L-Functions

Release 10.1

The Sage Development Team

Aug 21, 2023

CONTENTS

Sage includes several standard open source packages for computing with L-functions.

CHAPTER ONE

RUBINSTEIN'S L-FUNCTION CALCULATOR

This interface provides complete access to Rubinstein's lcalc calculator with extra PARI functionality compiled in and is a standard part of Sage.

Note: Each call to `lcalc` runs a complete `lcalc` process. On a typical Linux system, this entails about 0.3 seconds overhead.

AUTHORS:

- Michael Rubinstein (2005): released under GPL the C++ program lcalc
- William Stein (2006-03-05): wrote Sage interface to lcalc

class `sage.lfunctions.lcalc.LCalc`

Bases: `SageObject`

Rubinstein's L-functions Calculator

Type `lcalc.[tab]` for a list of useful commands that are implemented using the command line interface, but return objects that make sense in Sage. For each command the possible inputs for the L-function are:

- `''` - (default) the Riemann zeta function
- `'tau'` - the L function of the Ramanujan delta function
- elliptic curve E - where E is an elliptic curve over $\mathbf{Q}$; defines $L(E, s)$

You can also use the complete command-line interface of Rubinstein's L-functions calculations program via this class. Type `lcalc.help()` for a list of commands and how to call them.

analytic_rank(*L=''*)

Return the analytic rank of the L-function at the central critical point.

INPUT:

- `L` - defines L-function (default: Riemann zeta function)

OUTPUT: integer

Note: Of course this is not provably correct in general, since it is an open problem to compute analytic ranks provably correctly in general.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: lcalc.analytic_rank(E)
1
```

help()

twist_values(s, *dmin*, *dmax*, *L=''*)

Return values of $L(s, \chi_k)$ for each quadratic character χ_k whose discriminant d satisfies $d_{\min} \leq d \leq d_{\max}$.

INPUT:

- `s` - complex numbers
- `dmin` - integer
- `dmax` - integer
- `L` - defines L-function (default: Riemann zeta function)

OUTPUT:

- `list` - list of pairs (d, L(s,chi_d))

EXAMPLES:

```
sage: values = lcalc.twist_values(0.5, -10, 10)
sage: values[0][0]
-8
sage: values[0][1] # abs tol 1e-8
1.10042141 + 0.0*I
sage: values[1][0]
-7
sage: values[1][1] # abs tol 1e-8
1.14658567 + 0.0*I
sage: values[2][0]
-4
sage: values[2][1] # abs tol 1e-8
0.667691457 + 0.0*I
sage: values[3][0]
-3
sage: values[3][1] # abs tol 1e-8
0.480867558 + 0.0*I
sage: values[4][0]
5
sage: values[4][1] # abs tol 1e-8
0.231750947 + 0.0*I
sage: values[5][0]
8
sage: values[5][1] # abs tol 1e-8
0.373691713 + 0.0*I
```

twist_zeros(n, *dmin*, *dmax*, *L=''*)

Return first n real parts of nontrivial zeros for each quadratic character χ_k whose discriminant d satisfies $d_{\min} \leq d \leq d_{\max}$.

INPUT:

- `n` - integer

- `dmin` - integer
- `dmax` - integer
- `L` - defines L-function (default: Riemann zeta function)

OUTPUT:

- `dict` - keys are the discriminants d, and values are list of corresponding zeros.

EXAMPLES:

```
sage: lcalc.twist_zeros(3, -3, 6)
{-3: [8.03973716, 11.2492062, 15.7046192], 5: [6.64845335, 9.83144443, 11.
↪9588456]}
```

value(*s*, *L=''*)

Return $L(s)$ for s a complex number.

INPUT:

- `s` - complex number
- `L` - defines L-function (default: Riemann zeta function)

EXAMPLES:

```
sage: I = CC.0
sage: lcalc.value(0.5 + 100*I)
2.69261989 - 0.0203860296*I
```

Note, Sage can also compute zeta at complex numbers (using the PARI C library):

```
sage: (0.5 + 100*I).zeta()
2.69261988568132 - 0.0203860296025982*I
```

values_along_line(*s0*, *s1*, *number_samples*, *L=''*)

Return values of $L(s)$ at `number_samples` equally-spaced sample points along the line from s_0 to s_1 in the complex plane.

INPUT:

- `s0, s1` - complex numbers
- `number_samples` - integer
- `L` - defines L-function (default: Riemann zeta function)

OUTPUT:

- `list` - list of pairs (s, zeta(s)), where the s are equally spaced sampled points on the line from s0 to s1.

EXAMPLES:

```
sage: I = CC.0
sage: values = lcalc.values_along_line(0.5, 0.5+20*I, 5)
sage: values[0][0] # abs tol 1e-8
0.5
sage: values[0][1] # abs tol 1e-8
-1.46035451 + 0.0*I
sage: values[1][0] # abs tol 1e-8
```

(continues on next page)

(continued from previous page)

```
0.5 + 4.0*I
sage: values[1][1] # abs tol 1e-8
0.606783764 + 0.0911121400*I
sage: values[2][0] # abs tol 1e-8
0.5 + 8.0*I
sage: values[2][1] # abs tol 1e-8
1.24161511 + 0.360047588*I
sage: values[3][0] # abs tol 1e-8
0.5 + 12.0*I
sage: values[3][1] # abs tol 1e-8
1.01593665 - 0.745112472*I
sage: values[4][0] # abs tol 1e-8
0.5 + 16.0*I
sage: values[4][1] # abs tol 1e-8
0.938545408 + 1.21658782*I
```

Sometimes warnings are printed (by lcalc) when this command is run:

```
sage: E = EllipticCurve('389a')
sage: values = E.lseries().values_along_line(0.5, 3, 5)
sage: values[0][0] # abs tol 1e-8
0.0
sage: values[0][1] # abs tol 1e-8
0.209951303  + 0.0*I
sage: values[1][0] # abs tol 1e-8
0.5
sage: values[1][1] # abs tol 1e-8
0.0  + 0.0*I
sage: values[2][0] # abs tol 1e-8
1.0
sage: values[2][1] # abs tol 1e-8
0.133768433 - 0.0*I
sage: values[3][0] # abs tol 1e-8
1.5
sage: values[3][1] # abs tol 1e-8
0.360092864 - 0.0*I
sage: values[4][0] # abs tol 1e-8
2.0
sage: values[4][1] # abs tol 1e-8
0.552975867 + 0.0*I
```

zeros(n, $L=''$)

Return the imaginary parts of the first n nontrivial zeros of the L-function in the upper half plane, as 32-bit reals.

INPUT:

- `n` - integer
- `L` - defines L-function (default: Riemann zeta function)

This function also checks the Riemann Hypothesis and makes sure no zeros are missed. This means it looks for several dozen zeros to make sure none have been missed before outputting any zeros at all, so takes longer than `self.zeros_of_zeta_in_interval(...)`.

EXAMPLES:

```
sage: lcalc.zeros(4)                                # long time
[14.1347251, 21.0220396, 25.0108576, 30.4248761]
sage: lcalc.zeros(5, L='--tau')                     # long time
[9.22237940, 13.9075499, 17.4427770, 19.6565131, 22.3361036]
sage: lcalc.zeros(3, EllipticCurve('37a'))          # long time
[0.000000000, 5.00317001, 6.87039122]
```

zeros_in_interval(*x*, *y*, *stepsize*, *L=''*)

Return the imaginary parts of (most of) the nontrivial zeros of the L-function on the line $\Re(s) = 1/2$ with positive imaginary part between x and y, along with a technical quantity for each.

INPUT:

- `x, y, stepsize` - positive floating point numbers
- `L` - defines L-function (default: Riemann zeta function)

OUTPUT: list of pairs (zero, S(T)).

Rubinstein writes: The first column outputs the imaginary part of the zero, the second column a quantity related to $S(T)$ (it increases roughly by 2 whenever a sign change, i.e. pair of zeros, is missed). Higher up the critical strip you should use a smaller stepsize so as not to miss zeros.

EXAMPLES:

```
sage: lcalc.zeros_in_interval(10, 30, 0.1)
[(14.1347251, 0.184672916), (21.0220396, -0.0677893290), (25.0108576, -0.
↪0555872781)]
```

CHAPTER

TWO

WATKINS SYMMETRIC POWER L-FUNCTION CALCULATOR

SYMPOW is a package to compute special values of symmetric power elliptic curve L-functions. It can compute up to about 64 digits of precision. This interface provides complete access to sympow, which is a standard part of Sage (and includes the extra data files).

Note: Each call to `sympow` runs a complete `sympow` process. This incurs about 0.2 seconds overhead.

AUTHORS:

- Mark Watkins (2005-2006): wrote and released sympow
- William Stein (2006-03-05): wrote Sage interface

ACKNOWLEDGEMENT (from sympow readme):

- The quad-double package was modified from David Bailey's package: http://crd.lbl.gov/~dhbailey/mpdist/
- The `squfof` implementation was modified from Allan Steel's version of Arjen Lenstra's original LIP-based code.
- The `ec_ap` code was originally written for the kernel of MAGMA, but was modified to use small integers when possible.
- SYMPOW was originally developed using PARI, but due to licensing difficulties, this was eliminated. SYMPOW also does not use the standard math libraries unless Configure is run with the -lm option. SYMPOW still uses GP to compute the meshes of inverse Mellin transforms (this is done when a new symmetric power is added to datafiles).

class `sage.lfunctions.sympow.Sympow`

Bases: `SageObject`

Watkins Symmetric Power L-function Calculator

Type `sympow.[tab]` for a list of useful commands that are implemented using the command line interface, but return objects that make sense in Sage.

You can also use the complete command-line interface of sympow via this class. Type `sympow.help()` for a list of commands and how to call them.

`L`(*E*, *n*, *prec*)

Return $L(\mathrm{Sym}^{(n)}(E, \mathrm{edge}))$ to prec digits of precision, where edge is the *right* edge. Here n must be even.

INPUT:

- `E` - elliptic curve
- `n` - even integer
- `prec` - integer

OUTPUT:

- `string` - real number to prec digits of precision as a string.

Note: Before using this function for the first time for a given n, you may have to type `sympow('-new_data n')`, where `n` is replaced by your value of n.

If you would like to see the extensive output sympow prints when running this function, just type `set_verbose(2)`.

EXAMPLES:

These examples only work if you run `sympow -new_data 2` in a Sage shell first. Alternatively, within Sage, execute:

```
sage: sympow('-new_data 2')  # not tested
```

This command precomputes some data needed for the following examples.

```
sage: a = sympow.L(EllipticCurve('11a'), 2, 16)  # not tested
sage: a                                          # not tested
'1.057599244590958E+00'
sage: RR(a)                                      # not tested
1.05759924459096
```

Lderivs(*E*, *n*, *prec*, *d*)

Return 0^{th} to d^{th} derivatives of $L(\mathrm{Sym}^{(n)}(E, s)$ to prec digits of precision, where s is the right edge if n is even and the center if n is odd.

INPUT:

- `E` - elliptic curve
- `n` - integer (even or odd)
- `prec` - integer
- `d` - integer

OUTPUT: a string, exactly as output by sympow

Note: To use this function you may have to run a few commands like `sympow('-new_data 1d2')`, each which takes a few minutes. If this function fails it will indicate what commands have to be run.

EXAMPLES:

```
sage: print(sympow.Lderivs(EllipticCurve('11a'), 1, 16, 2))  # not tested
...
 1n0: 2.538418608559107E-01
 1w0: 2.538418608559108E-01
 1n1: 1.032321840884568E-01
 1w1: 1.059251499158892E-01
 1n2: 3.238743180659171E-02
 1w2: 3.414818600982502E-02
```

analytic_rank(E)

Return the analytic rank and leading L-value of the elliptic curve E.

INPUT:

- `E` - elliptic curve over Q

OUTPUT:

- `integer` - analytic rank
- `string` - leading coefficient (as string)

Note: The analytic rank is *not* computed provably correctly in general.

Note: In computing the analytic rank we consider $L^{(r)}(E, 1)$ to be 0 if $L^{(r)}(E, 1)/\Omega_E > 0.0001$.

EXAMPLES: We compute the analytic ranks of the lowest known conductor curves of the first few ranks:

```
sage: sympow.analytic_rank(EllipticCurve('11a'))
(0, '2.53842e-01')
sage: sympow.analytic_rank(EllipticCurve('37a'))
(1, '3.06000e-01')
sage: sympow.analytic_rank(EllipticCurve('389a'))
(2, '7.59317e-01')
sage: sympow.analytic_rank(EllipticCurve('5077a'))
(3, '1.73185e+00')
sage: sympow.analytic_rank(EllipticCurve([1, -1, 0, -79, 289]))
(4, '8.94385e+00')
sage: sympow.analytic_rank(EllipticCurve([0, 0, 1, -79, 342]))  # long time
(5, '3.02857e+01')
sage: sympow.analytic_rank(EllipticCurve([1, 1, 0, -2582, 48720]))  # long time
(6, '3.20781e+02')
sage: sympow.analytic_rank(EllipticCurve([0, 0, 0, -10012, 346900]))  # long
↪time
(7, '1.32517e+03')
```

help()

modular_degree(E)

Return the modular degree of the elliptic curve E, assuming the Stevens conjecture.

INPUT:

- `E` - elliptic curve over Q

OUTPUT:

- `integer` - modular degree

EXAMPLES: We compute the modular degrees of the lowest known conductor curves of the first few ranks:

```
sage: sympow.modular_degree(EllipticCurve('11a'))
1
sage: sympow.modular_degree(EllipticCurve('37a'))
2
```

(continues on next page)

(continued from previous page)

```
sage: sympow.modular_degree(EllipticCurve('389a'))
40
sage: sympow.modular_degree(EllipticCurve('5077a'))
1984
sage: sympow.modular_degree(EllipticCurve([1, -1, 0, -79, 289]))
334976
```

new_data(*n*)

Pre-compute data files needed for computation of n-th symmetric powers.

CHAPTER

THREE

DOKCHITSER'S L-FUNCTIONS CALCULATOR

Todo:

- add more examples from SAGE_EXTCODE/pari/dokchitser that illustrate use with Eisenstein series, number fields, etc.
- plug this code into number fields and modular forms code (elliptic curves are done).

AUTHORS:

- Tim Dokchitser (2002): original PARI code and algorithm (and the documentation below is based on Dokchitser's docs).
- William Stein (2006-03-08): Sage interface

class sage.lfunctions.dokchitser.**Dokchitser**(**args*, ***kwargs*)

Bases: SageObject

Dokchitser's L-functions Calculator

Create a Dokchitser L-series with

Dokchitser(conductor, gammaV, weight, eps, poles, residues, init, prec)

where

- conductor – integer, the conductor
- gammaV – list of Gamma-factor parameters, e.g. [0] for Riemann zeta, [0,1] for ell.curves, (see examples).
- weight – positive real number, usually an integer e.g. 1 for Riemann zeta, 2 for H^1 of curves/$\mathbf{Q}$
- eps – complex number; sign in functional equation
- poles – (default: []) list of points where $L^*(s)$ has (simple) poles; only poles with $Re(s) > weight/2$ should be included
- residues – vector of residues of $L^*(s)$ in those poles or set residues='automatic' (default value)
- prec – integer (default: 53) number of *bits* of precision

RIEMANN ZETA FUNCTION:

We compute with the Riemann Zeta function.

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1],␣
↪residues=[-1], init='1')
sage: L
Dokchitser L-series of conductor 1 and weight 1
```

(continues on next page)

(continued from previous page)

```
sage: L(1)
Traceback (most recent call last):
...
ArithmeticError
sage: L(2)
1.64493406684823
sage: L(2, 1.1)
1.64493406684823
sage: L.derivative(2)
-0.937548254315844
sage: h = RR('0.0000000000001')
sage: (zeta(2+h) - zeta(2.))/h
-0.937028232783632
sage: L.taylor_series(2, k=5)
1.64493406684823 - 0.937548254315844*z + 0.994640117149451*z^2 - 1.00002430047384*z^
↪3 + 1.00006193307...*z^4 + O(z^5)
```

RANK 1 ELLIPTIC CURVE:

We compute with the L-series of a rank 1 curve.

```
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser(algorithm='gp'); L
Dokchitser L-function associated to Elliptic Curve defined by y^2 + y = x^3 - x␣
↪over Rational Field
sage: L(1)
0.000000000000000
sage: L.derivative(1)
0.305999773834052
sage: L.derivative(1,2)
0.373095594536324
sage: L.num_coeffs()
48
sage: L.taylor_series(1,4)
0.000000000000000 + 0.305999773834052*z + 0.186547797268162*z^2 - 0.
↪136791463097188*z^3 + O(z^4)
sage: L.check_functional_equation()  # abs tol 1e-19
6.04442711160669e-18
```

RANK 2 ELLIPTIC CURVE:

We compute the leading coefficient and Taylor expansion of the L-series of a rank 2 elliptic curve.

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser(algorithm='gp')
sage: L.num_coeffs()
156
sage: L.derivative(1,E.rank())
1.51863300057685
sage: L.taylor_series(1,4)
-1.27685190980159e-23 + (7.23588070754027e-24)*z + 0.759316500288427*z^2 - 0.
↪430302337583362*z^3 + O(z^4)  # 32-bit
-2.72911738151096e-23 + (1.54658247036311e-23)*z + 0.759316500288427*z^2 - 0.
↪430302337583362*z^3 + O(z^4)  # 64-bit
```

NUMBER FIELD:

We compute with the Dedekind zeta function of a number field.

```
sage: x = var('x')
sage: K = NumberField(x**4 - x**2 - 1,'a')
sage: L = K.zeta_function(algorithm='gp')
sage: L.conductor
400
sage: L.num_coeffs()
264
sage: L(2)
1.10398438736918
sage: L.taylor_series(2,3)
1.10398438736918 - 0.215822638498759*z + 0.279836437522536*z^2 + O(z^3)
```

RAMANUJAN DELTA L-FUNCTION:

The coefficients are given by Ramanujan's tau function:

```
sage: L = Dokchitser(conductor=1, gammaV=[0,1], weight=12, eps=1)
sage: pari_precode = 'tau(n)=(5*sigma(n,3)+7*sigma(n,5))*n/12 - 35*sum(k=1,n-1,(6*k-
↪4*(n-k))*sigma(k,3)*sigma(n-k,5))'
sage: L.init_coeffs('tau(k)', pari_precode=pari_precode)
```

We redefine the default bound on the coefficients: Deligne's estimate on tau(n) is better than the default coefgrow(n)=`(4n)^{11/2}` (by a factor 1024), so re-defining coefgrow() improves efficiency (slightly faster).

