

## Contents

1 Installation 3  
   1.1 Prerequisites ................................................................. 3  
   1.2 Installation Procedure .................................................. 3  

2 Getting Started 5  

3 Tutorials 9  
   3.1 Riemann Theta Functions .................................................. 9  

4 Applications 13  
   4.1 Computing Bitangents of Plane Quartics ............................... 13  
   4.2 Genus Two and Three Solutions to the Kdovitsev-Petviashvili Equation .................................................. 13  

5 Reference 15  
   5.1 Primary Functionality ....................................................... 15  
   5.2 Module Reference .............................................................. 15  

6 Developer’s Guide 75  
   6.1 Development Prerequisites .................................................. 75  
   6.2 Development Topics ........................................................... 75  

7 Changelog 79  

8 Authors 81  

Bibliography 83  

Python Module Index 85
Abelfunctions is a library for computing with Abelian functions, Riemann surfaces, and algebraic curves. If you are new to Abelfunctions, begin with the Getting Started guide. More tutorials, application areas, and complete software reference is presented in the documentation below.
Installation

1.1 Prerequisites

Abelfunctions runs well with either Anaconda, the Enthought Python Distribution, or Sage. Specifically, Abelfunctions requires the following Python packages which are included in the aforementioned software:

- numpy
- scipy
- sympy
- networkx
- matplotlib
- cython

Additionally, the following packages are required to build the documentation.

- sphinx
- numpydoc
- releases
- sphinx_rtd_theme

1.2 Installation Procedure

1. **Download the Code**: There are two ways to do this:

   1. Download Abelfunctions.
   2. If you have git installed, run

      ```bash
      $ git clone https://github.com/cswiercz/abelfunctions.git
      ```

2. **Install**: Enter the top-level directory, `abelfunctions`, and run

   ```bash
   $ python setup.py install --user
   ```

   for a local installation. For a system-wide install, (or if you’re installing the package into Sage) run

   ```bash
   $ python setup.py install
   ```
CHAPTER 2

Getting Started

This document presents a brief overview of the capabilities of Abelfunctions by working through a simple example. For extended examples and particular applications, please see the Tutorials page. See the Installation guide if you need to obtain and install Abelfunctions.

Riemann Surfaces

One of the primary objects we can construct with abelfunctions is a Riemann surface. In particular, a Riemann surface $X$ obtained by desingularizing and compactifying a complex plane algebraic curve $f(x, y) = 0$. To create a Riemann surface, provide a SymPy (http://sympy.org) expression.

```python
>>> import sympy
>>> from sympy.abc import x, y
>>> from abelfunctions import *

>>> f = y**3 + 2*x**3*y - x**7
>>> X = RiemannSurface(f, x, y)
>>> X
Riemann surface defined by the algebraic curve -x**7 + 2*x**3*y + y**3
```

Abelfunctions can compute the singularities of the curve and uses the corresponding information to determine the genus.

```python
>>> print X.genus()
2
```

Holomorphic Differentials

We can compute a basis for the space of holomorphic differentials $\{\tilde{\omega}_1, \ldots, \tilde{\omega}_g\}$ defined on $X$. The affine part of the differentials are displayed when computed.

```python
>>> differentials = X.holomorphic_differentials()
>>> for omega in differentials:
...    print omega
x*y/(2*x**3 + 3*y**2)
x**3/(2*x**3 + 3*y**2)
```

Homology Basis

Dual to the holomorphic differentials is the first homology group of $X$. This space is spanned by a canonical basis of cycles $\{a_1, \ldots, a_g, b_1, \ldots, b_g\}$.

We can compute and plot the projections of these cycles in complex $x$ and $y$ planes.

```python
>>> a_cycles = X.a_cycles()
>>> b_cycles = X.b_cycles()
>>> xfig = a_cycles[0].plot_x(512)
```
Period Matrices and Riemann Matrices

Using the above two ingredients, we can compute the period matrix \( \tau = [A \mid B] \in \mathbb{C}^{g \times 2g} \) and Riemann matrix \( \Omega \in \mathbb{C}^{g \times g} \) of \( X \) where

\[
A_{ij} = \int_{a_j} \tilde{\omega}_i, \quad B_{ij} = \int_{b_j} \tilde{\omega}_i, \\
\Omega = A^{-1}B.
\]
```python
>>> import numpy
```n
generate numerical output

```python
>>> tau = X.period_matrix()
```

```python
[[ -1.381589e-12-1.201925j 1.849572e+00+0.600962j
  -7.064736e-01+2.174302j -1.849572e+00+2.545717j]
 [ 9.228812e-12+1.971464j 7.161762e-01-0.985732j
  -1.874974e+00-1.362248j -7.161762e-01+0.2327j ]]
```n

```python
>>> Omega = X.riemann_matrix()
```

```python
[[-1.309017+0.951057j -0.809017+0.587785j]
 [-0.809017+0.587785j -1.000000+1.175571j]]
```n

We numerically verify that \( \Omega \) is a Riemann matrix: a complex \( g \times g \) symmetric matrix with positive definite imaginary part.

```python
>>> import numpy.linalg
```n

```python
>>> numpy.linalg.norm(Omega - Omega.T)
3.64209384448e-11
```n

```python
>>> numpy.linalg.eigvals(Omega.imag)
[ 0.464905 1.661722]
```n

Riemann Theta Functions

Another major feature of Abelfunctions is the ability to efficiently compute the Riemann theta function \( \theta : \mathbb{C}^g \times h_g \)

\[
\theta(z, \Omega) = \sum_{n \in \mathbb{Z}^g} e^{2\pi i (\frac{1}{2} n \cdot \Omega n + n \cdot z)}
\]

where \( h_g \) is the space of \( g \times g \) Riemann matrices. Using the Riemann matrix computed above we can compute \( \theta(z, \Omega) \) for various \( z \in \mathbb{C}^2 \).
Abelfunctions is very efficient in computing the Riemann theta function for many values of \( z \). Here we plot the real and imaginary parts of \( \theta(z, \Omega) \) for \( z = (x + iy, 0) \) with \( x \in [0, 5], y \in [0, 1] \).

