<table>
<thead>
<tr>
<th></th>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rubinstein’s $L$-function calculator</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Watkins symmetric power $L$-function calculator</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Dokchitser’s $L$-functions calculator</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>Class for computing sums over zeros of motivic $L$-functions</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>$L$-functions from PARI</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>Indices and Tables</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>61</td>
</tr>
</tbody>
</table>
Sage includes several standard open source packages for computing with $L$-functions.
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

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

```python
>>> from sage.all import *

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

`help()`

**twist_values** \((s, d_{\text{min}}, d_{\text{max}}, L='')\)

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

**INPUT:**

- \(s\) – complex numbers
- \(d_{\text{min}}\) – integer
- \(d_{\text{max}}\) – integer
- \(L\) – defines \(L\)-function (default: Riemann zeta function)

**OUTPUT:**

- list – list of pairs \((d, L(s, \chi_d))\)

**EXAMPLES:**

```python
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
```

```python
>>> from sage.all import *

>>> values = lcalc.twist_values(RealNumber('0.5'), -Integer(10), Integer(10))
>>> values[Integer(0)][Integer(0)]
```

(continues on next page)
-8

<table>
<thead>
<tr>
<th>Values</th>
<th>1.10042141 + 0.0*I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>0</td>
</tr>
</tbody>
</table>

-7

<table>
<thead>
<tr>
<th>Values</th>
<th>1.1458567 + 0.0*I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>1</td>
</tr>
</tbody>
</table>

-4

<table>
<thead>
<tr>
<th>Values</th>
<th>0.667691457 + 0.0*I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>2</td>
</tr>
</tbody>
</table>

-3

<table>
<thead>
<tr>
<th>Values</th>
<th>0.480867558 + 0.0*I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>3</td>
</tr>
</tbody>
</table>

5

<table>
<thead>
<tr>
<th>Values</th>
<th>0.231750947 + 0.0*I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>4</td>
</tr>
</tbody>
</table>

8

<table>
<thead>
<tr>
<th>Values</th>
<th>0.373691713 + 0.0*I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>5</td>
</tr>
</tbody>
</table>

The `twist_zeros` function is defined as follows:

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

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

**INPUT:**

- \( n \) – integer
- \( d_{\text{min}} \) – integer
- \( d_{\text{max}} \) – integer
- \( L \) – defines \( L \)-function (default: Riemann zeta function)

**OUTPUT:**

- dict – keys are the discriminants \( d \), and values are list of corresponding zeros.

**EXAMPLES:**

```python
sage: lcalc.twist_zeros(3, -3, 6)
(-3: [8.03973716, 11.2492062, 15.7046192], 5: [6.64845335, 9.83144443, 11.9588456])
```

```python
from sage.all import *

>>> lcalc.twist_zeros(Integer(3), -Integer(3), Integer(6))
(-3: [8.03973716, 11.2492062, 15.7046192], 5: [6.64845335, 9.83144443, 11.9588456])
```

The `value` function is defined as follows:

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

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

**INPUT:**

- \( s \) – complex number
• \( L \) – defines \( L \)-function (default: Riemann zeta function)

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> I = CC.gen(0)
>>> lcalc.value(RealNumber(0.5) + Integer(100)*I)
2.69261989 - 0.0203860296*I
```

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

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

```python
>>> from sage.all import *
>>> (RealNumber(0.5) + Integer(100)*I).zeta()
2.69261988568132 - 0.0203860296025982*I
```

`values_along_line \((s_0, s_1, number\_samples, L='')\)`

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:**

• \( s_0, s_1 \) – complex numbers
• `number_samples` – integer
• \( L \) – defines \( L \)-function (default: Riemann zeta function)

**OUTPUT:**

• list – list of pairs \((s, \text{zeta}(s))\), where the \( s \) are equally spaced sampled points on the line from \( s_0 \) to \( s_1 \).

**EXAMPLES:**

```python
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
0.5 + 4.0*I
sage: values[1][1]  # abs tol 1e-8
0.606783764 + 0.091121400*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
```

(continues on next page)
Sometimes warnings are printed (by lcalc) when this command is run:

```python
\begin{verbatim}
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
\end{verbatim}
```
zeros \( (n, L=\text{''}) \)

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 \( \text{self.zeros_of_zeta_in_interval}(\ldots) \).

EXAMPLES:

```
sage: lcalc.zeros(4) \# long time
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 \( (x, y, \text{stepsize}, L=\text{''}) \)

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:
• \( x, y, \text{ stepsize} \) – 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 step size 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)]
```

```
>>> from sage.all import *

>>> lcalc.zeros_in_interval(Integer(10), Integer(30), RealNumber('0.1'))
[(14.1347251, 0.184672916), (21.0220396, -0.0677893290), (25.0108576, -0.0555872781)]
```
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).

```python
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 \( L(\text{Sym}^n(E, \text{edge})) \) to prec digits of precision, where edge is the right edge. Here \( n \) must be even.
    INPUT:
        • \( E \) – elliptic curve
        • \( n \) – even integer
        • \( \text{prec} \) – 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:

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

This command precomputes some data needed for the following examples.

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

```\( \text{Lderivs}(E, n, prec, d) \)

Return \( 0^{th} \) to \( d^{th} \) derivatives of \( L(\text{Sym}^{(n)}(E, s)) \) 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)
• \( \text{prec} \) – 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:

```plaintext
sage: print(sympow.Lderivs(EllipticCurve('11a'), 1, 16, 2))  # not tested
...
```
**analytic_rank** \((E)\)

