s |--> z, 5 |--> 7
```

We extend the nontrivial quadratic character:

```
sage: chi = SmoothCharacterGroupQp(5, QQ).character(1, [-1, 7])
sage: K.<z> = CyclotomicField(24); G.base_extend(K).extend_character(1, chi, z^6)
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 1, mapping s |--> z^6, 5 |--> 7
```

Extensions of higher level:

```
sage: K.<z> = CyclotomicField(20); rho = G.base_extend(K).extend_character(2,␣
↪chi, z); rho
Character of unramified extension Q_5(s)* (s^2 + 4*s + 2 = 0), of level 2,␣
↪mapping 11*s - 10 |--> z^5, 6 |--> 1, 5*s + 1 |--> -z^6, 5 |--> 7
sage: rho(3)
-1
```

Examples where it doesn't work:

```
sage: G.extend_character(1, chi, 1)
Traceback (most recent call last):
...
ValueError: Value at s must satisfy x^6 = chi(2) = -1, but it does not

sage: G = SmoothCharacterGroupQp(2, QQ); H =␣
↪SmoothCharacterGroupUnramifiedQuadratic(2, QQ)
sage: chi = G.character(3, [1, -1, 7])
sage: H.extend_character(2, chi, -1)
Traceback (most recent call last):
...
ValueError: Level of extended character cannot be smaller than level of␣
↪character of Qp
```

ideal(*c*)

Return the ideal p^c of `self.number_field()`. The result is cached, since we use the methods `idealstar()` and `ideallog()` which cache a Pari `bid` structure.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupUnramifiedQuadratic
sage: G = SmoothCharacterGroupUnramifiedQuadratic(7, QQ, 'a'); I = G.ideal(3);
↪ I
Fractional ideal (343)
sage: I is G.ideal(3)
True
```

number_field()

Return a number field of which this is the completion at p, defined by a polynomial whose discriminant is not divisible by p.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import␣
↪SmoothCharacterGroupUnramifiedQuadratic
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ, 'a').number_field()
Number Field in a with defining polynomial x^2 + 6*x + 3
sage: SmoothCharacterGroupUnramifiedQuadratic(5, QQ, 'b').number_field()
Number Field in b with defining polynomial x^2 + 4*x + 2
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ, 'c').number_field()
Number Field in c with defining polynomial x^2 + x + 1
```

quotient_gen(*level*)

Find an element generating the quotient

$$\mathcal{O}_F^\times / \mathbf{Z}_p^\times \cdot (1 + p^c \mathcal{O}_F),$$

where c is the given level.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupUnramifiedQuadratic
sage: G = SmoothCharacterGroupUnramifiedQuadratic(7,QQ)
sage: G.quotient_gen(1)
s
sage: G.quotient_gen(2)
-20*s - 21
sage: G.quotient_gen(3)
-69*s - 70
```

For $p = 2$ an error will be raised for level ≥ 3, as the quotient is not cyclic:

```
sage: G = SmoothCharacterGroupUnramifiedQuadratic(2,QQ)
sage: G.quotient_gen(1)
s
sage: G.quotient_gen(2)
-s + 2
sage: G.quotient_gen(3)
Traceback (most recent call last):
...
ValueError: Quotient group not cyclic
```

subgroup_gens (*level*)

A set of elements of $(\mathcal{O}_F/\mathfrak{p}^c)^\times$ generating the kernel of the reduction map to $(\mathcal{O}_F/\mathfrak{p}^{c-1})^\times$.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupUnramifiedQuadratic
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ).subgroup_gens(1)
[s]
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ).subgroup_gens(2)
[8, 7*s + 1]
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ).subgroup_gens(2)
[3, 2*s + 1]
```

unit_gens (*c*)

A list of generators $x_1, \dots, x_d$ of the abelian group $F^\times/(1 + \mathfrak{p}^c)^\times$, where c is the given level, satisfying no relations other than $x_i^{n_i} = 1$ for each i (where the integers n_i are returned by `exponents()`). We adopt the convention that the final generator x_d is a uniformiser (and $n_d = 0$).

ALGORITHM: Use Teichmueller lifts.

EXAMPLES:

```
sage: from sage.modular.local_comp.smoothchar import SmoothCharacterGroupUnramifiedQuadratic
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ).unit_gens(0)
[7]
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ).unit_gens(1)
[s, 7]
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ).unit_gens(2)
[22*s, 8, 7*s + 1, 7]
sage: SmoothCharacterGroupUnramifiedQuadratic(7, QQ).unit_gens(3)
[169*s + 49, 8, 7*s + 1, 7]
```

In the 2-adic case there can be more than 4 generators:

```
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ).unit_gens(0)
[2]
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ).unit_gens(1)
[s, 2]
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ).unit_gens(2)
[s, 2*s + 1, -1, 2]
sage: SmoothCharacterGroupUnramifiedQuadratic(2, QQ).unit_gens(3)
[s, 2*s + 1, 4*s + 1, -1, 2]
```

CHAPTER SEVEN

TYPE SPACES OF NEWFORMS

Let f be a new modular eigenform of level $\Gamma_1(N)$, and p a prime dividing N, with $N = Mp^r$ (M coprime to p). Suppose the power of p dividing the conductor of the character of f is p^c (so $c \le r$).

Then there is an integer u, which is $\min([r/2], r - c)$, such that any twist of f by a character mod p^u also has level N. The *type space* of f is the span of the modular eigensymbols corresponding to all of these twists, which lie in a space of modular symbols for a suitable Γ_H subgroup. This space is the key to computing the isomorphism class of the local component of the newform at p.

class `sage.modular.local_comp.type_space.`**TypeSpace**(*f*, *p*, *base_extend=True*)

Bases: `sage.structure.sage_object.SageObject`

The modular symbol type space associated to a newform, at a prime dividing the level.

character_conductor()

Exponent of p dividing the conductor of the character of the form.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().character_conductor()
0
```

conductor()

Exponent of p dividing the level of the form.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().conductor()
2
```

eigensymbol_subspace()

Return the subspace of self corresponding to the plus eigensymbols of f and its Galois conjugates (as a subspace of the vector space returned by *free_module()*).

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: T = example_type_space(); T.eigensymbol_subspace()
Vector space of degree 6 and dimension 1 over Number Field in a1 with␣
↪defining polynomial ...
Basis matrix:
[...]
sage: T.eigensymbol_subspace().is_submodule(T.free_module())
True
```

form()

The newform of which this is the type space.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().form()
q + ... + O(q^6)
```

free_module()

Return the underlying vector space of this type space.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().free_module()
Vector space of dimension 6 over Number Field in a1 with defining polynomial .
↪..
```

group()

Return a Γ_H group which is the level of all of the relevant twists of f.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().group()
Congruence Subgroup Gamma_H(98) with H generated by [43]
```

is_minimal()

Return True if there exists a newform g of level strictly smaller than N, and a Dirichlet character χ of p-power conductor, such that $f = g \otimes \chi$ where f is the form of which this is the type space. To find such a form, use *minimal_twist()*.

The result is cached.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().is_minimal()
True
sage: example_type_space(1).is_minimal()
False
```

minimal_twist()

Return a newform (not necessarily unique) which is a twist of the original form f by a Dirichlet character of p-power conductor, and which has minimal level among such twists of f.

An error will be raised if f is already minimal.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import TypeSpace, example_type_
↪space
sage: T = example_type_space(1)
sage: T.form().q_expansion(12)
q - q^2 + 2*q^3 + q^4 - 2*q^6 - q^8 + q^9 + O(q^12)
sage: g = T.minimal_twist()
sage: g.q_expansion(12)
q - q^2 - 2*q^3 + q^4 + 2*q^6 + q^7 - q^8 + q^9 + O(q^12)
sage: g.level()
```

(continues on next page)

(continued from previous page)

```
14
sage: TypeSpace(g, 7).is_minimal()
True
```

Test that trac ticket #13158 is fixed:

```
sage: f = Newforms(256,names='a')[0]
sage: T = TypeSpace(f,2)
sage: g = T.minimal_twist(); g
q - a*q^3 + O(q^6)
sage: g.level()
64
```

prime()

Return the prime p.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().prime()
7
```

rho(g)

Calculate the action of the group element g on the type space.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: T = example_type_space(2)
sage: m = T.rho([2,0,0,1]); m
[-1  1  0 -1]
[ 0  0 -1  1]
[ 0 -1 -1  1]
[ 1 -1 -2  2]
sage: v = T.eigensymbol_subspace().basis()[0]
sage: m * v == v
True
```

We test that it is a left action:

```
sage: T = example_type_space(0)
sage: a = [0,5,4,3]; b = [0,2,3,5]; ab = [1,4,2,2]
sage: T.rho(ab) == T.rho(a) * T.rho(b)
True
```

An odd level example:

```
sage: from sage.modular.local_comp.type_space import TypeSpace
sage: T = TypeSpace(Newform('54a'), 3)
sage: a = [0,1,3,0]; b = [2,1,0,1]; ab = [0,1,6,3]
sage: T.rho(ab) == T.rho(a) * T.rho(b)
True
```

tame_level()

The level away from p.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().tame_level()
2
```

u()

Largest integer u such that level of f_χ = level of f for all Dirichlet characters χ modulo p^u.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space().u()
1
sage: from sage.modular.local_comp.type_space import TypeSpace
sage: f = Newforms(Gamma1(5), 5, names='a')[0]
sage: TypeSpace(f, 5).u()
0
```

sage.modular.local_comp.type_space.**example_type_space**(*example_no=0*)

Quickly return an example of a type space. Used mainly to speed up doctesting.

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import example_type_space
sage: example_type_space()  # takes a while but caches stuff (21s on sage.math,␣
↪2012)
6-dimensional type space at prime 7 of form q + ... + O(q^6)
```

The above test takes a long time, but it precomputes and caches various things such that subsequent doctests can be very quick. So we don't want to mark it `# long time`.

sage.modular.local_comp.type_space.**find_in_space**(*f*, *A*, *base_extend=False*)

Given a Newform object f, and a space A of modular symbols of the same weight and level, find the subspace of A which corresponds to the Hecke eigenvalues of f.

If `base_extend = True`, this will return a 2-dimensional space generated by the plus and minus eigensymbols of f. If `base_extend = False` it will return a larger space spanned by the eigensymbols of f and its Galois conjugates.

(NB: "Galois conjugates" needs to be interpreted carefully – see the last example below.)

A should be an ambient space (because non-ambient spaces don't implement `base_extend`).

EXAMPLES:

```
sage: from sage.modular.local_comp.type_space import find_in_space
```

Easy case (f has rational coefficients):

```
sage: f = Newform('99a'); f
q - q^2 - q^4 - 4*q^5 + O(q^6)
sage: A = ModularSymbols(GammaH(99, [13]))
sage: find_in_space(f, A)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 25␣
↪for Congruence Subgroup Gamma_H(99) with H generated by [13] of weight 2 with␣
↪sign 0 and over Rational Field
```

Harder case:

```
sage: f = Newforms(23, names='a')[0]
sage: A = ModularSymbols(Gamma1(23))
sage: find_in_space(f, A, base_extend=True)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 45␣
↪for Gamma_1(23) of weight 2 with sign 0 and over Number Field in a0 with␣
↪defining polynomial x^2 + x - 1
sage: find_in_space(f, A, base_extend=False)
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 45␣
↪for Gamma_1(23) of weight 2 with sign 0 and over Rational Field
```

An example with character, indicating the rather subtle behaviour of `base_extend`:

```
sage: chi = DirichletGroup(5).0
sage: f = Newforms(chi, 7, names='c')[0]; f  # long time (4s on sage.math, 2012)
q + c0*q^2 + (zeta4*c0 - 5*zeta4 + 5)*q^3 + ((-5*zeta4 - 5)*c0 + 24*zeta4)*q^4 +␣
↪((10*zeta4 - 5)*c0 - 40*zeta4 - 55)*q^5 + O(q^6)
sage: find_in_space(f, ModularSymbols(Gamma1(5), 7), base_extend=True)  # long␣
↪time
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 12␣
↪for Gamma_1(5) of weight 7 with sign 0 and over Number Field in c0 with␣
↪defining polynomial x^2 + (5*zeta4 + 5)*x - 88*zeta4 over its base field
sage: find_in_space(f, ModularSymbols(Gamma1(5), 7), base_extend=False)  # long␣
↪time (27s on sage.math, 2012)
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 12␣
↪for Gamma_1(5) of weight 7 with sign 0 and over Cyclotomic Field of order 4 and␣
↪degree 2
```

Note that the base ring in the second example is $\mathbf{Q}(\zeta_4)$ (the base ring of the character of f), *not* $\mathbf{Q}$.

CHAPTER

EIGHT

HELPER FUNCTIONS FOR LOCAL COMPONENTS

This module contains various functions relating to lifting elements of $\mathrm{SL}_2(\mathbf{Z}/N\mathbf{Z})$ to $\mathrm{SL}_2(\mathbf{Z})$, and other related problems.

`sage.modular.local_comp.liftings.`**`lift_for_SL`**`(A, N=None)`

Lift a matrix A from $SL_m(\mathbf{Z}/N\mathbf{Z})$ to $SL_m(\mathbf{Z})$.

This follows [Shi1971], Lemma 1.38, p. 21.

INPUT:

- `A` – a square matrix with coefficients in $\mathbf{Z}/N\mathbf{Z}$ (or $\mathbf{Z}$)
- `N` – the modulus (optional) required only if the matrix `A` has coefficients in $\mathbf{Z}$

EXAMPLES:

```
sage: from sage.modular.local_comp.liftings import lift_for_SL
sage: A = matrix(Zmod(11), 4, 4, [6, 0, 0, 9, 1, 6, 9, 4, 4, 4, 8, 0, 4, 0, 0, 8])
sage: A.det()
1
sage: L = lift_for_SL(A)
sage: L.det()
1
sage: (L - A) == 0
True

sage: B = matrix(Zmod(19), 4, 4, [1, 6, 10, 4, 4, 14, 15, 4, 13, 0, 1, 15, 15, 15,
↪ 17, 10])
sage: B.det()
1
sage: L = lift_for_SL(B)
sage: L.det()
1
sage: (L - B) == 0
True
```

`sage.modular.local_comp.liftings.`**`lift_gen_to_gamma1`**`(m, n)`

Return four integers defining a matrix in $\mathrm{SL}_2(\mathbf{Z})$ which is congruent to $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \pmod m$ and lies in the subgroup $\begin{pmatrix} 1 & * \\ 0 & 1 \end{pmatrix} \pmod n$.

This is a special case of *`lift_to_gamma1()`*, and is coded as such.

INPUT:

- m, n – coprime positive integers

EXAMPLES:

```
sage: from sage.modular.local_comp.liftings import lift_gen_to_gamma1
sage: A = matrix(ZZ, 2, lift_gen_to_gamma1(9, 8)); A
[441  62]
[ 64   9]
sage: A.change_ring(Zmod(9))
[0 8]
[1 0]
sage: A.change_ring(Zmod(8))
[1 6]
[0 1]
sage: type(lift_gen_to_gamma1(9, 8)[0])
<type 'sage.rings.integer.Integer'>
```

`sage.modular.local_comp.liftings.`**`lift_matrix_to_sl2z`**(A, N)

Given a list of length 4 representing a 2x2 matrix over $\mathbf{Z}/N\mathbf{Z}$ with determinant 1 (mod N), lift it to a 2x2 matrix over $\mathbf{Z}$ with determinant 1.

This is a special case of *lift_to_gamma1()*, and is coded as such.

INPUT:

- `A` – list of 4 integers defining a 2×2 matrix
- N – positive integer

EXAMPLES:

```
sage: from sage.modular.local_comp.liftings import lift_matrix_to_sl2z
sage: lift_matrix_to_sl2z([10, 11, 3, 11], 19)
[29, 106, 3, 11]
sage: type(_[0])
<type 'sage.rings.integer.Integer'>
sage: lift_matrix_to_sl2z([2,0,0,1], 5)
Traceback (most recent call last):
...
ValueError: Determinant is 2 mod 5, should be 1
```

`sage.modular.local_comp.liftings.`**`lift_ramified`**(g, p, u, n)

Given four integers a, b, c, d with $p \mid c$ and $ad - bc = 1 \pmod{p^u}$, find a', b', c', d' congruent to $a, b, c, d \pmod{p^u}$, with $c' = c \pmod{p^{u+1}}$, such that $a'd' - b'c'$ is exactly 1, and $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is in $\Gamma_1(n)$.

Algorithm: Uses *lift_to_gamma1()* to get a lifting modulo p^u, and then adds an appropriate multiple of the top row to the bottom row in order to get the bottom-left entry correct modulo p^{u+1}.

EXAMPLES:

```
sage: from sage.modular.local_comp.liftings import lift_ramified
sage: lift_ramified([2,2,3,2], 3, 1, 1)
[5, 8, 3, 5]
sage: lift_ramified([8,2,12,2], 3, 2, 23)
[323, 110, -133584, -45493]
sage: type(lift_ramified([8,2,12,2], 3, 2, 23)[0])
<type 'sage.rings.integer.Integer'>
```

`sage.modular.local_comp.liftings.`**`lift_to_gamma1`**(g, m, n)

If `g = [a,b,c,d]` is a list of integers defining a 2×2 matrix whose determinant is $1 \pmod{m}$, return a list of integers giving the entries of a matrix which is congruent to $g \pmod{m}$ and to $\begin{pmatrix} 1 & * \\ 0 & 1 \end{pmatrix} \pmod{n}$. Here m and n must be coprime.

INPUT:

- g – list of 4 integers defining a 2×2 matrix
- m, n – coprime positive integers

Here m and n should be coprime positive integers. Either of m and n can be 1. If $n = 1$, this still makes perfect sense; this is what is called by the function `lift_matrix_to_sl2z()`. If $m = 1$ this is a rather silly question, so we adopt the convention of always returning the identity matrix.

The result is always a list of Sage integers (unlike `lift_to_sl2z`, which tends to return Python ints).

EXAMPLES:

```
sage: from sage.modular.local_comp.liftings import lift_to_gamma1
sage: A = matrix(ZZ, 2, lift_to_gamma1([10, 11, 3, 11], 19, 5)); A
[371  68]
[ 60  11]
sage: A.det() == 1
True
sage: A.change_ring(Zmod(19))
[10 11]
[ 3 11]
sage: A.change_ring(Zmod(5))
[1 3]
[0 1]
sage: m = list(SL2Z.random_element())
sage: n = lift_to_gamma1(m, 11, 17)
sage: assert matrix(Zmod(11), 2, n) == matrix(Zmod(11),2,m)
sage: assert matrix(Zmod(17), 2, [n[0], 0, n[2], n[3]]) == 1
sage: type(lift_to_gamma1([10,11,3,11],19,5)[0])
<type 'sage.rings.integer.Integer'>
```

Tests with $m = 1$ and with $n = 1$:

```
sage: lift_to_gamma1([1,1,0,1], 5, 1)
[1, 1, 0, 1]
sage: lift_to_gamma1([2,3,11,22], 1, 5)
[1, 0, 0, 1]
```

`sage.modular.local_comp.liftings.`**`lift_uniformiser_odd`**`(p, u, n)`

Construct a matrix over $\mathbf{Z}$ whose determinant is p, and which is congruent to $\begin{pmatrix} 0 & -1 \\ p & 0 \end{pmatrix} \pmod{p^u}$ and to $\begin{pmatrix} p & 0 \\ 0 & 1 \end{pmatrix} \pmod{n}$.

This is required for the local components machinery in the "ramified" case (when the exponent of p dividing the level is odd).

EXAMPLES:

```
sage: from sage.modular.local_comp.liftings import lift_uniformiser_odd
sage: lift_uniformiser_odd(3, 2, 11)
[432, 377, 165, 144]
sage: type(lift_uniformiser_odd(3, 2, 11)[0])
<type 'sage.rings.integer.Integer'>
```

CHAPTER NINE

ETA-PRODUCTS ON MODULAR CURVES $X_0(N)$

This package provides a class for representing eta-products, which are meromorphic functions on modular curves of the form

$$\prod_{d|N} \eta(q^d)^{r_d}$$

where $\eta(q)$ is Dirichlet's eta function $q^{1/24} \prod_{n=1}^{\infty}(1-q^n)$. These are useful for obtaining explicit models of modular curves.

See trac ticket #3934 for background.

AUTHOR:

- David Loeffler (2008-08-22): initial version

`sage.modular.etaproducts.AllCusps(N)`

Return a list of CuspFamily objects corresponding to the cusps of $X_0(N)$.

INPUT:

- `N` - (integer): the level

EXAMPLES:

```
sage: AllCusps(18)
[(Inf), (c_{2}), (c_{3,1}), (c_{3,2}), (c_{6,1}), (c_{6,2}), (c_{9}), (0)]
sage: AllCusps(0)
Traceback (most recent call last):
...
ValueError: N must be positive
```

class `sage.modular.etaproducts.CuspFamily(N, width, label=None)`

Bases: `sage.structure.sage_object.SageObject`

A family of elliptic curves parametrising a region of $X_0(N)$.

level()

The level of this cusp.

EXAMPLES:

```
sage: e = CuspFamily(10, 1)
sage: e.level()
10
```

sage_cusp()

Return the corresponding element of $\mathbb{P}^1(\mathbf{Q})$.

EXAMPLES:

```
sage: CuspFamily(10, 1).sage_cusp() # not implemented
Infinity
```

width()

The width of this cusp.

EXAMPLES:

```
sage: e = CuspFamily(10, 1)
sage: e.width()
1
```

sage.modular.etaproducts.**EtaGroup**(*level*)

Create the group of eta products of the given level.

EXAMPLES:

```
sage: EtaGroup(12)
Group of eta products on X_0(12)
sage: EtaGroup(1/2)
Traceback (most recent call last):
...
TypeError: Level (=1/2) must be a positive integer
sage: EtaGroup(0)
Traceback (most recent call last):
...
ValueError: Level (=0) must be a positive integer
```

class sage.modular.etaproducts.**EtaGroupElement**(*parent*, *rdict*)

Bases: sage.structure.element.MultiplicativeGroupElement

Create an eta product object. Usually called implicitly via EtaGroup_class.__call__ or the EtaProduct factory function.

EXAMPLES:

```
sage: EtaGroupElement(EtaGroup(8), {1:24, 2:-24})
Eta product of level 8 : (eta_1)^24 (eta_2)^-24
sage: g = _; g == loads(dumps(g))
True
```

degree()

Return the degree of self as a map $X_0(N) \to \mathbb{P}^1$, which is equal to the sum of all the positive coefficients in the divisor of self.

EXAMPLES:

```
sage: e = EtaProduct(12, {1:-336, 2:576, 3:696, 4:-216, 6:-576, 12:-144})
sage: e.degree()
230
```

divisor()

Return the divisor of `self`, as a formal sum of CuspFamily objects.

EXAMPLES:

```
sage: e = EtaProduct(12, {1:-336, 2:576, 3:696, 4:-216, 6:-576, 12:-144})
sage: e.divisor() # FormalSum seems to print things in a random order?
-131*(Inf) - 50*(c_{2}) + 11*(0) + 50*(c_{6}) + 169*(c_{4}) - 49*(c_{3})
```

(continues on next page)

(continued from previous page)

```
sage: e = EtaProduct(2^8, {8:1,32:-1})
sage: e.divisor() # random
-(c_{2}) - (Inf) - (c_{8,2}) - (c_{8,3}) - (c_{8,4}) - (c_{4,2}) - (c_{8,1}) -
↪ (c_{4,1}) + (c_{32,4}) + (c_{32,3}) + (c_{64,1}) + (0) + (c_{32,2}) + (c_
↪{64,2}) + (c_{128}) + (c_{32,1})
```

level()

Return the level of this eta product.

EXAMPLES:

```
sage: e = EtaProduct(3, {3:12, 1:-12})
sage: e.level()
3
sage: EtaProduct(12, {6:6, 2:-6}).level() # not the lcm of the d's
12
sage: EtaProduct(36, {6:6, 2:-6}).level() # not minimal
36
```

order_at_cusp(*cusp*)

Return the order of vanishing of `self` at the given cusp.

INPUT:

- `cusp` – a CuspFamily object

OUTPUT:

- an integer

EXAMPLES:

```
sage: e = EtaProduct(2, {2:24, 1:-24})
sage: e.order_at_cusp(CuspFamily(2, 1)) # cusp at infinity
1
sage: e.order_at_cusp(CuspFamily(2, 2)) # cusp 0
-1
```

q_expansion(*n*)

The q-expansion of self at the cusp at infinity.