First, we setup a grid of complex numbers.

```python
>>> N = 128
>>> x = numpy.linspace(-1,1,N)
>>> y = numpy.linspace(0.1,0.6,N)
>>> X,Y = numpy.meshgrid(x,y)
>>> Z = X + 1.0j*Y
>>> Z = Z.flatten()
```

Next, we evaluate the Riemann theta function.

```python
>>> U = RiemannTheta([[z,0] for z in Z], Omega)
>>> U = U.reshape(N,N)
```

Finally, we plot the real and imaginary parts of the Riemann theta function at each of these values.

```python
>>> import matplotlib
>>> import matplotlib.pyplot as plt

>>> fig = plt.figure(figsize=(16,6))

>>> ax_real = fig.add_subplot(1,2,1, projection='3d')
>>> ax_real.plot_surface(X, Y, U.real, cmap='jet')

>>> ax_imag = fig.add_subplot(1,2,2, projection='3d')
>>> ax_imag.plot_surface(X, Y, U.imag, cmap='jet')

>>> fig.show()
```
Each of the documents listed below give an in-depth tutorial on a particular feature or module of Abelfunctions.

### 3.1 Riemann Theta Functions

Abelfunctions provides tools for computing with the Riemann theta function. The implementation is fast and works well with Numpy data types.

The Riemann theta function $\theta(z, \Omega)$ is defined by

$$\theta(z, \Omega) = \sum_{n \in \mathbb{Z}^g} e^{2\pi i (\frac{1}{2} n \cdot \Omega n + n \cdot z)}$$

where $h_g$ is the space of $g \times g$ Riemann matrices. It can be computed by calling `RiemannTheta()`, which is located in `abelfunctions.riemann_theta.riemann_theta.RiemannTheta` of the code.

**Evaluating Theta**

First, we need a $3 \times 3$ Riemann matrix: a complex symmetric with positive definite imaginary part. Throughout this document we will use the following Riemann matrix as an example:

$$\Omega = \begin{pmatrix} i & 1/2 & 1/2 \\ 1/2 & i & 1/2 \\ 1/2 & 1/2 & i \end{pmatrix}$$

```python
>>> import numpy
>>> import matplotlib.pyplot as plt
>>> from abelfunctions import RiemannTheta

>>> Omega = [[1.j, 0.5, 0.5],
            [0.5, 1.j, 0.5],
            [0.5, 0.5, 1.j]]
```

We evaluate the Riemann theta function like so:

```python
>>> z = [0, 0, 0]
>>> RiemannTheta(z, Omega)
(1.2362529854052204+9.161987717498445e-35j)
```

Optionally, we can obtain the “exponential” and “oscillatory” parts of the Riemann theta function separately. We define

$$\theta(z, \Omega) = e^{\pi i Y^{-1} \tilde{\theta}(z, \Omega)}$$

```
where $y$ and $Y$ are the imaginary parts of the vector $z$ and the Riemann matrix $\Omega$, respectively. The function $\hat{\theta}$ is referred to as the oscillatory part of $\theta$ and the exponential factor before it is referred to as the exponential part. It is useful to consider these two components separately in order to control the double-exponential growth of $\theta$ in certain applications.

We compute these separate components using `RiemannTheta.exponential_part` and `RiemannTheta.oscillatory_part`.

```python
>>> # exponential_part only computes the exponent π * (y . Yinv*y)
>>> u = RiemannTheta.exponential_part(z, Omega)
>>> v = RiemannTheta.oscillatory_part(z, Omega)
>>> numpy.exp(u) * v
(1.2362529854052204+9.1619877174984454e-35j)
```

Derivatives of $\theta$ in the unit component directions can be computed by supplying an optional `derivs` keyword. That is, derivatives of the form

$$\frac{\partial}{\partial z_i} \theta((z_1, \ldots, z_g), \Omega)$$

To compute the first $z_1^1$ partial provide the vector `derivs=[[1,0,...,0]]`. The `derivs` keyword works with the components of $\theta$ as well. Below we compute the three unit directional derivatives of $\theta$ at the point $z = [0.2 + 0.5i, 0.3 - 0.1i, -0.1 + 0.2i]$.

```python
>>> z = [0.2+0.5j, 0.3-0.1j, -0.1+0.2j]
>>> RiemannTheta(z, Omega, derivs=[[1,0,0]])
(-5.72959007264-0.891993752803j)
>>> RiemannTheta(z, Omega, derivs=[[0,1,0]])
(-0.163009877076-0.650792690979j)
>>> RiemannTheta(z, Omega, derivs=[[0,0,1]])
(1.01154060736+0.0305285337127j)
```

Second derivatives are computed similarly.

```python
>>> RiemannTheta(z, Omega, derivs=[[1,0,0],[1,0,0]])
(-5.5449651294619331+35.991137389193163j)
```

### Plotting Slices of Theta

Continuing with this example, we wish to evaluate $\theta(z, \Omega)$ on 1-complex dimensional slice of $\mathbb{C}^3$ and plot the real and imaginary parts of the result. In particular, we evaluate $\theta$ on

$$W = \{(z_1, 0, 0) : z_1 \in [0, 1/4] \times [0, 1/4]i\}.$$  

First, we have to generate this set of vectors. We use a `numpy meshgrid` since we eventually want to display a contour plot.

```python
>>> n = 64
>>> x = numpy.linspace(0,0.25,n)
>>> y = numpy.linspace(0,0.25,n)
>>> X, Y = numpy.meshgrid(x, y)
>>> z = (X + 1.j*Y).flatten()
>>> W = [[zl,0,0] for zl in z]
```

To evaluate the Riemann theta function on each of the vectors in $W$ we simply pass the list in as an argument `RiemannTheta(W, Omega)`. One could
Finally, we plot the real and imaginary parts of the result. Before doing so, we need to resize the output array to match the dimensions of the $X$- and $Y$-coordinate arrays.