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

**INPUT:**

- \(E\) – elliptic curve over \(\mathbb{Q}\)

**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:

```python
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, 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:
new_data \( (n) \)

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

```python
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/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.

```python
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
```
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.
˓→0.00002430047384*z^3 + 1.00006193307...*z^4 + O(z^5)

>>> from sage.all import *
>>> L = Dokchitser(conductor=Integer(1), gammaV=[Integer(0)], weight=Integer(1),
˓→eps=Integer(1), poles=[Integer(1)], residues=[-Integer(1)], init='1')
>>> L
Dokchitser L-series of conductor 1 and weight 1
>>> L(Integer(1))
Traceback (most recent call last):
...
ArithmeticError
>>> L(Integer(2))
1.64493406684823
>>> L(Integer(2), RealNumber('1.1'))
1.64493406684823
>>> L.derivative(Integer(2))
-0.937548254315844
>>> h = RR('0.0000000000001')
>>> (zeta(Integer(2)+h) - zeta(RealNumber('2.')))/h
-0.937028232783632
>>> L.taylor_series(Integer(2), k=Integer(5))
1.64493406684823 - 0.937548254315844*z + 0.994640117149451*z^2 - 1.
˓→0.0002430047384*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.
\[ \text{Continued from previous page} \]

\[
\begin{align*}
\rightarrow & 136791463097188*z^3 + O(z^4) \\
\text{sage: } & L.\text{check}\text{ functional equation()} \quad \# \text{ abs tol 1e-19} \\
& 6.04442711160669e-18
\end{align*}
\]

```python
>>> from sage.all import *
>>> E = EllipticCurve('37a')
>>> L = E.iseries().dokchitser(algorithm='gp'); L
Dokchitser L-function associated to Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
>>> L(Integer(1))
0.000000000000000
>>> L.derivative(Integer(1))
0.305999773834052
>>> L.derivative(Integer(1),Integer(2))
0.37309594536324
>>> L.num_coeffs()
48
>>> L.taylor_series(Integer(1),Integer(4))
0.000000000000000 + 0.305999773834052*z + 0.186547797268162*z^2 - 0.136791463097188*z^3 + O(z^4)
>>> 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.

```python
sage: E = EllipticCurve('389a')
sage: L = E.iseries().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 - 0.136791463097188*z^3 + O(z^4)
```

```python
>>> from sage.all import *
>>> E = EllipticCurve('389a')
>>> L = E.iseries().dokchitser(algorithm='gp')
>>> L.num_coeffs()
156
>>> L.derivative(Integer(1),E.rank())
1.51863300057685
>>> L.taylor_series(Integer(1),E.rank())
-1.27685190980159e-23 + (7.23588070754027e-24)*z + 0.759316500288427*z^2 - 0.136791463097188*z^3 + O(z^4)
```

**NUMBERFIELD:**
We compute with the Dedekind zeta function of a number field.
```python
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:

```python
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 coef-
grow(n)=(4n)^{11/2} (by a factor 1024), so re-defining coefgrow() improves efficiency (slightly faster).

```python
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.

```python
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)
```

```python
>>> from sage.all import *
>>> L(Integer(1))
0.0374412812685155
>>> L(Integer(1), RealNumber('1.1'))
0.0374412812685155
>>> L.taylor_series(Integer(1),Integer(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:**

```python
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
```

```python
>>> from sage.all import *
>>> L = Dokchitser(conductor=Integer(1), gammaV=[Integer(0)], weight=Integer(1), eps=Integer(1), poles=[Integer(1)], residues=[-Integer(1)], init='1')
>>> L.check_functional_equation()  # abs tol 1e-19
-2.71050543121376e-20
```

If we choose the sign in functional equation for the $\zeta$ function incorrectly, the functional equation doesn’t check out.

```python
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=-11, poles=[1], residues=[-1], init='1')
sage: L.check_functional_equation()
-9.73967861488124
```
```python
>>> from sage.all import *
>>> L = Dokchitser(conductor=Integer(1), gammaV=[Integer(0)],
    → weight=Integer(1), eps=-Integer(11), poles=[Integer(1)], residues=[-
    → Integer(1)], init='1')
>>> 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** ($v, cutoff=1, w=None, pari_precode='', max_imaginary_part=0, max_asympt_coeffs=40$)

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)
- Functions, Release 10.4

- `pari_precode` – some code to execute in pari before calling `initL.data`
- `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:**

```python
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):

```python
sage: L(14)
0.998583063162746
sage: a = delta_qexp(1000)
sage: sum(a[n]/float(n)^14 for n in reversed(range(1,1000)))
0.9985830631627461