```
sage: L.num_coeffs()
12
sage: L.set_coeff_growth('2*n^(11/2)')
sage: L.num_coeffs()
11
```

Now we're ready to evaluate, etc.

```
sage: L(1)
0.0374412812685155
sage: L(1, 1.1)
0.0374412812685155
sage: L.taylor_series(1,3)
0.0374412812685155 + 0.0709221123619322*z + 0.0380744761270520*z^2 + O(z^3)
```

check_functional_equation(*T=1.2*)

Verifies how well numerically the functional equation is satisfied, and also determines the residues if `self.poles != []` and residues='automatic'.

More specifically: for $T > 1$ (default 1.2), `self.check_functional_equation(T)` should ideally return 0 (to the current precision).

- if what this function returns does not look like 0 at all, probably the functional equation is wrong (i.e. some of the parameters gammaV, conductor etc., or the coefficients are wrong),
- if checkfeq(T) is to be used, more coefficients have to be generated (approximately T times more), e.g. call cflength(1.3), initLdata("a(k)",1.3), checkfeq(1.3)
- T=1 always (!) returns 0, so T has to be away from 1

- default value $T = 1.2$ seems to give a reasonable balance
- if you don't have to verify the functional equation or the L-values, call num_coeffs(1) and initLdata("a(k)",1), you need slightly less coefficients.

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1],␣
↪residues=[-1], init='1')
sage: L.check_functional_equation()  # abs tol 1e-19
-2.71050543121376e-20
```

If we choose the sign in functional equation for the ζ function incorrectly, the functional equation doesn't check out.

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=-11, poles=[1],␣
↪residues=[-1], init='1')
sage: L.check_functional_equation()
-9.73967861488124
```

derivative(*s*, *k=1*)

Return the k-th derivative of the L-series at s.

Warning: If k is greater than the order of vanishing of L at s you may get nonsense.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser(algorithm='gp')
sage: L.derivative(1,E.rank())
1.51863300057685
```

gp()

Return the gp interpreter that is used to implement this Dokchitser L-function.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: L = E.lseries().dokchitser(algorithm='gp')
sage: L(2)
0.546048036215014
sage: L.gp()
PARI/GP interpreter
```

init_coeffs(*v*, *cutoff=1*, *w=None*, *pari_precode=''*, *max_imaginary_part=0*, *max_asymp_coeffs=40*)

Set the coefficients a_n of the L-series.

If $L(s)$ is not equal to its dual, pass the coefficients of the dual as the second optional argument.

INPUT:

- `v` – list of complex numbers or string (pari function of k)
- `cutoff` – real number = 1 (default: 1)
- `w` – list of complex numbers or string (pari function of k)

- `pari_precode` – some code to execute in pari before calling initLdata
- `max_imaginary_part` – (default: 0): redefine if you want to compute L(s) for s having large imaginary part,
- `max_asymp_coeffs` – (default: 40): at most this many terms are generated in asymptotic series for phi(t) and G(s,t).

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0,1], weight=12, eps=1)
sage: pari_precode = 'tau(n)=(5*sigma(n,3)+7*sigma(n,5))*n/12 - 35*sum(k=1,n-1,
↪(6*k-4*(n-k))*sigma(k,3)*sigma(n-k,5))'
sage: L.init_coeffs('tau(k)', pari_precode=pari_precode)
```

Evaluate the resulting L-function at a point, and compare with the answer that one gets "by definition" (of L-function attached to a modular form):

```
sage: L(14)
0.998583063162746
sage: a = delta_qexp(1000)
sage: sum(a[n]/float(n)^14 for n in range(1,1000))
0.9985830631627459
```

Illustrate that one can give a list of complex numbers for v (see github issue #10937):

```
sage: L2 = Dokchitser(conductor=1, gammaV=[0,1], weight=12, eps=1)
sage: L2.init_coeffs(list(delta_qexp(1000))[1:])
sage: L2(14)
0.998583063162746
```

num_coeffs(*T=1*)

Return number of coefficients a_n that are needed in order to perform most relevant L-function computations to the desired precision.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: L = E.lseries().dokchitser(algorithm='gp')
sage: L.num_coeffs()
26
sage: E = EllipticCurve('5077a')
sage: L = E.lseries().dokchitser(algorithm='gp')
sage: L.num_coeffs()
568
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1],␣
↪residues=[-1], init='1')
sage: L.num_coeffs()
4
```

Verify that `num_coeffs` works with non-real spectral parameters, e.g. for the L-function of the level 10 Maass form with eigenvalue 2.7341055592527126:

```
sage: ev = 2.7341055592527126
sage: L = Dokchitser(conductor=10, gammaV=[ev*i, -ev*i],weight=2,eps=1)
```

(continues on next page)

(continued from previous page)

```
sage: L.num_coeffs()
26
```

set_coeff_growth(*coefgrow*)

You might have to redefine the coefficient growth function if the a_n of the L-series are not given by the following PARI function:

```
coefgrow(n) = if(length(Lpoles),
                1.5*n^(vecmax(real(Lpoles))-1),
                sqrt(4*n)^(weight-1));
```

INPUT:

- `coefgrow` – string that evaluates to a PARI function of n that defines a coefgrow function.

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0,1], weight=12, eps=1)
sage: pari_precode = 'tau(n)=(5*sigma(n,3)+7*sigma(n,5))*n/12 - 35*sum(k=1,n-1,
↪(6*k-4*(n-k))*sigma(k,3)*sigma(n-k,5))'
sage: L.init_coeffs('tau(k)', pari_precode=pari_precode)
sage: L.set_coeff_growth('2*n^(11/2)')
sage: L(1)
0.0374412812685155
```

taylor_series(*a=0*, *k=6*, *var='z'*)

Return the first k terms of the Taylor series expansion of the L-series about a.

This is returned as a series in `var`, where you should view `var` as equal to $s - a$. Thus this function returns the formal power series whose coefficients are $L^{(n)}(a)/n!$.