INPUT:

- `n` (integer): number of terms to calculate

OUTPUT:

- a power series over $\mathbf{Z}$ in the variable q, with a *relative* precision of $1 + O(q^n)$.

ALGORITHM: Calculates eta to (n/m) terms, where m is the smallest integer dividing self.level() such that self.r(m) != 0. Then multiplies.

EXAMPLES:

```
sage: EtaProduct(36, {6:6, 2:-6}).q_expansion(10)
q + 6*q^3 + 27*q^5 + 92*q^7 + 279*q^9 + O(q^11)
sage: R.<q> = ZZ[[]]
sage: EtaProduct(2,{2:24,1:-24}).q_expansion(100) == delta_qexp(101)(q^2)/
↪delta_qexp(101)(q)
True
```

qexp (*n*)
Alias for `self.q_expansion()`.

EXAMPLES:

```
sage: e = EtaProduct(36, {6:8, 3:-8})
sage: e.qexp(10)
q + 8*q^4 + 36*q^7 + O(q^10)
sage: e.qexp(30) == e.q_expansion(30)
True
```

r (*d*)
Return the exponent r_d of $\eta(q^d)$ in self.

EXAMPLES:

```
sage: e = EtaProduct(12, {2:24, 3:-24})
sage: e.r(3)
-24
sage: e.r(4)
0
```

class `sage.modular.etaproducts.`**EtaGroup_class** (*level*)
Bases: `sage.groups.old.AbelianGroup`

The group of eta products of a given level under multiplication.

basis (*reduce=True*)
Produce a basis for the free abelian group of eta-products of level N (under multiplication), attempting to find basis vectors of the smallest possible degree.

INPUT:

- `reduce` - a boolean (default True) indicating whether or not to apply LLL-reduction to the calculated basis

EXAMPLES:

```
sage: EtaGroup(5).basis()
[Eta product of level 5 : (eta_1)^6 (eta_5)^-6]
sage: EtaGroup(12).basis()
[Eta product of level 12 : (eta_1)^2 (eta_2)^1 (eta_3)^2 (eta_4)^-1 (eta_6)^-
↪7 (eta_12)^3,
Eta product of level 12 : (eta_1)^-4 (eta_2)^2 (eta_3)^4 (eta_6)^-2,
Eta product of level 12 : (eta_1)^-1 (eta_2)^3 (eta_3)^3 (eta_4)^-2 (eta_6)^-
↪9 (eta_12)^6,
Eta product of level 12 : (eta_1)^1 (eta_2)^-1 (eta_3)^-3 (eta_4)^-2 (eta_6)^
↪7 (eta_12)^-2,
Eta product of level 12 : (eta_1)^-6 (eta_2)^9 (eta_3)^2 (eta_4)^-3 (eta_6)^-
↪3 (eta_12)^1]
sage: EtaGroup(12).basis(reduce=False) # much bigger coefficients
[Eta product of level 12 : (eta_2)^24 (eta_12)^-24,
Eta product of level 12 : (eta_1)^-336 (eta_2)^576 (eta_3)^696 (eta_4)^-216␣
↪(eta_6)^-576 (eta_12)^-144,
Eta product of level 12 : (eta_1)^-8 (eta_2)^-2 (eta_6)^2 (eta_12)^8,
Eta product of level 12 : (eta_1)^1 (eta_2)^9 (eta_3)^13 (eta_4)^-4 (eta_6)^-
↪15 (eta_12)^-4,
Eta product of level 12 : (eta_1)^15 (eta_2)^-24 (eta_3)^-29 (eta_4)^9 (eta_
↪6)^24 (eta_12)^5]
```

ALGORITHM: An eta product of level N is uniquely determined by the integers r_d for $d|N$ with $d < N$, since $\sum_{d|N} r_d = 0$. The valid r_d are those that satisfy two congruences modulo 24, and one congruence modulo 2 for every prime divisor of N. We beef up the congruences modulo 2 to congruences modulo 24 by multiplying by 12. To calculate the kernel of the ensuing map $\mathbf{Z}^m \to (\mathbf{Z}/24\mathbf{Z})^n$ we lift it arbitrarily to an integer matrix and calculate its Smith normal form. This gives a basis for the lattice.

This lattice typically contains "large" elements, so by default we pass it to the reduce_basis() function which performs LLL-reduction to give a more manageable basis.

level()

Return the level of self. EXAMPLES:

```
sage: EtaGroup(10).level()
10
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: EtaGroup(12).one()
Eta product of level 12 : 1
```

reduce_basis(*long_etas*)

Produce a more manageable basis via LLL-reduction.

INPUT:

- `long_etas` - a list of EtaGroupElement objects (which should all be of the same level)

OUTPUT:

- a new list of EtaGroupElement objects having hopefully smaller norm

ALGORITHM: We define the norm of an eta-product to be the L^2 norm of its divisor (as an element of the free **Z**-module with the cusps as basis and the standard inner product). Applying LLL-reduction to this gives a basis of hopefully more tractable elements. Of course we'd like to use the L^1 norm as this is just twice the degree, which is a much more natural invariant, but L^2 norm is easier to work with!

EXAMPLES:

```
sage: EtaGroup(4).reduce_basis([ EtaProduct(4, {1:8,2:24,4:-32}),␣
↪EtaProduct(4, {1:8, 4:-8})])
[Eta product of level 4 : (eta_1)^8 (eta_4)^-8,
Eta product of level 4 : (eta_1)^-8 (eta_2)^24 (eta_4)^-16]
```

`sage.modular.etaproducts.`**EtaProduct**(*level*, *dict*)

Create an EtaGroupElement object representing the function $\prod_{d|N} \eta(q^d)^{r_d}$. Checks the criteria of Ligozat to ensure that this product really is the q-expansion of a meromorphic function on X_0(N).

INPUT:

- `level` - (integer): the N such that this eta product is a function on X_0(N).
- `dict` - (dictionary): a dictionary indexed by divisors of N such that the coefficient of $\eta(q^d)$ is r[d]. Only nonzero coefficients need be specified. If Ligozat's criteria are not satisfied, a ValueError will be raised.

OUTPUT:

- an EtaGroupElement object, whose parent is the EtaGroup of level N and whose coefficients are the given dictionary.

Note: The dictionary dict does not uniquely specify N. It is possible for two EtaGroupElements with different N's to be created with the same dictionary, and these represent different objects (although they will have the same q-expansion at the cusp ∞).

EXAMPLES:

```
sage: EtaProduct(3, {3:12, 1:-12})
Eta product of level 3 : (eta_1)^-12 (eta_3)^12
sage: EtaProduct(3, {3:6, 1:-6})
Traceback (most recent call last):
...
ValueError: sum d r_d (=12) is not 0 mod 24
sage: EtaProduct(3, {4:6, 1:-6})
Traceback (most recent call last):
...
ValueError: 4 does not divide 3
```

`sage.modular.etaproducts.`**`eta_poly_relations`**(*eta_elements, degree, labels=['x1', 'x2'], verbose=False*)

Find polynomial relations between eta products.

INPUT:

- `eta_elements` - (list): a list of EtaGroupElement objects. Not implemented unless this list has precisely two elements. degree
- `degree` - (integer): the maximal degree of polynomial to look for.
- `labels` - (list of strings): labels to use for the polynomial returned.
- `verbose`` - (boolean, default False): if True, prints information as it goes.

OUTPUT: a list of polynomials which is a Groebner basis for the part of the ideal of relations between eta_elements which is generated by elements up to the given degree; or None, if no relations were found.

ALGORITHM: An expression of the form $\sum_{0\leq i,j\leq d} a_{ij} x^i y^j$ is zero if and only if it vanishes at the cusp infinity to degree at least $v = d(deg(x) + deg(y))$. For all terms up to q^v in the q-expansion of this expression to be zero is a system of $v + k$ linear equations in d^2 coefficients, where k is the number of nonzero negative coefficients that can appear.

Solving these equations and calculating a basis for the solution space gives us a set of polynomial relations, but this is generally far from a minimal generating set for the ideal, so we calculate a Groebner basis.

As a test, we calculate five extra terms of q-expansion and check that this doesn't change the answer.

EXAMPLES:

```
sage: from sage.modular.etaproducts import eta_poly_relations
sage: t = EtaProduct(26, {2:2,13:2,26:-2,1:-2})
sage: u = EtaProduct(26, {2:4,13:2,26:-4,1:-2})
sage: eta_poly_relations([t, u], 3)
sage: eta_poly_relations([t, u], 4)
[x1^3*x2 - 13*x1^3 - 4*x1^2*x2 - 4*x1*x2 - x2^2 + x2]
```

Use verbose=True to see the details of the computation:

```
sage: eta_poly_relations([t, u], 3, verbose=True)
Trying to find a relation of degree 3
Lowest order of a term at infinity = -12
```

(continues on next page)

(continued from previous page)

```
Highest possible degree of a term = 15
Trying all coefficients from q^-12 to q^15 inclusive
No polynomial relation of order 3 valid for 28 terms
Check:
Trying all coefficients from q^-12 to q^20 inclusive
No polynomial relation of order 3 valid for 33 terms
```

```
sage: eta_poly_relations([t, u], 4, verbose=True)
Trying to find a relation of degree 4
Lowest order of a term at infinity = -16
Highest possible degree of a term = 20
Trying all coefficients from q^-16 to q^20 inclusive
Check:
Trying all coefficients from q^-16 to q^25 inclusive
[x1^3*x2 - 13*x1^3 - 4*x1^2*x2 - 4*x1*x2 - x2^2 + x2]
```

`sage.modular.etaproducts.`**`num_cusps_of_width`**(*N*, *d*)

Return the number of cusps on $X_0(N)$ of width d.

INPUT:

- `N` - (integer): the level
- `d` - (integer): an integer dividing N, the cusp width

EXAMPLES:

```
sage: from sage.modular.etaproducts import num_cusps_of_width
sage: [num_cusps_of_width(18,d) for d in divisors(18)]
[1, 1, 2, 2, 1, 1]
sage: num_cusps_of_width(4,8)
Traceback (most recent call last):
...
ValueError: N and d must be positive integers with d|N
```

`sage.modular.etaproducts.`**`qexp_eta`**(*ps_ring*, *prec*)

Return the q-expansion of $\eta(q)/q^{1/24}$, where $\eta(q)$ is Dedekind's function

$$\eta(q) = q^{1/24} \prod_{n=1}^{\infty} (1 - q^n),$$

as an element of ps_ring, to precision prec.

INPUT:

- `ps_ring` – (PowerSeriesRing): a power series ring
- `prec` – (integer): the number of terms to compute

OUTPUT: An element of ps_ring which is the q-expansion of $\eta(q)/q^{1/24}$ truncated to prec terms.

ALGORITHM: We use the Euler identity

$$\eta(q) = q^{1/24}(1 + \sum_{n\geq 1} (-1)^n (q^{n(3n+1)/2} + q^{n(3n-1)/2})$$

to compute the expansion.

EXAMPLES:

```
sage: from sage.modular.etaproducts import qexp_eta
sage: qexp_eta(ZZ[['q']], 100)
1 - q - q^2 + q^5 + q^7 - q^12 - q^15 + q^22 + q^26 - q^35 - q^40 + q^51 + q^57 -
↪q^70 - q^77 + q^92 + O(q^100)
```

CHAPTER TEN

THE SPACE OF P-ADIC WEIGHTS

A p-adic weight is a continuous character $\mathbf{Z}_p^\times \to \mathbf{C}_p^\times$. These are the $\mathbf{C}_p$-points of a rigid space over $\mathbf{Q}_p$, which is isomorphic to a disjoint union of copies (indexed by $(\mathbf{Z}/p\mathbf{Z})^\times$) of the open unit p-adic disc.

Sage supports both "classical points", which are determined by the data of a Dirichlet character modulo p^m for some m and an integer k (corresponding to the character $z \mapsto z^k \chi(z)$) and "non-classical points" which are determined by the data of an element of $(\mathbf{Z}/p\mathbf{Z})^\times$ and an element $w \in \mathbf{C}_p$ with $|w - 1| < 1$.

EXAMPLES:

```
sage: W = pAdicWeightSpace(17)
sage: W
Space of 17-adic weight-characters defined over '17-adic Field with capped relative␣
↪precision 20'
sage: R.<x> = QQ[]
sage: L = Qp(17).extension(x^2 - 17, names='a'); L.rename('L')
sage: W.base_extend(L)
Space of 17-adic weight-characters defined over 'L'
```

We create a simple element of $\mathcal{W}$: the algebraic character, $x \mapsto x^6$:

```
sage: kappa = W(6)
sage: kappa(5)
15625
sage: kappa(5) == 5^6
True
```

A locally algebraic character, $x \mapsto x^6\chi(x)$ for χ a Dirichlet character mod p:

```
sage: kappa2 = W(6, DirichletGroup(17, Qp(17)).0^8)
sage: kappa2(5) == -5^6
True
sage: kappa2(13) == 13^6
True
```

A non-locally-algebraic character, sending the generator 18 of $1 + 17\mathbf{Z}_{17}$ to 35 and acting as $\mu \mapsto \mu^4$ on the group of 16th roots of unity:

```
sage: kappa3 = W(35 + O(17^20), 4, algebraic=False)
sage: kappa3(2)
16 + 8*17 + ... + O(17^20)
```

AUTHORS:

- David Loeffler (2008-9)

class sage.modular.overconvergent.weightspace.**AlgebraicWeight**(*parent*, *k*, *chi=None*)

Bases: *sage.modular.overconvergent.weightspace.WeightCharacter*

A point in weight space corresponding to a locally algebraic character, of the form $x \mapsto \chi(x)x^k$ where k is an integer and χ is a Dirichlet character modulo p^n for some n.

Lvalue()

Return the value of the p-adic L-function of $\mathbf{Q}$ evaluated at this weight-character.

If the character is $x \mapsto x^k\chi(x)$ where $k > 0$ and χ has conductor a power of p, this is an element of the number field generated by the values of χ, equal to the value of the complex L-function $L(1-k,\chi)$. If χ is trivial, it is equal to $(1-p^{k-1})\zeta(1-k)$.

At present this is not implemented in any other cases, except the trivial character (for which the value is ∞).

Todo: Implement this more generally using the Amice transform machinery in sage/schemes/elliptic_curves/padic_lseries.py, which should clearly be factored out into a separate class.

EXAMPLES:

```
sage: pAdicWeightSpace(7)(4).Lvalue() == (1 - 7^3)*zeta__exact(-3)
True
sage: pAdicWeightSpace(7)(5, DirichletGroup(7, Qp(7)).0^4).Lvalue()
0
sage: pAdicWeightSpace(7)(6, DirichletGroup(7, Qp(7)).0^4).Lvalue()
1 + 2*7 + 7^2 + 3*7^3 + 3*7^5 + 4*7^6 + 2*7^7 + 5*7^8 + 2*7^9 + 3*7^10 + 6*7^
↪11 + 2*7^12 + 3*7^13 + 5*7^14 + 6*7^15 + 5*7^16 + 3*7^17 + 6*7^18 + O(7^19)
```

chi()

If this character is $x \mapsto x^k\chi(x)$ for an integer k and a Dirichlet character χ, return χ.

EXAMPLES:

```
sage: kappa = pAdicWeightSpace(29)(13, DirichletGroup(29, Qp(29)).0^14)
sage: kappa.chi()
Dirichlet character modulo 29 of conductor 29 mapping 2 |--> 28 + 28*29 +␣
↪28*29^2 + ... + O(29^20)
```

k()

If this character is $x \mapsto x^k\chi(x)$ for an integer k and a Dirichlet character χ, return k.

EXAMPLES:

```
sage: kappa = pAdicWeightSpace(29)(13, DirichletGroup(29, Qp(29)).0^14)
sage: kappa.k()
13
```

teichmuller_type()

Return the Teichmuller type of this weight-character κ.

This is the unique $t \in \mathbf{Z}/(p-1)\mathbf{Z}$ such that $\kappa(\mu) = \mu^t$ for μ a $(p-1)$-st root of 1.

For $p = 2$ this does not make sense, but we still want the Teichmuller type to correspond to the index of the component of weight space in which κ lies, so we return 1 if κ is odd and 0 otherwise.

EXAMPLES:

```
sage: pAdicWeightSpace(11)(2, DirichletGroup(11,QQ).0).teichmuller_type()
7
sage: pAdicWeightSpace(29)(13, DirichletGroup(29, Qp(29)).0).teichmuller_
↪type()
14
sage: pAdicWeightSpace(2)(3, DirichletGroup(4,QQ).0).teichmuller_type()
0
```

class sage.modular.overconvergent.weightspace.**ArbitraryWeight**(*parent*, *w*, *t*)
Bases: *sage.modular.overconvergent.weightspace.WeightCharacter*

Create the element of p-adic weight space in the given component mapping 1 + p to w.

Here w must be an element of a p-adic field, with finite precision.

EXAMPLES:

```
sage: pAdicWeightSpace(17)(1 + 17^2 + O(17^3), 11, False)
[1 + 17^2 + O(17^3), 11]
```

teichmuller_type()
Return the Teichmuller type of this weight-character κ.

This is the unique $t \in \mathbf{Z}/(p-1)\mathbf{Z}$ such that $\kappa(\mu) = \mu^t$ for mu a $(p-1)$-st root of 1.

For $p = 2$ this does not make sense, but we still want the Teichmuller type to correspond to the index of the component of weight space in which κ lies, so we return 1 if κ is odd and 0 otherwise.

EXAMPLES:

```
sage: pAdicWeightSpace(17)(1 + 3*17 + 2*17^2 + O(17^3), 8, False).teichmuller_
↪type()
8
sage: pAdicWeightSpace(2)(1 + 2 + O(2^2), 1, False).teichmuller_type()
1
```

class sage.modular.overconvergent.weightspace.**WeightCharacter**(*parent*)
Bases: sage.structure.element.Element

Abstract base class representing an element of the p-adic weight space $Hom(\mathbf{Z}_p^\times, \mathbf{C}_p^\times)$.

Lvalue()
Return the value of the p-adic L-function of $\mathbf{Q}$, which can be regarded as a rigid-analytic function on weight space, evaluated at this character.

EXAMPLES:

```
sage: W = pAdicWeightSpace(11)
sage: sage.modular.overconvergent.weightspace.WeightCharacter(W).Lvalue()
Traceback (most recent call last):
...
NotImplementedError
```

base_extend(*R*)
Extend scalars to the base ring R (which must have a canonical map from the current base ring)

EXAMPLES:

```
sage: w = pAdicWeightSpace(17, QQ)(3)
sage: w.base_extend(Qp(17))
3
```

is_even()

Return True if this weight-character sends -1 to +1.

EXAMPLES:

```
sage: pAdicWeightSpace(17)(0).is_even()
True
sage: pAdicWeightSpace(17)(11).is_even()
False
sage: pAdicWeightSpace(17)(1 + 17 + O(17^20), 3, False).is_even()
False
sage: pAdicWeightSpace(17)(1 + 17 + O(17^20), 4, False).is_even()
True
```

is_trivial()

Return True if and only if this is the trivial character.

EXAMPLES:

```
sage: pAdicWeightSpace(11)(2).is_trivial()
False
sage: pAdicWeightSpace(11)(2, DirichletGroup(11, QQ).0).is_trivial()
False
sage: pAdicWeightSpace(11)(0).is_trivial()
True
```

one_over_Lvalue()

Return the reciprocal of the p-adic L-function evaluated at this weight-character.

If the weight-character is odd, then the L-function is zero, so an error will be raised.

EXAMPLES:

```
sage: pAdicWeightSpace(11)(4).one_over_Lvalue()
-12/133
sage: pAdicWeightSpace(11)(3, DirichletGroup(11, QQ).0).one_over_Lvalue()
-1/6
sage: pAdicWeightSpace(11)(3).one_over_Lvalue()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
sage: pAdicWeightSpace(11)(0).one_over_Lvalue()
0
sage: type(_)
<type 'sage.rings.integer.Integer'>
```

pAdicEisensteinSeries(*ring*, *prec=20*)

Calculate the q-expansion of the p-adic Eisenstein series of given weight-character, normalised so the constant term is 1.

EXAMPLES:

```
sage: kappa = pAdicWeightSpace(3)(3, DirichletGroup(3,QQ).0)
sage: kappa.pAdicEisensteinSeries(QQ[['q']], 20)
1 - 9*q + 27*q^2 - 9*q^3 - 117*q^4 + 216*q^5 + 27*q^6 - 450*q^7 + 459*q^8 -
↪9*q^9 - 648*q^10 + 1080*q^11 - 117*q^12 - 1530*q^13 + 1350*q^14 + 216*q^15 -
↪ 1845*q^16 + 2592*q^17 + 27*q^18 - 3258*q^19 + O(q^20)
```

values_on_gens()

If κ is this character, calculate the values $(\kappa(r), t)$ where r is $1 + p$ (or 5 if $p = 2$) and t is the unique element of $\mathbf{Z}/(p-1)\mathbf{Z}$ such that $\kappa(\mu) = \mu^t$ for μ a (p-1)st root of unity. (If $p = 2$, we take t to be 0 or 1 according to whether κ is odd or even.) These two values uniquely determine the character κ.

EXAMPLES:

```
sage: W = pAdicWeightSpace(11); W(2).values_on_gens()
(1 + 2*11 + 11^2 + O(11^20), 2)
sage: W(2, DirichletGroup(11, QQ).0).values_on_gens()
(1 + 2*11 + 11^2 + O(11^20), 7)
sage: W(1 + 2*11 + O(11^5), 4, algebraic = False).values_on_gens()
(1 + 2*11 + O(11^5), 4)
```

class sage.modular.overconvergent.weightspace.**WeightSpace_class**(*p*, *base_ring*)

Bases: sage.structure.parent_base.ParentWithBase

The space of p-adic weight-characters $\mathcal{W} = \mathrm{Hom}(\mathbf{Z}_p^\times, \mathbf{C}_p^\times)$.

This is isomorphic to a disjoint union of $(p-1)$ open discs of radius 1 (or 2 such discs if $p = 2$), with the parameter on the open disc corresponding to the image of $1 + p$ (or 5 if $p = 2$)

base_extend(*R*)

Extend scalars to the ring R.

There must be a canonical coercion map from the present base ring to R.

EXAMPLES:

```
sage: W = pAdicWeightSpace(3, QQ)
sage: W.base_extend(Qp(3))
Space of 3-adic weight-characters defined over '3-adic Field with capped␣
↪relative precision 20'
sage: W.base_extend(IntegerModRing(12))
Traceback (most recent call last):
...
TypeError: No coercion map from 'Rational Field' to 'Ring of integers modulo␣
↪12' is defined
```

prime()

Return the prime p such that this is a p-adic weight space.

EXAMPLES:

```
sage: pAdicWeightSpace(17).prime()
17
```

zero()

Return the zero of this weight space.

EXAMPLES:

```
sage: W = pAdicWeightSpace(17)
sage: W.zero()
0
```

sage.modular.overconvergent.weightspace.**WeightSpace_constructor**(*p*, *base_ring=None*)

Construct the p-adic weight space for the given prime p.

A p-adic weight is a continuous character $\mathbf{Z}_p^\times \to \mathbf{C}_p^\times$. These are the $\mathbf{C}_p$-points of a rigid space over $\mathbf{Q}_p$, which is isomorphic to a disjoint union of copies (indexed by $(\mathbf{Z}/p\mathbf{Z})^\times$) of the open unit p-adic disc.

Note that the "base ring" of a p-adic weight is the smallest ring containing the image of $\mathbf{Z}$; in particular, although the default base ring is $\mathbf{Q}_p$, base ring $\mathbf{Q}$ will also work.

EXAMPLES:

```
sage: pAdicWeightSpace(3) # indirect doctest
Space of 3-adic weight-characters defined over '3-adic Field with capped relative␣
↪precision 20'
sage: pAdicWeightSpace(3, QQ)
Space of 3-adic weight-characters defined over 'Rational Field'
sage: pAdicWeightSpace(10)
Traceback (most recent call last):
...
ValueError: p must be prime
```

CHAPTER

ELEVEN

OVERCONVERGENT P-ADIC MODULAR FORMS FOR SMALL PRIMES

This module implements computations of Hecke operators and U_p-eigenfunctions on p-adic overconvergent modular forms of tame level 1, where p is one of the primes $\{2, 3, 5, 7, 13\}$, using the algorithms described in [Loe2007].