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

```python
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:**
Verify that \texttt{num\_coeffs} works with non-real spectral parameters, e.g. for the L-function of the level 10 Maass form with eigenvalue 2.734105592527126:

\begin{verbatim}
sage: ev = 2.734105592527126
sage: L = Dokchitser(conductor=10, gammaV=[ev*i, -ev*i], weight=2, eps=1)
sage: L.num_coeffs()
26
\end{verbatim}

\texttt{set\_coeff\_growth} (\texttt{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:

\begin{verbatim}
coefgrow(n) = if(length(Lpoles),
    1.5*n^(vecmax(real(Lpoles))-1),
    sqrt(4*n)^(weight-1));
\end{verbatim}

\textbf{INPUT:}

- \texttt{coefgrow} – string that evaluates to a PARI function of \(n\) that defines a \texttt{coefgrow} function.

\textbf{EXAMPLES:}
\texttt{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 \texttt{var}, where you should view \texttt{var} as equal to $s - a$. Thus this function returns the formal power series whose coefficients are $L(a)/n!$.

\textbf{INPUT:}

- $a$ – complex number (default: 0); point about which to expand
- $k$ – integer (default: 6), series is $O(\text{\texttt{var}}^k)$
- $\texttt{var}$ – string (default: \texttt{\textquote{z}}), variable of power series

\textbf{EXAMPLES:}

\begin{verbatim}
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1], residues=[-1], init='1')
sage: L.taylor_series(Integer(2), Integer(3))
1.64493406684823 - 0.937548254315844*z + 0.994640117149451*z^2 + O(z^3)
sage: E = EllipticCurve('37a')
sage: L = E.1series().dokchitser(algorithm='gp')
sage: L.taylor_series(Integer(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)
\end{verbatim}

We compute a Taylor series where each coefficient is to high precision.
sage: E = EllipticCurve('389a')
sage: L = E.heart().dokchitser(200, algorithm='gp')
sage: L.taylor_series(1,3)
\[e^{-82} + (...)z + 0.\]
\[=75931650028842677023019260789472201907809751649492435158581z^2 + O(z^3)\]

>>> from sage.all import *
>>> E = EllipticCurve('389a')
>>> L = E.heart().dokchitser(Integer(200), algorithm='gp')
>>> L.taylor_series(Integer(1),Integer(3))
\[e^{-82} + (...)z + 0.\]
\[=75931650028842677023019260789472201907809751649492435158581z^2 + O(z^3)\]

Check that 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)

>>> from sage.all import *
>>> L = EllipticCurve("24a1").modular_form().lseries()
>>> L.taylor_series(-Integer(1), Integer(3))
0.000000000000000 - 0.702565506265199*z + 0.638929001045535*z^2 + O(z^3)

Check that 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

>>> from sage.all import *
>>> 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)
>>> L2.taylor_series(Integer(0),Integer(4))
0.000000000000000 - 0.357620466127498*z + 0.273373112603865*z^2 + 0.
\[=303362857047671*z^3 + O(z^4)\]
>>> L2.taylor_series(Integer(0),Integer(1))
O(z^1)
>>> L2(Integer(0))
0.000000000000000

sage.lfunctions.dokchitser.reduce_load_dokchitser(D)
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:

```python
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
```

```python
>>> from sage.all import *
>>> E = EllipticCurve("389a")
>>> 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
```

```python
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 $\gamma$ 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 $\Delta$ 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 \( \eta \) is the Euler-Mascheroni constant = 0.5772...; the crossover point is at conductor ~8.3\times10^8. 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 \( \Delta \), 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:

\texttt{LFunctionZeroSum()} \texttt{EllipticCurve.root_number()} \texttt{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 \( \Delta \) will yield tight rank estimates.

\begin{verbatim}
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
\end{verbatim}

This is especially true for elliptic curves with large rank.

\begin{verbatim}
>>> from sage.all import *
>>> E = EllipticCurve("11a")
>>> E.rank()
0
>>> Z = LFunctionZeroSum(E)
>>> Z.analytic_rank_upper_bound(max_Delta=Integer(1),ncpus=Integer(1))
0

>>> E = EllipticCurve([-Integer(39),Integer(123)])
>>> E.rank()
1
>>> Z = LFunctionZeroSum(E)
>>> Z.analytic_rank_upper_bound(max_Delta=Integer(1))
1
\end{verbatim}

\begin{verbatim}
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)
\end{verbatim}
However, some curves have \( L \)-functions with low-lying zeroes, and for these larger values of \( \Delta \) must be used to get tight estimates.

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

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 \( \approx 2.815 \).
This method is can be called on curves with large conductor.

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 \( \epsilon^{2\pi \Delta} \). The example below is of the rank 28 curve discovered by Elkies that is the elliptic curve of (currently) largest known rank.

(continues on next page)
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 \(-a_p \log(p)/p^e\), 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 \(-\left(\alpha_p^e + \beta_p^e\right) \log(p)/p^e\), where \(\alpha_p\) and \(\beta_p\) 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 \(2 \log(n)/\sqrt{n}\) in magnitude.

EXAMPLES:

```python
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)
(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)
```

```python
from sage.all import *
```

```python
>>> E = EllipticCurve("11a")
```

```python
>>> Z = LFunctionZeroSum(E)
```

```python
>>> for n in range(Integer(12)): print((n, Z.cn(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)
```
elliptic_curve()

Return the elliptic curve associated with self.

EXAMPLES:

```python
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:

```python
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: sage.all.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 $\eta$ is the Euler-Mascheroni constant $\approx 0.5772\ldots$ 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:

```python
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
```
>>> from sage.all import *
>>> E = EllipticCurve("389a")
>>> Z = LFunctionZeroSum(E)
>>> Z.C0() # tol 1.0e-13
0.5666969404983447
>>> Z.C0(include_euler_gamma=False) # tol 1.0e-13
1.1439126053998776

\texttt{cnlist}\ (n, python\_floats=False)

Return a list of Dirichlet coefficient of the logarithmic derivative of the \(L\)-function attached to \texttt{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: \texttt{False}); if \texttt{True} return a list of Python floats instead of Sage Real Double Field elements.