```python
>>> V = RiemannTheta(W, Omega)
>>> V
[ 1.23625299 +9.16198772e-35j  1.23623096 +4.61397926e-20j
  1.23616488 +1.85236701e-19j ... 1.17416789 -1.62839443e-01j
  1.16973862 -1.62986765e-01j 1.16530683 -1.63032759e-01j]
```

```python
>>> import matplotlib.pyplot as plt
```

```python
>>> V.resize((n,n))
>>> plt.contourf(X,Y,V.real)
>>> plt.colorbar()
>>> plt.title(r'$Re\, \theta(z,\Omega)$', size=16)
>>> plt.xlabel('$Re\, z_1$')
>>> plt.ylabel('$Im\, z_1$')
>>> plt.show()
```

```python
>>> plt.contourf(X,Y,V.imag)
>>> plt.colorbar()
>>> plt.title(r'$Re\, \theta(z,\Omega)$', size=16)
>>> plt.xlabel('$Re\, z_1$')
>>> plt.ylabel('$Im\, z_1$')
>>> plt.show()
```
Below is a listing of applications demonstrating how to use abelfunctions to solve various problems. Some of these solutions and applications are the subject of Chris Swierczewski’s Ph.D. thesis.

Please consult the Writing Documentation page if you would like to contribute a new application of abelfunctions to the documentation.

4.1 Computing Bitangents of Plane Quartics

(This is a stub.)

4.2 Genus Two and Three Solutions to the Kadomtsev-Petviashvili Equation

(This is a stub.)
5.1 Primary Functionality

Below is a list of the primary functions offered by abelfunctions. This is not a comprehensive list of all functions written in abelfunctions but, rather, the functions and classes one will most likely interact with when solving problems using the software.

- `abelfunctions.riemann_surface.RiemannSurface`
- `abelfunctions.divisor.Place`
- `abelfunctions.divisor.Divisor`
- (and several others)

5.2 Module Reference

For an in-depth reference to a specific module or function please take a look at the automatically-generated documentation below. Links to the corresponding source code are provided as well.

5.2.1 Abel Map and Jacobian `abelfunctions.abelmap`

Given a Riemann surface \( X \) and a fixed place \( P_0 \in X \) The Abel Map is a function \( A : X \to \mathbb{C}^g \) given by

\[
A(P) := \left( \int_{P_0}^P \omega_1, \ldots, \int_{P_0}^P \omega_g \right)
\]

where \( \omega_1, \ldots, \omega_g \) are the normalized basis of holomorphic one-forms defined on \( X \). The Abel map can be accessed via `AbelMap()`, which is an instantiation of the class `AbelMap_Function`. `AbelMap_Function` makes use of `Jacobian`, also defined here. `Jacobian` defines the Jacobian of a Riemann surface.

\[
J(X) := \mathbb{C}^g / (\mathbb{Z}^g + \Omega \mathbb{Z}^g)
\]

where \( \Omega \) is the Riemann matrix of the Riemann surface. The primary use of the Jacobian is to reduce vectors in \( \mathbb{C}^g \) modulo this lattice. The Jacobian is constructed with `Jacobian(X)`.
AbelMap Function

AbelMap Function() The Abel Map.
Jacobian(X) The Jacobian of a Riemann Surface.

Functions

fractional_part(z[, tol]) Returns the fractional part of a vector.

Notes

The implementation of \texttt{AbelMap Function} is based on the algorithm described in \cite{CTAM}. When calling \texttt{AbelMap()} on a place \( P \in X \) a path on the Riemannn surface is constructed to it from the base place \( P_0 \in X \). See \texttt{abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory.path_to_place()} for details on how this path is constructed.

Once this path is constructed, each of the non-normalized basis Abelian differentials of the first kind computed using \texttt{abelfunctions.integralbasis.integral_basis()} are integrated along it. Multiplying the result by \( A^{-1} \), the inverse of the \( a \)-cycles periods, is equivalent to integrating the normalized holomorphic differentials.

The implementation of \texttt{Jacobian} uses the fact that

References

Examples

We construct a Riemann surface corresponding to the curve \( f(x, y) = x^2 y^3 - x^4 + 1 \) and evaluate the Abel map at the single place \( P \in X \) lying above the point \( x = 0 \).

```python
>>> import numpy; numpy.set_printoptions(precision=4, suppress=True)
>>> from sympy.abc import x, y
>>> from abelfunctions import RiemannSurface, AbelMap, Jacobian

>>> f = x**2*y**3 - x**4 + 1
>>> X = RiemannSurface(f, x, y)
>>> P = X(0)[0]
>>> AbelMap(P)
[-0.5261+0.0864j, 0.0669+0.6392j, -0.7495+1.1037j, -1.5030+1.0356j]
```

The Abel map can be computed between two different places. Let \( Q \in X \) be the sum of all places lying above the point \( x = 2 \). We compute \( A(P, Q) \) below.

