Sage Reference Manual: L-Functions

Release 6.3

The Sage Development Team

August 11, 2014

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: `sage.structure.sage_object.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 $\mathbb{Q}$; 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()
x.__init__(...) initializes x; see help(type(x)) for signature

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: lcalc.twist_values(0.5, -10, 10)
[(-8, 1.10042141), (-7, 1.14658567), (-4, 0.667691457), (-3, 0.480867558), (5, 0.231750947),
```

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: lcalc.values_along_line(0.5, 0.5+20*I, 5)
[(0.500000000, -1.46035451), (0.500000000 + 4.00000000*I, 0.606783764 + 0.0911121400*I), (0.
```

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

```
sage: E = EllipticCurve('389a')
sage: E.lseries().values_along_line(0.5, 3, 5)
[(0.000000000, 0.209951303),
 (0.500000000, -...e-16),
 (1.00000000, 0.133768433),
 (1.50000000, 0.360092864),
 (2.00000000, 0.552975867)]
```

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

INPUT:

•`E` - elliptic curve

- n - even integer
- 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()

x.__init__(...) initializes x; see help(type(x)) for signature

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
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

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

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).

class `sage.lfunctions.dokchitser.`**`Dokchitser`**(*conductor*, *gammaV*, *weight*, *eps*, *poles*=[], *residues='automatic'*, *prec=53*, *init=None*)

Bases: `sage.structure.sage_object.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.

```
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 - 1.00002430047384*z^3 + 1.000061
```

RANK 1 ELLIPTIC CURVE:

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

```
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser(); L
Dokchitser L-function associated to Elliptic Curve defined by y^2 + y = x^3 - x over Rational Fi
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()
6.11218974800000e-18                              # 32-bit
6.04442711160669e-18                              # 64-bit
```

RANK 2 ELLIPTIC CURVE:

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

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser()
sage: L.num_coeffs ()
156
sage: L.derivative(1,E.rank())
1.51863300057685
sage: L.taylor_series(1,4)
2.90759778535572e-20 + (-1.64772676916085e-20)*z + 0.759316500288427*z^2 - 0.430302337583362*z^3
-3.11623283109075e-21 + (1.76595961125962e-21)*z + 0.759316500288427*z^2 - 0.430302337583362*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))*sig
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).

```
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], ini
sage: L.check_functional_equation()
-1.35525271600000e-20                          # 32-bit
-2.71050543121376e-20                          # 64-bit
```

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], i
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()
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()
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))
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 trac 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
```

TESTS:

Verify that setting the w parameter does not raise an error (see trac 10937). Note that the meaning of w does not seem to be documented anywhere in Dokchitser's package yet, so there is no claim that the example below is meaningful!

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

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

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.

EXAMPLE:

```
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))
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 $O(\text{“}var\text{“}^k)$
- `var` - string (default: 'z'), variable of power series

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1], residues=[-1], ini
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()
sage: L.taylor_series(1)
0.000000000000000 + 0.305999773834052*z + 0.186547797268162*z^2 - 0.136791463097188*z^3 + 0.
```

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

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser(200)
sage: L.taylor_series(1,3)
6.22401886341037743482734469656208012888363286519732345731 33e-73 + (-3.527132447498646306292
```

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

x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER

FOUR

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

l

INDEX

S

T

V

Z