OUTPUT:

A list of real numbers

See also:

\texttt{cn()}

\textbf{Todo:} Speed this up; make more efficient

\textbf{EXAMPLES:}

```python
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)
```

```python
>>> from sage.all import *
>>> E = EllipticCurve("11a")
>>> Z = LFunctionZeroSum(E)
>>> cnlist = Z.cnlist(Integer(11))
>>> for n in range(Integer(12)): print((n, cnlist[n])) # tol 1.0e-13
(0, 0.0)
(1, 0.0)
(2, 0.6931471805599453)
```
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 $0.5 \leq \Re(s) \leq 1.5$. 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:**

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

```python
>>> from sage.all import *
>>> E = EllipticCurve([Integer(23),Integer(100)])
>>> Z = LFunctionZeroSum(E)
>>> Z.completed_logarithmic_derivative(Integer(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 $s$. 
digamma (s, include_constant_term=True)

Return the digamma function \( \psi(s) \) on the complex input \( s \).

This is given by

\[ \psi(s) = -\gamma + \sum_{k=1}^{\infty} \frac{s-1}{k(k+s-1)} \]

where \( \gamma \) is the Euler-Mascheroni constant = 0.5772156649... .

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-Mascheron 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:

```python
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
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

>>> from sage.all import *
>>> Z.digamma(Integer(3), include_constant_term=False)
1.5
>>> Z.digamma(Integer(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

>>> from sage.all import *
>>> E = EllipticCurve("389a")
>>> Z = LFunctionZeroSum(E)
>>> Z.level()
389

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

Compute the value of the logarithmic derivative $L'/L$ 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 $L'/L$; convergence is controlled by the distance of s from the critical strip $0.5 \leq \Re(s) \leq 1.5$. 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:
Increasing the number of terms should see the truncation error decrease.

```
sage: Z.logarithmic_derivative(2.2, num_terms=Integer(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)
```

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:

```python
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.

```python
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:

```python
sage: E = EllipticCurve("389a")
sage: Z = LFunctionZeroSum(E)
sage: Z.weight()
2
```
```python
>>> from sage.all import *
>>> E = EllipticCurve("389a")
>>> Z = LFunctionZeroSum(E)
>>> 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 \( \gamma \) 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 \( \Delta \) 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 \( \Delta \) is increased. If \( \tau \) 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 \( \tau = 0 \). `self` must be attached to an elliptic curve over \( \mathbb{Q} \) 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) \( \Delta \) values. `self` must be attached to an elliptic curve over \( \mathbb{Q} \) 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 \( \Delta \), 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 \( \tau \). 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
def E = EllipticCurve("389a"); E.rank()
def Z = LFunctionZeroSum(E)
def E.lseries().zeros(3)
[0.000000000, 0.000000000, 2.87609907]
def Z.zerosum(Delta=1, function="sincsquared_fast")
2.037500084595065
def Z.zerosum(Delta=1, function="sincsquared_parallel")
2.037500084595065
def Z.zerosum(Delta=1, function="sincsquared")
2.0375000845950644
def Z.zerosum(Delta=1, tau=2.876, function="sincsquared")
1.075551295651154
def Z.zerosum(Delta=1, tau=1.2, function="sincsquared")
0.1083155377490683
def Z.zerosum(Delta=1, function="gaussian")
2.056890425029435
```

```sage
def Z.zerosum(Delta=1, tau=2.876, function="sincsquared")
1.075551295651154
def Z.zerosum(Delta=1, tau=1.2, function="sincsquared")
0.1083155377490683
def Z.zerosum(Delta=1, function="gaussian")
2.056890425029435
```
CHAPTER FIVE

L-FUNCTIONS FROM PARI

This is a wrapper around the general PARI L-functions functionality.

AUTHORS:
• Frédéric Chapoton (2018) interface

```python
class sage.lfunctions.pari.LFunction (lfun, prec=None)
    Bases: SageObject

Build the L-function from a PARI L-function.
```

**Rank 1 elliptic curve**

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

```python
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser(algorithm="pari"); L
PARI 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()
50
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 4e-19
1.08420217248550e-19
```

```python
>>> from sage.all import *
>>> E = EllipticCurve('37a')
>>> L = E.lseries().dokchitser(algorithm="pari"); L
PARI L-function associated to Elliptic Curve defined by y^2 + y = x^3 - x over...
→Rational Field
>>> L(Integer(1))
0.000000000000000
>>> L.derivative(Integer(1))
0.305999773834052
>>> L.derivative(Integer(1), Integer(2))
```

(continues on next page)
0.373095594536324
>>> L.num_coeffs()
50
>>> L.taylor_series(Integer(1), Integer(4))
0.000000000000000 + 0.305999773834052*z + 0.186547797268162*z^2 - 0.
˓→136791463097188*z^3 + O(z^4)
>>> L.check_functional_equation()  # abs tol 4e-19
1.08420217248550e-19

Rank 2 elliptic curve

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

```sage
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser(algorithm="pari")
sage: L.num_coeffs()
163
sage: L.derivative(1, E.rank())
1.51863300057685
sage: L.taylor_series(1, 4)
...
<...>
```