```python
>>> Q = sum(X(2))
>>> AbelMap(P, Q)
[ 0.1468-0.0985j, 0.8467+0.6989j, 0.0996+1.0083j, -1.1003+0.8159j]
```

Contents

class abelfunctions.abelmap.AbelMap Function

Bases: object

The Abel Map.

\[
A(P) := \left( \int_{P_0}^P \omega_1, \ldots, \int_{P_0}^P \omega_g \right)
\]
By default, the Abel map is computed from the base place of the Riemann surface. Optionally, a starting place can be provided. That is, \( \text{AbelMap}(P, Q) \) returns

\[
A(P) := \left( \int_Q \omega_1, \ldots, \int_{P_0} \omega_g \right).
\]

The argument, \( Q \), can also be a divisor.

**Methods**

\[\text{eval(*args)} \quad \text{Evaluate the Abel map at a place or divisor.}\]

\[\text{_eval_primitive(P)} \quad \text{Primitive evaluation of the Abel map at a single place, } P.\]

\[\text{_eval_primitive(P)} \quad \text{Primitive evaluation of the Abel map at a single place, } P.\]

**_eval_primitive(P)**

Primitive evaluation of the Abel map at a single place, \( P \).

In the case when the input to \( \text{AbelMap\_Function.eval()} \) is a divisor the Abel map is computed at each place first. Always starts from the base place \( P_0 \) of the Riemann surface.

**Parameters**

\( P \) : Place

The target place.

**Returns**

\( \text{value} : \text{complex array} \)

A complex g-vector equal to the Abel map evaluated at \( P \).

\[\text{eval(*args)} \quad \text{Evaluate the Abel map at a place or divisor.}\]

When only one argument (a place or divisor) \( D \) is provided, return \( A(P_0, D) \). If two arguments are given, a place \( P \) and a divisor \( D \), then return \( A(P, D) \).

Constructs a \( \text{RiemannSurfacePath} \gamma : [0, 1] \to X \) starting at the base place \( P_0 \) and ending at \( P \). Then, simply integrates each of the holomorphic one-forms along this path.

**Parameters**

\( P \) : Place, optional

If two arguments are given, the first argument corresponds to the starting place of the Abel map. By default, this is the base place of the Riemann surface.

\( D \) : Place or Divisor

The target place or divisor of the Abel Map.

**Returns**

\( \text{value} : \text{complex array} \)

The Abel map \( A(P, D) \).

**class abelfunctions.abelmap.Jacobian(X)**

\[\text{Bases: object}\]

The Jacobian of a Riemann Surface.

The Jacobian of a Riemann surface is defined by

\[
J(X) := \mathbb{C}^g / (\mathbb{Z}^g + \Omega \mathbb{Z}^g)
\]

where \( \Omega \) is the Riemann matrix of the Riemann surface. The primary use of the Jacobian is to reduce vectors in \( \mathbb{C}^g \) modulo this lattice. The Jacobian is constructed with \( \text{Jacobian}(X) \).
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omega</td>
<td>(complex matrix) The Riemann matrix of the surface.</td>
</tr>
<tr>
<td>g</td>
<td>(int) The genus of the Riemann surface</td>
</tr>
<tr>
<td>M</td>
<td>(complex matrix) A transformation matrix used to decompose a vector $z$ into its components $z = u + \Omega v$.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>components($z$)</td>
<td>Decomposes $z$ into its lattice components $z = u + \Omega v$.</td>
</tr>
<tr>
<td>eval($z$)</td>
<td>Reduce $z$ modulo the lattice defining the Jacobian.</td>
</tr>
</tbody>
</table>

components($z$)
Decomposes $z$ into its lattice components $z = u + \Omega v$.

Parameters $z$ : complex array
An array / vector of g components.

Returns $u, v$ : complex array
Arrays / vectors of g components.

Notes
This is often used with reduce_components(). In some cases when $z$ is equal lattice element, floating point error can case reduce_components() to incorrectly indicate this. (Two zero vectors should be returned in this case.) At the end of this routine we round the results to the nearest 15th decimal place in an attempt to rectify this error.

eval($z$)
Reduce $z$ modulo the lattice defining the Jacobian.

Parameters $z$ : complex array
An array / vector of g components.

Returns $z_{mod}$ : complex array
The vector $z$ reduced modulo the lattice Lambda.

abelfunctions.abelmap.fractional_part($z, tol=1e-08$)
Returns the fractional part of a vector.

This function is different from numpy.floor, which also determines the fractional part of a vector, in the sense that it can handle components that are close to integers. That is fractional_part(0.999999999999) and fractional_part(1.000000000001) should both return 0 since they’re close to an integer.

This is primarily used in Jacobian to reduce a vector in $\mathbb{C}^g$ modulo the lattice $\mathbb{Z}^g + \Omega \mathbb{Z}^g$.

Parameters $z$ : double array
$tol=1e-8$ : double
Tolerance for determining when an entry is close to an integer.

Returns $w$ : array
5.2.2 Analytic Continuation `abelfunctions.analytic_continuation`

Objects for performing analytic continuation along a `RiemannSurfacePath` object and, in fact, is a member object of class: `RiemannSurfacePath`.

This module contains an abstract class: `AnalyticContinuator`

### Classes

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AnalyticContinuator</code></td>
<td>Abstract class for analytically continuing along a curve.</td>
</tr>
<tr>
<td><code>AnalyticContinuatorPuiseux</code></td>
<td>Riemann surface path analytic continuation using Puiseux series.</td>
</tr>
</tbody>
</table>

### Functions

### Examples

### Contents

```python
class abelfunctions.analytic_continuation.AnalyticContinuator
Bases: object

Abstract class for analytically continuing along a curve.

An analytic continuator object dictates how to continue a y-fibre from one x-point \( x_i \) to another point \( x_{i+1} \) along a `RiemannSurfacePath` object. (Technically, a `RiemannSurfacePathPrimitive` object which serves as the base class for all path types.)

#### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>(RiemannSurface) The Riemann surface on which analytic continuation takes place</td>
</tr>
<tr>
<td>gamma</td>
<td>(RiemannSurfacePathPrimitive) The path along which the analytic continuation is performed.</td>
</tr>
</tbody>
</table>

#### Methods

```python
analytically_continue()  Analytically continues the fibre \( y_i \) from \( x_i \) to \( x_{i+1} \).
```

```python
analytically_continue()  Analytically continues the fibre \( y_i \) from \( x_i \) to \( x_{i+1} \).
```

#### Parameters

- \( x_i \): complex
  - The starting complex x-value.
- \( y_i \): complex[:]
  - The starting complex y-fibre lying above \( x_i \).
- \( x_{i+1} \): complex
  - The target complex x-value.

#### Returns

- complex[:]
  - The y-fibre lying above \( x_{i+1} \).
**integrate()**
Integrate omega on the path using this analytic continuator.

**Parameters**
- **omega**: Differential

**Returns**
- complex

**parameterize()**
Returns the differential omega parameterized on the path.

Given a differential math: $\omega = \omega(x,y)dx$, **parameterize** returns the differential

$$\omega_s(s) = \omega(\gamma_x(s), \gamma_y(s))\gamma'_x(s)$$

where $s \in [0, 1]$ and $\gamma_x, \gamma_y$ and the x- and y-components of the path **gamma** using this analytic continuator.

**Parameters**
- **omega**: Differential

**Returns**
- function

```python
class abelfunctions.analytic_continuation.AnalyticContinuatorPuiseux
Bases: abelfunctions.analytic_continuation.AnalyticContinuator
```

Riemann surface path analytic continuation using Puiseux series.