- [Loe2007]

AUTHORS:

- David Loeffler (August 2008): initial version
- David Loeffler (March 2009): extensively reworked
- Lloyd Kilford (May 2009): add `slopes()` method
- David Loeffler (June 2009): miscellaneous bug fixes and usability improvements

11.1 The Theory

Let p be one of the above primes, so $X_0(p)$ has genus 0, and let

$$f_p = \sqrt[p-1]{\frac{\Delta(pz)}{\Delta(z)}}$$

(an η-product of level p – see module `sage.modular.etaproducts`). Then one can show that f_p gives an isomorphism $X_0(p) \to \mathbb{P}^1$. Furthermore, if we work over $\mathbf{C}_p$, the r-overconvergent locus on $X_0(p)$ (or of $X_0(1)$, via the canonical subgroup lifting), corresponds to the p-adic disc

$$|f_p|_p \leq p^{\frac{12r}{p-1}}.$$

(This is Theorem 1 of [Loe2007].)

Hence if we fix an element c with $|c| = p^{-\frac{12r}{p-1}}$, the space $S_k^\dagger(1, r)$ of overconvergent p-adic modular forms has an orthonormal basis given by the functions $(cf)^n$. So any element can be written in the form $E_k \times \sum_{n\geq 0} a_n(cf)^n$, where $a_n \to 0$ as $N \to \infty$, and any such sequence a_n defines a unique overconvergent form.

One can now find the matrix of Hecke operators in this basis, either by calculating q-expansions, or (for the special case of U_p) using a recurrence formula due to Kolberg.

11.2 An Extended Example

We create a space of 3-adic modular forms:

```
sage: M = OverconvergentModularForms(3, 8, 1/6, prec=60)
```

Creating an element directly as a linear combination of basis vectors.

```
sage: f1 = M.3 + M.5; f1.q_expansion()
27*q^3 + 1055916/1093*q^4 + 19913121/1093*q^5 + 268430112/1093*q^6 + ...
sage: f1.coordinates(8)
[0, 0, 0, 1, 0, 1, 0, 0]
```

We can coerce from elements of classical spaces of modular forms:

```
sage: f2 = M(CuspForms(3, 8).0); f2
3-adic overconvergent modular form of weight-character 8 with q-expansion q + 6*q^2 -
↪27*q^3 - 92*q^4 + 390*q^5 - 162*q^6 ...
```

We express this in a basis, and see that the coefficients go to zero very fast:

```
sage: [x.valuation(3) for x in f2.coordinates(60)]
[+Infinity, -1, 3, 6, 10, 13, 18, 20, 24, 27, 31, 34, 39, 41, 45, 48, 52, 55, 61, 62,
↪66, 69, 73, 76, 81, 83, 87, 90, 94, 97, 102, 104, 108, 111, 115, 118, 124, 125, 129,
↪ 132, 136, 139, 144, 146, 150, 153, 157, 160, 165, 167, 171, 174, 178, 181, 188,
↪188, 192, 195, 199, 202]
```

This form has more level at p, and hence is less overconvergent:

```
sage: f3 = M(CuspForms(9, 8).0); [x.valuation(3) for x in f3.coordinates(60)]
[+Infinity, -1, -1, 0, -4, -4, -2, -3, 0, 0, -1, -1, 1, 0, 3, 3, 3, 3, 5, 3, 7, 7, 6,
↪6, 8, 7, 10, 10, 8, 8, 10, 9, 12, 12, 12, 12, 14, 12, 17, 16, 15, 15, 17, 16, 19,
↪19, 18, 18, 20, 19, 22, 22, 22, 22, 24, 21, 25, 26, 24, 24]
```

An error will be raised for forms which are not sufficiently overconvergent:

```
sage: M(CuspForms(27, 8).0)
Traceback (most recent call last):
...
ValueError: Form is not overconvergent enough (form is only 1/12-overconvergent)
```

Let's compute some Hecke operators. Note that the coefficients of this matrix are p-adically tiny:

```
sage: M.hecke_matrix(3, 4).change_ring(Qp(3,prec=1))
[        1 + O(3)                0                0                0]
[               0    2*3^3 + O(3^4)    2*3^3 + O(3^4)      3^2 + O(3^3)]
[               0    2*3^7 + O(3^8)    2*3^8 + O(3^9)      3^6 + O(3^7)]
[               0 2*3^10 + O(3^11)  2*3^10 + O(3^11)   2*3^9 + O(3^10)]
```

We compute the eigenfunctions of a 4x4 truncation:

```
sage: efuncs = M.eigenfunctions(4)
sage: for i in [1..3]:
....:     print(efuncs[i].q_expansion(prec=4).change_ring(Qp(3,prec=20)))
(1 + O(3^20))*q + (2*3 + 3^15 + 3^16 + 3^17 + 2*3^19 + 2*3^20 + O(3^21))*q^2 + (2*3^3
↪+ 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^10 + 2*3^11 + 2*3^12 + 2*3^13
↪+ 2*3^14 + 2*3^15 + 2*3^16 + 3^17 + 2*3^18 + 2*3^19 + 3^21 + 3^22 + O(3^23))*q^3 +
↪O(q^4)
(1 + O(3^20))*q + (3 + 2*3^2 + 3^3 + 3^4 + 3^12 + 3^13 + 2*3^14 + 3^15 + 2*3^17 + 3^
↪18 + 3^19 + 3^20 + O(3^21))*q^2 + (3^7 + 3^13 + 2*3^14 + 2*3^15 + 3^16 + 3^17 + 2*3^
↪18 + 3^20 + 2*3^21 + 2*3^22 + 2*3^23 + 2*3^25 + O(3^27))*q^3 + O(q^4)
```

(continues on next page)

(continued from previous page)

```
(1 + O(3^20))*q + (2*3 + 3^3 + 2*3^4 + 3^6 + 2*3^8 + 3^9 + 3^10 + 2*3^11 + 2*3^13 + 3^
↪16 + 3^18 + 3^19 + 3^20 + O(3^21))*q^2 + (3^9 + 2*3^12 + 3^15 + 3^17 + 3^18 + 3^19␣
↪+ 3^20 + 2*3^22 + 2*3^23 + 2*3^27 + 2*3^28 + O(3^29))*q^3 + O(q^4)
```

The first eigenfunction is a classical cusp form of level 3:

```
sage: (efuncs[1] - M(CuspForms(3, 8).0)).valuation()
13
```

The second is an Eisenstein series!

```
sage: (efuncs[2] - M(EisensteinForms(3, 8).1)).valuation()
10
```

The third is a genuinely new thing (not a classical modular form at all); the coefficients are almost certainly not algebraic over **Q**. Note that the slope is 9, so Coleman's classicality criterion (forms of slope $< k - 1$ are classical) does not apply.

```
sage: a3 = efuncs[3].q_expansion()[3]; a3
3^9 + 2*3^12 + 3^15 + 3^17 + 3^18 + 3^19 + 3^20 + 2*3^22 + 2*3^23 + 2*3^27 + 2*3^28 +␣
↪3^32 + 3^33 + 2*3^34 + 3^38 + 2*3^39 + 3^40 + 2*3^41 + 3^44 + 3^45 + 3^46 + 2*3^47␣
↪+ 2*3^48 + 3^49 + 3^50 + 2*3^51 + 2*3^52 + 3^53 + 2*3^54 + 3^55 + 3^56 + 3^57 + 2*3^
↪58 + 2*3^59 + 3^60 + 2*3^61 + 2*3^63 + 2*3^64 + 3^65 + 2*3^67 + 3^68 + 2*3^69 + 2*3^
↪71 + 3^72 + 2*3^74 + 3^75 + 3^76 + 3^79 + 3^80 + 2*3^83 + 2*3^84 + 3^85 + 2*3^87 +␣
↪3^88 + 2*3^89 + 2*3^90 + 2*3^91 + 3^92 + O(3^98)
sage: efuncs[3].slope()
9
```

---

class sage.modular.overconvergent.genus0.**OverconvergentModularFormElement**(*parent*, *gexp=None*, *qexp=None*)

Bases: `sage.structure.element.ModuleElement`

A class representing an element of a space of overconvergent modular forms.

EXAMPLES:

```
sage: K.<w> = Qp(5).extension(x^7 - 5); s = OverconvergentModularForms(5, 6, 1/21,
↪ base_ring=K).0
sage: s == loads(dumps(s))
True
```

additive_order()

Return the additive order of this element (required attribute for all elements deriving from sage.modules.ModuleElement).

EXAMPLES:

```
sage: M = OverconvergentModularForms(13, 10, 1/2, base_ring = Qp(13).
↪extension(x^2 - 13,names='a'))
sage: M.gen(0).additive_order()
+Infinity
sage: M(0).additive_order()
1
```

base_extend(*R*)

Return a copy of self but with coefficients in the given ring.

EXAMPLES:

```
sage: M = OverconvergentModularForms(7, 10, 1/2, prec=5)
sage: f = M.1
sage: f.base_extend(Qp(7, 4))
7-adic overconvergent modular form of weight-character 10 with q-expansion (7␣
↪+ O(7^5))*q + (6*7 + 4*7^2 + 7^3 + 6*7^4 + O(7^5))*q^2 + (5*7 + 5*7^2 + 7^4␣
↪+ O(7^5))*q^3 + (7^2 + 4*7^3 + 3*7^4 + 2*7^5 + O(7^6))*q^4 + O(q^5)
```

coordinates(*prec=None*)

Return the coordinates of this modular form in terms of the basis of this space.

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 0, 1/2, prec=15)
sage: f = (M.0 + M.3); f.coordinates()
[1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: f.coordinates(6)
[1, 0, 0, 1, 0, 0]
sage: OverconvergentModularForms(3, 0, 1/6)(f).coordinates(6)
[1, 0, 0, 729, 0, 0]
sage: f.coordinates(100)
Traceback (most recent call last):
...
ValueError: Precision too large for space
```

eigenvalue()

Return the U_p-eigenvalue of this eigenform. Raises an error unless this element was explicitly flagged as an eigenform, using the _notify_eigen function.

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
sage: f.eigenvalue()
3^2 + 3^4 + 2*3^6 + 3^7 + 3^8 + 2*3^9 + 2*3^10 + 3^12 + 3^16 + 2*3^17 + 3^18␣
↪+ 3^20 + 2*3^21 + 3^22 + 2*3^23 + 3^25 + 3^26 + 2*3^27 + 2*3^29 + 3^30 + 3^
↪31 + 3^32 + 3^33 + 3^34 + 3^36 + 3^40 + 2*3^41 + 3^43 + 3^44 + 3^45 + 3^46␣
↪+ 3^48 + 3^49 + 3^50 + 2*3^51 + 3^52 + 3^54 + 2*3^57 + 2*3^59 + 3^60 + 3^61␣
↪+ 2*3^63 + 2*3^66 + 2*3^67 + 3^69 + 2*3^72 + 3^74 + 2*3^75 + 3^76 + 2*3^77␣
↪+ 2*3^78 + 2*3^80 + 3^81 + 2*3^82 + 3^84 + 2*3^85 + 2*3^86 + 3^87 + 3^88 +␣
↪2*3^89 + 2*3^91 + 3^93 + 3^94 + 3^95 + 3^96 + 3^98 + 2*3^99 + O(3^100)
sage: M.gen(4).eigenvalue()
Traceback (most recent call last):
...
TypeError: eigenvalue only defined for eigenfunctions
```

gexp()

Return the formal power series in g corresponding to this overconvergent modular form (so the result is F where this modular form is $E_k^* \times F(g)$, where g is the appropriately normalised parameter of $X_0(p)$).

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
```

(continues on next page)

(continued from previous page)

```
sage: f.gexp()
(3^-3 + O(3^95))*g + (3^-1 + 1 + 2*3 + 3^2 + 2*3^3 + 3^5 + 3^7 + 3^10 + 3^11␣
↪+ 3^14 + 3^15 + 3^16 + 2*3^19 + 3^21 + 3^22 + 2*3^23 + 2*3^24 + 3^26 + 2*3^
↪27 + 3^29 + 3^31 + 3^34 + 2*3^35 + 2*3^36 + 3^38 + 2*3^39 + 3^41 + 2*3^42 +␣
↪2*3^43 + 2*3^44 + 2*3^46 + 2*3^47 + 3^48 + 2*3^49 + 2*3^50 + 3^51 + 2*3^54␣
↪+ 2*3^55 + 2*3^56 + 3^57 + 2*3^58 + 2*3^59 + 2*3^60 + 3^61 + 3^62 + 3^63 +␣
↪3^64 + 2*3^65 + 3^67 + 3^68 + 2*3^69 + 3^70 + 2*3^71 + 2*3^74 + 3^76 + 2*3^
↪77 + 3^78 + 2*3^79 + 2*3^80 + 3^84 + 2*3^85 + 2*3^86 + 3^88 + 2*3^89 + 3^91␣
↪+ 3^92 + 2*3^94 + 3^95 + O(3^97))*g^2 + O(g^3)
```

governing_term(r)

The degree of the series term with largest norm on the r-overconvergent region.

EXAMPLES:

```
sage: o=OverconvergentModularForms(3, 0, 1/2)
sage: f=o.eigenfunctions(10)[1]
sage: f.governing_term(1/2)
1
```

is_eigenform()

Return True if this is an eigenform. At present this returns False unless this element was explicitly flagged as an eigenform, using the _notify_eigen function.

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
sage: f.is_eigenform()
True
sage: M.gen(4).is_eigenform()
False
```

is_integral()

Test whether or not this element has q-expansion coefficients that are p-adically integral. This should always be the case with eigenfunctions, but sometimes if n is very large this breaks down for unknown reasons!

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 0, 1/3)
sage: q = QQ[['q']].gen()
sage: M(q - 17*q^2 + O(q^3)).is_integral()
True
sage: M(q - q^2/2 + 6*q^7  + O(q^9)).is_integral()
False
```

prec()

Return the series expansion precision of this overconvergent modular form. (This is not the same as the p-adic precision of the coefficients.)

EXAMPLES:

```
sage: OverconvergentModularForms(5, 6, 1/3,prec=15).gen(1).prec()
15
```

prime()

If this is a p-adic modular form, return p.

EXAMPLES:

```
sage: OverconvergentModularForms(2, 0, 1/2).an_element().prime()
2
```

q_expansion(*prec=None*)

Return the q-expansion of self, to as high precision as it is known.

EXAMPLES:

```
sage: OverconvergentModularForms(3, 4, 1/2).gen(0).q_expansion()
1 - 120/13*q - 1080/13*q^2 - 120/13*q^3 - 8760/13*q^4 - 15120/13*q^5 - 1080/
↪13*q^6 - 41280/13*q^7 - 5400*q^8 - 120/13*q^9 - 136080/13*q^10 - 159840/
↪13*q^11 - 8760/13*q^12 - 263760/13*q^13 - 371520/13*q^14 - 15120/13*q^15 -␣
↪561720/13*q^16 - 45360*q^17 - 1080/13*q^18 - 823200/13*q^19 + O(q^20)
```

r_ord(*r*)

The p-adic valuation of the norm of self on the r-overconvergent region.

EXAMPLES:

```
sage: o=OverconvergentModularForms(3, 0, 1/2)
sage: t = o([1, 1, 1/3])
sage: t.r_ord(1/2)
1
sage: t.r_ord(2/3)
3
```

slope()

Return the slope of this eigenform, i.e. the valuation of its U_p-eigenvalue. Raises an error unless this element was explicitly flagged as an eigenform, using the _notify_eigen function.

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
sage: f.slope()
2
sage: M.gen(4).slope()
Traceback (most recent call last):
...
TypeError: slope only defined for eigenfunctions
```

valuation()

Return the p-adic valuation of this form (i.e. the minimum of the p-adic valuations of its coordinates).

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: (M.7).valuation()
0
sage: (3^18 * (M.2)).valuation()
18
```

valuation_plot(*rmax=None*)

Draw a graph depicting the growth of the norm of this overconvergent modular form as it approaches the boundary of the overconvergent region.

EXAMPLES:

```
sage: o=OverconvergentModularForms(3, 0, 1/2)
sage: f=o.eigenfunctions(4)[1]
sage: f.valuation_plot()
Graphics object consisting of 1 graphics primitive
```

weight()

Return the weight of this overconvergent modular form.

EXAMPLES:

```
sage: M = OverconvergentModularForms(13, 10, 1/2, base_ring = Qp(13).
↪extension(x^2 - 13,names='a'))
sage: M.gen(0).weight()
10
```

sage.modular.overconvergent.genus0.**OverconvergentModularForms**(*prime*, *weight*, *radius*, *base_ring=Rational Field*, *prec=20*, *char=None*)

Create a space of overconvergent p-adic modular forms of level $\Gamma_0(p)$, over the given base ring. The base ring need not be a p-adic ring (the spaces we compute with typically have bases over $\mathbf{Q}$).

INPUT:

- `prime` - a prime number p, which must be one of the primes $\{2, 3, 5, 7, 13\}$, or the congruence subgroup $\Gamma_0(p)$ where p is one of these primes.
- `weight` - an integer (which at present must be 0 or ≥ 2), the weight.
- `radius` - a rational number in the interval $\left(0, \frac{p}{p+1}\right)$, the radius of overconvergence.
- `base_ring` (default: $\mathbf{Q}$), a ring over which to compute. This need not be a p-adic ring.
- `prec` - an integer (default: 20), the number of q-expansion terms to compute.
- `char` - a Dirichlet character modulo p or `None` (the default). Here `None` is interpreted as the trivial character modulo p.

The character χ and weight k must satisfy $(-1)^k = \chi(-1)$, and the base ring must contain an element v such that $\mathrm{ord}_p(v) = \frac{12r}{p-1}$ where r is the radius of overconvergence (and ord_p is normalised so $\mathrm{ord}_p(p) = 1$).

EXAMPLES:

```
sage: OverconvergentModularForms(3, 0, 1/2)
Space of 3-adic 1/2-overconvergent modular forms of weight-character 0 over␣
↪Rational Field
sage: OverconvergentModularForms(3, 16, 1/2)
Space of 3-adic 1/2-overconvergent modular forms of weight-character 16 over␣
↪Rational Field
sage: OverconvergentModularForms(3, 3, 1/2, char = DirichletGroup(3,QQ).0)
Space of 3-adic 1/2-overconvergent modular forms of weight-character (3, 3, [-1])␣
↪over Rational Field
```

class sage.modular.overconvergent.genus0.**OverconvergentModularFormsSpace**(*prime*, *weight*, *radius*, *base_ring*, *prec*, *char*)

Bases: sage.modules.module.Module

A space of overconvergent modular forms of level $\Gamma_0(p)$, where p is a prime such that $X_0(p)$ has genus 0.

Elements are represented as power series, with a formal power series F corresponding to the modular form $E_k^* \times F(g)$ where E_k^* is the p-deprived Eisenstein series of weight-character k, and g is a uniformiser of $X_0(p)$ normalised so that the r-overconvergent region $X_0(p)_{\geq r}$ corresponds to $|g| \leq 1$.

base_extend(*ring*)

Return the base extension of self to the given base ring. There must be a canonical map to this ring from the current base ring, otherwise a TypeError will be raised.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 0, 1/2, base_ring = Qp(2))
sage: M.base_extend(Qp(2).extension(x^2 - 2, names="w"))
Space of 2-adic 1/2-overconvergent modular forms of weight-character 0 over 2-
↪adic Eisenstein Extension ...
sage: M.base_extend(QQ)
Traceback (most recent call last):
...
TypeError: Base extension of self (over '2-adic Field with capped relative␣
↪precision 20') to ring 'Rational Field' not defined.
```

change_ring(*ring*)

Return the space corresponding to self but over the given base ring.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 0, 1/2)
sage: M.change_ring(Qp(2))
Space of 2-adic 1/2-overconvergent modular forms of weight-character 0 over 2-
↪adic Field with ...
```

character()

Return the character of self. For overconvergent forms, the weight and the character are unified into the concept of a weight-character, so this returns exactly the same thing as self.weight().

EXAMPLES:

```
sage: OverconvergentModularForms(3, 0, 1/2).character()
0
sage: type(OverconvergentModularForms(3, 0, 1/2).character())
<class '...weightspace.AlgebraicWeight'>
sage: OverconvergentModularForms(3, 3, 1/2, char=DirichletGroup(3,QQ).0).
↪character()
(3, 3, [-1])
```

coordinate_vector(*x*)

Write x as a vector with respect to the basis given by self.basis(). Here x must be an element of this space or something that can be converted into one. If x has precision less than the default precision of self, then the returned vector will be shorter.

EXAMPLES:

```
sage: M = OverconvergentModularForms(Gamma0(3), 0, 1/3, prec=4)
sage: M.coordinate_vector(M.gen(2))
(0, 0, 1, 0)
sage: q = QQ['q'].gen(); M.coordinate_vector(q - q^2 + O(q^4))
(0, 1/9, -13/81, 74/243)
sage: M.coordinate_vector(q - q^2 + O(q^3))
(0, 1/9, -13/81)
```

cps_u(*n*, *use_recurrence=False*)

Compute the characteristic power series of U_p acting on self, using an n x n matrix.

EXAMPLES:

```
sage: OverconvergentModularForms(3, 16, 1/2, base_ring=Qp(3)).cps_u(4)
1 + O(3^20) + (2 + 2*3 + 2*3^2 + 2*3^4 + 3^5 + 3^6 + 3^7 + 3^11 + 3^12 + 2*3^
↪14 + 3^16 + 3^18 + O(3^19))*T + (2*3^3 + 3^5 + 3^6 + 3^7 + 2*3^8 + 2*3^9 +␣
↪2*3^10 + 3^11 + 3^12 + 2*3^13 + 2*3^16 + 2*3^18 + O(3^19))*T^2 + (2*3^15 +␣
↪2*3^16 + 2*3^19 + 2*3^20 + 2*3^21 + O(3^22))*T^3 + (3^17 + 2*3^18 + 3^19 +␣
↪3^20 + 3^22 + 2*3^23 + 2*3^25 + 3^26 + O(3^27))*T^4
sage: OverconvergentModularForms(3, 16, 1/2, base_ring=Qp(3), prec=30).cps_
↪u(10)
1 + O(3^20) + (2 + 2*3 + 2*3^2 + 2*3^4 + 3^5 + 3^6 + 3^7 + 2*3^15 + O(3^
↪16))*T + (2*3^3 + 3^5 + 3^6 + 3^7 + 2*3^8 + 2*3^9 + 2*3^10 + 2*3^11 + 2*3^
↪12 + 2*3^13 + 3^14 + 3^15 + O(3^16))*T^2 + (3^14 + 2*3^15 + 2*3^16 + 3^17 +␣
↪3^18 + O(3^19))*T^3 + (3^17 + 2*3^18 + 3^19 + 3^20 + 3^21 + O(3^24))*T^4 +␣
↪(3^29 + 2*3^32 + O(3^33))*T^5 + (2*3^44 + O(3^45))*T^6 + (2*3^59 + O(3^
↪60))*T^7 + (2*3^78 + O(3^79))*T^8
```

Note: Uses the Hessenberg form of the Hecke matrix to compute the characteristic polynomial. Because of the use of relative precision here this tends to give better precision in the p-adic coefficients.

eigenfunctions(*n*, *F=None*, *exact_arith=True*)

Calculate approximations to eigenfunctions of self. These are the eigenfunctions of self.hecke_matrix(p, n), which are approximations to the true eigenfunctions. Returns a list of OverconvergentModularFormElement objects, in increasing order of slope.

INPUT:

- `n` - integer. The size of the matrix to use.
- `F` - None, or a field over which to calculate eigenvalues. If the field is None, the current base ring is used. If the base ring is not a p-adic ring, an error will be raised.
- `exact_arith` - True or False (default True). If True, use exact rational arithmetic to calculate the matrix of the U operator and its characteristic power series, even when the base ring is an inexact p-adic ring. This is typically slower, but more numerically stable.

NOTE: Try using `set_verbose(1, 'sage/modular/overconvergent')` to get more feedback on what is going on in this algorithm. For even more feedback, use 2 instead of 1.

EXAMPLES:

```
sage: X = OverconvergentModularForms(2, 2, 1/6).eigenfunctions(8, Qp(2, 100))
sage: X[1]
2-adic overconvergent modular form of weight-character 2 with q-expansion (1␣
↪+ O(2^74))*q + (2^4 + 2^5 + 2^9 + 2^10 + 2^12 + 2^13 + 2^15 + 2^17 + 2^19 +␣
↪2^20 + 2^21 + 2^23 + 2^28 + 2^30 + 2^31 + 2^32 + 2^34 + 2^36 + 2^37 + 2^39
↪+ 2^40 + 2^43 + 2^44 + 2^45 + 2^47 + 2^48 + 2^52 + 2^53 + 2^54 + 2^55 + 2^
↪56 + 2^58 + 2^59 + 2^60 + 2^61 + 2^67 + 2^68 + 2^70 + 2^71 + 2^72 + 2^74 +␣
↪2^76 + O(2^78))*q^2 + (2^2 + 2^7 + 2^8 + 2^9 + 2^12 + 2^13 + 2^16 + 2^17 +␣
↪2^21 + 2^23 + 2^25 + 2^28 + 2^33 + 2^34 + 2^36 + 2^37 + 2^42 + 2^45 + 2^47␣
↪+ 2^49 + 2^50 + 2^51 + 2^54 + 2^55 + 2^58 + 2^60 + 2^61 + 2^67 + 2^71 + 2^
↪72 + O(2^76))*q^3 + (2^8 + 2^11 + 2^14 + 2^19 + 2^21 + 2^22 + 2^24 + 2^25 +␣
↪2^26 + 2^27 + 2^28 + 2^29 + 2^32 + 2^33 + 2^35 + 2^36 + 2^44 + 2^45 + 2^46
```

(continues on next page)

(continued from previous page)

```
sage: [x.slope() for x in X]
[0, 4, 8, 14, 16, 18, 26, 30]
```

gen (*i*)

Return the ith module generator of self.

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 2, 1/2, prec=4)
sage: M.gen(0)
3-adic overconvergent modular form of weight-character 2 with q-expansion 1 +␣
↪12*q + 36*q^2 + 12*q^3 + O(q^4)
sage: M.gen(1)
3-adic overconvergent modular form of weight-character 2 with q-expansion␣
↪27*q + 648*q^2 + 7290*q^3 + O(q^4)
sage: M.gen(30)
3-adic overconvergent modular form of weight-character 2 with q-expansion O(q^
↪4)
```

gens ()

Return a generator object that iterates over the (infinite) set of basis vectors of self.