INPUT:

- `a` – complex number (default: 0); point about which to expand
- `k` – integer (default: 6), series is $O(``var``^k)$
- `var` – string (default: 'z'), variable of power series

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1],␣
↪residues=[-1], init='1')
sage: L.taylor_series(2, 3)
1.64493406684823 - 0.937548254315844*z + 0.994640117149451*z^2 + O(z^3)
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser(algorithm='gp')
sage: L.taylor_series(1)
0.000000000000000 + 0.305999773834052*z + 0.186547797268162*z^2 - 0.
↪136791463097188*z^3 + 0.0161066468496401*z^4 + 0.0185955175398802*z^5 + O(z^6)
```

We compute a Taylor series where each coefficient is to high precision.

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser(200, algorithm='gp')
sage: L.taylor_series(1,3)
```

(continues on next page)

(continued from previous page)

```
...e-82 + (...e-82)*z + 0.
↪759316500288426770230192607894722019078097516494924351585781*z^2 + O(z^3)
```

Check that github issue #25402 is fixed:

```
sage: L = EllipticCurve("24a1").modular_form().lseries()
sage: L.taylor_series(-1, 3)
0.000000000000000 - 0.702565506265199*z + 0.638929001045535*z^2 + O(z^3)
```

Check that github issue #25965 is fixed:

```
sage: L2 = EllipticCurve("37a1").modular_form().lseries(); L2
L-series associated to the cusp form q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + O(q^6)
sage: L2.taylor_series(0,4)
0.000000000000000 - 0.357620466127498*z + 0.273373112603865*z^2 + 0.
↪303362857047671*z^3 + O(z^4)
sage: L2.taylor_series(0,1)
O(z^1)
sage: L2(0)
0.000000000000000
```

sage.lfunctions.dokchitser.**reduce_load_dokchitser**(*D*)

CHAPTER FOUR

CLASS FOR COMPUTING SUMS OVER ZEROS OF MOTIVIC L-FUNCTIONS

All computations are done to double precision.

AUTHORS:

- Simon Spicer (2014-10): first version

`sage.lfunctions.zero_sums.LFunctionZeroSum`(*X*, **args*, ***kwds*)

Constructor for the LFunctionZeroSum class.

INPUT:

- `X` – A motivic object. Currently only implemented for X = an elliptic curve over the rational numbers.

OUTPUT:

An LFunctionZeroSum object.

EXAMPLES:

```
sage: E = EllipticCurve("389a")
sage: Z = LFunctionZeroSum(E); Z
Zero sum estimator for L-function attached to Elliptic Curve defined by y^2 + y = x^
↪3 + x^2 - 2*x over Rational Field
```

class `sage.lfunctions.zero_sums.LFunctionZeroSum_EllipticCurve`

Bases: *LFunctionZeroSum_abstract*

Subclass for computing certain sums over zeros of an elliptic curve L-function without having to determine the zeros themselves.

`analytic_rank_upper_bound`(*max_Delta=None*, *adaptive=True*, *root_number='compute'*, *bad_primes=None*, *ncpus=None*)

Return an upper bound for the analytic rank of the L-function $L_E(s)$ attached to `self`, conditional on the Generalized Riemann Hypothesis, via computing the zero sum $\sum_\gamma f(\Delta\gamma)$, where γ ranges over the imaginary parts of the zeros of $L(E, s)$ along the critical strip, $f(x) = \left(\frac{\sin(\pi x)}{\pi x}\right)^2$, and Δ is the tightness parameter whose maximum value is specified by max_Delta.

This computation can be run on curves with very large conductor (so long as the conductor is known or quickly computable) when Delta is not too large (see below).

Uses Bober's rank bounding method as described in [Bob2013].

INPUT:

- `max_Delta` – (default: `None`) If not `None`, a positive real value specifying the maximum Delta value used in the zero sum; larger values of Delta yield better bounds - but runtime is exponential in Delta. If left as `None`, Delta is set to $\min\left\{\frac{1}{\pi}\left(\log(N+1000)/2-\log(2\pi)-\eta\right), 2.5\right\}$, where N is the conductor of the curve attached to `self`, and η is the Euler-Mascheroni constant $= 0.5772...$; the crossover point is at conductor ~8.3*10^8. For the former value, empirical results show that for about 99.7% of all curves the returned value is the actual analytic rank.
- `adaptive` – (default: `True`) Boolean
  - If `True`, the computation is first run with small and then successively larger Delta values up to max_Delta. If at any point the computed bound is 0 (or 1 when root_number is -1 or `True`), the computation halts and that value is returned; otherwise the minimum of the computed bounds is returned.
  - If `False`, the computation is run a single time with Delta=max_Delta, and the resulting bound returned.
- `root_number` – (default: "compute") String or integer
  - `"compute"` – the root number of self is computed and used to (possibly) lower the analytic rank estimate by 1.
  - `"ignore"` – the above step is omitted
  - `1` – this value is assumed to be the root number of self. This is passable so that rank estimation can be done for curves whose root number has been precomputed.
  - `-1` – this value is assumed to be the root number of self. This is passable so that rank estimation can be done for curves whose root number has been precomputed.
- `bad_primes` – (default: `None`) If not `None`, a list of the primes of bad reduction for the curve attached to `self`. This is passable so that rank estimation can be done for curves of large conductor whose bad primes have been precomputed.
- `ncpus` – (default: `None`) If not `None`, a positive integer defining the maximum number of CPUs to be used for the computation. If left as `None`, the maximum available number of CPUs will be used. Note: Multiple processors will only be used for Delta values >= 1.75.

Note: Output will be incorrect if the incorrect root number is specified.

Warning: Zero sum computation time is exponential in the tightness parameter Δ, roughly doubling for every increase of 0.1 thereof. Using $\Delta = 1$ (and adaptive=False) will yield a runtime of a few milliseconds; $\Delta = 2$ takes a few seconds, and $\Delta = 3$ may take upwards of an hour. Increase beyond this at your own risk!

OUTPUT:

A non-negative integer greater than or equal to the analytic rank of self. If the returned value is 0 or 1 (the latter if parity is not False), then this is the true analytic rank of self.

Note: If you use set_verbose(1), extra information about the computation will be printed.