Number field

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

```sage
sage: x = var('x')
sage: K = NumberField(x**4 - x**2 - 1, 'a')
sage: L = K.zeta_function(algorithm="pari")
sage: L.conductor
400
sage: L.num_coeffs()
348
sage: L(2)
1.10398438736918
sage: L.taylor_series(2, 3)
1.10398438736918 - 0.215822638498759*z + 0.279836437522536*z^2 + O(z^3)
```

```sage
>>> from sage.all import *
>>> E = EllipticCurve('389a')
>>> L = E.lseries().dokchitser(algorithm="pari")
>>> L.num_coeffs()
163
>>> L.derivative(Integer(1), E.rank())
1.51863300057685
>>> L.taylor_series(Integer(1), Integer(4))
...
<...>
```
Ramanujan $\Delta$ L-function

The coefficients are given by Ramanujan’s $\tau$ function:

```
sage: from sage.lfunctions.pari import lfun_generic, LFunction
sage: lf = lfun_generic(conductor=1, gammaV=[0, 1], weight=12, eps=1)
sage: tau = pari(k->vector(k,n,(5*sigma(n,3)+7*sigma(n,5))*n/12 - 35*sum(k=1,n-1,
    \rightarrow (6*k-4*(n-k))*sigma(k,3)*sigma(n-k,5))))
sage: lf.init_coeffs(tau)
sage: L = LFunction(lf)
```

Now we are ready to evaluate, etc.

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

\textbf{\texttt{Lambda}(s)}

Evaluate the completed L-function at $s$.

\textbf{EXAMPLES:}

```
sage: from sage.lfunctions.pari import lfun_number_field, LFunction
sage: L = LFunction(lfun_number_field(QQ))
sage: L.Lambda(2) 0.523598775598299
sage: L.Lambda(1 - 2) 0.523598775598299
```
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_number_field, LFunction
>>> L = LFunction(lfun_number_field(QQ))
>>> L.Lambda(Integer(2))
0.523598775598299
>>> L.Lambda(Integer(1) - Integer(2))
0.523598775598299

\texttt{check_functional_equation()}

Verify how well numerically the functional equation is satisfied.

If what this function returns does not look like 0 at all, probably the functional equation is wrong, i.e. some of the parameters \( \gamma_V, \text{conductor}, \text{etc.}, \) or the coefficients are wrong.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.lfunctions.pari import lfun_generic, LFunction
sage: lf = lf
fun_generic(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1],...
->residues=[1], v=pari('k->vector(k,n,1)'))
sage: L = LFunction(lf)
sage: L.check_functional_equation()
4.33680868994202e-19
\end{verbatim}

If we choose the sign in functional equation for the \( \zeta \) function incorrectly, the functional equation does not check out:

\begin{verbatim}
sage: lf = lf
fun_generic(conductor=1, gammaV=[0], weight=1, eps=-1, poles=[1],...
->residues=[1])
sage: lf.init_coeffs([1]*2000)
sage: L = LFunction(lf)
sage: L.check_functional_equation()
16.0000000000000
\end{verbatim}

\textbf{property conductor}

Return the conductor.

\textbf{EXAMPLES:}
derivative \((s, D=1)\)

Return the derivative of the L-function at point \(s\) and order \(D\).

INPUT:

- \(s\) – complex number
- \(D\) – optional integer (default 1)

EXAMPLES:

```python
sage: from sage.lfunctions.pari import lfun_number_field, LFunction
sage: L = LFunction(lfun_number_field(QQ))
sage: L.derivative(2)
-0.937548254315844
```

```python
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_number_field, LFunction

>>> L = LFunction(lfun_number_field(QQ))

>>> L.derivative(Integer(2))
-0.937548254315844
```

hardy \((t)\)

Evaluate the associated Hardy function at \(t\).

This only works for real \(t\).

EXAMPLES:

```python
sage: from sage.lfunctions.pari import lfun_number_field, LFunction
sage: L = LFunction(lfun_number_field(QQ))
```

```python
sage: L.hardy(2)
-0.962008487244041
```

```python
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_number_field, LFunction

>>> L = LFunction(lfun_number_field(QQ))

>>> L.hardy(Integer(2))
-0.962008487244041
```

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:

```python
sage: E = EllipticCurve('11a')
sage: L = E.iloop().dokchitser(algorithm="pari")
sage: L.num_coeffs()
27
sage: E = EllipticCurve('5077a')
sage: L = E.iloop().dokchitser(algorithm="pari")
sage: L.num_coeffs()
591
sage: from sage.lfunctions.pari import lfun_generic, LFunction
sage: lf = lfun_generic(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1],
→residues=[-1], v=pari('k->vector(k,n,1)'))
sage: L = LFunction(lf)
sage: L.num_coeffs()
4
```

```python
>>> from sage.all import *
>>> E = EllipticCurve('11a')
>>> L = E.iloop().dokchitser(algorithm="pari")
>>> L.num_coeffs()
27
>>> E = EllipticCurve('5077a')
>>> L = E.iloop().dokchitser(algorithm="pari")
>>> L.num_coeffs()
591
>>> from sage.lfunctions.pari import lfun_generic, LFunction
>>> lf = lfun_generic(conductor=Integer(1), gammaV=[Integer(0)],
→weight=Integer(1), eps=Integer(1), poles=[Integer(1)], residues=[-Integer(1)], v=pari('k->vector(k,n,1)'))
>>> L = LFunction(lf)
>>> L.num_coeffs()
4
```

`taylor_series(s, k=6, var='z')`

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

This is returned as a formal power series in $\var$.