We must use Puiseux series in order to analytically continue a y-fibre to a discriminant point $x = b$. The initial y-fibre establishes the ordering of the `PuiseuxXSeries` at $x = b$.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RS</strong></td>
<td>(RiemannSurface) The Riemann surface on which analytic continuation takes place</td>
</tr>
<tr>
<td><strong>gamma</strong></td>
<td>(RiemannSurfacePathPrimitive) The path along which the analytic continuation is performed.</td>
</tr>
<tr>
<td><strong>center</strong></td>
<td>(sympy.Expr) The center of the Puiseux series expansions. Usually a discriminant point of the underlying curve of the Riemann surface.</td>
</tr>
<tr>
<td><strong>puiseux_series</strong></td>
<td>(list, PuiseuxXSeries) An ordered list of <code>PuiseuxXSeries</code> corresponding to each branch of the curve.</td>
</tr>
</tbody>
</table>

**Methods**

```python
analytically_continue
_compute_puisseux_series()
```

**_compute_puisseux_series()**
Return the Puiseux series at the center in the correct order.

In order to analytically continue from the regular places at the beginning of the path $x = a$ to the discriminant places at the end of the path $x = b$, we need to compute all of the `PuiseuxXSeries` at $x = b$. There are two steps to this calculation:

- compute enough terms of the Puiseux series centered at $x = b$ in order to accurately capture the y-roots at $x = a$.
- permute the series accordingly to match up with the y-roots at $x = a$.

**Parameters**
- **gamma**: RiemannSurfacePathPrimitive

**Returns**
- list

A list of ordered Puiseux series corresponding to each branch at $x = a$. 

---

Chapter 5. Reference
Notes

A more efficient method of extending the PuiseuxTSeries could probably be designed. In particular, some series may "converge" faster than others or may have much higher order terms than the others.

**integrate()**
Integrate the differential along the underlying path.

When integrating a holomorphic differential to place corresponding to a discriminant point of the curve care needs to be taken since the denominator of the differential vanishes at this point.

**Parameters**
- **omega** : Differential

**parameterize()**
Returns the differential omega parameterized on the path.

Given a differential math:`omega = omega(x,y)dx`, parameterize returns the differential

```latex
\omega_\gamma(s) = \omega(\gamma_x(s), \gamma_y(s))\gamma'_x(s)
```

where `s \in [0, 1]` and `\gamma_x`, `\gamma_y` and the x- and y-components of the path `gamma` using this analytic continuator.

**Parameters**
- **omega** : Differential

**Returns**
function

### 5.2.3 Analytic Continuation: Smale’s Alpha Theory

This module implements subclass of :class:`AnalyticContinuator` which uses Smale’s alpha theory for analytically continuing y-roots. This AnalyticContinuator is only effective away from the branch points and singular points of the curve since its primary mechanism is Newton iteration. A different AnalyticContinuator, such as :class:`AnalyticContinuatorPuiseux`, is required in order to analytically continue to such points.

**Functions**

- factorial
- newton
- smale_alpha
- smale_beta
- smale_gamma

**Classes**

- UnivariatePolynomial
- MultivariatePolynomial

**Globals:**

ABECPFUNCTIONS_SMALE_ALPHA0

**class** `abelfunctions.analytic_continuation.smale.AnalyticContinuatorSmale`

**Bases:** `abelfunctions.analytic_continuation.AnalyticContinuator`

Riemann surface path analytic continuation using Smale’s alpha theory.

When sufficiently far away from branch points and singular point of the curve we can use Newton iteration to analytically continue the y-roots of the curve along paths in \( \mathbb{C}_x \). Smale’s alpha theory is used to determine an optimal step size in \( \mathbb{C}_x \) to ensure that Newton iteration will not only succeed with each y-root but the y-roots will not “collide” or swap places with each other. See [XXX REFERENCE XXX] for more information.

**Note:** This class uses the functions `newton()`, `smale_alpha()`, `smale_beta()`, and `smale_gamma()`, defined in this module.

## 5.2. Module Reference
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>(RiemannSurface) The Riemann surface on which analytic continuation takes place.</td>
</tr>
<tr>
<td>gamma</td>
<td>(RiemannSurfacePathPrimitive) The path along which the analytic continuation is performed.</td>
</tr>
<tr>
<td>df</td>
<td>(MultivariatePolynomial[:]) A list of all of the y-derivatives of the curve, ( f = f(x,y) ). These are used by Smale’s alpha theory.</td>
</tr>
</tbody>
</table>

### Methods

#### `analytically_continue`

Analytically continues the fibre \( y_i \) from \( x_i \) to \( x_{i+1} \) using Smale’s alpha theory.

```python
analytically_continue()
```

**Parameters**

- **gamma** : RiemannSurfacePathPrimitive
  - A Riemann surface path-type object.
- **xi** : complex
  - The starting complex x-value.
- **yi** : complex[:]
  - The starting complex y-fibre lying above \( x_i \).
- **xip1** : complex
  - The target complex x-value.

**Returns**

- complex[:]
  - The y-fibre lying above \( x_{i+1} \).

#### `integrate`

Integrate \( \omega \) on the path using this analytic continuator.

```python
integrate()
```

**Parameters**

- **omega** : Differential

**Returns**

- complex
  - The y-fibre lying above \( x_{i+1} \).

#### `parameterize`

Returns the differential omega parameterized on the path.

```python
parameterize()
```

**Parameters**

- **omega** : Differential

**Returns**

- function
  - The differential omega parameterized on the path.

#### 5.2.4 Differentials `abelfunctions.differentials`

This module contains functions for computing a basis of holomorphic differentials of a Riemann surface given by a complex plane algebraic curve \( f \in \mathbb{C}[x,y] \). A differential \( \omega = h(x,y)dx \) defined on a Riemann surface \( X \) is holomorphic on \( X \) if it is holomorphic at every point on \( X \).
The function `differentials()` computes the basis of holomorphic differentials from an input algebraic curve $f = f(x, y)$. The differentials themselves are encapsulated in a `Differential` Cython class.

**Classes**

- **Differential** A differential one-form which can be defined on a Riemann surface.

**Functions**

- `differentials` Returns a basis of holomorphic differentials on Riemann surface.
- `mnuk_conditions` Determine the Mnuk conditions on the coefficients of $P$.

**References**

**Examples**

**Contents**

- `abelfunctions.differentials.AbelianDifferentialSecondKind`
  - Bases: `abelfunctions.differentials.Differential`
  
  Defines an Abelian Differential of the second kind.

  An Abelian differential of the second kind is one constructed in the following way: given a place $P \in X$ and a positive integer $m$ an Abelian differential of second kind is a meromorphic differential with a pole only at $P$ of order $m + 1$.