EXAMPLES:

```
sage: o = OverconvergentModularForms(3, 12, 1/2)
sage: t = o.gens()
sage: next(t)
3-adic overconvergent modular form of weight-character 12 with q-expansion 1 -
↪ 32760/61203943*q - 67125240/61203943*q^2 - ...
sage: next(t)
3-adic overconvergent modular form of weight-character 12 with q-expansion␣
↪27*q + 19829193012/61203943*q^2 + 146902585770/61203943*q^3 + ...
```

gens_dict ()

Return a dictionary mapping the names of generators of this space to their values. (Required by parent class definition.) As this does not make any sense here, this raises a TypeError.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 4, 1/6)
sage: M.gens_dict()
Traceback (most recent call last):
...
TypeError: gens_dict does not make sense as number of generators is infinite
```

hecke_matrix (*m*, *n*, *use_recurrence=False*, *exact_arith=False*)

Calculate the matrix of the T_m operator in the basis of this space, truncated to an $n \times n$ matrix. Conventions are that operators act on the left on column vectors (this is the opposite of the conventions of the sage.modules.matrix_morphism class!) Uses naive q-expansion arguments if use_recurrence=False and uses the Kolberg style recurrences if use_recurrence=True.

The argument "exact_arith" causes the computation to be done with rational arithmetic, even if the base ring is an inexact p-adic ring. This is useful as there can be precision loss issues (particularly with use_recurrence=False).

EXAMPLES:

```
sage: OverconvergentModularForms(2, 0, 1/2).hecke_matrix(2, 4)
[    1     0     0     0]
[    0    24    64     0]
[    0    32  1152  4608]
[    0     0  3072 61440]
sage: OverconvergentModularForms(2, 12, 1/2, base_ring=pAdicField(2)).hecke_
↪matrix(2, 3) * (1 + O(2^2))
[        1 + O(2^2)                  0                  0]
[                 0        2^3 + O(2^5)        2^6 + O(2^8)]
[                 0        2^4 + O(2^6) 2^7 + 2^8 + O(2^9)]
sage: OverconvergentModularForms(2, 12, 1/2, base_ring=pAdicField(2)).hecke_
↪matrix(2, 3, exact_arith=True)
[                             1                              0                              ␣
↪               0]
[                             0                33881928/1414477                             ␣
↪              64]
[                             0 -192898739923312/2000745183529                            ␣
↪1626332544/1414477]
```

hecke_operator (*f*, *m*)

Given an element f and an integer m, calculates the Hecke operator T_m acting on f.

The input may be either a "bare" power series, or an OverconvergentModularFormElement object; the return value will be of the same type.

EXAMPLES:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.1
sage: M.hecke_operator(f, 3)
3-adic overconvergent modular form of weight-character 0 with q-expansion␣
↪2430*q + 265356*q^2 + 10670373*q^3 + 249948828*q^4 + 4113612864*q^5 +␣
↪52494114852*q^6 + O(q^7)
sage: M.hecke_operator(f.q_expansion(), 3)
2430*q + 265356*q^2 + 10670373*q^3 + 249948828*q^4 + 4113612864*q^5 +␣
↪52494114852*q^6 + O(q^7)
```

is_exact ()

True if elements of this space are represented exactly, i.e., there is no precision loss when doing arithmetic. As this is never true for overconvergent modular forms spaces, this returns False.

EXAMPLES:

```
sage: OverconvergentModularForms(13, 12, 0).is_exact()
False
```

ngens ()

The number of generators of self (as a module over its base ring), i.e. infinity.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 4, 1/6)
sage: M.ngens()
+Infinity
```

normalising_factor ()

The normalising factor c such that $g = cf$ is a parameter for the r-overconvergent disc in $X_0(p)$, where f is the standard uniformiser.

EXAMPLES:

```
sage: L.<w> = Qp(7).extension(x^2 - 7)
sage: OverconvergentModularForms(7, 0, 1/4, base_ring=L).normalising_factor()
w + O(w^41)
```

prec()

Return the series precision of self. Note that this is different from the p-adic precision of the base ring.

EXAMPLES:

```
sage: OverconvergentModularForms(3, 0, 1/2).prec()
20
sage: OverconvergentModularForms(3, 0, 1/2,prec=40).prec()
40
```

prime()

Return the residue characteristic of self, i.e. the prime p such that this is a p-adic space.

EXAMPLES:

```
sage: OverconvergentModularForms(5, 12, 1/3).prime()
5
```

radius()

The radius of overconvergence of this space.

EXAMPLES:

```
sage: OverconvergentModularForms(3, 0, 1/3).radius()
1/3
```

recurrence_matrix(*use_smithline=True*)

Return the recurrence matrix satisfied by the coefficients of U, that is a matrix $R = (r_{rs})_{r,s=1\dots p}$ such that $u_{ij} = \sum_{r,s=1}^{p} r_{rs} u_{i-r,j-s}$. Uses an elegant construction which I believe is due to Smithline. See [Loe2007].

EXAMPLES:

```
sage: OverconvergentModularForms(2, 0, 0).recurrence_matrix()
[  48    1]
[4096    0]
sage: OverconvergentModularForms(2, 0, 1/2).recurrence_matrix()
[48 64]
[64  0]
sage: OverconvergentModularForms(3, 0, 0).recurrence_matrix()
[   270     36      1]
[ 26244    729      0]
[531441      0      0]
sage: OverconvergentModularForms(5, 0, 0).recurrence_matrix()
[     1575      1300       315        30         1]
[   162500     39375      3750       125         0]
[  4921875    468750     15625         0         0]
[ 58593750   1953125         0         0         0]
[244140625         0         0         0         0]
sage: OverconvergentModularForms(7, 0, 0).recurrence_matrix()
[       4018        8624        5915        1904         322          28     ␣
↪       1]
[     422576      289835       93296       15778        1372          49     ␣
↪       0]
```

(continues on next page)

(continued from previous page)

```
[   14201915     4571504      773122       67228        2401           0  ␣
↪      0]
[  224003696    37882978     3294172      117649           0           0  ␣
↪      0]
[ 1856265922   161414428     5764801           0           0           0  ␣
↪      0]
[ 7909306972   282475249           0           0           0           0  ␣
↪      0]
[13841287201           0           0           0           0           0  ␣
↪      0]
sage: OverconvergentModularForms(13, 0, 0).recurrence_matrix()
[         15145          124852          354536 ...
```

slopes (*n*, *use_recurrence=False*)

Compute the slopes of the U_p operator acting on self, using an n x n matrix.

EXAMPLES:

```
sage: OverconvergentModularForms(5,2,1/3,base_ring=Qp(5),prec=100).slopes(5)
[0, 2, 5, 6, 9]
sage: OverconvergentModularForms(2,1,1/3,char=DirichletGroup(4,QQ).0).
↪slopes(5)
[0, 2, 4, 6, 8]
```

weight ()

Return the character of self. For overconvergent forms, the weight and the character are unified into the concept of a weight-character, so this returns exactly the same thing as self.character().

EXAMPLES:

```
sage: OverconvergentModularForms(3, 0, 1/2).weight()
0
sage: type(OverconvergentModularForms(3, 0, 1/2).weight())
<class '...weightspace.AlgebraicWeight'>
sage: OverconvergentModularForms(3, 3, 1/2, char=DirichletGroup(3,QQ).0).
↪weight()
(3, 3, [-1])
```

zero ()

Return the zero of this space.

EXAMPLES:

```
sage: K.<w> = Qp(13).extension(x^2-13); M = OverconvergentModularForms(13, 20,
↪ radius=1/2, base_ring=K)
sage: K.zero()
0
```

CHAPTER

TWELVE

ATKIN/HECKE SERIES FOR OVERCONVERGENT MODULAR FORMS

This file contains a function *hecke_series()* to compute the characteristic series $P(t)$ modulo p^m of the Atkin/Hecke operator U_p upon the space of p-adic overconvergent modular forms of level $\Gamma_0(N)$. The input weight `k` can also be a list `klist` of weights which must all be congruent modulo $(p-1)$.

Two optional parameters `modformsring` and `weightbound` can be specified, and in most cases for levels $N > 1$ they can be used to obtain the output more quickly. When $m \leq k - 1$ the output $P(t)$ is also equal modulo p^m to the reverse characteristic polynomial of the Atkin operator U_p on the space of classical modular forms of weight k and level $\Gamma_0(Np)$. In addition, provided $m \leq (k-2)/2$ the output $P(t)$ is equal modulo p^m to the reverse characteristic polynomial of the Hecke operator T_p on the space of classical modular forms of weight k and level $\Gamma_0(N)$. The function is based upon the main algorithm in [Lau2011], and has linear running time in the logarithm of the weight k.

AUTHORS:

- Alan G.B. Lauder (2011-11-10): original implementation.
- David Loeffler (2011-12): minor optimizations in review stage.

EXAMPLES:

The characteristic series of the U_11 operator modulo 11^10 on the space of 11-adic overconvergent modular forms of level 1 and weight 10000:

```
sage: hecke_series(11,1,10000,10)
10009319650*x^4 + 25618839103*x^3 + 6126165716*x^2 + 10120524732*x + 1
```

The characteristic series of the U_5 operator modulo 5^5 on the space of 5-adic overconvergent modular forms of level 3 and weight 1000:

```
sage: hecke_series(5,3,1000,5)
1875*x^6 + 1250*x^5 + 1200*x^4 + 1385*x^3 + 1131*x^2 + 2533*x + 1
```

The characteristic series of the U_7 operator modulo 7^5 on the space of 7-adic overconvergent modular forms of level 5 and weight 1000. Here the optional parameter `modformsring` is set to true:

```
sage: hecke_series(7,5,1000,5,modformsring = True)  # long time (21s on sage.math,␣
↪2012)
12005*x^7 + 10633*x^6 + 6321*x^5 + 6216*x^4 + 5412*x^3 + 4927*x^2 + 4906*x + 1
```

The characteristic series of the U_13 operator modulo 13^5 on the space of 13-adic overconvergent modular forms of level 2 and weight 10000. Here the optional parameter `weightbound` is set to 4:

```
sage: hecke_series(13,2,10000,5,weightbound = 4)  # long time (17s on sage.math, 2012)
325156*x^5 + 109681*x^4 + 188617*x^3 + 220858*x^2 + 269566*x + 1
```

A list containing the characteristic series of the U_23 operator modulo 23^10 on the spaces of 23-adic overconvergent modular forms of level 1 and weights 1000 and 1022, respectively.

```
sage: hecke_series(23,1,[1000,1022],10)
[7204610645852*x^6 + 2117949463923*x^5 + 24152587827773*x^4 + 31270783576528*x^3 +␣
↪30336366679797*x^2
+ 29197235447073*x + 1, 32737396672905*x^4 + 36141830902187*x^3 + 16514246534976*x^2␣
↪+ 38886059530878*x + 1]
```

`sage.modular.overconvergent.hecke_series.`**`complementary_spaces`**(*N*, *p*, *k0*, *n*, *mdash*, *elldashp*, *elldash*, *modformsring*, *bound*)

Returns a list `Ws`, each element in which is a list `Wi` of q-expansions modulo $(p^{\text{mdash}}, q^{\text{elldashp}})$. The list `Wi` is a basis for a choice of complementary space in level $\Gamma_0(N)$ and weight k to the image of weight $k-(p-1)$ forms under multiplication by the Eisenstein series E_{p-1}.

The lists `Wi` play the same role as W_i in Step 2 of Algorithm 2 in [Lau2011]. (The parameters `k0,n,mdash, elldash,elldashp = elldash*p` are defined as in Step 1 of that algorithm when this function is used in *`hecke_series()`*.) However, the complementary spaces are computed in a different manner, combining a suggestion of David Loeffler with one of John Voight. That is, one builds these spaces recursively using random products of forms in low weight, first searching for suitable products modulo (p, q^{elldash}), and then later reconstructing only the required products to the full precision modulo $(p^{\text{mdash}}, q^{elldashp})$. The forms in low weight are chosen from either bases of all forms up to weight `bound` or from a (tentative) generating set for the ring of all modular forms, according to whether `modformsring` is `False` or `True`.

INPUT:

- `N` – positive integer at least 2 and not divisible by p (level).
- `p` – prime at least 5.
- `k0` – integer in range 0 to `p-1`.
- `n,mdash,elldashp,elldash` – positive integers.
- `modformsring` – True or False.
- `bound` – positive (even) integer (ignored if `modformsring` is True).

OUTPUT:

- list of lists of q-expansions modulo `(p^\text{mdash},q^\text{elldashp})`.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import complementary_spaces
sage: complementary_spaces(2,5,0,3,2,5,4,true,6) # random
[[1],
[1 + 23*q + 24*q^2 + 19*q^3 + 7*q^4 + O(q^5)],
[1 + 21*q + 2*q^2 + 17*q^3 + 14*q^4 + O(q^5)],
[1 + 19*q + 9*q^2 + 11*q^3 + 9*q^4 + O(q^5)]]
sage: complementary_spaces(2,5,0,3,2,5,4,false,6) # random
[[1],
[3 + 4*q + 2*q^2 + 12*q^3 + 11*q^4 + O(q^5)],
[2 + 2*q + 14*q^2 + 19*q^3 + 18*q^4 + O(q^5)],
[6 + 8*q + 10*q^2 + 23*q^3 + 4*q^4 + O(q^5)]]
```

`sage.modular.overconvergent.hecke_series.`**`complementary_spaces_modp`**(*N*, *p*, *k0*, *n*, *elldash*, *LWBModp*, *bound*)

Returns a list of lists of lists of lists `[j,a]`. The pairs `[j,a]` encode the choice of the a-th element in the

j-th list of the input LWBModp, i.e., the a-th element in a particular basis modulo (p, q^{elldash}) for the space of modular forms of level $\Gamma_0(N)$ and weight $2(j+1)$. The list [[j_1,a_1],...,[j_r,a_r]] then encodes the product of the r modular forms associated to each [j_i,a_i]; this has weight $k + (p-1)i$ for some $0 \leq i \leq n$; here the i is such that this *list of lists* occurs in the ith list of the output. The ith list of the output thus encodes a choice of basis for the complementary space W_i which occurs in Step 2 of Algorithm 2 in [Lau2011]. The idea is that one searches for this space W_i first modulo (p, q^{elldash}) and then, having found the correct products of generating forms, one can reconstruct these spaces modulo $(p^{\text{mdash}}, q^{\text{elldashp}})$ using the output of this function. (This idea is based upon a suggestion of John Voight.)

INPUT:

- N – positive integer at least 2 and not divisible by p (level).
- p – prime at least 5.
- k0 – integer in range 0 to $p-1$.
- n,elldash – positive integers.
- LWBModp – list of lists of q-expansions over $GF(p)$.
- bound – positive even integer (twice the length of the list LWBModp).

OUTPUT:

- list of list of list of lists.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import random_low_weight_
↪bases, complementary_spaces_modp
sage: LWB = random_low_weight_bases(2,5,2,4,6)
sage: LWBModp = [[f.change_ring(Zmod(5)) for f in x] for x in LWB]
sage: complementary_spaces_modp(2,5,0,3,4,LWBModp,6) # random, indirect doctest
[[[]], [[[0, 0], [0, 0]]], [[[0, 0], [2, 1]]], [[[0, 0], [0, 0], [0, 0], [2, 1]]]]
```

sage.modular.overconvergent.hecke_series.**compute_G**(*p*, *F*)

Given a power series $F \in R[[q]]^\times$, for some ring R, and an integer p, compute the quotient

$$\frac{F(q)}{F(q^p)}.$$

Used by *level1_UpGj()* and by *higher_level_UpGj()*, with F equal to the Eisenstein series E_{p-1}.

INPUT:

- p – integer
- F – power series (with invertible constant term)

OUTPUT:

the power series $F(q)/F(q^p)$, to the same precision as F

EXAMPLES:

```
sage: E = sage.modular.overconvergent.hecke_series.eisenstein_series_qexp(2, 12,␣
↪Zmod(9),normalization="constant")
sage: sage.modular.overconvergent.hecke_series.compute_G(3, E)
1 + 3*q + 3*q^4 + 6*q^7 + O(q^12)
```

sage.modular.overconvergent.hecke_series.**compute_Wi**(*k*, *p*, *h*, *hj*, *E4*, *E6*)

This function computes a list W_i of q-expansions, together with an auxiliary quantity h^j (see below) which is to be used on the next call of this function. (The precision is that of input q-expansions.)

The list W_i is a certain subset of a basis of the modular forms of weight k and level 1. Suppose (a, b) is the pair of non-negative integers with $4a + 6b = k$ and a minimal among such pairs. Then this space has a basis given by

$$\{\Delta^j E_6^{b-2j} E_4^a : 0 \leq j < d\}$$

where d is the dimension.

What this function returns is the subset of the above basis corresponding to $e \leq j < d$ where e is the dimension of the space of modular forms of weight $k - (p - 1)$. This set is a basis for the complement of the image of the weight $k - (p - 1)$ forms under multiplication by E_{p-1}.

This function is used repeatedly in the construction of the Katz expansion basis. Hence considerable care is taken to reuse steps in the computation wherever possible: we keep track of powers of the form $h = \Delta/E_6^2$.

INPUT:

- `k` – non-negative integer.
- `p` – prime at least 5.
- `h` – q-expansion of h (to some finite precision).
- `hj` – q-expansion of h^j where j is the dimension of the space of modular forms of level 1 and weight $k - (p - 1)$ (to same finite precision).
- `E4` – q-expansion of `E_4` (to same finite precision).
- `E6` – q-expansion of `E_6` (to same finite precision).

The Eisenstein series q-expansions should be normalized to have constant term 1.

OUTPUT:

- list of q-expansions (to same finite precision), and q-expansion.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import compute_Wi
sage: p = 17
sage: prec = 10
sage: k = 24
sage: S = Zmod(17^3)
sage: E4 = eisenstein_series_qexp(4, prec, K=S, normalization="constant")
sage: E6 = eisenstein_series_qexp(6, prec, K=S, normalization="constant")
sage: h = delta_qexp(prec, K=S) / E6^2
sage: j = dimension_modular_forms(1, k - (p-1))
sage: hj = h**j
sage: c = compute_Wi(k,p,h,hj,E4,E6); c
([q + 3881*q^2 + 4459*q^3 + 4665*q^4 + 2966*q^5 + 1902*q^6 + 1350*q^7 + 3836*q^8␣
↪+ 1752*q^9 + O(q^10), q^2 + 4865*q^3 + 1080*q^4 + 4612*q^5 + 1343*q^6 + 1689*q^
↪7 + 3876*q^8 + 1381*q^9 + O(q^10)], q^3 + 2952*q^4 + 1278*q^5 + 3225*q^6 +␣
↪1286*q^7 + 589*q^8 + 122*q^9 + O(q^10))
sage: c == ([delta_qexp(10) * E6^2, delta_qexp(10)^2], h**3)
True
```

`sage.modular.overconvergent.hecke_series.`**`compute_elldash`**$(p, N, k0, n)$

Returns the "Sturm bound" for the space of modular forms of level $\Gamma_0(N)$ and weight $k_0 + n(p - 1)$.

See also:

`sturm_bound()`

INPUT:

- `p` – prime.
- `N` – positive integer (level).
- `k0`, `n` - non-negative integers not both zero.

OUTPUT:

- positive integer.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import compute_elldash
sage: compute_elldash(11,5,4,10)
53
```

`sage.modular.overconvergent.hecke_series.`**`ech_form`**`(A, p)`

Returns echelon form of matrix `A` over the ring of integers modulo p^m, for some prime p and $m \geq 1$.

Todo: This should be moved to `sage.matrix.matrix_modn_dense` at some point.

INPUT:

- `A` – matrix over `Zmod(p^m)` for some m.
- `p` - prime p.

OUTPUT:

- matrix over `Zmod(p^m)`.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import ech_form
sage: A = MatrixSpace(Zmod(5**3),3)([1,2,3,4,5,6,7,8,9])
sage: ech_form(A,5)
[1 2 3]
[0 1 2]
[0 0 0]
```

`sage.modular.overconvergent.hecke_series.`**`hecke_series`**`(p, N, klist, m, modformsring=False, weightbound=6)`

Returns the characteristic series modulo p^m of the Atkin operator U_p acting upon the space of p-adic overconvergent modular forms of level $\Gamma_0(N)$ and weight `klist`. The input `klist` may also be a list of weights congruent modulo $(p-1)$, in which case the output is the corresponding list of characteristic series for each k in `klist`; this is faster than performing the computation separately for each k, since intermediate steps in the computation may be reused.

If `modformsring` is True, then for $N > 1$ the algorithm computes at one step `ModularFormsRing(N).generators()`. This will often be faster but the algorithm will default to `modformsring = False` if the generators found are not p-adically integral. Note that `modformsring` is ignored for $N = 1$ and the ring structure of modular forms is *always* used in this case.

When `modformsring` is False and $N > 1$, *weightbound* is a bound set on the weight of generators for a certain subspace of modular forms. The algorithm will often be faster if `weightbound = 4`, but it may fail to terminate for certain exceptional small values of N, when this bound is too small.

The algorithm is based upon that described in [Lau2011].

INPUT:

- p – a prime greater than or equal to 5.
- N – a positive integer not divisible by p.
- klist – either a list of integers congruent modulo $(p-1)$, or a single integer.
- m – a positive integer.
- modformsring – True or False (optional, default False). Ignored if $N = 1$.
- weightbound – a positive even integer (optional, default 6). Ignored if $N = 1$ or modformsring is True.

OUTPUT:

Either a list of polynomials or a single polynomial over the integers modulo p^m.

EXAMPLES:

```
sage: hecke_series(5,7,10000,5, modformsring = True) # long time (3.4s)
250*x^6 + 1825*x^5 + 2500*x^4 + 2184*x^3 + 1458*x^2 + 1157*x + 1
sage: hecke_series(7,3,10000,3, weightbound = 4)
196*x^4 + 294*x^3 + 197*x^2 + 341*x + 1
sage: hecke_series(19,1,[10000,10018],5)
[1694173*x^4 + 2442526*x^3 + 1367943*x^2 + 1923654*x + 1,
130321*x^4 + 958816*x^3 + 2278233*x^2 + 1584827*x + 1]
```

Check that silly weights are handled correctly:

```
sage: hecke_series(5, 7, [2, 3], 5)
Traceback (most recent call last):
...
ValueError: List of weights must be all congruent modulo p-1 = 4, but given list␣
↪contains 2 and 3 which are not congruent
sage: hecke_series(5, 7, [3], 5)
[1]
sage: hecke_series(5, 7, 3, 5)
1
```

sage.modular.overconvergent.hecke_series.**hecke_series_degree_bound**(p, N, k, m)

Returns the Wan bound on the degree of the characteristic series of the Atkin operator on p-adic overconvergent modular forms of level $\Gamma_0(N)$ and weight k when reduced modulo p^m. This bound depends only upon p, $k \pmod{p-1}$, and N. It uses Lemma 3.1 in [Wan1998].