See also:

`LFunctionZeroSum()` `EllipticCurve.root_number()` `set_verbose()`

EXAMPLES:

For most elliptic curves with small conductor the central zero(s) of $L_E(s)$ are fairly isolated, so small values of Δ will yield tight rank estimates.

```
sage: E = EllipticCurve("11a")
sage: E.rank()
0
sage: Z = LFunctionZeroSum(E)
sage: Z.analytic_rank_upper_bound(max_Delta=1,ncpus=1)
0

sage: E = EllipticCurve([-39,123])
sage: E.rank()
1
sage: Z = LFunctionZeroSum(E)
sage: Z.analytic_rank_upper_bound(max_Delta=1)
1
```

This is especially true for elliptic curves with large rank.

```
sage: for r in range(9):
....:     E = elliptic_curves.rank(r)[0]
....:     print((r, E.analytic_rank_upper_bound(max_Delta=1,
....:     adaptive=False,root_number="ignore")))
(0, 0)
(1, 1)
(2, 2)
(3, 3)
(4, 4)
(5, 5)
(6, 6)
(7, 7)
(8, 8)
```

However, some curves have L-functions with low-lying zeroes, and for these larger values of Δ must be used to get tight estimates.

```
sage: E = EllipticCurve("974b1")
sage: r = E.rank(); r
0
sage: Z = LFunctionZeroSum(E)
sage: Z.analytic_rank_upper_bound(max_Delta=1,root_number="ignore")
1
sage: Z.analytic_rank_upper_bound(max_Delta=1.3,root_number="ignore")
0
```

Knowing the root number of E allows us to use smaller Delta values to get tight bounds, thus speeding up runtime considerably.

```
sage: Z.analytic_rank_upper_bound(max_Delta=0.6,root_number="compute")
0
```

The are a small number of curves which have pathologically low-lying zeroes. For these curves, this method will produce a bound that is strictly larger than the analytic rank, unless very large values of Delta are used. The following curve (“256944c1” in the Cremona tables) is a rank 0 curve with a zero at 0.0256…; the smallest Delta value for which the zero sum is strictly less than 2 is ~2.815.

```
sage: E = EllipticCurve([0, -1, 0, -7460362000712, -7842981500851012704])
sage: N,r = E.conductor(),E.analytic_rank(); N, r
(256944, 0)
sage: E.analytic_rank_upper_bound(max_Delta=1,adaptive=False)
2
sage: E.analytic_rank_upper_bound(max_Delta=2,adaptive=False)
2
```

This method is can be called on curves with large conductor.

```
sage: E = EllipticCurve([-2934,19238])
sage: Z = LFunctionZeroSum(E)
sage: Z.analytic_rank_upper_bound()
1
```

And it can bound rank on curves with *very* large conductor, so long as you know beforehand/can easily compute the conductor and primes of bad reduction less than $e^{2\pi\Delta}$. The example below is of the rank 28 curve discovered by Elkies that is the elliptic curve of (currently) largest known rank.

```
sage: a4 = -20067762415575526585033208209338542750930230312178956502
sage: a6 =␣
↪34481611795030556467032985690390720374855944359319180361266008296291939448732243429
sage: E = EllipticCurve([1,-1,1,a4,a6])
sage: bad_primes = [2,3,5,7,11,13,17,19,48463]
sage: N =␣
↪345560110835754734153225386490160523119851150579373313890059518947214472478145663538015414987
sage: Z = LFunctionZeroSum(E,N)
sage: Z.analytic_rank_upper_bound(max_Delta=2.37,adaptive=False, # long time
....: root_number=1,bad_primes=bad_primes,ncpus=2)
32
```

cn(n)

Return the nth Dirichlet coefficient of the logarithmic derivative of the L-function attached to `self`, shifted so that the critical line lies on the imaginary axis.

The returned value is zero if n is not a perfect prime power; when $n = p^e$ for p a prime of bad reduction it is $-a_p^e log(p)/p^e$, where a_p is $+1, -1$ or 0 according to the reduction type of p; and when $n = p^e$ for a prime p of good reduction, the value is $-(\alpha_p^e + \beta_p^e)\log(p)/p^e$, where α_p and β_p are the two complex roots of the characteristic equation of Frobenius at p on E.

INPUT:

- `n` – non-negative integer

OUTPUT:

A real number which (by Hasse's Theorem) is at most $2\frac{log(n)}{\sqrt{n}}$ in magnitude.

EXAMPLES:

```
sage: E = EllipticCurve("11a")
sage: Z = LFunctionZeroSum(E)
sage: for n in range(12): print((n, Z.cn(n))) # tol 1.0e-13
(0, 0.0)
(1, 0.0)
(2, 0.6931471805599453)
```

(continues on next page)

(continued from previous page)

```
(3, 0.3662040962227033)
(4, 0.0)
(5, -0.32188758248682003)
(6, 0.0)
(7, 0.555974328301518)
(8, -0.34657359027997264)
(9, 0.6103401603711721)
(10, 0.0)
(11, -0.21799047934530644)
```

elliptic_curve()

Return the elliptic curve associated with `self`.

EXAMPLES:

```
sage: E = EllipticCurve([23,100])
sage: Z = LFunctionZeroSum(E)
sage: Z.elliptic_curve()
Elliptic Curve defined by y^2 = x^3 + 23*x + 100 over Rational Field
```

lseries()

Return the L-series associated with `self`.

EXAMPLES:

```
sage: E = EllipticCurve([23,100])
sage: Z = LFunctionZeroSum(E)
sage: Z.lseries()
Complex L-series of the Elliptic Curve defined by y^2 = x^3 + 23*x + 100 over␣
↪Rational Field
```

class sage.lfunctions.zero_sums.**LFunctionZeroSum_abstract**

Bases: `SageObject`

Abstract class for computing certain sums over zeros of a motivic L-function without having to determine the zeros themselves.

C0(*include_euler_gamma=True*)

Return the constant term of the logarithmic derivative of the completed L-function attached to `self`.

This is equal to $-\eta + \log(N)/2 - \log(2\pi)$, where η is the Euler-Mascheroni constant $= 0.5772...$ and N is the level of the form attached to `self`.

INPUT:

- `include_euler_gamma` – bool (default: `True`); if set to `False`, return the constant $\log(N)/2 - \log(2\pi)$, i.e., do not subtract off the Euler-Mascheroni constant.