  `valuation_divisor()`
  
  Returns the valuation divisor of the Abelian differential of the second kind.

  **Parameters** none
  **Returns** Divisor

- `abelfunctions.differentials.Differential`
  - Bases: object
  
  A differential one-form which can be defined on a Riemann surface.

**Notes**

To reduce the number of discriminant points to check for computing the valuation divisor we keep separate the numerator and denominator of the Differential. This behavior may change after implementing different types of differentials.

**Attributes**

- `numer, denom` (MultivariatePolynomial) Fast multivariate polynomial objects representing the numerator and denominator of the differential.
Methods
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>eval</strong></td>
<td>Evaluate the differential at the complex point ((x, y)).</td>
</tr>
<tr>
<td><strong>as_numer_denom</strong></td>
<td>Returns the differential as a numerator, denominator pair.</td>
</tr>
<tr>
<td><strong>as_sympy_expr</strong></td>
<td>Returns the differential as a Sympy expression.</td>
</tr>
</tbody>
</table>

**_find_necessary_xvalues()**

Returns a list of x-points over which the places appearing in the valuation divisor are found.

**valuation_divisor()** requires a necessary list of x-values from which to compute places which may appear in the valuation divisor of the differential.

In the case when `self.denom` is equal to \(\partial_y f\) we simply use the discriminant points of the curve.

**Parameters none**

**Returns list**

**as_numer_denom()**

Returns the differential as a numerator, denominator pair.

**Returns list, sympy.Expr**

**as_sympy_expr()**

Returns the differential as a Sympy expression.

**Returns sympy.Expr**

**centered_at_place()**

Rewrite the differential in terms of the local coordinates at \(P\).

If \(P\) is a regular place, then returns `self` as a sympy expression. Otherwise, if \(P\) is a discriminant place \(P(t) = \{x(t), y(t)\}\) then returns

\[
\omega|_P = q(x(t), y(t))x'(t)/\partial_y f(x(t), y(t)).
\]

**Parameters P : Place**

**order : int, optional**

Passed to `PuiseuxTSeries.eval_y()`.

**Returns sympy.Expr**

**eval()**

Evaluate the differential at the complex point \((x, y)\).

**Parameters x,y : complex**

**Returns complex**

Returns the value \(\omega(x, y)\).

**evaluate()**

Evaluates `omega` along the path at \(N\) uniform points.

**Note:** Note: right now it doesn’t matter what the values in \(t\) are. This function will simply turn \(t\) into a bunch of uniformly distributed points between 0 and 1.

**Parameters omega : Differential**

**t : double[:]**

An array of \(t\) between 0 and 1.
Returns complex[:]
The differential omega evaluated along the path at $N$ points.

`localize()`
Same as `centered_at_place()`.

`plot()`
Plot the differential along the RiemannSurfacePath $\gamma$.

Parameters $\gamma$ : RiemannSurfacePath
A path along which to evaluate the differential.

$N$ : int
Number of interpolating points to use when plotting the value of the differential along the path $\gamma$

`grid` : boolean
(Default: `False`) If true, draw gridlines at each “segment” of the parameterized RiemannSurfacePath. See the RiemannSurfacePath documentation for more information.

Returns `matplotlib.Figure`

`valuation_divisor()`
Returns the valuation divisor of the differential.
This is a generic algorithm for computing valuation divisors and should only be used if nothing is known about the differential in question.

If the differential is Abelian of the first kind (holomorphic) then create an instance of AbelianDifferentialFirstKind. Similarly, if the differential is Abelian of the second kind then create an instance of AbelianDifferentialSecondKind. These implement versions of `valuation_divisor()` that use properties of the differential to save on time.

Parameters none

Returns `Divisor`

`abelfunctions.differentials.differentials()`
Returns a basis of holomorphic differentials on Riemann surface.

The surface is given by the desingularization and compactification of the affine complex plane algebraic curve $f = f(x,y)$.

Parameters $f$ : sympy.Expr

$x$ : sympy.Symbol

$y$ : sympy.Symbol

Returns list, Differential

`abelfunctions.differentials.fast_expand()`
Quickly compute the Taylor expansion of $\text{numer}/\text{denom}$.

Parameters $\text{numer}$, $\text{denom}$ : sympy.Expr

Polynomials in $t$.

$t$ : sympy.Symbol

The dependent variable of $\text{numer}$ and $\text{denom}$.

`order` : int
The desired order of the expansion.

Returns sympy.Expr

abelfunctions.differentials.mnuk_conditions()
Determine the Mnuk conditions on the coefficients of \( P \).

Determine the conditions on the coefficients \( c \) of \( P \) at the integral basis element \( b \) modulo the curve \( g = g(u,v) \).
See [Mnuk] for details.

Parameters g : sympy.Expr
    u : sympy.Symbol
    v : sympy.Symbol
    b : sympy.Expr
        An integral basis element.
    P : sympy.Expr
        A generic adjoint polynomial as provided by differentials(). Only one instance is created for caching and performance purposes.
    c : list, sympy.Symbol
        A list of the unknown symbolic coefficients we wish to solve for.

Returns list, sympy.Expr
    A list of expressions from which a system of equations is build to determine the differentials.

5.2.5 Places and Divisors divisor

A module defining places and divisors on a Riemann surface. A regular affine place is simply given as an \((x_0, y_0)\) tuple on the curve \( C : f(x, y) = 0 \) from which the Riemann surface is defined.

Classes

| Divisor(RS, d) | A divisor on a Riemann surface. |
| Place(RS[, name]) | A Place on a Riemann surface. |

Functions

ZeroDivisor(RS) A class representing the zero divisor on a Riemann surface.

Examples

Contents

class abelfunctions.divisor.DiscriminantPlace(RS, P, **kwds)
    Bases: abelfunctions.divisor.Place
    A discriminant place on a Riemann surface.
class abelfunctions.divisor.Divisor(RS, d)
    Bases: object

A divisor on a Riemann surface.
A divisor is a formal sum of places \( P_i \) along with multiplicities \( m_i \) usually written,
\[
D = m_1 P_1 + \cdots + m_k P_k,
\]
where the places \( P_i \) are distinct.

Attributes

- **places** *(list)* A list of the distinct places in this divisor.
- **multiplicities** *(list)* The corresponding multiplicities of the places in this divisor.

class abelfunctions.divisor.Place(RS, name=None)
    Bases: abelfunctions.divisor.Divisor

A Place on a Riemann surface.