INPUT:

- p – prime at least 5.
- N – positive integer not divisible by p.
- k – even integer.
- m – positive integer.

OUTPUT:

A non-negative integer.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import hecke_series_degree_
↪bound
sage: hecke_series_degree_bound(13,11,100,5)
39
```

sage.modular.overconvergent.hecke_series.**higher_level_UpGj**(*p*, *N*, *klist*, *m*, *modformsring*, *bound*, *extra_data=False*)

Return a list `[A_k]` of square matrices over `IntegerRing(p^m)` parameterised by the weights k in `klist`.

The matrix A_k is the finite square matrix which occurs on input p, k, N and m in Step 6 of Algorithm 2 in [Lau2011].

Notational change from paper: In Step 1 following Wan we defined j by $k = k_0 + j(p-1)$ with $0 \leq k_0 < p-1$. Here we replace j by `kdiv` so that we may use j as a column index for matrices.)

INPUT:

- `p` – prime at least 5.
- `N` – integer at least 2 and not divisible by p (level).
- `klist` – list of integers all congruent modulo (p-1) (the weights).
- `m` – positive integer.
- `modformsring` – `True` or `False`.
- `bound` – (even) positive integer.
- `extra_data` – (default: `False`) boolean.

OUTPUT:

- list of square matrices. If `extra_data` is `True`, return also extra intermediate data, namely the matrix E in [Lau2011] and the integers `elldash` and `mdash`.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import higher_level_UpGj
sage: higher_level_UpGj(5,3,[4],2,true,6)
[
[ 1  0  0  0  0  0]
[ 0  1  0  0  0  0]
[ 0  7  0  0  0  0]
[ 0  5 10 20  0  0]
[ 0  7 20  0 20  0]
[ 0  1 24  0 20  0]
]
sage: len(higher_level_UpGj(5,3,[4],2,true,6,extra_data=True))
4
```

sage.modular.overconvergent.hecke_series.**higher_level_katz_exp**(*p*, *N*, *k0*, *m*, *mdash*, *elldash*, *elldashp*, *modformsring*, *bound*)

Returns a matrix e of size `ell x elldashp` over the integers modulo p^{mdash}, and the Eisenstein series $E_{p-1} = 1 + \dots \bmod (p^{\text{mdash}}, q^{\text{elldashp}})$. The matrix e contains the coefficients of the elements $e_{i,s}$ in the Katz expansions basis in Step 3 of Algorithm 2 in [Lau2011] when one takes as input to that algorithm p,‘N‘,‘m‘ and k and define `k0`, `mdash`, `n`, `elldash`, `elldashp = ell*dashp` as in Step 1.

INPUT:

- `p` – prime at least 5.
- `N` – positive integer at least 2 and not divisible by p (level).
- `k0` – integer in range 0 to p-1.
- `m,mdash,elldash,elldashp` – positive integers.
- `modformsring` – True or False.
- `bound` – positive (even) integer.

OUTPUT:

- matrix and q-expansion.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import higher_level_katz_exp
sage: e,Ep1 = higher_level_katz_exp(5,2,0,1,2,4,20,true,6)
sage: e
[ 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  1 18 23 19  6  9  9 17  7  3 17 12  8 22  8 11 19  1  5]
[ 0  0  1 11 20 16  0  8  4  0 18 15 24  6 15 23  5 18  7 15]
[ 0  0  0  1  4 16 23 13  6  5 23  5  2 16  4 18 10 23  5 15]
sage: Ep1
1 + 15*q + 10*q^2 + 20*q^3 + 20*q^4 + 15*q^5 + 5*q^6 + 10*q^7 +
5*q^9 + 10*q^10 + 5*q^11 + 10*q^12 + 20*q^13 + 15*q^14 + 20*q^15 + 15*q^16 +
10*q^17 + 20*q^18 + O(q^20)
```

`sage.modular.overconvergent.hecke_series.`**`is_valid_weight_list`**(*klist*, *p*)

This function checks that `klist` is a nonempty list of integers all of which are congruent modulo $(p-1)$. Otherwise, it will raise a ValueError.

INPUT:

- `klist` – list of integers.
- `p` – prime.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import is_valid_weight_list
sage: is_valid_weight_list([10,20,30],11)
sage: is_valid_weight_list([-3, 1], 5)
sage: is_valid_weight_list([], 3)
Traceback (most recent call last):
...
ValueError: List of weights must be non-empty
sage: is_valid_weight_list([-3, 2], 5)
Traceback (most recent call last):
...
ValueError: List of weights must be all congruent modulo p-1 = 4, but given list␣
↪contains -3 and 2 which are not congruent
```

`sage.modular.overconvergent.hecke_series.`**`katz_expansions`**(*k0*, *p*, *ellp*, *mdash*, *n*)

Returns a list e of q-expansions, and the Eisenstein series $E_{p-1} = 1 + \ldots$, all modulo $(p^{\text{mdash}}, q^{\text{ellp}})$. The list e contains the elements $e_{i,s}$ in the Katz expansions basis in Step 3 of Algorithm 1 in [Lau2011] when one takes as input to that algorithm p,m and k and define `k0`, `mdash`, n, `ellp = ell*p` as in Step 1.

INPUT:

- `k0` – integer in range 0 to p-1.
- `p` – prime at least 5.
- `ellp,mdash,n` – positive integers.

OUTPUT:

- list of q-expansions and the Eisenstein series E_{p-1} modulo $(p^{\text{mdash}}, q^{\text{ellp}})$.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import katz_expansions
sage: katz_expansions(0,5,10,3,4)
([1 + O(q^10), q + 6*q^2 + 27*q^3 + 98*q^4 + 65*q^5 + 37*q^6 + 81*q^7 + 85*q^8 +
↪62*q^9 + O(q^10)],
1 + 115*q + 35*q^2 + 95*q^3 + 20*q^4 + 115*q^5 + 105*q^6 + 60*q^7 + 25*q^8 + 55*q^
↪9 + O(q^10))
```

sage.modular.overconvergent.hecke_series.**level1_UpGj**(*p*, *klist*, *m*, *extra_data=False*)

Return a list $[A_k]$ of square matrices over `IntegerRing(p^m)` parameterised by the weights k in `klist`.

The matrix A_k is the finite square matrix which occurs on input p, k and m in Step 6 of Algorithm 1 in [Lau2011].

Notational change from paper: In Step 1 following Wan we defined j by $k = k_0 + j(p-1)$ with $0 \leq k_0 < p-1$. Here we replace j by `kdiv` so that we may use j as a column index for matrices.

INPUT:

- `p` – prime at least 5.
- `klist` – list of integers congruent modulo $(p-1)$ (the weights).
- `m` – positive integer.
- `extra_data` – (default: `False`) boolean

OUTPUT:

- list of square matrices. If `extra_data` is `True`, return also extra intermediate data, namely the matrix E in [Lau2011] and the integers `elldash` and `mdash`.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import level1_UpGj
sage: level1_UpGj(7,[100],5)
[
[    1   980  4802     0     0]
[    0 13727 14406     0     0]
[    0 13440  7203     0     0]
[    0  1995  4802     0     0]
[    0  9212 14406     0     0]
]
sage: len(level1_UpGj(7,[100],5,extra_data=True))
4
```

sage.modular.overconvergent.hecke_series.**low_weight_bases**(*N*, *p*, *m*, *NN*, *weightbound*)

Return a list of integral bases of modular forms of level N and (even) weight at most `weightbound`, as q-expansions modulo (p^m, q^{NN}).

These forms are obtained by reduction mod p^m from an integral basis in Hermite normal form (so they are not necessarily in reduced row echelon form mod p^m, but they are not far off).

INPUT:

- N – positive integer (level).
- p – prime.
- m, NN – positive integers.
- weightbound – (even) positive integer.

OUTPUT:

- list of lists of q-expansions modulo (p^m, q^{NN}).

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import low_weight_bases
sage: low_weight_bases(2,5,3,5,6)
[[1 + 24*q + 24*q^2 + 96*q^3 + 24*q^4 + O(q^5)],
[1 + 115*q^2 + 35*q^4 + O(q^5), q + 8*q^2 + 28*q^3 + 64*q^4 + O(q^5)],
[1 + 121*q^2 + 118*q^4 + O(q^5), q + 32*q^2 + 119*q^3 + 24*q^4 + O(q^5)]]
```

sage.modular.overconvergent.hecke_series.**low_weight_generators**(*N*, *p*, *m*, *NN*)

Returns a list of lists of modular forms, and an even natural number. The first output is a list of lists of modular forms reduced modulo (p^m, q^{NN}) which generate the $(\mathbf{Z}/p^m\mathbf{Z})$-algebra of mod p^m modular forms of weight at most 8, and the second output is the largest weight among the forms in the generating set.

We (Alan Lauder and David Loeffler, the author and reviewer of this patch) conjecture that forms of weight at most 8 are always sufficient to generate the algebra of mod p^m modular forms of all weights. (We believe 6 to be sufficient, and we can prove that 4 is sufficient when there are no elliptic points, but using weights up to 8 acts as a consistency check.)

INPUT:

- N – positive integer (level).
- p – prime.
- m, NN – positive integers.

OUTPUT:

a tuple consisting of:

- a list of lists of q-expansions modulo (p^m, q^{NN}),
- an even natural number (twice the length of the list).

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import low_weight_generators
sage: low_weight_generators(3,7,3,10)
([[1 + 12*q + 36*q^2 + 12*q^3 + 84*q^4 + 72*q^5 + 36*q^6 + 96*q^7 + 180*q^8 +␣
↪12*q^9 + O(q^10)],
[1 + 240*q^3 + 102*q^6 + 203*q^9 + O(q^10)],
[1 + 182*q^3 + 175*q^6 + 161*q^9 + O(q^10)]], 6)
sage: low_weight_generators(11,5,3,10)
([[1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 +␣
↪O(q^10),
q + 123*q^2 + 124*q^3 + 2*q^4 + q^5 + 2*q^6 + 123*q^7 + 123*q^9 + O(q^10)],
[q + 116*q^4 + 115*q^5 + 102*q^6 + 121*q^7 + 96*q^8 + 106*q^9 + O(q^10)]], 4)
```

sage.modular.overconvergent.hecke_series.**random_low_weight_bases**(*N*, *p*, *m*, *NN*, *weightbound*)

Returns list of random integral bases of modular forms of level N and (even) weight at most weightbound with coefficients reduced modulo (p^m, q^{NN}).

INPUT:

- N – positive integer (level).
- p – prime.
- m, NN – positive integers.
- weightbound – (even) positive integer.

OUTPUT:

- list of lists of q-expansions modulo (p^m, q^{NN}).

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import random_low_weight_bases
sage: S = random_low_weight_bases(3,7,2,5,6); S # random
[[4 + 48*q + 46*q^2 + 48*q^3 + 42*q^4 + O(q^5)],
[3 + 5*q + 45*q^2 + 22*q^3 + 22*q^4 + O(q^5),
1 + 3*q + 27*q^2 + 27*q^3 + 23*q^4 + O(q^5)],
[2*q + 4*q^2 + 16*q^3 + 48*q^4 + O(q^5),
2 + 6*q + q^2 + 3*q^3 + 43*q^4 + O(q^5),
1 + 2*q + 6*q^2 + 14*q^3 + 4*q^4 + O(q^5)]]
sage: S[0][0].parent()
Power Series Ring in q over Ring of integers modulo 49
sage: S[0][0].prec()
5
```

sage.modular.overconvergent.hecke_series.**random_new_basis_modp**(*N*, *p*, *k*, *LWBModp*, *TotalBasisModp*, *elldash*, *bound*)

Returns NewBasisCode. Here $NewBasisCode$ is a list of lists of lists [j,a]. This encodes a choice of basis for the ith complementary space W_i, as explained in the documentation for the function *complementary_spaces_modp()*.

INPUT:

- N – positive integer at least 2 and not divisible by p (level).
- p – prime at least 5.
- k – non-negative integer.
- LWBModp – list of list of q-expansions modulo (p, q^{elldash}).
- TotalBasisModp – matrix over GF(p).
- elldash - positive integer.
- bound – positive even integer (twice the length of the list LWBModp).

OUTPUT:

- A list of lists of lists [j,a].

Note: As well as having a non-trivial return value, this function also modifies the input matrix TotalBasisModp.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import random_low_weight_
↪bases, complementary_spaces_modp
sage: LWB = random_low_weight_bases(2,5,2,4,6)
sage: LWBModp = [ [f.change_ring(GF(5)) for f in x] for x in LWB]
sage: complementary_spaces_modp(2,5,2,3,4,LWBModp,4) # random, indirect doctest
[[[[0, 0]]], [[[0, 0], [1, 1]]], [[[0, 0], [1, 0], [1, 1]]], [[[0, 0], [1, 0], [1,
↪ 1], [1, 1]]]]
```

sage.modular.overconvergent.hecke_series.**random_solution**(*B*, *K*)

Returns a random solution in non-negative integers to the equation $a_1 + 2a_2 + 3a_3 + ... + Ba_B = K$, using a greedy algorithm.

Note that this is *much* faster than using `WeightedIntegerVectors.random_element()`.

INPUT:

- `B`, `K` – non-negative integers.

OUTPUT:

- list.

EXAMPLES:

```
sage: from sage.modular.overconvergent.hecke_series import random_solution
sage: random_solution(5,10)
[1, 1, 1, 1, 0]
```

CHAPTER

THIRTEEN

MODULE OF SUPERSINGULAR POINTS

The module of divisors on the modular curve $X_0(N)$ over F_p supported at supersingular points.

AUTHORS:

- William Stein
- David Kohel
- Iftikhar Burhanuddin

EXAMPLES:

```
sage: x = SupersingularModule(389)
sage: m = x.T(2).matrix()
sage: a = m.change_ring(GF(97))
sage: D = a.decomposition()
sage: D[:3]
[
(Vector space of degree 33 and dimension 1 over Finite Field of size 97
Basis matrix:
[ 0  0  0  1 96 96  1  0 95  1  1  1  1 95  2 96  0  0 96  0 96  0 96  2 96 96  0  1 ␣
↪0  2  1 95  0], True),
(Vector space of degree 33 and dimension 1 over Finite Field of size 97
Basis matrix:
[ 0  1 96 16 75 22 81  0  0 17 17 80 80  0  0 74 40  1 16 57 23 96 81  0 74 23  0 24 ␣
↪0  0 73  0  0], True),
(Vector space of degree 33 and dimension 1 over Finite Field of size 97
Basis matrix:
[ 0  1 96 90 90  7  7  0  0 91  6  6 91  0  0 91  0 13  7  0  6 84 90  0  6 91  0 90 ␣
↪0  0  7  0  0], True)
]
sage: len(D)
9
```

We compute a Hecke operator on a space of huge dimension!:

```
sage: X = SupersingularModule(next_prime(10000))
sage: t = X.T(2).matrix()          # long time (21s on sage.math, 2011)
sage: t.nrows()                    # long time
835
```

`sage.modular.ssmod.ssmod.`**`Phi2_quad`**(*J3*, *ssJ1*, *ssJ2*)

Return a certain quadratic polynomial over a finite field in indeterminate J3.

The roots of the polynomial along with ssJ1 are the neighboring/2-isogenous supersingular j-invariants of ssJ2.

INPUT:

- `J3` – indeterminate of a univariate polynomial ring defined over a finite field with p^2 elements where p is a prime number
- `ssJ2`, `ssJ2` – supersingular j-invariants over the finite field

OUTPUT:

- polynomial – defined over the finite field

EXAMPLES:

The following code snippet produces a factor of the modular polynomial $\Phi_2(x, j_{in})$, where j_{in} is a supersingular j-invariant defined over the finite field with 37^2 elements:

```
sage: F = GF(37^2, 'a')
sage: X = PolynomialRing(F, 'x').gen()
sage: j_in = supersingular_j(F)
sage: poly = sage.modular.ssmod.ssmod.Phi_polys(2,X,j_in)
sage: poly.roots()
[(8, 1), (27*a + 23, 1), (10*a + 20, 1)]
sage: sage.modular.ssmod.ssmod.Phi2_quad(X, F(8), j_in)
x^2 + 31*x + 31
```

Note: Given a root (j1,j2) to the polynomial $Phi_2(J1, J2)$, the pairs (j2,j3) not equal to (j2,j1) which solve $Phi_2(j2, j3)$ are roots of the quadratic equation:

J3^2 + (-j2^2 + 1488*j2 + (j1 - 162000))*J3 + (-j1 + 1488)*j2^2 + (1488*j1 + 40773375)*j2 + j1^2 - 162000*j1 + 8748000000

This will be of use to extend the 2-isogeny graph, once the initial three roots are determined for $Phi_2(J1, J2)$.

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

`sage.modular.ssmod.ssmod.`**`Phi_polys`**(*L*, *x*, *j*)

Return a certain polynomial of degree $L + 1$ in the indeterminate x over a finite field.

The roots of the **modular** polynomial $\Phi(L, x, j)$ are the L-isogenous supersingular j-invariants of j.

INPUT:

- `L` – integer
- `x` – indeterminate of a univariate polynomial ring defined over a finite field with p^2 elements, where p is a prime number
- `j` – supersingular j-invariant over the finite field

OUTPUT:

- polynomial – defined over the finite field

EXAMPLES:

The following code snippet produces the modular polynomial $\Phi_L(x, j_{in})$, where j_{in} is a supersingular j-invariant defined over the finite field with 7^2 elements:

```
sage: F = GF(7^2, 'a')
sage: X = PolynomialRing(F, 'x').gen()
sage: j_in = supersingular_j(F)
sage: sage.modular.ssmod.ssmod.Phi_polys(2,X,j_in)
x^3 + 3*x^2 + 3*x + 1
sage: sage.modular.ssmod.ssmod.Phi_polys(3,X,j_in)
x^4 + 4*x^3 + 6*x^2 + 4*x + 1
sage: sage.modular.ssmod.ssmod.Phi_polys(5,X,j_in)
x^6 + 6*x^5 + x^4 + 6*x^3 + x^2 + 6*x + 1
sage: sage.modular.ssmod.ssmod.Phi_polys(7,X,j_in)
x^8 + x^7 + x + 1
sage: sage.modular.ssmod.ssmod.Phi_polys(11,X,j_in)
x^12 + 5*x^11 + 3*x^10 + 3*x^9 + 5*x^8 + x^7 + x^5 + 5*x^4 + 3*x^3 + 3*x^2 + 5*x␣
↪+ 1
sage: sage.modular.ssmod.ssmod.Phi_polys(13,X,j_in)
x^14 + 2*x^7 + 1
```

class sage.modular.ssmod.ssmod.**SupersingularModule**(*prime=2*, *level=1*, *base_ring=Integer Ring*)

Bases: sage.modular.hecke.module.HeckeModule_free_module

The module of supersingular points in a given characteristic, with given level structure.

The characteristic must not divide the level.

Note: Currently, only level 1 is implemented.

EXAMPLES:

```
sage: S = SupersingularModule(17)
sage: S
Module of supersingular points on X_0(1)/F_17 over Integer Ring
sage: S = SupersingularModule(16)
Traceback (most recent call last):
...
ValueError: the argument prime must be a prime number
sage: S = SupersingularModule(prime=17, level=34)
Traceback (most recent call last):
...
ValueError: the argument level must be coprime to the argument prime
sage: S = SupersingularModule(prime=17, level=5)
Traceback (most recent call last):
...
NotImplementedError: supersingular modules of level > 1 not yet implemented
```

dimension()

Return the dimension of the space of modular forms of weight 2 and level equal to the level associated to self.

INPUT:

- self – SupersingularModule object

OUTPUT:

- integer – dimension, nonnegative

EXAMPLES:

```
sage: S = SupersingularModule(7)
sage: S.dimension()
1

sage: S = SupersingularModule(15073)
sage: S.dimension()
1256

sage: S = SupersingularModule(83401)
sage: S.dimension()
6950
```

Note: The case of level > 1 has not yet been implemented.

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

free_module()

EXAMPLES:

```
sage: X = SupersingularModule(37)
sage: X.free_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

This illustrates the fix at trac ticket #4306:

```
sage: X = SupersingularModule(389)
sage: X.basis()
((1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪0, 0, 0, 0, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,␣
↪0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪1, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 1, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 1, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 1, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 1, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 1, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 1, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 1))
```

hecke_matrix(L)

Return the L^{th} Hecke matrix.

INPUT:

- `self` – SupersingularModule object
- `L` – integer, positive

OUTPUT:

- matrix – sparse integer matrix

EXAMPLES:

This example computes the action of the Hecke operator T_2 on the module of supersingular points on $X_0(1)/F_{37}$:

```
sage: S = SupersingularModule(37)
sage: M = S.hecke_matrix(2)
sage: M
[1 1 1]
[1 0 2]
[1 2 0]
```

This example computes the action of the Hecke operator T_3 on the module of supersingular points on $X_0(1)/F_{67}$:

```
sage: S = SupersingularModule(67)
sage: M = S.hecke_matrix(3)
sage: M
[0 0 0 0 2 2]
[0 0 1 1 1 1]
[0 1 0 2 0 1]
[0 1 2 0 1 0]
[1 1 0 1 0 1]
[1 1 1 0 1 0]
```

Note: The first list — list_j — returned by the supersingular_points function are the rows *and* column indexes of the above hecke matrices and its ordering should be kept in mind when interpreting these matrices.

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

level()

This function returns the level associated to `self`.

INPUT:

- `self` – SupersingularModule object

OUTPUT:

- integer – the level, positive

EXAMPLES:

```
sage: S = SupersingularModule(15073)
sage: S.level()
1
```

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

prime()

Return the characteristic of the finite field associated to `self`.

INPUT:

- `self` – SupersingularModule object

OUTPUT:

- integer – characteristic, positive

EXAMPLES:

```
sage: S = SupersingularModule(19)
sage: S.prime()
19
```

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

rank()

Return the dimension of the space of modular forms of weight 2 and level equal to the level associated to `self`.

INPUT:

- `self` – SupersingularModule object

OUTPUT:

- integer – dimension, nonnegative

EXAMPLES:

```
sage: S = SupersingularModule(7)
sage: S.dimension()
1

sage: S = SupersingularModule(15073)
sage: S.dimension()
1256

sage: S = SupersingularModule(83401)
sage: S.dimension()
6950
```

Note: The case of level > 1 has not yet been implemented.

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

supersingular_points()

Compute the supersingular j-invariants over the finite field associated to `self`.

INPUT:

- `self` – SupersingularModule object

OUTPUT:

- **list_j, dict_j – list_j is the list of supersingular** j-invariants, dict_j is a dictionary with these j-invariants as keys and their indexes as values. The latter is used to speed up j-invariant look-up. The indexes are based on the order of their *discovery*.

EXAMPLES:

The following examples calculate supersingular j-invariants over finite fields with characteristic 7, 11 and 37:

```
sage: S = SupersingularModule(7)
sage: S.supersingular_points()
([6], {6: 0})

sage: S = SupersingularModule(11)
sage: S.supersingular_points()[0]
[1, 0]

sage: S = SupersingularModule(37)
sage: S.supersingular_points()[0]
[8, 27*a + 23, 10*a + 20]
```

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

upper_bound_on_elliptic_factors (*p=None*, *ellmax=2*)

Return an upper bound (provably correct) on the number of elliptic curves of conductor equal to the level of this supersingular module.

INPUT:

- `p` – (default: 997) prime to work modulo

ALGORITHM: Currently we only use T_2. Function will be extended to use more Hecke operators later.

The prime p is replaced by the smallest prime that does not divide the level.

EXAMPLES:

```
sage: SupersingularModule(37).upper_bound_on_elliptic_factors()
2
```

(There are 4 elliptic curves of conductor 37, but only 2 isogeny classes.)

weight ()

Return the weight associated to `self`.

INPUT:

- `self` – SupersingularModule object

OUTPUT:

- integer – weight, positive

EXAMPLES:

```
sage: S = SupersingularModule(19)
sage: S.weight()
2
```

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

sage.modular.ssmod.ssmod.**dimension_supersingular_module**(*prime*, *level=1*)

Return the dimension of the Supersingular module, which is equal to the dimension of the space of modular forms of weight 2 and conductor equal to `prime` times `level`.

INPUT:

- `prime` – integer, prime
- `level` – integer, positive

OUTPUT:

- dimension – integer, nonnegative

EXAMPLES:

The code below computes the dimensions of Supersingular modules with level=1 and prime = 7, 15073 and 83401:

```
sage: dimension_supersingular_module(7)
1

sage: dimension_supersingular_module(15073)
1256

sage: dimension_supersingular_module(83401)
6950
```

Note: The case of level > 1 has not been implemented yet.

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin - burhanud@usc.edu

sage.modular.ssmod.ssmod.**supersingular_D**(*prime*)

Return a fundamental discriminant D of an imaginary quadratic field, where the given prime does not split.

See Silverman's Advanced Topics in the Arithmetic of Elliptic Curves, page 184, exercise 2.30(d).

INPUT:

- prime – integer, prime

OUTPUT:

- D – integer, negative

EXAMPLES:

These examples return *supersingular discriminants* for 7, 15073 and 83401:

```
sage: supersingular_D(7)
-4

sage: supersingular_D(15073)
-15

sage: supersingular_D(83401)
-7
```

AUTHORS:

- David Kohel - kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin - burhanud@usc.edu

`sage.modular.ssmod.ssmod.`**`supersingular_j`**`(`*FF*`)`

Return a supersingular j-invariant over the finite field FF.

INPUT:

- `FF` – finite field with p^2 elements, where p is a prime number

OUTPUT:

- finite field element – a supersingular j-invariant defined over the finite field FF

EXAMPLES:

The following examples calculate supersingular j-invariants for a few finite fields:

```
sage: supersingular_j(GF(7^2, 'a'))
6
```

Observe that in this example the j-invariant is not defined over the prime field:

```
sage: supersingular_j(GF(15073^2, 'a'))
4443*a + 13964
sage: supersingular_j(GF(83401^2, 'a'))
67977
```

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

CHAPTER

FOURTEEN

BRANDT MODULES

14.1 Introduction

This tutorial outlines the construction of Brandt modules in Sage. The importance of this construction is that it provides us with a method to compute modular forms on $\Gamma_0(N)$ as outlined in Pizer's paper [Piz1980]. In fact there exists a non-canonical Hecke algebra isomorphism between the Brandt modules and a certain subspace of $S_2(\Gamma_0(pM))$ which contains all the newforms.

The Brandt module is the free abelian group on right ideal classes of a quaternion order together with a natural Hecke action determined by Brandt matrices.

14.1.1 Quaternion Algebras

A quaternion algebra over $\mathbf{Q}$ is a central simple algebra of dimension 4 over $\mathbf{Q}$. Such an algebra A is said to be ramified at a place v of $\mathbf{Q}$ if and only if $A_v = A \otimes \mathbf{Q}_v$ is a division algebra. Otherwise A is said to be split at v.

`A = QuaternionAlgebra(p)` returns the quaternion algebra A over $\mathbf{Q}$ ramified precisely at the places p and ∞.

`A = QuaternionAlgebra(k,a,b)` returns a quaternion algebra with basis $\{1, i, j, j\}$ over $\mathbb{K}$ such that $i^2 = a$, $j^2 = b$ and $ij = k$.

An order R in a quaternion algebra is a 4-dimensional lattice on A which is also a subring containing the identity.

`R = A.maximal_order()` returns a maximal order R in the quaternion algebra A.

An Eichler order $\mathcal{O}$ in a quaternion algebra is the intersection of two maximal orders. The level of $\mathcal{O}$ is its index in any maximal order containing it.

`O = A.order_of_level_N` returns an Eichler order $\mathcal{O}$ in A of level N where p does not divide N.

A right $\mathcal{O}$-ideal I is a lattice on A such that $I_p = a_p\mathcal{O}$ (for some $a_p \in A_p^*$) for all $p < \infty$. Two right $\mathcal{O}$-ideals I and J are said to belong to the same class if $I = aJ$ for some $a \in A^*$. (Left $\mathcal{O}$-ideals are defined in a similar fashion.)

The right order of I is defined to be the set of elements in A which fix I under right multiplication.

`right_order(R, basis)` returns the right ideal of I in R given a basis for the right ideal I contained in the maximal order R.

`ideal_classes(self)` returns a tuple of all right ideal classes in self which, for the purpose of constructing the Brandt module B(p,M), is taken to be an Eichler order of level M.

The implementation of this method is especially interesting. It depends on the construction of a Hecke module defined as a free abelian group on right ideal classes of a quaternion algebra with the following action

$$T_n[I] = \sum_{\phi} [J]$$

where $(n, pM) = 1$ and the sum is over cyclic $\mathcal{O}$-module homomorphisms $\phi : I \rightarrow J$ of degree n up to isomorphism of J. Equivalently one can sum over the inclusions of the submodules $J \rightarrow n^{-1}I$. The rough idea is to start with the trivial ideal class containing the order $\mathcal{O}$ itself. Using the method `cyclic_submodules(self, I, p)` one computes $T_p([\mathcal{O}])$ for some prime integer p not dividing the level of the order $\mathcal{O}$. Apply this method repeatedly and test for equivalence among resulting ideals. A theorem of Serre asserts that one gets a complete set of ideal class representatives after a finite number of repetitions.

One can prove that two ideals I and J are equivalent if and only if there exists an element $\alpha \in I\overline{J}$ such $N(\alpha) = N(I)N(J)$.

`is_equivalent(I,J)` returns true if I and J are equivalent. This method first compares the theta series of I and J. If they are the same, it computes the theta series of the lattice $I\bar{(}J)$. It returns true if the n^{th} coefficient of this series is nonzero where $n = N(J)N(I)$.

The theta series of a lattice L over the quaternion algebra A is defined as

$$\theta_L(q) = \sum_{x \in L} q^{\frac{N(x)}{N(L)}}$$

`L.theta_series(T,q)` returns a power series representing $\theta_L(q)$ up to a precision of $\mathcal{O}(q^{T+1})$.

14.1.2 Hecke Structure

The Hecke structure defined on the Brandt module is given by the Brandt matrices which can be computed using the definition of the Hecke operators given earlier.

`hecke_matrix_from_defn(self,n)` returns the matrix of the n-th Hecke operator $B_0(n)$ acting on self, computed directly from the definition.

However, one can efficiently compute Brandt matrices using theta series. In fact, let $\{I_1, ..., I_h\}$ be a set of right $\mathcal{O}$-ideal class representatives. The (i,j) entry in the Brandt matrix $B_0(n)$ is the product of the n^{th} coefficient in the theta series of the lattice $I_i\overline{I_j}$ and the first coefficient in the theta series of the lattice $I_i\overline{I_i}$.

`compute_hecke_matrix_brandt(self,n)` returns the n-th Hecke matrix, computed using theta series.

EXAMPLES:

```
sage: B = BrandtModule(23)

sage: B.maximal_order()
Order of Quaternion Algebra (-1, -23) with base ring Rational Field with basis (1/2 +
↪1/2*j, 1/2*i + 1/2*k, j, k)

sage: B.right_ideals()
(Fractional ideal (2 + 2*j, 2*i + 2*k, 4*j, 4*k), Fractional ideal (2 + 2*j, 2*i +
↪6*k, 8*j, 8*k), Fractional ideal (2 + 10*j + 8*k, 2*i + 8*j + 6*k, 16*j, 16*k))

sage: B.hecke_matrix(2)
[1 2 0]
[1 1 1]
[0 3 0]

sage: B.brandt_series(3)
[1/4 + q + q^2 + O(q^3)      1/4 + q^2 + O(q^3)           1/4 + O(q^3)]
[  1/2 + 2*q^2 + O(q^3) 1/2 + q + q^2 + O(q^3)   1/2 + 3*q^2 + O(q^3)]
[           1/6 + O(q^3)      1/6 + q^2 + O(q^3)       1/6 + q + O(q^3)]
```

REFERENCES:

- [Piz1980]
- [Koh2000]

14.1.3 Further Examples

We decompose a Brandt module over both $\mathbf{Z}$ and $\mathbf{Q}$.

```
sage: B = BrandtModule(43, base_ring=ZZ); B
Brandt module of dimension 4 of level 43 of weight 2 over Integer Ring
sage: D = B.decomposition()
sage: D
[
Subspace of dimension 1 of Brandt module of dimension 4 of level 43 of weight 2 over␣
↪Integer Ring,
Subspace of dimension 1 of Brandt module of dimension 4 of level 43 of weight 2 over␣
↪Integer Ring,
Subspace of dimension 2 of Brandt module of dimension 4 of level 43 of weight 2 over␣
↪Integer Ring
]
sage: D[0].basis()
((0, 0, 1, -1),)
sage: D[1].basis()
((1, 2, 2, 2),)
sage: D[2].basis()
((1, 1, -1, -1), (0, 2, -1, -1))
sage: B = BrandtModule(43, base_ring=QQ); B
Brandt module of dimension 4 of level 43 of weight 2 over Rational Field
sage: B.decomposition()[2].basis()
((1, 0, -1/2, -1/2), (0, 1, -1/2, -1/2))
```

AUTHORS:

- Jon Bober
- Alia Hamieh
- Victoria de Quehen
- William Stein
- Gonzalo Tornaria

`sage.modular.quatalg.brandt.`**`BrandtModule`**(*N*, *M=1*, *weight=2*, *base_ring=Rational Field*, *use_cache=True*)

Return the Brandt module of given weight associated to the prime power p^r and integer M, where p and M are coprime.

INPUT:

- N – a product of primes with odd exponents
- M – an integer coprime to q (default: 1)
- `weight` – an integer that is at least 2 (default: 2)
- `base_ring` – the base ring (default: `QQ`)
- `use_cache` – whether to use the cache (default: `True`)

OUTPUT:

a Brandt module

EXAMPLES:

```
sage: BrandtModule(17)
Brandt module of dimension 2 of level 17 of weight 2 over Rational Field
sage: BrandtModule(17,15)
Brandt module of dimension 32 of level 17*15 of weight 2 over Rational Field
sage: BrandtModule(3,7)
Brandt module of dimension 2 of level 3*7 of weight 2 over Rational Field
sage: BrandtModule(3,weight=2)
Brandt module of dimension 1 of level 3 of weight 2 over Rational Field
sage: BrandtModule(11, base_ring=ZZ)
Brandt module of dimension 2 of level 11 of weight 2 over Integer Ring
sage: BrandtModule(11, base_ring=QQbar)
Brandt module of dimension 2 of level 11 of weight 2 over Algebraic Field
```

The `use_cache` option determines whether the Brandt module returned by this function is cached:

```
sage: BrandtModule(37) is BrandtModule(37)
True
sage: BrandtModule(37,use_cache=False) is BrandtModule(37,use_cache=False)
False
```

class sage.modular.quatalg.brandt.**BrandtModuleElement**(*parent*, *x*)

Bases: `sage.modular.hecke.element.HeckeModuleElement`

EXAMPLES:

```
sage: B = BrandtModule(37)
sage: x = B([1,2,3]); x
(1, 2, 3)
sage: parent(x)
Brandt module of dimension 3 of level 37 of weight 2 over Rational Field
```

monodromy_pairing(*x*)

Return the monodromy pairing of `self` and `x`.

EXAMPLES:

```
sage: B = BrandtModule(5,13)
sage: B.monodromy_weights()
(1, 3, 1, 1, 1, 3)
sage: (B.0 + B.1).monodromy_pairing(B.0 + B.1)
4
```

class sage.modular.quatalg.brandt.**BrandtModule_class**(*N*, *M*, *weight*, *base_ring*)

Bases: `sage.modular.hecke.ambient_module.AmbientHeckeModule`

A Brandt module.

EXAMPLES:

```
sage: BrandtModule(3, 10)
Brandt module of dimension 4 of level 3*10 of weight 2 over Rational Field
```

Element

alias of *BrandtModuleElement*

M()

Return the auxiliary level (prime to p part) of the quaternion order used to compute this Brandt module.

EXAMPLES:

```
sage: BrandtModule(7,5,2,ZZ).M()
5
```

N()

Return ramification level N.

EXAMPLES:

```
sage: BrandtModule(7,5,2,ZZ).N()
7
```

brandt_series(*prec*, *var='q'*)

Return matrix of power series $\sum T_n q^n$ to the given precision.

Note that the Hecke operators in this series are always over $\mathbf{Q}$, even if the base ring of this Brandt module is not $\mathbf{Q}$.

INPUT:

- `prec` – positive integer
- `var` – string (default: q)

OUTPUT:

matrix of power series with coefficients in $\mathbf{Q}$

EXAMPLES:

```
sage: B = BrandtModule(11)
sage: B.brandt_series(2)
[1/4 + q + O(q^2)     1/4 + O(q^2)]
[    1/6 + O(q^2) 1/6 + q + O(q^2)]
sage: B.brandt_series(5)
[1/4 + q + q^2 + 2*q^3 + 5*q^4 + O(q^5)   1/4 + 3*q^2 + 3*q^3 + 3*q^4 + O(q^
↪5)]
[  1/6 + 2*q^2 + 2*q^3 + 2*q^4 + O(q^5)         1/6 + q + q^3 + 4*q^4 + O(q^
↪5)]
```

Asking for a smaller precision works:

```
sage: B.brandt_series(3)
[1/4 + q + q^2 + O(q^3)    1/4 + 3*q^2 + O(q^3)]
[  1/6 + 2*q^2 + O(q^3)        1/6 + q + O(q^3)]
sage: B.brandt_series(3,var='t')
[1/4 + t + t^2 + O(t^3)    1/4 + 3*t^2 + O(t^3)]
[  1/6 + 2*t^2 + O(t^3)        1/6 + t + O(t^3)]
```

character()

The character of this space.

Always trivial.

EXAMPLES:

```
sage: BrandtModule(11,5).character()
Dirichlet character modulo 55 of conductor 1 mapping 12 |--> 1, 46 |--> 1
```

cyclic_submodules (I, p)

Return a list of rescaled versions of the fractional right ideals J such that J contains I and the quotient has group structure the product of two cyclic groups of order p.

We emphasize again that J is rescaled to be integral.

INPUT:

- I – ideal I in R = self.order_of_level_N()
- p – prime p coprime to self.level()

OUTPUT:

list of the $p + 1$ fractional right R-ideals that contain I such that J/I is GF(p) x GF(p).

EXAMPLES:

```
sage: B = BrandtModule(11)
sage: I = B.order_of_level_N().unit_ideal()
sage: B.cyclic_submodules(I, 2)
[Fractional ideal (1/2 + 3/2*j + k, 1/2*i + j + 1/2*k, 2*j, 2*k),
 Fractional ideal (1/2 + 1/2*i + 1/2*j + 1/2*k, i + k, j + k, 2*k),
 Fractional ideal (1/2 + 1/2*j + k, 1/2*i + j + 3/2*k, 2*j, 2*k)]
sage: B.cyclic_submodules(I, 3)
[Fractional ideal (1/2 + 1/2*j, 1/2*i + 5/2*k, 3*j, 3*k),
 Fractional ideal (1/2 + 3/2*j + 2*k, 1/2*i + 2*j + 3/2*k, 3*j, 3*k),
 Fractional ideal (1/2 + 3/2*j + k, 1/2*i + j + 3/2*k, 3*j, 3*k),
 Fractional ideal (1/2 + 5/2*j, 1/2*i + 1/2*k, 3*j, 3*k)]
sage: B.cyclic_submodules(I, 11)
Traceback (most recent call last):
...
ValueError: p must be coprime to the level
```

eisenstein_subspace ()

Return the 1-dimensional subspace of `self` on which the Hecke operators T_p act as $p + 1$ for p coprime to the level.

Note: This function assumes that the base field has characteristic 0.

EXAMPLES:

```
sage: B = BrandtModule(11); B.eisenstein_subspace()
Subspace of dimension 1 of Brandt module of dimension 2 of level 11 of weight␣
↪2 over Rational Field
sage: B.eisenstein_subspace() is B.eisenstein_subspace()
True
sage: BrandtModule(3,11).eisenstein_subspace().basis()
((1, 1),)
sage: BrandtModule(7,10).eisenstein_subspace().basis()
((1, 1, 1, 1/2, 1, 1, 1/2, 1, 1, 1),)
sage: BrandtModule(7,10,base_ring=ZZ).eisenstein_subspace().basis()
((2, 2, 2, 1, 2, 2, 1, 2, 2, 2),)
```

free_module ()

Return the underlying free module of the Brandt module.

EXAMPLES:

```
sage: B = BrandtModule(10007,389)
sage: B.free_module()
Vector space of dimension 325196 over Rational Field
```

hecke_matrix (*n*, *algorithm='default'*, *sparse=False*, *B=None*)

Return the matrix of the n-th Hecke operator.

INPUT:

- n – integer
- `algorithm` – string (default: 'default')
  - 'default' – let Sage guess which algorithm is best
  - 'direct' – use cyclic subideals (generally much better when you want few Hecke operators and the dimension is very large); uses 'theta' if n divides the level.
  - 'brandt' – use Brandt matrices (generally much better when you want many Hecke operators and the dimension is very small; bad when the dimension is large)
- `sparse` – bool (default: `False`)
- B – integer or `None` (default: `None`); in direct algorithm, use theta series to this precision as an initial check for equality of ideal classes.

EXAMPLES:

```
sage: B = BrandtModule(3,7); B.hecke_matrix(2)
[0 3]
[1 2]
sage: B.hecke_matrix(5, algorithm='brandt')
[0 6]
[2 4]
sage: t = B.hecke_matrix(11, algorithm='brandt', sparse=True); t
[ 6  6]
[ 2 10]
sage: type(t)
<type 'sage.matrix.matrix_rational_sparse.Matrix_rational_sparse'>
sage: B.hecke_matrix(19, algorithm='direct', B=2)
[ 8 12]
[ 4 16]
```

is_cuspidal ()

Return whether `self` is cuspidal, i.e. has no Eisenstein part.

EXAMPLES:

```
sage: B = BrandtModule(3, 4)
sage: B.is_cuspidal()
False
sage: B.eisenstein_subspace()
Brandt module of dimension 1 of level 3*4 of weight 2 over Rational Field
```

maximal_order ()

Return a maximal order in the quaternion algebra associated to this Brandt module.

EXAMPLES:

```
sage: BrandtModule(17).maximal_order()
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with␣
↪basis (1/2 + 1/2*j, 1/2*i + 1/2*k, -1/3*j - 1/3*k, k)
sage: BrandtModule(17).maximal_order() is BrandtModule(17).maximal_order()
True
```

monodromy_weights()

Return the weights for the monodromy pairing on this Brandt module.

The weights are associated to each ideal class in our fixed choice of basis. The weight of an ideal class $[I]$ is half the number of units of the right order I.

NOTE: The base ring must be $\mathbf{Q}$ or $\mathbf{Z}$.

EXAMPLES:

```
sage: BrandtModule(11).monodromy_weights()
(2, 3)
sage: BrandtModule(37).monodromy_weights()
(1, 1, 1)
sage: BrandtModule(43).monodromy_weights()
(2, 1, 1, 1)
sage: BrandtModule(7,10).monodromy_weights()
(1, 1, 1, 2, 1, 1, 2, 1, 1, 1)
sage: BrandtModule(5,13).monodromy_weights()
(1, 3, 1, 1, 1, 3)
sage: BrandtModule(2).monodromy_weights()
(12,)
sage: BrandtModule(2,7).monodromy_weights()
(3, 3)
```

order_of_level_N()

Return Eichler order of level $N = p^{2r+1}M$ in the quaternion algebra.

EXAMPLES:

```
sage: BrandtModule(7).order_of_level_N()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis␣
↪(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
sage: BrandtModule(7,13).order_of_level_N()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis␣
↪(1/2 + 1/2*j + 12*k, 1/2*i + 9/2*k, j + 11*k, 13*k)
sage: BrandtModule(7,3*17).order_of_level_N()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis␣
↪(1/2 + 1/2*j + 35*k, 1/2*i + 65/2*k, j + 19*k, 51*k)
```

quaternion_algebra()

Return the quaternion algebra A over $\mathbf{Q}$ ramified precisely at p and infinity used to compute this Brandt module.

EXAMPLES:

```
sage: BrandtModule(997).quaternion_algebra()
Quaternion Algebra (-2, -997) with base ring Rational Field
sage: BrandtModule(2).quaternion_algebra()
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: BrandtModule(3).quaternion_algebra()
Quaternion Algebra (-1, -3) with base ring Rational Field
sage: BrandtModule(5).quaternion_algebra()
```

(continues on next page)

(continued from previous page)

```
Quaternion Algebra (-2, -5) with base ring Rational Field
sage: BrandtModule(17).quaternion_algebra()
Quaternion Algebra (-17, -3) with base ring Rational Field
```

right_ideals(*B=None*)

Return sorted tuple of representatives for the equivalence classes of right ideals in `self`.

OUTPUT:

sorted tuple of fractional ideals

EXAMPLES:

```
sage: B = BrandtModule(23)
sage: B.right_ideals()
(Fractional ideal (2 + 2*j, 2*i + 2*k, 4*j, 4*k),
 Fractional ideal (2 + 2*j, 2*i + 6*k, 8*j, 8*k),
 Fractional ideal (2 + 10*j + 8*k, 2*i + 8*j + 6*k, 16*j, 16*k))
```

class sage.modular.quatalg.brandt.**BrandtSubmodule**(*ambient*, *submodule*, *dual_free_module=None*, *check=True*)

Bases: `sage.modular.hecke.submodule.HeckeSubmodule`

sage.modular.quatalg.brandt.**basis_for_left_ideal**(*R*, *gens*)

Return a basis for the left ideal of R with given generators.

INPUT:

- R – quaternion order
- `gens` – list of elements of R

OUTPUT:

list of four elements of R

EXAMPLES:

```
sage: B = BrandtModule(17); A = B.quaternion_algebra(); i,j,k = A.gens()
sage: sage.modular.quatalg.brandt.basis_for_left_ideal(B.maximal_order(), [i+j,i-
↪j,2*k,A(3)])
[1/2 + 1/6*j + 2/3*k, 1/2*i + 1/2*k, 1/3*j + 1/3*k, k]
sage: sage.modular.quatalg.brandt.basis_for_left_ideal(B.maximal_order(),␣
↪[3*(i+j),3*(i-j),6*k,A(3)])
[3/2 + 1/2*j + 2*k, 3/2*i + 3/2*k, j + k, 3*k]
```

sage.modular.quatalg.brandt.**benchmark_magma**(*levels*, *silent=False*)

INPUT:

- `levels` – list of pairs (p, M) where p is a prime not dividing M
- `silent` – bool, default `False`; if `True` suppress printing during computation

OUTPUT:

list of 4-tuples ('magma', p, M, tm), where tm is the CPU time in seconds to compute T2 using Magma

EXAMPLES:

```
sage: a = sage.modular.quatalg.brandt.benchmark_magma([(11,1), (37,1), (43,1),
↪(97,1)])   # optional - magma
('magma', 11, 1, ...)
('magma', 37, 1, ...)
('magma', 43, 1, ...)
('magma', 97, 1, ...)
sage: a = sage.modular.quatalg.brandt.benchmark_magma([(11,2), (37,2), (43,2),
↪(97,2)])   # optional - magma
('magma', 11, 2, ...)
('magma', 37, 2, ...)
('magma', 43, 2, ...)
('magma', 97, 2, ...)
```

`sage.modular.quatalg.brandt.`**`benchmark_sage`**(*levels*, *silent=False*)

INPUT:

- `levels` – list of pairs (p, M) where p is a prime not dividing M
- `silent` – bool, default `False`; if `True` suppress printing during computation

OUTPUT:

list of 4-tuples ('sage', p, M, tm), where tm is the CPU time in seconds to compute T2 using Sage

EXAMPLES:

```
sage: a = sage.modular.quatalg.brandt.benchmark_sage([(11,1), (37,1), (43,1), (97,
↪1)])
('sage', 11, 1, ...)
('sage', 37, 1, ...)
('sage', 43, 1, ...)
('sage', 97, 1, ...)
sage: a = sage.modular.quatalg.brandt.benchmark_sage([(11,2), (37,2), (43,2), (97,
↪2)])
('sage', 11, 2, ...)
('sage', 37, 2, ...)
('sage', 43, 2, ...)
('sage', 97, 2, ...)
```

`sage.modular.quatalg.brandt.`**`class_number`**(p, r, M)

Return the class number of an order of level $N = p^r M$ in the quaternion algebra over $\mathbf{Q}$ ramified precisely at p and infinity.

This is an implementation of Theorem 1.12 of [Piz1980].

INPUT:

- p – a prime
- r – an odd positive integer (default: 1)
- M – an integer coprime to q (default: 1)

OUTPUT:

Integer

EXAMPLES:

```
sage: sage.modular.quatalg.brandt.class_number(389,1,1)
33
sage: sage.modular.quatalg.brandt.class_number(389,1,2)   # TODO -- right?
```

(continues on next page)

(continued from previous page)

```
97
sage: sage.modular.quatalg.brandt.class_number(389,3,1)   # TODO -- right?
4892713
```

sage.modular.quatalg.brandt.**maximal_order**(*A*)

Return a maximal order in the quaternion algebra ramified at p and infinity.

This is an implementation of Proposition 5.2 of [Piz1980].

INPUT:

- A – quaternion algebra ramified precisely at p and infinity

OUTPUT:

a maximal order in A

EXAMPLES:

```
sage: A = BrandtModule(17).quaternion_algebra()
sage: sage.modular.quatalg.brandt.maximal_order(A)
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with basis (1/
↪2 + 1/2*j, 1/2*i + 1/2*k, -1/3*j - 1/3*k, k)

sage: A = QuaternionAlgebra(17,names='i,j,k')
sage: A.maximal_order()
Order of Quaternion Algebra (-3, -17) with base ring Rational Field with basis (1/
↪2 + 1/2*i, 1/2*j - 1/2*k, -1/3*i + 1/3*k, -k)
```

sage.modular.quatalg.brandt.**quaternion_order_with_given_level**(*A, level*)

Return an order in the quaternion algebra A with given level.

This is implemented only when the base field is the rational numbers.

INPUT:

- `level` – The level of the order to be returned. Currently this is only implemented when the level is divisible by at most one power of a prime that ramifies in this quaternion algebra.

EXAMPLES:

```
sage: from sage.modular.quatalg.brandt import quaternion_order_with_given_level,␣
↪maximal_order
sage: A.<i,j,k> = QuaternionAlgebra(5)
sage: level = 2 * 5 * 17
sage: O = quaternion_order_with_given_level(A, level)
sage: M = maximal_order(A)
sage: L = O.free_module()
sage: N = M.free_module()
sage: L.index_in(N) == level/5  #check that the order has the right index in the␣
↪maximal order
True
```

sage.modular.quatalg.brandt.**right_order**(*R, basis*)

Given a basis for a left ideal I, return the right order in the quaternion order R of elements x such that Ix is contained in I.

INPUT:

- R – order in quaternion algebra

- `basis` – basis for an ideal I

OUTPUT:

order in quaternion algebra

EXAMPLES:

We do a consistency check with the ideal equal to a maximal order:

```
sage: B = BrandtModule(17); basis = sage.modular.quatalg.brandt.basis_for_left_
↪ideal(B.maximal_order(), B.maximal_order().basis())
sage: sage.modular.quatalg.brandt.right_order(B.maximal_order(), basis)
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with basis (1/
↪2 + 1/6*j + 2/3*k, 1/2*i + 1/2*k, 1/3*j + 1/3*k, k)
sage: basis
[1/2 + 1/6*j + 2/3*k, 1/2*i + 1/2*k, 1/3*j + 1/3*k, k]

sage: B = BrandtModule(17); A = B.quaternion_algebra(); i,j,k = A.gens()
sage: basis = sage.modular.quatalg.brandt.basis_for_left_ideal(B.maximal_order(),␣
↪[i*j-j])
sage: sage.modular.quatalg.brandt.right_order(B.maximal_order(), basis)
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with basis (1/
↪2 + 1/2*i + 1/2*j + 17/2*k, i, j + 8*k, 9*k)
```

CHAPTER

FIFTEEN

THE SET $\mathbb{P}^1(K)$ OF CUSPS OF A NUMBER FIELD K

AUTHORS:

- Maite Aranes (2009): Initial version

EXAMPLES:

The space of cusps over a number field k:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: kCusps = NFCusps(k); kCusps
Set of all cusps of Number Field in a with defining polynomial x^2 + 5
sage: kCusps is NFCusps(k)
True
```

Define a cusp over a number field:

```
sage: NFCusp(k, a, 2/(a+1))
Cusp [a - 5: 2] of Number Field in a with defining polynomial x^2 + 5
sage: kCusps((a,2))
Cusp [a: 2] of Number Field in a with defining polynomial x^2 + 5
sage: NFCusp(k,oo)
Cusp Infinity of Number Field in a with defining polynomial x^2 + 5
```

Different operations with cusps over a number field:

```
sage: alpha = NFCusp(k, 3, 1/a + 2); alpha
Cusp [a + 10: 7] of Number Field in a with defining polynomial x^2 + 5
sage: alpha.numerator()
a + 10
sage: alpha.denominator()
7
sage: alpha.ideal()
Fractional ideal (7, a + 3)
sage: M = alpha.ABmatrix(); M # random
[a + 10, 2*a + 6, 7, a + 5]
sage: NFCusp(k, oo).apply(M)
Cusp [a + 10: 7] of Number Field in a with defining polynomial x^2 + 5
```

Check Gamma0(N)-equivalence of cusps:

```
sage: N = k.ideal(3)
sage: alpha = NFCusp(k, 3, a + 1)
sage: beta = kCusps((2, a - 3))
sage: alpha.is_Gamma0_equivalent(beta, N)
True
```

Obtain transformation matrix for equivalent cusps:

```
sage: t, M = alpha.is_Gamma0_equivalent(beta, N, Transformation=True)
sage: M[2] in N
True
sage: M[0]*M[3] - M[1]*M[2] == 1
True
sage: alpha.apply(M) == beta
True
```

List representatives for Gamma_0(N) - equivalence classes of cusps:

```
sage: Gamma0_NFCusps(N)
[Cusp [0: 1] of Number Field in a with defining polynomial x^2 + 5,
Cusp [1: 3] of Number Field in a with defining polynomial x^2 + 5,
...]
```

sage.modular.cusps_nf.**Gamma0_NFCusps**(*N*)

Return a list of inequivalent cusps for $\Gamma_0(N)$, i.e., a set of representatives for the orbits of `self` on $\mathbb{P}^1(k)$.

INPUT:

- `N` – an integral ideal of the number field k (the level).

OUTPUT:

A list of inequivalent number field cusps.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: N = k.ideal(3)
sage: L = Gamma0_NFCusps(N)
```

The cusps in the list are inequivalent:

```
sage: any(L[i].is_Gamma0_equivalent(L[j], N)
....:     for i in range(len(L)) for j in range(len(L)) if i < j)
False
```

We test that we obtain the right number of orbits:

```
sage: from sage.modular.cusps_nf import number_of_Gamma0_NFCusps
sage: len(L) == number_of_Gamma0_NFCusps(N)
True
```

Another example:

```
sage: k.<a> = NumberField(x^4 - x^3 -21*x^2 + 17*x + 133)
sage: N = k.ideal(5)
sage: from sage.modular.cusps_nf import number_of_Gamma0_NFCusps
sage: len(Gamma0_NFCusps(N)) == number_of_Gamma0_NFCusps(N) # long time (over 1␣
↪sec)
True
```

class sage.modular.cusps_nf.**NFCusp**(*number_field, a, b=None, parent=None, lreps=None*)

Bases: `sage.structure.element.Element`

Create a number field cusp, i.e., an element of $\mathbb{P}^1(k)$.

A cusp on a number field is either an element of the field or infinity, i.e., an element of the projective line over the number field. It is stored as a pair (a,b), where a, b are integral elements of the number field.

INPUT:

- `number_field` – the number field over which the cusp is defined.
- `a` – it can be a number field element (integral or not), or a number field cusp.
- `b` – (optional) when present, it must be either Infinity or coercible to an element of the number field.
- `lreps` – (optional) a list of chosen representatives for all the ideal classes of the field. When given, the representative of the cusp will be changed so its associated ideal is one of the ideals in the list.

OUTPUT:

`[a: b]` – a number field cusp.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: NFCusp(k, a, 2)
Cusp [a: 2] of Number Field in a with defining polynomial x^2 + 5
sage: NFCusp(k, (a,2))
Cusp [a: 2] of Number Field in a with defining polynomial x^2 + 5
sage: NFCusp(k, a, 2/(a+1))
Cusp [a - 5: 2] of Number Field in a with defining polynomial x^2 + 5
```

Cusp Infinity:

```
sage: NFCusp(k, 0)
Cusp [0: 1] of Number Field in a with defining polynomial x^2 + 5
sage: NFCusp(k, oo)
Cusp Infinity of Number Field in a with defining polynomial x^2 + 5
sage: NFCusp(k, 3*a, oo)
Cusp [0: 1] of Number Field in a with defining polynomial x^2 + 5
sage: NFCusp(k, a + 5, 0)
Cusp Infinity of Number Field in a with defining polynomial x^2 + 5
```

Saving and loading works:

```
sage: alpha = NFCusp(k, a, 2/(a+1))
sage: loads(dumps(alpha))==alpha
True
```

Some tests:

```
sage: I*I
-1
sage: NFCusp(k, I)
Traceback (most recent call last):
...
TypeError: unable to convert I to a cusp of the number field
```

```
sage: NFCusp(k, oo, oo)
Traceback (most recent call last):
...
TypeError: unable to convert (+Infinity, +Infinity) to a cusp of the number field
```

```
sage: NFCusp(k, 0, 0)
Traceback (most recent call last):
...
TypeError: unable to convert (0, 0) to a cusp of the number field
```

```
sage: NFCusp(k, "a + 2", a)
Cusp [-2*a + 5: 5] of Number Field in a with defining polynomial x^2 + 5
```

```
sage: NFCusp(k, NFCusp(k, oo))
Cusp Infinity of Number Field in a with defining polynomial x^2 + 5
sage: c = NFCusp(k, 3, 2*a)
sage: NFCusp(k, c, a + 1)
Cusp [-a - 5: 20] of Number Field in a with defining polynomial x^2 + 5
sage: L.<b> = NumberField(x^2 + 2)
sage: NFCusp(L, c)
Traceback (most recent call last):
...
ValueError: Cannot coerce cusps from one field to another
```

ABmatrix()

Return AB-matrix associated to the cusp `self`.

Given R a Dedekind domain and A, B ideals of R in inverse classes, an AB-matrix is a matrix realizing the isomorphism between R+R and A+B. An AB-matrix associated to a cusp [a1: a2] is an AB-matrix with A the ideal associated to the cusp (A=<a1, a2>) and first column given by the coefficients of the cusp.

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 + 11)
sage: alpha = NFCusp(k, oo)
sage: alpha.ABmatrix()
[1, 0, 0, 1]
```

```
sage: alpha = NFCusp(k, 0)
sage: alpha.ABmatrix()
[0, -1, 1, 0]
```

Note that the AB-matrix associated to a cusp is not unique, and the output of the `ABmatrix` function may change.

```
sage: alpha = NFCusp(k, 3/2, a-1)
sage: M = alpha.ABmatrix()
sage: M # random
[-a^2 - a - 1, -3*a - 7, 8, -2*a^2 - 3*a + 4]
sage: M[0] == alpha.numerator() and M[2]==alpha.denominator()
True
```

An AB-matrix associated to a cusp alpha will send Infinity to alpha:

```
sage: alpha = NFCusp(k, 3, a-1)
sage: M = alpha.ABmatrix()
sage: (k.ideal(M[1], M[3])*alpha.ideal()).is_principal()
True
sage: M[0] == alpha.numerator() and M[2]==alpha.denominator()
True
sage: NFCusp(k, oo).apply(M) == alpha
True
```

apply(*g*)

Return g(`self`), where `g` is a 2x2 matrix, which we view as a linear fractional transformation.

INPUT:

- `g` – a list of integral elements [a, b, c, d] that are the entries of a 2x2 matrix.

OUTPUT:

A number field cusp, obtained by the action of `g` on the cusp `self`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 23)
sage: beta = NFCusp(k, 0, 1)
sage: beta.apply([0, -1, 1, 0])
Cusp Infinity of Number Field in a with defining polynomial x^2 + 23
sage: beta.apply([1, a, 0, 1])
Cusp [a: 1] of Number Field in a with defining polynomial x^2 + 23
```

denominator()

Return the denominator of the cusp `self`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 1)
sage: c = NFCusp(k, a, 2)
sage: c.denominator()
2
sage: d = NFCusp(k, 1, a + 1);d
Cusp [1: a + 1] of Number Field in a with defining polynomial x^2 + 1
sage: d.denominator()
a + 1
sage: NFCusp(k, oo).denominator()
0
```

ideal()

Return the ideal associated to the cusp `self`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 23)
sage: alpha = NFCusp(k, 3, a-1)
sage: alpha.ideal()
Fractional ideal (3, 1/2*a - 1/2)
sage: NFCusp(k, oo).ideal()
Fractional ideal (1)
```

is_Gamma0_equivalent(*other*, *N*, *Transformation=False*)

Check if cusps `self` and `other` are $\Gamma_0(N)$- equivalent.

INPUT:

- `other` – a number field cusp or a list of two number field elements which define a cusp.
- `N` – an ideal of the number field (level)

OUTPUT:

- bool – `True` if the cusps are equivalent.

- a transformation matrix – (if `Transformation=True`) a list of integral elements [a, b, c, d] which are the entries of a 2x2 matrix M in $\Gamma_0(N)$ such that M * `self` = `other` if `other` and `self` are $\Gamma_0(N)$- equivalent. If `self` and `other` are not equivalent it returns zero.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-10)
sage: N = K.ideal(a-1)
sage: alpha = NFCusp(K, 0)
sage: beta = NFCusp(K, oo)
sage: alpha.is_Gamma0_equivalent(beta, N)
False
sage: alpha.is_Gamma0_equivalent(beta, K.ideal(1))
True
sage: b, M = alpha.is_Gamma0_equivalent(beta, K.ideal(1),Transformation=True)
sage: alpha.apply(M)
Cusp Infinity of Number Field in a with defining polynomial x^3 - 10
```

```
sage: k.<a> = NumberField(x^2+23)
sage: N = k.ideal(3)
sage: alpha1 = NFCusp(k, a+1, 4)
sage: alpha2 = NFCusp(k, a-8, 29)
sage: alpha1.is_Gamma0_equivalent(alpha2, N)
True
sage: b, M = alpha1.is_Gamma0_equivalent(alpha2, N, Transformation=True)
sage: alpha1.apply(M) == alpha2
True
sage: M[2] in N
True
```

is_infinity()

Return `True` if this is the cusp infinity.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 1)
sage: NFCusp(k, a, 2).is_infinity()
False
sage: NFCusp(k, 2, 0).is_infinity()
True
sage: NFCusp(k, oo).is_infinity()
True
```

number_field()

Return the number field of definition of the cusp `self`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 2)
sage: alpha = NFCusp(k, 1, a + 1)
sage: alpha.number_field()
Number Field in a with defining polynomial x^2 + 2
```

numerator()

Return the numerator of the cusp `self`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 1)
sage: c = NFCusp(k, a, 2)
sage: c.numerator()
a
sage: d = NFCusp(k, 1, a)
sage: d.numerator()
1
sage: NFCusp(k, oo).numerator()
1
```

sage.modular.cusps_nf.**NFCusps**(*number_field*)

The set of cusps of a number field K, i.e. $\mathbb{P}^1(K)$.

INPUT:

- `number_field` – a number field

OUTPUT:

The set of cusps over the given number field.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: kCusps = NFCusps(k); kCusps
Set of all cusps of Number Field in a with defining polynomial x^2 + 5
sage: kCusps is NFCusps(k)
True
```

Saving and loading works:

```
sage: loads(kCusps.dumps()) == kCusps
True
```

class sage.modular.cusps_nf.**NFCuspsSpace**(*number_field*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

The set of cusps of a number field. See `NFCusps` for full documentation.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: kCusps = NFCusps(k); kCusps
Set of all cusps of Number Field in a with defining polynomial x^2 + 5
```

number_field()

Return the number field that this set of cusps is attached to.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 1)
sage: kCusps = NFCusps(k)
sage: kCusps.number_field()
Number Field in a with defining polynomial x^2 + 1
```

zero()

Return the zero cusp.

Note: This method just exists to make some general algorithms work. It is not intended that the returned cusp is an additive neutral element.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: kCusps = NFCusps(k)
sage: kCusps.zero()
Cusp [0: 1] of Number Field in a with defining polynomial x^2 + 5
```

`sage.modular.cusps_nf.`**`NFCusps_ideal_reps_for_levelN`**`(`*N*, *nlists=1*`)`

Return a list of lists (`nlists` different lists) of prime ideals, coprime to `N`, representing every ideal class of the number field.

INPUT:

- `N` – number field ideal.
- `nlists` – optional (default 1). The number of lists of prime ideals we want.

OUTPUT:

A list of lists of ideals representatives of the ideal classes, all coprime to `N`, representing every ideal.

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 + 11)
sage: N = k.ideal(5, a + 1)
sage: from sage.modular.cusps_nf import NFCusps_ideal_reps_for_levelN
sage: NFCusps_ideal_reps_for_levelN(N)
[(Fractional ideal (1), Fractional ideal (2, a + 1))]
sage: L = NFCusps_ideal_reps_for_levelN(N, 3)
sage: all(len(L[i]) == k.class_number() for i in range(len(L)))
True
```

```
sage: k.<a> = NumberField(x^4 - x^3 -21*x^2 + 17*x + 133)
sage: N = k.ideal(6)
sage: from sage.modular.cusps_nf import NFCusps_ideal_reps_for_levelN
sage: NFCusps_ideal_reps_for_levelN(N)
[(Fractional ideal (1),
  Fractional ideal (67, a + 17),
  Fractional ideal (127, a + 48),
  Fractional ideal (157, a - 19))]
sage: L = NFCusps_ideal_reps_for_levelN(N, 5)
sage: all(len(L[i]) == k.class_number() for i in range(len(L)))
True
```

`sage.modular.cusps_nf.`**`list_of_representatives`**`(`*N*`)`

Return a list of ideals, coprime to the ideal `N`, representatives of the ideal classes of the corresponding number field.

Note: This list, used every time we check $\Gamma_0(N)$ - equivalence of cusps, is cached.

INPUT:

- `N` – an ideal of a number field.

OUTPUT:

A list of ideals coprime to the ideal N, such that they are representatives of all the ideal classes of the number field.

EXAMPLES:

```
sage: from sage.modular.cusps_nf import list_of_representatives
sage: k.<a> = NumberField(x^4 + 13*x^3 - 11)
sage: N = k.ideal(713, a + 208)
sage: L = list_of_representatives(N); L
(Fractional ideal (1),
 Fractional ideal (47, a - 9),
 Fractional ideal (53, a - 16))
```

sage.modular.cusps_nf.**number_of_Gamma0_NFCusps**(*N*)

Return the total number of orbits of cusps under the action of the congruence subgroup $\Gamma_0(N)$.

INPUT:

- N – a number field ideal.

OUTPUT:

ingeter – the number of orbits of cusps under Gamma0(N)-action.

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 + 11)
sage: N = k.ideal(2, a+1)
sage: from sage.modular.cusps_nf import number_of_Gamma0_NFCusps
sage: number_of_Gamma0_NFCusps(N)
4
sage: L = Gamma0_NFCusps(N)
sage: len(L) == number_of_Gamma0_NFCusps(N)
True
sage: k.<a> = NumberField(x^2 + 7)
sage: N = k.ideal(9)
sage: number_of_Gamma0_NFCusps(N)
6
sage: N = k.ideal(a*9 + 7)
sage: number_of_Gamma0_NFCusps(N)
24
```

sage.modular.cusps_nf.**units_mod_ideal**(*I*)

Return integral elements of the number field representing the images of the global units modulo the ideal I.

INPUT:

- I – number field ideal.

OUTPUT:

A list of integral elements of the number field representing the images of the global units modulo the ideal I. Elements of the list might be equivalent to each other mod I.

EXAMPLES:

```
sage: from sage.modular.cusps_nf import units_mod_ideal
sage: k.<a> = NumberField(x^2 + 1)
sage: I = k.ideal(a + 1)
sage: units_mod_ideal(I)
```

(continues on next page)

(continued from previous page)

```
[1]
sage: I = k.ideal(3)
sage: units_mod_ideal(I)
[1, a, -1, -a]
```

```
sage: from sage.modular.cusps_nf import units_mod_ideal
sage: k.<a> = NumberField(x^3 + 11)
sage: k.unit_group()
Unit group with structure C2 x Z of Number Field in a with defining polynomial x^
↪3 + 11
sage: I = k.ideal(5, a + 1)
sage: units_mod_ideal(I)
[1,
2*a^2 + 4*a - 1,
...]
```

```
sage: from sage.modular.cusps_nf import units_mod_ideal
sage: k.<a> = NumberField(x^4 - x^3 -21*x^2 + 17*x + 133)
sage: k.unit_group()
Unit group with structure C6 x Z of Number Field in a with defining polynomial x^
↪4 - x^3 - 21*x^2 + 17*x + 133
sage: I = k.ideal(3)
sage: U = units_mod_ideal(I)
sage: all(U[j].is_unit() and (U[j] not in I) for j in range(len(U)))
True
```

CHAPTER

SIXTEEN

HYPERGEOMETRIC MOTIVES

This is largely a port of the corresponding package in Magma. One important conventional difference: the motivic parameter t has been replaced with $1/t$ to match the classical literature on hypergeometric series. (E.g., see [BeukersHeckman])

The computation of Euler factors is currently only supported for primes p of good reduction. That is, it is required that $v_p(t) = v_p(t-1) = 0$.

AUTHORS:

- Frédéric Chapoton
- Kiran S. Kedlaya

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([30], [1,2,3,5]))
sage: H.alpha_beta()
([1/30, 7/30, 11/30, 13/30, 17/30, 19/30, 23/30, 29/30],
[0, 1/5, 1/3, 2/5, 1/2, 3/5, 2/3, 4/5])
sage: H.M_value() == 30**30 / (15**15 * 10**10 * 6**6)
True
sage: H.euler_factor(2, 7)
T^8 + T^5 + T^3 + 1
```

REFERENCES:

- [BeukersHeckman]
- [Benasque2009]
- [Kat1991]
- [MagmaHGM]
- [Fedorov2015]
- [Roberts2017]
- [Roberts2015]
- [BeCoMe]
- [Watkins]

class sage.modular.hypergeometric_motive.**HypergeometricData**(*cyclotomic=None*, *alpha_beta=None*, *gamma_list=None*)

Bases: object

Creation of hypergeometric motives.

INPUT:

three possibilities are offered, each describing a quotient of products of cyclotomic polynomials.

- `cyclotomic` – a pair of lists of nonnegative integers, each integer k represents a cyclotomic polynomial Φ_k
- `alpha_beta` – a pair of lists of rationals, each rational represents a root of unity
- `gamma_list` – a pair of lists of nonnegative integers, each integer n represents a polynomial $x^n - 1$

In the last case, it is also allowed to send just one list of signed integers where signs indicate to which part the integer belongs to.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(cyclotomic=([2],[1]))
Hypergeometric data for [1/2] and [0]

sage: Hyp(alpha_beta=([1/2],[0]))
Hypergeometric data for [1/2] and [0]
sage: Hyp(alpha_beta=([1/5,2/5,3/5,4/5],[0,0,0,0]))
Hypergeometric data for [1/5, 2/5, 3/5, 4/5] and [0, 0, 0, 0]

sage: Hyp(gamma_list=([5],[1,1,1,1,1]))
Hypergeometric data for [1/5, 2/5, 3/5, 4/5] and [0, 0, 0, 0]
sage: Hyp(gamma_list=([5,-1,-1,-1,-1,-1]))
Hypergeometric data for [1/5, 2/5, 3/5, 4/5] and [0, 0, 0, 0]
```

H_value (*p*, *f*, *t*, *ring=None*)

Return the trace of the Frobenius, computed in terms of Gauss sums using the hypergeometric trace formula.

INPUT:

- p – a prime number
- f – an integer such that $q = p^f$
- t – a rational parameter
- `ring` – optional (default `UniversalCyclotomicfield`)

The ring could be also `ComplexField(n)` or `QQbar`.

OUTPUT:

an integer

> **Warning:** This is apparently working correctly as can be tested using ComplexField(70) as value ring.
>
> Using instead UniversalCyclotomicfield, this is much slower than the p-adic version *padic_H_value()*.