INPUT:

- $s$ – complex number; point about which to expand
- $k$ – optional integer (default: 6), series is $O(\var^k)$
- $\var$ – optional string (default: ‘$z$’), variable of power series

EXAMPLES:

```python
sage: from sage.lfunctions.pari import lfun_number_field, LFunction
sage: lf = lfun_number_field(QQ)
sage: L = LFunction(lf)
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.iloop().dokchitser(algorithm="pari")
sage: L.taylor_series(1)
```

(continues on next page)
0.000000000000000 + 0.305999773834052*z + 0.186547797268162*z^2 - 0.
\[ \rightarrow 136791463097188*z^3 + 0.161066468496401*z^4 + 0.0185955175398802*z^5 + O(z^6) \]

```python
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_number_field, LFunction
>>> lf = lfun_number_field(QQ)
>>> L = LFunction(lf)
>>> L.taylor_series(Integer(2), Integer(3))
1.64493406684823 - 0.937548254315844*z + 0.994640117149451*z^2 + O(z^3)
```

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

```python
sage: E = EllipticCurve('389a')
sage: L = E.ihseries().dokchitser(200,algorithm="pari")
sage: L.taylor_series(Integer(1), Integer(3))
2.6e-63 + (...e-63)*z + 0.75931650028842677023019260789472201907809751649492435158581*z^2 + O(z^3)
```

Check that Issue #25402 is fixed:

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

```python
>>> from sage.all import *
>>> E = EllipticCurve('389a')
>>> L = E.ihseries().dokchitser(Integer(200),algorithm="pari")
>>> L.taylor_series(Integer(1), Integer(3))
2.6e-63 + (...e-63)*z + 0.75931650028842677023019260789472201907809751649492435158581*z^2 + O(z^3)
```

```python
>>> from sage.all import *
>>> L = EllipticCurve("24a1").ihseries().lseries()
>>> L.taylor_series(-1, 3)
0.000000000000000 - 0.702565506265199*z + 0.638929001045535*z^2 + O(z^3)
```

```
 zeros(maxi)  
 Return the zeros with imaginary part bounded by maxi.

 EXAMPLES:
```
```python
sage: from sage.lfunctions.pari import lfun_number_field, LFunction
sage: lf = lfun_number_field(QQ)
sage: L = LFunction(lf)
sage: L.zeros(20)
[14.1347251417347]  
```
sage.lfunctions.pari.lfun_character(chi)
Create the L-function of a primitive Dirichlet character.
If the given character is not primitive, it is replaced by its associated primitive character.

OUTPUT:

one pari:lfun object

EXAMPLES:

```python
sage: from sage.lfunctions.pari import lfun_character, LFunction
sage: chi = DirichletGroup(6).gen().primitive_character()
sage: L = LFunction(lfun_character(chi))
sage: L(3)
0.884023811750080
```

sage.lfunctions.pari.lfun_delta()
Return the L-function of Ramanujan’s Delta modular form.

EXAMPLES:

```python
sage: from sage.lfunctions.pari import lfun_delta, LFunction
sage: L = LFunction(lfun_delta())
sage: L(1)
0.0374412812685155
```

sage.lfunctions.pari.lfun_elliptic_curve(E)
Create the L-function of an elliptic curve.

OUTPUT:

one pari:lfun object

EXAMPLES:

```python
sage: from sage.lfunctions.pari import lfun_elliptic_curve, LFunction
sage: E = EllipticCurve('11a1')
```
sage: L = LFunction(lfun_elliptic_curve(E))
sage: L(3)
0.752723147389051
sage: L(1)
0.253841860855911

>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_elliptic_curve, LFunction
>>> E = EllipticCurve('11a1')
>>> L = LFunction(lfun_elliptic_curve(E))
>>> L(Integer(3))
0.752723147389051
>>> L(Integer(1))
0.253841860855911

Over number fields:

sage: K.<a> = QuadraticField(2)
sage: E = EllipticCurve([1, a])
sage: L = LFunction(lfun_elliptic_curve(E))
sage: L(3)
1.00412346717019

```python
>>> from sage.all import *
>>> K = QuadraticField(Integer(2), names=('a',)); (a,) = K._first_ngens(1)
>>> E = EllipticCurve([Integer(1), a])
>>> L = LFunction(lfun_elliptic_curve(E))
>>> L(Integer(3))
1.00412346717019
```

```
sage.lfunctions.pari.lfun_eta_quotient (scalings, exponents)
Return the L-function of an eta-quotient.
This uses pari:lfunetaquo.

INPUT:
• scalings – a list of integers, the scaling factors
• exponents – a list of integers, the exponents

EXAMPLES:
sage: from sage.lfunctions.pari import lfun_eta_quotient, LFunction
sage: L = LFunction(lfun_eta_quotient([1], [24]))
sage: L(1)
0.0374412812685155

sage: lfun_eta_quotient([6], [4])
[[Vecsmall([7]), [Vecsmall([6]), Vecsmall([4]), 0]], 0, [0, 1], 2, 36, 1]
sage: lfun_eta_quotient([2, 1, 4], [5, -2, -2])
Traceback (most recent call last):
  ...
PariError: sorry, noncuspidal eta quotient is not yet implemented
```
```python
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_eta_quotient, LFunction
>>> L = LFunction(lfun_eta_quotient([Integer(1)], [Integer(24)]))
>>> L(Integer(1))
0.0374412812685155

>>> lfun_eta_quotient([Integer(6)], [Integer(4)])
[[Vecsmall([7]), [Vecsmall([6]), Vecsmall([4]), 0]], 0, [0, 1], 2, 36, 1]

>>> lfun_eta_quotient([Integer(2), Integer(1), Integer(4)], [Integer(5), -Integer(2), -Integer(2)])
Traceback (most recent call last):
  ...
PariError: sorry, noncuspidal eta quotient is not yet implemented
```

class sage.lfunctions.pari.lfun_generic(conductor, gammaV, weight, eps, poles=[], residues='automatic', prec=None, *args, **kwds)

Bases: object

Create a PARI $L$-function (pari:lfun instance).