Is abelfunctions a Riemann surface is obtained by desingularizing and compactifying a complex algebraic curve. Every place \( P \) on the resulting Riemann surface can be described in terms of a Puiseux series in a local parameter \( t \).
\[
P = (x(t), y(t)) = \alpha + \lambda t^\gamma y(t) = \sum_k \beta_k t^{n_k}
\]
When \( \alpha \) is not a discriminant point or infinity of the curve then \( P = (x(0), y(0)) = (\alpha, \beta) \) is sufficient in representing the place from the curve. Otherwise, multiple places may have the same projection onto the curve.

Attributes

- **x** *(complex)* The x-projection of the place onto the underlying curve.
- **y** *(complex)* The y-projection of the place onto the underlying curve.
- **name** *(string, optional)* A name given to the place. Used when the place is a term in a \texttt{Divisor}.

Methods

- **is_discriminant**
- **valuation(omega)**

Returns the valuation of \( \omega \) at this place.

The valuation of \( \omega \) at this place \( P \) is an integer \( m \) such that
\[
\omega|_P = ct^m + O(t^{m+1})
\]
where \( P = P(0) \) is given in terms of some local parameter \( t \).

This method is a key ingredient in determining the valuation divisor of a differential \( \omega \).

Parameters omega: Differential

Returns int
class abelfunctions.divisor.RegularPlace(RS, x, y, **kwds)
    Bases: abelfunctions.divisor.Place
    A regular place on a Riemann surface.

class abelfunctions.divisor.ZeroDivisor(RS)
    Bases: abelfunctions.divisor.Divisor
    A class representing the zero divisor on a Riemann surface.

5.2.6 Integral Basis abelfunctions.integralbasis

A module for computing integral bases of algebraic function fields of the form \( O(X) = \mathbb{C}[x, y]/(f(x, y)) \) where \( X : f(x, y) = 0 \). The algorithm is based off of the paper “An Algorithm for Computing an Integral Basis in an Algebraic Function Field” by Mark van Hoeij [vHoeij].

An integral basis for \( O(X) \) is a set of \( \beta_i \in \mathbb{C}(x, y) \) such that

\[
O(X) = \beta_1 \mathbb{C}[x, y] + \cdots + \beta_g \mathbb{C}[x, y].
\]

This data is necessary for computing a basis for the space of holomorphic differentials \( \Omega_X^1 \) defined on the Riemann surface \( X \) which is implemented in `differentials`.

Functions

\[ \text{integral_basis}(f, x, y) \] Returns the integral basis of the algebraic function field of \( f \).

References

Examples

We compute an integral basis of the curve \( f(x, y) = (x^2 - x + 1)y^2 - 2x^2y + x^4 \).

```python
# from abelfunctions import *
# import sympy
# from sympy.abc import x,y
# f = (x**2 - x + 1)*y**2 - 2*x**2*y + x**4
# X = RiemannSurface(f,x,y)
# b = integral_basis(X)
# sympy.pprint(b, use_unicode=False)
```

\[
xy - y
\]

\[
[1, y - -------]
\]

\[
2
\]

\[
\times
\]

Contents

abelfunctions.integralbasis.Int(i, px)

Computes \( \text{Int}_i = \sum_{k \neq i} v(p_i - p_k) \).

\text{Int} is used in `compute_expansion_bounds()` for determining sufficient bounds on Puiseux series expansions.
Parameters i: int

p: list, PuiseuxXSeries

A list of PuiseuxXSeries.

Returns sympy.Rational

abelfunctions.integralbasis._integral_basis_monic(f, x, y)

Returns the integral basis of a monic curve.

Called by integral_basis() after monicizing its input curve.

Parameters f: sympy.Expr

x: sympy.Symbol

y: sympy.Symbol

Returns list, sympy.Expr

A list of rational functions representing an integral basis of the monic curve.

See also:

integral_basis generic integral basis function

abelfunctions.integralbasis.compute_bd(f, x, y, df, r, alpha, a)

Determine the next integral basis element from those already computed.

Parameters f: sympy.Expr

x: sympy.Symbol

y: sympy.Symbol

b: list

The current set of integral basis elements.

df: list

The set of irreducible factors.

r: list of PuiseuxTSeries

A list of lists of truncated Puiseux series centered each of the x-values in the list alpha.

alpha: list of complex

The roots of each irreducible factor in k.

a: list of sympy.Symbols

Returns sympy.Expression

The next integral basis element.

abelfunctions.integralbasis.compute_expansion_bounds(px)

Returns a list of necessary bounds on each Puiseux series in px.

Computes the expansion bounds \( N_1, \ldots, N_n \) such that for all polynomials \( G \in L[x, y] \) the truncation \( r_i \) of the Puiseux series \( p_i \) satisfying \( v(r_i - p_i) > N_i \) satisfies the relation

\[ \forall M, i, v(G(r_i)) > M \]
if and only if

\[ \forall M, i, v(G(p_i)) > M. \]

That is, the truncations \( r_i \) are sufficiently long so that polynomial evaluation of \( r_i \) and \( p_i \) has the same valuation.

**Parameters** `px` : list, PuiseuxXSeries

**Returns** list, int

A list of degree bounds for each PuiseuxXSeries in `px`.

`abelfunctions.integralbasis.compute_series_truncations(f, x, y, alpha)`

Computes Puiseux series at \( x = \alpha \) with necessary terms.

The Puiseux series expansions of \( f = f(x, y) \) centered at \( \alpha \) are computed up to the number of terms needed for the integral basis algorithm to be successful. The expansion degree bounds are determined by `compute_expansion_bounds()`.

**Parameters** `f` : sympy.Expr

- `x` : sympy.Symbol
- `y` : sympy.Symbol
- `alpha` : sympy.Complex

**Returns** list, sympy.Expression

A list of Puiseux series expansions cetnered at \( x = \alpha \) with enough terms to compute integral bases as SymPy expressions.

`abelfunctions.integralbasis.integral_basis(f, x)`

Returns the integral basis of the algebraic function field of \( f \).