EXAMPLES:

```
sage: E = EllipticCurve("389a")
sage: Z = LFunctionZeroSum(E)
sage: Z.C0() # tol 1.0e-13
0.5666969404983447
sage: Z.C0(include_euler_gamma=False) # tol 1.0e-13
1.1439126053998776
```

cnlist(*n*, *python_floats=False*)

Return a list of Dirichlet coefficient of the logarithmic derivative of the L-function attached to `self`, shifted so that the critical line lies on the imaginary axis, up to and including n.

The i-th element of the returned list is a[i].

INPUT:

- `n` – non-negative integer
- `python_floats` – bool (default: `False`); if `True` return a list of Python floats instead of Sage Real Double Field elements.

OUTPUT:

A list of real numbers

See also:

cn()

Todo: Speed this up; make more efficient

EXAMPLES:

```
sage: E = EllipticCurve("11a")
sage: Z = LFunctionZeroSum(E)
sage: cnlist = Z.cnlist(11)
sage: for n in range(12): print((n, cnlist[n])) # tol 1.0e-13
(0, 0.0)
(1, 0.0)
(2, 0.6931471805599453)
(3, 0.3662040962227033)
(4, 0.0)
(5, -0.32188758248682003)
(6, 0.0)
(7, 0.555974328301518)
(8, -0.34657359027997264)
(9, 0.6103401603711721)
(10, 0.0)
(11, -0.21799047934530644)
```

completed_logarithmic_derivative(*s*, *num_terms=10000*)

Compute the value of the completed logarithmic derivative $\frac{\Lambda'}{\Lambda}$ at the point s to *low* precision, where $\Lambda = N^{s/2}(2\pi)^s\Gamma(s)L(s)$ and L is the L-function attached to `self`.

Warning: This is computed naively by evaluating the Dirichlet series for $\frac{L'}{L}$; the convergence thereof is controlled by the distance of s from the critical strip $0.5 <= \Re(s) <= 1.5$. You may use this method to attempt to compute values inside the critical strip; however, results are then *not* guaranteed to be correct to any number of digits.

INPUT:

- `s` – Real or complex value
- `num_terms` – (default: 10000) the maximum number of terms summed in the Dirichlet series.

OUTPUT:

A tuple (z,err), where z is the computed value, and err is an upper bound on the truncation error in this value introduced by truncating the Dirichlet sum.

Note: For the default term cap of 10000, a value accurate to all 53 bits of a double precision floating point number is only guaranteed when $|\Re(s-1)| > 4.58$, although in practice inputs closer to the critical strip will still yield computed values close to the true value.

See also:

`logarithmic_derivative()`

EXAMPLES:

```
sage: E = EllipticCurve([23,100])
sage: Z = LFunctionZeroSum(E)
sage: Z.completed_logarithmic_derivative(3) # tol 1.0e-13
(6.64372066048195, 6.584671359095225e-06)
```

Complex values are handled. The function is odd about s=1, so the value at 2-s should be minus the value at s.

```
sage: Z.completed_logarithmic_derivative(complex(-2.2,1)) # tol 1.0e-13
(-6.898080633125154 + 0.22557015394248361*I, 5.623853049808912e-11)
sage: Z.completed_logarithmic_derivative(complex(4.2,-1)) # tol 1.0e-13
(6.898080633125154 - 0.22557015394248361*I, 5.623853049808912e-11)
```

digamma(*s*, *include_constant_term=True*)

Return the digamma function $F(s)$ on the complex input s.

This is given by $F(s) = -\eta + \sum_{k=1}^{\infty} \frac{s-1}{k(k+s-1)}$, where η is the Euler-Mascheroni constant $= 0.5772156649\ldots$.

This function is needed in the computing the logarithmic derivative of the L-function attached to `self`.

INPUT:

- `s` – A complex number
- `include_constant_term` – (default: `True`) boolean; if set to `False`, only the value of the sum over k is returned without subtracting off the Euler-Mascheroni constant, i.e. the returned value is equal to $\sum_{k=1}^{\infty} \frac{s-1}{k(k+s-1)}$.

OUTPUT:

A real double precision number if the input is real and not a negative integer; Infinity if the input is a negative integer, and a complex number otherwise.

EXAMPLES:

```
sage: Z = LFunctionZeroSum(EllipticCurve("37a"))
sage: Z.digamma(3.2) # tol 1.0e-13
0.9988388912865993
sage: Z.digamma(3.2,include_constant_term=False) # tol 1.0e-13
1.576054556188132
sage: Z.digamma(1+I) # tol 1.0e-13
0.09465032062247625 + 1.076674047468581*I
```

(continues on next page)

(continued from previous page)

```
sage: Z.digamma(-2)
+Infinity
```

Evaluating the sum without the constant term at the positive integers n returns the (n-1)th harmonic number.

```
sage: Z.digamma(3,include_constant_term=False)
1.5
sage: Z.digamma(6,include_constant_term=False)
2.283333333333333
```

level()

Return the level of the form attached to `self`.

If `self` was constructed from an elliptic curve, then this is equal to the conductor of E.

EXAMPLES:

```
sage: E = EllipticCurve("389a")
sage: Z = LFunctionZeroSum(E)
sage: Z.level()
389
```

logarithmic_derivative(*s*, *num_terms=10000*, *as_interval=False*)

Compute the value of the logarithmic derivative $\frac{L'}{L}$ at the point s to *low* precision, where L is the L-function attached to `self`.

> **Warning:** The value is computed naively by evaluating the Dirichlet series for $\frac{L'}{L}$; convergence is controlled by the distance of s from the critical strip $0.5 <= \Re(s) <= 1.5$. You may use this method to attempt to compute values inside the critical strip; however, results are then *not* guaranteed to be correct to any number of digits.

INPUT:

- `s` – Real or complex value
- `num_terms` – (default: 10000) the maximum number of terms summed in the Dirichlet series.

OUTPUT:

A tuple (z,err), where z is the computed value, and err is an upper bound on the truncation error in this value introduced by truncating the Dirichlet sum.

> **Note:** For the default term cap of 10000, a value accurate to all 53 bits of a double precision floating point number is only guaranteed when $|\Re(s-1)| > 4.58$, although in practice inputs closer to the critical strip will still yield computed values close to the true value.