The arguments are:

```
lfun_generic(conductor, gammaV, weight, eps, poles, residues, init)
```

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/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)
- `init` – list of coefficients

RIEMANN ZETA FUNCTION:

We compute with the Riemann Zeta function:

```
sage: from sage.lfunctions.pari import lfun_generic, LFunction
sage: lf = lfun_generic(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1], residues=[1])
sage: lf.init_coeffs([1]*2000)
```

```
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_generic, LFunction
>>> lf = lfun_generic(conductor=Integer(1), gammaV=[Integer(0)], weight=Integer(1), eps=Integer(1), poles=[Integer(1)], residues=[Integer(1)])
>>> lf.init_coeffs([Integer(1)]*Integer(2000))
```

Now we can wrap this PARI $L$-function into one Sage $L$-function:
init_coeffs (v, cutoff=None, w=None)

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 unary function
- cutoff – unused
- w – list of complex numbers or unary function

EXAMPLES:

```python
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_generic, LFunction
sage: lf = lfun_generic(conductor=1, gammaV=[0, 1], weight=12, eps=1)
```

```python
sage: pari_coeffs = pari(k->vector(k,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))))
```

```python
sage: lf.init_coeffs(pari_coeffs)
```
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 = LFunction(lf)
sage: L(14)
0.998583063162746
sage: a = delta_qexp(1000)
sage: sum(a[n]/float(n)**14 for n in reversed(range(1,1000)))
0.9985830631627461
```

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

```
sage: l2 = lfun_generic(conductor=1, gammaV=[0, 1], weight=12, eps=1)
sage: L2 = LFunction(l2)
sage: L2(14)
0.998583063162746
```

Return the L-function of a curve of genus 2.

INPUT:

- \( C \) – hyperelliptic curve of genus 2

Currently, the model needs to be minimal at 2.

This uses pari:lfungenus2.

EXAMPLES:
sage: L = LFunction(lfun_genus2(C))
...  
sage: L(3)
0.965946926261520

sage: C = HyperellipticCurve(x^2 + x, x^3 + x^2 + 1)
sage: L = LFunction(lfun_genus2(C))
sage: L(2)
0.364286342944359

```python
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_genus2, LFunction
>>> x = polygen(QQ, 'x')
>>> C = HyperellipticCurve(x**Integer(5) + x + Integer(1))
>>> L = LFunction(lfun_genus2(C))
...  
>>> L(Integer(3))
0.965946926261520

>>> C = HyperellipticCurve(x**Integer(2) + x, x**Integer(3) + x**Integer(2) + Integer(1))
>>> L = LFunction(lfun_genus2(C))
>>> L(Integer(2))
0.364286342944359
```

`sage.lfunctions.pari.lfun_hgm(motif, t)`

Create the L-function of an hypergeometric motive.

OUTPUT:

one pari:lfun object

EXAMPLES:

```python
sage: from sage.lfunctions.pari import lfun_hgm, LFunction
sage: from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
sage: H = Hyp(gamma_list=(3,-1,-1,-1))
(...) 
```

sage.lfunctions.pari.lfun_number_field(K)

Create the Dedekind zeta function of a number field.

OUTPUT:

one pari:lfun object

EXAMPLES:

```python
>>> from sage.all import *
>>> from sage.lfunctions.pari import lfun_genus2, LFunction
>>> from sage.modular.hypergeometric_motive import HypergeometricData as Hyp
... 
>>> H = Hyp(gamma_list=(Integer(3),-Integer(1),-Integer(1),-Integer(1)))
>>> L = LFunction(lfun_hgm(H, Integer(1)/Integer(5)))
>>> L(Integer(3))
0.901925346034773
```
sage: from sage.lfunctions.pari import lfun_number_field, LFunction

sage: L = LFunction(lfun_number_field(QQ))

sage: L(3)
1.20205690315959

sage: K = NumberField(x**2 - 2, 'a')

sage: L = LFunction(lfun_number_field(K))

sage: L(3)
1.15202784126080

sage: L(0)
...
0.000000000000000

sage.lfunctions.pari.lfun_quadratic_form(qf)

Return the \( L \)-function of a positive definite quadratic form.

This uses pari:lfunqf.

EXAMPLES:

sage: from sage.lfunctions.pari import lfun_quadratic_form, LFunction

sage: Q = QuadraticForm(ZZ, 2, [2, 3, 4])

sage: L = LFunction(lfun_quadratic_form(Q))

sage: L(3)
0.377597233183583

>>> from sage.all import *

>>> from sage.lfunctions.pari import lfun_quadratic_form, LFunction

>>> Q = QuadraticForm(ZZ, Integer(2), [Integer(2), Integer(3), Integer(4)])

>>> L = LFunction(lfun_quadratic_form(Q))

>>> L(Integer(3))
0.377597233183583
CHAPTER
SIX

INDICES AND TABLES

• Index
• Module Index
• Search Page
sage.lfunctions.dokchitser, 17
sage.lfunctions.lcalc, 3
sage.lfunctions.pari, 43
sage.lfunctions.sympow, 11
sage.lfunctions.zero_sums, 27
INDEX

A
analytic_rank() (sage.lfunctions.lcalc.LCalc method), 3
analytic_rank() (sage.lfunctions.sympow.Sympow method), 13
analytic_rank_upper_bound() (sage.lfunctions.zero_sums.LFunctionZeroSum_EllipticCurve method), 27

C
c0() (sage.lfunctions.zero_sums.LFunctionZeroSum_abstract method), 33
check_functional_equation() (sage.lfunctions.dokchitser.Dokchitser method), 21
check_functional_equation() (sage.lfunctions.pari.LFunction method), 46
cn() (sage.lfunctions.zero_sums.LFunctionZeroSum_EllipticCurve method), 32
cnlist() (sage.lfunctions.zero_sums.LFunctionZeroSum_abstract method), 34
completed_logarithmic_derivative() (sage.lfunctions.zero_sums.LFunctionZeroSum_abstract method), 35
conductor (sage.lfunctions.pari.LFunction property), 46

D
derivative() (sage.lfunctions.dokchitser.Dokchitser method), 22
derivative() (sage.lfunctions.pari.LFunction method), 47
digamma() (sage.lfunctions.zero_sums.LFunctionZeroSum_abstract method), 36
Dokchitser (class in sage.lfunctions.dokchitser), 17

E
elliptic_curve() (sage.lfunctions.zero_sums.LFunctionZeroSum_EllipticCurve method), 32

G
gp() (sage.lfunctions.dokchitser.Dokchitser method), 22

H
hardy() (sage.lfunctions.pari.LFunction method), 47
help() (sage.lfunctions.lcalc.LCalc method), 4
help() (sage.lfunctions.sympow.Sympow method), 14
init_coeffs() (sage.lfunctions.dokchitser.Dokchitser method), 22
init_coeffs() (sage.lfunctions.pari.lfun_generic method), 53

L
L() (sage.lfunctions.sympow.Sympow method), 11
Lambda() (sage.lfunctions.pari.LFunction method), 45
LCalc (class in sage.lfunctions.lcalc), 3
Lderivs() (sage.lfunctions.sympow.Sympow method), 12
level() (sage.lfunctions.zero_sums.LFunctionZeroSum_abstract method), 37
lfun_character() (in module sage.lfunctions.pari), 50
lfun_delta() (in module sage.lfunctions.pari), 50
lfun_elliptic_curve() (in module sage.lfunctions.pari), 50
lfun_eta_quotient() (in module sage.lfunctions.pari), 51
lfun_generic (class in sage.lfunctions.pari), 52
lfun_genus2() (in module sage.lfunctions.pari), 54
lfun_hgm() (in module sage.lfunctions.pari), 55
lfun_number_field() (in module sage.lfunctions.pari), 55
lfun_quadratic_form() (in module sage.lfunctions.pari), 56
LFunction (class in sage.lfunctions.pari), 43
LFunctionZeroSum() (in module sage.lfunctions.zero_sums), 27
LFunctionZeroSum_abstract (class in sage.lfunctions.zero_sums), 33
LFunctionZeroSum_EllipticCurve (class in sage.lfunctions.zero_sums), 27
logarithmic_derivative() (sage.lfunctions.zero_sums.LFunctionZeroSum_abstract method), 61
L-Functions, Release 10.4

\[ \text{lseries()} \quad (\text{sage.lfunctions.zero_sums.LFunctionZeroSum.EllipticCurve method}, 33) \]

\[ \text{module} \quad \text{sage.lfunctions.dokchitser, 17} \]
\[ \text{sage.lfunctions.lcalc, 3} \]
\[ \text{sage.lfunctions.pari, 43} \]
\[ \text{sage.lfunctions.sympow, 11} \]
\[ \text{sage.lfunctions.zero_sums, 27} \]

\[ \text{method, 37} \]

\[ \text{method, 39} \]

\[ \text{method, 6} \]

\[ \text{method, 39} \]

\[ \text{module} \quad \text{sage.lfunctions.dokchitser, 17} \]
\[ \text{sage.lfunctions.lcalc, 3} \]
\[ \text{sage.lfunctions.pari, 43} \]
\[ \text{sage.lfunctions.sympow, 11} \]
\[ \text{sage.lfunctions.zero_sums, 27} \]

\[ \text{set_coeff_growth()} \quad (\text{sage.lfunctions.dokchitser.Dokchitser method}, 24) \]
\[ \text{Sympow} \quad (\text{class in sage.lfunctions.sympow}, 11) \]

\[ \text{method, 14} \]

\[ \text{new_data()} \quad (\text{sage.lfunctions.sympow.Sympow method}, 15) \]
\[ \text{num_coeffs()} \quad (\text{sage.lfunctions.dokchitser.Dokchitser method}, 23) \]
\[ \text{num_coeffs()} \quad (\text{sage.lfunctions.pari.LFunction method}, 47) \]

\[ \text{reduce_load_dokchitser()} \quad \text{(in module sage.lfunctions.dokchitser), 26} \]

\[ \text{module, 17} \]
\[ \text{module, 3} \]
\[ \text{module, 43} \]
\[ \text{module, 11} \]
\[ \text{module, 27} \]

\[ \text{method, 8} \]

\[ \text{method, 49} \]
\[ \text{method, 8} \]
\[ \text{method, 40} \]

\[ \text{method, 6} \]

\[ \text{method, 5} \]

\[ \text{method, 5} \]