EXAMPLES:

With values in the UniversalCyclotomicField (slow):

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(alpha_beta=([1/2]*4,[0]*4))
sage: [H.H_value(3,i,-1) for i in range(1,3)]
[0, -12]
sage: [H.H_value(5,i,-1) for i in range(1,3)]
[-4, 276]
sage: [H.H_value(7,i,-1) for i in range(1,3)]  # not tested
[0, -476]
sage: [H.H_value(11,i,-1) for i in range(1,3)]  # not tested
[0, -4972]
sage: [H.H_value(13,i,-1) for i in range(1,3)]  # not tested
[-84, -1420]
```

With values in ComplexField:

```
sage: [H.H_value(5,i,-1, ComplexField(60)) for i in range(1,3)]
[-4, 276]
```

Check issue from trac ticket #28404:

```
sage: H1 = Hyp(cyclotomic=([1,1,1],[6,2]))
sage: H2 = Hyp(cyclotomic=([6,2],[1,1,1]))
sage: [H1.H_value(5,1,i) for i in range(2,5)]
[1, -4, -4]
sage: [H2.H_value(5,1,QQ(i)) for i in range(2,5)]
[-4, 1, -4]
```

REFERENCES:

- [BeCoMe] (Theorem 1.3)
- [Benasque2009]

M_value()

Return the M coefficient that appears in the trace formula.

OUTPUT:

a rational

See also:

canonical_scheme()

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(alpha_beta=([1/6,1/3,2/3,5/6],[1/8,3/8,5/8,7/8]))
sage: H.M_value()
729/4096
sage: Hyp(alpha_beta=(([1/2,1/2,1/2,1/2],[0,0,0,0]))).M_value()
256
sage: Hyp(cyclotomic=([5],[1,1,1,1])).M_value()
3125
```

alpha()

Return the first tuple of rational arguments.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).alpha()
[1/2]
```

alpha_beta()

Return the pair of lists of rational arguments.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).alpha_beta()
([1/2], [0])
```

beta()

Return the second tuple of rational arguments.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).beta()
[0]
```

canonical_scheme(*t=None*)

Return the canonical scheme.

This is a scheme that contains this hypergeometric motive in its cohomology.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([3],[4]))
sage: H.gamma_list()
[-1, 2, 3, -4]
sage: H.canonical_scheme()
Spectrum of Quotient of Multivariate Polynomial Ring
in X0, X1, Y0, Y1 over Fraction Field of Univariate Polynomial Ring
in t over Rational Field by the ideal
(X0 + X1 - 1, Y0 + Y1 - 1, (-t)*X0^2*X1^3 + 27/64*Y0*Y1^4)

sage: H = Hyp(gamma_list=[-2, 3, 4, -5])
sage: H.canonical_scheme()
Spectrum of Quotient of Multivariate Polynomial Ring
in X0, X1, Y0, Y1 over Fraction Field of Univariate Polynomial Ring
in t over Rational Field by the ideal
(X0 + X1 - 1, Y0 + Y1 - 1, (-t)*X0^3*X1^4 + 1728/3125*Y0^2*Y1^5)
```

REFERENCES:

[Kat1991], section 5.4

cyclotomic_data()

Return the pair of tuples of indices of cyclotomic polynomials.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).cyclotomic_data()
([2], [1])
```

defining_polynomials()

Return the pair of products of cyclotomic polynomials.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/4,3/4],[0,0])).defining_polynomials()
(x^2 + 1, x^2 - 2*x + 1)
```

degree()

Return the degree.

This is the sum of the Hodge numbers.

See also:

hodge_numbers()

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).degree()
1
sage: Hyp(gamma_list=([2,2,4],[8])).degree()
4
sage: Hyp(cyclotomic=([5,6],[1,1,2,2,3])).degree()
6
sage: Hyp(cyclotomic=([3,8],[1,1,1,2,6])).degree()
6
sage: Hyp(cyclotomic=([3,3],[2,2,4])).degree()
4
```

euler_factor(*t*, *p*, *cache_p=False*)

Return the Euler factor of the motive H_t at prime p.

INPUT:

- t – rational number, not 0 or 1
- p – prime number of good reduction

OUTPUT:

a polynomial

See [Benasque2009] for explicit examples of Euler factors.

For odd weight, the sign of the functional equation is +1. For even weight, the sign is computed by a recipe found in 11.1 of [Watkins].

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(alpha_beta=([1/2]*4,[0]*4))
sage: H.euler_factor(-1, 5)
15625*T^4 + 500*T^3 - 130*T^2 + 4*T + 1

sage: H = Hyp(gamma_list=[-6,-1,4,3])
sage: H.weight(), H.degree()
(1, 2)
sage: t = 189/125
sage: [H.euler_factor(1/t,p) for p in [11,13,17,19,23,29]]
```

(continues on next page)

(continued from previous page)

```
[11*T^2 + 4*T + 1,
13*T^2 + 1,
17*T^2 + 1,
19*T^2 + 1,
23*T^2 + 8*T + 1,
29*T^2 + 2*T + 1]

sage: H = Hyp(cyclotomic=([6,2],[1,1,1]))
sage: H.weight(), H.degree()
(2, 3)
sage: [H.euler_factor(1/4,p) for p in [5,7,11,13,17,19]]
[125*T^3 + 20*T^2 + 4*T + 1,
 343*T^3 - 42*T^2 - 6*T + 1,
 -1331*T^3 - 22*T^2 + 2*T + 1,
 -2197*T^3 - 156*T^2 + 12*T + 1,
 4913*T^3 + 323*T^2 + 19*T + 1,
 6859*T^3 - 57*T^2 - 3*T + 1]

sage: H = Hyp(alpha_beta=([1/12,5/12,7/12,11/12],[0,1/2,1/2,1/2]))
sage: H.weight(), H.degree()
(2, 4)
sage: t = -5
sage: [H.euler_factor(1/t,p) for p in [11,13,17,19,23,29]]
[-14641*T^4 - 1210*T^3 + 10*T + 1,
 -28561*T^4 - 2704*T^3 + 16*T + 1,
 -83521*T^4 - 4046*T^3 + 14*T + 1,
 130321*T^4 + 14440*T^3 + 969*T^2 + 40*T + 1,
 279841*T^4 - 25392*T^3 + 1242*T^2 - 48*T + 1,
 707281*T^4 - 7569*T^3 + 696*T^2 - 9*T + 1]
```

This is an example of higher degree:

```
sage: H = Hyp(cyclotomic=([11], [7, 12]))
sage: H.euler_factor(2, 13)
371293*T^10 - 85683*T^9 + 26364*T^8 + 1352*T^7 - 65*T^6 + 394*T^5 - 5*T^4 +␣
↪8*T^3 + 12*T^2 - 3*T + 1
sage: H.euler_factor(2, 19) # long time
2476099*T^10 - 651605*T^9 + 233206*T^8 - 77254*T^7 + 20349*T^6 - 4611*T^5 +␣
↪1071*T^4 - 214*T^3 + 34*T^2 - 5*T + 1
```

REFERENCES:

- [Roberts2015]
- [Watkins]

gamma_array()

Return the dictionary $\{v : \gamma_v\}$ for the expression

$$\prod_v (T^v - 1)^{\gamma_v}$$

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).gamma_array()
{1: -2, 2: 1}
sage: Hyp(cyclotomic=([6,2],[1,1,1])).gamma_array()
{1: -3, 3: -1, 6: 1}
```

gamma_list()

Return a list of integers describing the $x^n - 1$ factors.

Each integer n stands for $(x^{|n|} - 1)^{\mathrm{sgn}(n)}$.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).gamma_list()
[-1, -1, 2]

sage: Hyp(cyclotomic=([6,2],[1,1,1])).gamma_list()
[-1, -1, -1, -3, 6]

sage: Hyp(cyclotomic=([3],[4])).gamma_list()
[-1, 2, 3, -4]
```

gauss_table(*p, f, prec*)

Return (and cache) a table of Gauss sums used in the trace formula.

See also:

gauss_table_full()

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([3],[4]))
sage: H.gauss_table(2, 2, 4)
(4, [1 + 2 + 2^2 + 2^3, 1 + 2 + 2^2 + 2^3, 1 + 2 + 2^2 + 2^3])
```

gauss_table_full()

Return a dict of all stored tables of Gauss sums.

The result is passed by reference, and is an attribute of the class; consequently, modifying the result has global side effects. Use with caution.

See also:

gauss_table()

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([3],[4]))
sage: H.euler_factor(2, 7, cache_p=True)
7*T^2 - 3*T + 1
sage: H.gauss_table_full()[(7, 1)]
(2, array('l', [-1, -29, -25, -48, -47, -22]))
```

Clearing cached values:

```
sage: H = Hyp(cyclotomic=([3],[4]))
sage: H.euler_factor(2, 7, cache_p=True)
7*T^2 - 3*T + 1
sage: d = H.gauss_table_full()
sage: d.clear() # Delete all entries of this dict
sage: H1 = Hyp(cyclotomic=([5],[12]))
sage: d1 = H1.gauss_table_full()
sage: len(d1.keys()) # No cached values
0
```

has_symmetry_at_one()
If `True`, the motive H(t=1) is a direct sum of two motives.

Note that simultaneous exchange of (t,1/t) and (alpha,beta) always gives the same motive.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=[[1/2]*16,[0]*16]).has_symmetry_at_one()
True
```

REFERENCES:

- [Roberts2017]

hodge_function(*x*)
Evaluate the Hodge polygon as a function.

See also:

hodge_numbers(), *hodge_polynomial()*, *hodge_polygon_vertices()*

sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp sage: H = Hyp(cyclotomic=([6,10],[3,12])) sage: H.hodge_function(3) 2 sage: H.hodge_function(4) 4

hodge_numbers()
Return the Hodge numbers.

See also:

degree(), *hodge_polynomial()*, `hodge_polygon()`

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([3],[6]))
sage: H.hodge_numbers()
[1, 1]

sage: H = Hyp(cyclotomic=([4],[1,2]))
sage: H.hodge_numbers()
[2]

sage: H = Hyp(gamma_list=([8,2,2,2],[6,4,3,1]))
sage: H.hodge_numbers()
[1, 2, 2, 1]

sage: H = Hyp(gamma_list=([5],[1,1,1,1,1]))
sage: H.hodge_numbers()
[1, 1, 1, 1]

sage: H = Hyp(gamma_list=[6,1,-4,-3])
sage: H.hodge_numbers()
[1, 1]

sage: H = Hyp(gamma_list=[-3]*4 + [1]*12)
sage: H.hodge_numbers()
[1, 1, 1, 1, 1, 1, 1, 1]
```

REFERENCES:

- [Fedorov2015]

hodge_polygon_vertices()

Return the vertices of the Hodge polygon.

See also:

hodge_numbers(), *hodge_polynomial()*, *hodge_function()*

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([6,10],[3,12]))
sage: H.hodge_polygon_vertices()
[(0, 0), (1, 0), (3, 2), (5, 6), (6, 9)]
sage: H = Hyp(cyclotomic=([2,2,2,2,3,3,3,6,6],[1,1,4,5,9]))
sage: H.hodge_polygon_vertices()
[(0, 0), (1, 0), (4, 3), (7, 9), (10, 18), (13, 30), (14, 35)]
```

hodge_polynomial()

Return the Hodge polynomial.

See also:

hodge_numbers(), *hodge_polygon_vertices()*, *hodge_function()*

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([6,10],[3,12]))
sage: H.hodge_polynomial()
(T^3 + 2*T^2 + 2*T + 1)/T^2
sage: H = Hyp(cyclotomic=([2,2,2,2,3,3,3,6,6],[1,1,4,5,9]))
sage: H.hodge_polynomial()
(T^5 + 3*T^4 + 3*T^3 + 3*T^2 + 3*T + 1)/T^2
```

is_primitive()

Return whether this data is primitive.

See also:

primitive_index(), *primitive_data()*

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(cyclotomic=([3],[4])).is_primitive()
True
sage: Hyp(gamma_list=[-2, 4, 6, -8]).is_primitive()
False
sage: Hyp(gamma_list=[-3, 6, 9, -12]).is_primitive()
False
```

padic_H_value(*p*, *f*, *t*, *prec=None*, *cache_p=False*)

Return the p-adic trace of Frobenius, computed using the Gross-Koblitz formula.

If left unspecified, $prec$ is set to the minimum p-adic precision needed to recover the Euler factor.

If $cache_p$ is True, then the function caches an intermediate result which depends only on p and f. This leads to a significant speedup when iterating over t.

INPUT:

- p – a prime number

- f – an integer such that $q = p^f$
- t – a rational parameter
- `prec` – precision (optional)
- `cache_p` - a boolean

OUTPUT:

an integer

EXAMPLES:

From Benasque report [Benasque2009], page 8:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(alpha_beta=([1/2]*4,[0]*4))
sage: [H.padic_H_value(3,i,-1) for i in range(1,3)]
[0, -12]
sage: [H.padic_H_value(5,i,-1) for i in range(1,3)]
[-4, 276]
sage: [H.padic_H_value(7,i,-1) for i in range(1,3)]
[0, -476]
sage: [H.padic_H_value(11,i,-1) for i in range(1,3)]
[0, -4972]
```

From [Roberts2015] (but note conventions regarding t):

```
sage: H = Hyp(gamma_list=[-6,-1,4,3])
sage: t = 189/125
sage: H.padic_H_value(13,1,1/t)
0
```

REFERENCES:

- [MagmaHGM]

primitive_data()

Return a primitive version.

See also:

is_primitive(), primitive_index()

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([3],[4]))
sage: H2 = Hyp(gamma_list=[-2, 4, 6, -8])
sage: H2.primitive_data() == H
True
```

primitive_index()

Return the primitive index.

See also:

is_primitive(), primitive_data()

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(cyclotomic=([3],[4])).primitive_index()
1
sage: Hyp(gamma_list=[-2, 4, 6, -8]).primitive_index()
2
sage: Hyp(gamma_list=[-3, 6, 9, -12]).primitive_index()
3
```

sign (t, p)

Return the sign of the functional equation for the Euler factor of the motive H_t at the prime p.

For odd weight, the sign of the functional equation is +1. For even weight, the sign is computed by a recipe found in 11.1 of [Watkins] (when 0 is not in alpha).

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(cyclotomic=([6,2],[1,1,1]))
sage: H.weight(), H.degree()
(2, 3)
sage: [H.sign(1/4,p) for p in [5,7,11,13,17,19]]
[1, 1, -1, -1, 1, 1]

sage: H = Hyp(alpha_beta=([1/12,5/12,7/12,11/12],[0,1/2,1/2,1/2]))
sage: H.weight(), H.degree()
(2, 4)
sage: t = -5
sage: [H.sign(1/t,p) for p in [11,13,17,19,23,29]]
[-1, -1, -1, 1, 1, 1]
```

We check that trac ticket #28404 is fixed:

```
sage: H = Hyp(cyclotomic=([1,1,1],[6,2]))
sage: [H.sign(4,p) for p in [5,7,11,13,17,19]]
[1, 1, -1, -1, 1, 1]
```

swap_alpha_beta ()

Return the hypergeometric data with `alpha` and `beta` exchanged.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(alpha_beta=([1/2],[0]))
sage: H.swap_alpha_beta()
Hypergeometric data for [0] and [1/2]
```

twist ()

Return the twist of this data.

This is defined by adding $1/2$ to each rational in α and β.

This is an involution.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(alpha_beta=([1/2],[0]))
sage: H.twist()
Hypergeometric data for [0] and [1/2]
```

(continues on next page)

(continued from previous page)

```
sage: H.twist().twist() == H
True

sage: Hyp(cyclotomic=([6],[1,2])).twist().cyclotomic_data()
([3], [1, 2])
```

weight()

Return the motivic weight of this motivic data.

EXAMPLES:

With rational inputs:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: Hyp(alpha_beta=([1/2],[0])).weight()
0
sage: Hyp(alpha_beta=([1/4,3/4],[0,0])).weight()
1
sage: Hyp(alpha_beta=([1/6,1/3,2/3,5/6],[0,0,1/4,3/4])).weight()
1
sage: H = Hyp(alpha_beta=([1/6,1/3,2/3,5/6],[1/8,3/8,5/8,7/8]))
sage: H.weight()
1
```

With cyclotomic inputs:

```
sage: Hyp(cyclotomic=([6,2],[1,1,1])).weight()
2
sage: Hyp(cyclotomic=([6],[1,2])).weight()
0
sage: Hyp(cyclotomic=([8],[1,2,3])).weight()
0
sage: Hyp(cyclotomic=([5],[1,1,1,1])).weight()
3
sage: Hyp(cyclotomic=([5,6],[1,1,2,2,3])).weight()
1
sage: Hyp(cyclotomic=([3,8],[1,1,1,2,6])).weight()
2
sage: Hyp(cyclotomic=([3,3],[2,2,4])).weight()
1
```

With gamma list input:

```
sage: Hyp(gamma_list=([8,2,2,2],[6,4,3,1])).weight()
3
```

zigzag(*x*, *flip_beta=False*)

Count `alpha`'s at most `x` minus `beta`'s at most `x`.

This function is used to compute the weight and the Hodge numbers. With $flip_beta$ set to True, replace each b in β with $1 - b$.

See also:

weight(), *hodge_numbers()*

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(alpha_beta=([1/6,1/3,2/3,5/6],[1/8,3/8,5/8,7/8]))
sage: [H.zigzag(x) for x in [0, 1/3, 1/2]]
[0, 1, 0]
sage: H = Hyp(cyclotomic=([5],[1,1,1,1]))
sage: [H.zigzag(x) for x in [0,1/6,1/4,1/2,3/4,5/6]]
[-4, -4, -3, -2, -1, 0]
```

sage.modular.hypergeometric_motive.**alpha_to_cyclotomic**(*alpha*)

Convert from a list of rationals arguments to a list of integers.

The input represents arguments of some roots of unity.

The output represent a product of cyclotomic polynomials with exactly the given roots. Note that the multiplicity of r/s in the list must be independent of r; otherwise, a `ValueError` will be raised.

This is the inverse of *cyclotomic_to_alpha()*.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import alpha_to_cyclotomic
sage: alpha_to_cyclotomic([0])
[1]
sage: alpha_to_cyclotomic([1/2])
[2]
sage: alpha_to_cyclotomic([1/5,2/5,3/5,4/5])
[5]
sage: alpha_to_cyclotomic([0, 1/6, 1/3, 1/2, 2/3, 5/6])
[1, 2, 3, 6]
sage: alpha_to_cyclotomic([1/3,2/3,1/2])
[2, 3]
```

sage.modular.hypergeometric_motive.**capital_M**(*n*)

Auxiliary function, used to describe the canonical scheme.

INPUT:

- `n` – an integer

OUTPUT:

a rational

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import capital_M
sage: [capital_M(i) for i in range(1,8)]
[1, 4, 27, 64, 3125, 432, 823543]
```

sage.modular.hypergeometric_motive.**characteristic_polynomial_from_traces**(*traces*, *d*, *q*, *i*, *sign*)

Given a sequence of traces $t_1, \ldots, t_k$, return the corresponding characteristic polynomial with Weil numbers as roots.

The characteristic polynomial is defined by the generating series

$$P(T) = \exp\left(-\sum_{k\geq 1} t_k \frac{T^k}{k}\right)$$

and should have the property that reciprocals of all roots have absolute value $q^{i/2}$.

INPUT:

- `traces` – a list of integers $t_1, \dots, t_k$
- `d` – the degree of the characteristic polynomial
- `q` – power of a prime number
- `i` – integer, the weight in the motivic sense
- `sign` – integer, the sign

OUTPUT:

a polynomial

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import characteristic_polynomial_
↪from_traces
sage: characteristic_polynomial_from_traces([1, 1], 1, 3, 0, -1)
-T + 1
sage: characteristic_polynomial_from_traces([25], 1, 5, 4, -1)
-25*T + 1

sage: characteristic_polynomial_from_traces([3], 2, 5, 1, 1)
5*T^2 - 3*T + 1
sage: characteristic_polynomial_from_traces([1], 2, 7, 1, 1)
7*T^2 - T + 1

sage: characteristic_polynomial_from_traces([20], 3, 29, 2, 1)
24389*T^3 - 580*T^2 - 20*T + 1
sage: characteristic_polynomial_from_traces([12], 3, 13, 2, -1)
-2197*T^3 + 156*T^2 - 12*T + 1

sage: characteristic_polynomial_from_traces([36,7620], 4, 17, 3, 1)
24137569*T^4 - 176868*T^3 - 3162*T^2 - 36*T + 1
sage: characteristic_polynomial_from_traces([-4,276], 4, 5, 3, 1)
15625*T^4 + 500*T^3 - 130*T^2 + 4*T + 1
sage: characteristic_polynomial_from_traces([4,-276], 4, 5, 3, 1)
15625*T^4 - 500*T^3 + 146*T^2 - 4*T + 1
sage: characteristic_polynomial_from_traces([22, 484], 4, 31, 2, -1)
-923521*T^4 + 21142*T^3 - 22*T + 1
```

`sage.modular.hypergeometric_motive.`**`cyclotomic_to_alpha`**(*cyclo*)

Convert a list of indices of cyclotomic polynomials to a list of rational numbers.

The input represents a product of cyclotomic polynomials.

The output is the list of arguments of the roots of the given product of cyclotomic polynomials.

This is the inverse of *`alpha_to_cyclotomic()`*.

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import cyclotomic_to_alpha
sage: cyclotomic_to_alpha([1])
[0]
sage: cyclotomic_to_alpha([2])
[1/2]
sage: cyclotomic_to_alpha([5])
```

(continues on next page)

(continued from previous page)

```
[1/5, 2/5, 3/5, 4/5]
sage: cyclotomic_to_alpha([1,2,3,6])
[0, 1/6, 1/3, 1/2, 2/3, 5/6]
sage: cyclotomic_to_alpha([2,3])
[1/3, 1/2, 2/3]
```

sage.modular.hypergeometric_motive.**cyclotomic_to_gamma**(*cyclo_up*, *cyclo_down*)

Convert a quotient of products of cyclotomic polynomials to a quotient of products of polynomials $x^n - 1$.

INPUT:

- `cyclo_up` – list of indices of cyclotomic polynomials in the numerator
- `cyclo_down` – list of indices of cyclotomic polynomials in the denominator

OUTPUT:

a dictionary mapping an integer n to the power of $x^n - 1$ that appears in the given product

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import cyclotomic_to_gamma
sage: cyclotomic_to_gamma([6], [1])
{2: -1, 3: -1, 6: 1}
```

sage.modular.hypergeometric_motive.**enumerate_hypergeometric_data**(*d*, *weight=None*)

Return an iterator over parameters of hypergeometric motives (up to swapping).

INPUT:

- `d` – the degree
- `weight` – optional integer, to specify the motivic weight

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import enumerate_hypergeometric_
↪data as enum
sage: l = [H for H in enum(6, weight=2) if H.hodge_numbers()[0] == 1]
sage: len(l)
112
```

sage.modular.hypergeometric_motive.**gamma_list_to_cyclotomic**(*galist*)

Convert a quotient of products of polynomials $x^n - 1$ to a quotient of products of cyclotomic polynomials.

INPUT:

- `galist` – a list of integers, where an integer n represents the power $(x^{|n|} - 1)^{\mathrm{sgn}(n)}$

OUTPUT:

a pair of list of integers, where k represents the cyclotomic polynomial Φ_k

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import gamma_list_to_cyclotomic
sage: gamma_list_to_cyclotomic([-1, -1, 2])
([2], [1])

sage: gamma_list_to_cyclotomic([-1, -1, -1, -3, 6])
([2, 6], [1, 1, 1])
```

(continues on next page)

(continued from previous page)

```
sage: gamma_list_to_cyclotomic([-1, 2, 3, -4])
([3], [4])

sage: gamma_list_to_cyclotomic([8,2,2,2,-6,-4,-3,-1])
([2, 2, 8], [3, 3, 6])
```

sage.modular.hypergeometric_motive.**possible_hypergeometric_data**(*d*, *weight=None*)

Return the list of possible parameters of hypergeometric motives (up to swapping).

INPUT:

- `d` – the degree
- `weight` – optional integer, to specify the motivic weight

EXAMPLES:

```
sage: from sage.modular.hypergeometric_motive import possible_hypergeometric_data␣
↪as P
sage: [len(P(i,weight=2)) for i in range(1, 7)]
[0, 0, 10, 30, 93, 234]
```

CHAPTER SEVENTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[CO1977] H. Cohen, J. Oesterlé, *Dimensions des espaces de formes modulaires*, p. 69-78 in Modular functions in one variable VI. Lecture Notes in Math. 627, Springer-Verlag, NewYork, 1977.

PYTHON MODULE INDEX

m

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

O

P

Q

R

S