EXAMPLES:

```
sage: E = EllipticCurve([23,100])
sage: Z = LFunctionZeroSum(E)
sage: Z.logarithmic_derivative(10) # tol 1.0e-13
(5.648066742632698e-05, 1.0974102859764345e-34)
```

(continues on next page)

(continued from previous page)

```
sage: Z.logarithmic_derivative(2.2) # tol 1.0e-13
(0.5751257063594758, 0.024087912696974387)
```

Increasing the number of terms should see the truncation error decrease.

```
sage: Z.logarithmic_derivative(2.2,num_terms=50000) # long time # rel tol 1.0e-
↪14
(0.5751579645060139, 0.008988775519160675)
```

Attempting to compute values inside the critical strip gives infinite error.

```
sage: Z.logarithmic_derivative(1.3) # tol 1.0e-13
(5.442994413920786, +Infinity)
```

Complex inputs and inputs to the left of the critical strip are allowed.

```
sage: Z.logarithmic_derivative(complex(3,-1)) # tol 1.0e-13
(0.04764548578052381 + 0.16513832809989326*I, 6.584671359095225e-06)
sage: Z.logarithmic_derivative(complex(-3,-1.1)) # tol 1.0e-13
(-13.908452173241546 + 2.591443099074753*I, 2.7131584736258447e-14)
```

The logarithmic derivative has poles at the negative integers.

```
sage: Z.logarithmic_derivative(-3) # tol 1.0e-13
(-Infinity, 2.7131584736258447e-14)
```

ncpus(*n=None*)

Set or return the number of CPUs to be used in parallel computations.

If called with no input, the number of CPUs currently set is returned; else this value is set to n. If n is 0 then the number of CPUs is set to the max available.

INPUT:

- `n` – (default: `None`) If not `None`, a nonnegative integer

OUTPUT:

If n is not `None`, returns a positive integer

EXAMPLES:

```
sage: Z = LFunctionZeroSum(EllipticCurve("389a"))
sage: Z.ncpus()
1
sage: Z.ncpus(2)
sage: Z.ncpus()
2
```

The following output will depend on the system that Sage is running on.

```
sage: Z.ncpus(0)
sage: Z.ncpus()            # random
4
```

weight()

Return the weight of the form attached to `self`.

If `self` was constructed from an elliptic curve, then this is 2.

EXAMPLES:

```
sage: E = EllipticCurve("389a")
sage: Z = LFunctionZeroSum(E)
sage: Z.weight()
2
```

zerosum(*Delta=1*, *tau=0*, *function='sincsquared_fast'*, *ncpus=None*)

Bound from above the analytic rank of the form attached to `self`.

This bound is obtained by computing $\sum_\gamma f(\Delta(\gamma-\tau))$, where γ ranges over the imaginary parts of the zeros of $L_E(s)$ along the critical strip, and $f(x)$ is an appropriate even continuous L_2 function such that $f(0) = 1$.

If $\tau = 0$, then as Δ increases this sum converges from above to the analytic rank of the L-function, as $f(0) = 1$ is counted with multiplicity r, and the other terms all go to 0 uniformly.

INPUT:

- `Delta` – positive real number (default: 1) parameter denoting the tightness of the zero sum.
- `tau` – real parameter (default: 0) denoting the offset of the sum to be computed. When $\tau = 0$ the sum will converge to the analytic rank of the L-function as Δ is increased. If τ is the value of the imaginary part of a noncentral zero, the limit will be 1 (assuming the zero is simple); otherwise, the limit will be 0. Currently only implemented for the sincsquared and cauchy functions; otherwise ignored.
- `function` – string (default: "sincsquared_fast") - the function $f(x)$ as described above. Currently implemented options for f are
  - `sincsquared` – $f(x) = \left(\frac{\sin(\pi x)}{\pi x}\right)^2$
  - `gaussian` – $f(x) = e^{-x^2}$
  - `sincsquared_fast` – Same as "sincsquared", but implementation optimized for elliptic curve L-functions, and tau must be 0. self must be attached to an elliptic curve over $\mathbf{Q}$ given by its global minimal model, otherwise the returned result will be incorrect.
  - `sincsquared_parallel` – Same as "sincsquared_fast", but optimized for parallel computation with large (>2.0) Δ values. self must be attached to an elliptic curve over $\mathbf{Q}$ given by its global minimal model, otherwise the returned result will be incorrect.
  - `cauchy` – $f(x) = \frac{1}{1+x^2}$; this is only computable to low precision, and only when $\Delta < 2$.
- `ncpus` – (default: `None`) If not `None`, a positive integer defining the number of CPUs to be used for the computation. If left as `None`, the maximum available number of CPUs will be used. Only implemented for algorithm="sincsquared_parallel"; ignored otherwise.

Warning: Computation time is exponential in Δ, roughly doubling for every increase of 0.1 thereof. Using $\Delta = 1$ will yield a computation time of a few milliseconds; $\Delta = 2$ takes a few seconds, and $\Delta = 3$ takes upwards of an hour. Increase at your own risk beyond this!

OUTPUT:

A positive real number that bounds from above the number of zeros with imaginary part equal to τ. When $\tau = 0$ this is an upper bound for the L-function's analytic rank.

See also:

`analytic_rank_bound()` for more documentation and examples on calling this method on elliptic curve L-functions.

EXAMPLES:

```
sage: E = EllipticCurve("389a"); E.rank()
2
sage: Z = LFunctionZeroSum(E)
sage: E.lseries().zeros(3)
[0.000000000, 0.000000000, 2.87609907]
sage: Z.zerosum(Delta=1,function="sincsquared_fast") # tol 1.0e-13
2.037500084595065
sage: Z.zerosum(Delta=1,function="sincsquared_parallel") # tol 1.0e-11
2.037500084595065
sage: Z.zerosum(Delta=1,function="sincsquared") # tol 1.0e-13
2.0375000845950644
sage: Z.zerosum(Delta=1,tau=2.876,function="sincsquared") # tol 1.0e-13
1.075551295651154
sage: Z.zerosum(Delta=1,tau=1.2,function="sincsquared") # tol 1.0e-13
0.10831555377490683
sage: Z.zerosum(Delta=1,function="gaussian") # tol 1.0e-13
2.056890425029435
```

CHAPTER

FIVE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

l

INDEX

S

T

V

W

Z