An integral basis for the algebraic function field \( O(X) \) is a set of \( \beta_i \in C(x, y) \) such that

\[ O(X) = \beta_1 C[x, y] + \cdots + \beta_y C[x, y]. \]

**Parameters** `f` : sympy.Expr

- `x` : sympy.Symbol
- `y` : sympy.Symbol

**Returns** list, sympy.Expr

A list of rational functions representing an integral basis.

`abelfunctions.integralbasis.solve_coefficient_system(equations, vars, **kwds)`

Solve the linear system of `equations` with resp. to `vars`.

The systems of equations considered in this problem is always linear. This function constructs the linear system and solves it. The format is similair to `sympy.solve`.

**Parameters** `equations` : list

A system of equations in `vars`.

- `vars` : list
  A list of variables to solve for in `equations`.

**Returns** list or `None`

If a unique solution exists, returns as a list. Otherwise, returns `None`. 
5.2.7 Polynomials

Fast, Cython-level implementation of Univariate and Multivariate polynomials.

Authors

• Chris Swierczewski (April 2014)

class abelfunctions.polynomials.MultivariatePolynomial

Bases: object

Fast complex multivariate polynomial.

Converts a SymPy bivariate polynomial to a polynomial object that performs fast complex evaluation.

Note: coefficients are stored in self.c in “reverse order”. That is, the polynomial is given by

\[ c[0](x)y^{\text{deg}} + c[1](x)y^{\text{deg}-1} + \ldots + c[\text{deg}](x) \]

Attributes

<table>
<thead>
<tr>
<th>deg</th>
<th>(int) The degree of the polynomial.</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>(UnivariatePolynomial[:]) The coefficients of the polynomial starting with the degree deg term and ending with the constant term.</td>
</tr>
</tbody>
</table>

Methods

- eval(complex z) Evaluate the polynomial at the complex point z.
- is_zero_poly() Returns True if the polynomial is the zero polynomial.

class abelfunctions.polynomials.UnivariatePolynomial

Bases: object

Fast complex univariate polynomial.

Converts a SymPy univariate polynomial to a faster univariate polynomial. Used by MultivariatePolynomial.

Note: coefficients are stored in self.c in “reverse order”. That is, the polynomial is given by

\[ c[0]x^{\text{deg}} + c[1]x^{\text{deg}-1} + \ldots + c[\text{deg}] \]

Attributes

<table>
<thead>
<tr>
<th>deg</th>
<th>(int) The degree of the polynomial.</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>(int[:]) The coefficients of the polynomial starting with the degree deg term and ending with the constant term.</td>
</tr>
</tbody>
</table>

Methods

- eval(complex z) Evaluate the polynomial at the complex point z.
5.2.8 Puiseux Series abelfunctions.puiseux


Classes

- **PuiseuxTSeries**: A Puiseux t-series about some place \((\alpha, \beta) \in X\).
- **PuiseuxXSeries**: A Puiseux x-series centered at \(x = x_0\).

Functions

- **puiseux**: Singular parts of the Puiseux series above \(x = \alpha\).

References

Examples

Contents

class abelfunctions.puiseux.PuiseuxTSeries(f, x, y, x0, singular_data[, ...])

Bases: object

A Puiseux t-series about some place \((\alpha, \beta) \in X\).

A parametric Puiseux series \(P(t)\) centered at \((x, y) = (\alpha, \beta)\) is given in terms of a pair of functions

\[
x(t) = \alpha + \lambda t^e,
\]

\[
y(t) = \sum_{h=0}^{\infty} \alpha_h t^{eh},
\]

where \(x(0) = \alpha, y(0) = \beta\).

The primary reference for the notation and computational method of these Puiseux series is D. Duval.

Attributes

- **f**: (sympy.Expr)
- **x**: (sympy.Symbol)
- **y**: (sympy.Symbol)
- **x0**: (complex) The x-center of the Puiseux series expansion.
- **ramification_index**: (sympy.Rational) The ramification index \(e\).
- **terms**: (list) A list of exponent-coefficient pairs representing the y-series.
- **order**: (int) The order of the Puiseux series expansion.

Methods

- **xseries([all_conjugates])**: Returns the corresponding x-series.

Continued on next page
Table 5.16 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extend(order)</code></td>
<td>Extends the series in place.</td>
</tr>
<tr>
<td><code>eval_x(t)</code></td>
<td>Evaluate the x-part of the Puiseux series at t.</td>
</tr>
<tr>
<td><code>eval_y(t, order)</code></td>
<td>Evaluate of the y-part of the Puiseux series at t.</td>
</tr>
</tbody>
</table>

**_initialize_extension (extension_polynomial, t, y)_**

Set up regular part extension machinery.

RootOfs in expressions are not preserved under this transformation. (that is, actual algebraic representations are calculated.) each RootOf is temporarily replaced by a dummy variable

**Parameters** None

**Returns** None

**add_term (order=None)**

Extend the y-series terms in-place using Newton iteration.

The modular Newton iteration algorithm in `newton_iteration()` efficiently computes the series up to order $t^{2^n}$ where $2^n$ is the smallest power of two greater than the current order.

**coerce_to_numerical ()**

Coerces coefficients and data to numerical types.

In numerical situations it is best to work with numerical types instead of symbolic ones for performance purposes. When `coerce_to_numerical` is executed all internal data structures are converted to Numpy data types.

List of data coerced to numerical types:

- x-part terms
- y-part terms
- y-part series extension data

**Note:** Symbolic coefficient data is lost once this is performed.

**Parameters** — None

**Returns** — None

**eval_x (t)**

Evaluate the x-part of the Puiseux series at t.

**Parameters** t : sympy.Expr or complex

**Returns** complex

**eval_y (t, order=None)**

Evaluate of the y-part of the Puiseux series at t.

The y-part can be evaluated up to a certain order or with a certain number of terms.

**Parameters** t : complex or complex

nterms : int, optional

If provided, only evaluates using nterms in the y-part of the series. If set to zero, will evaluate the principal part of the series: the terms in the series which distinguishes places with the same x-projection.

order : int, optional
If provided, only evaluates up to order.

Returns complex

Notes

This can be sped up using a Holder-like fast exponent evaluation trick.

\textbf{extend} (order=None)
Extends the series in place.

Computes additional terms in the Puiseux series up to the specified order or with \textit{nterms} number of non-zero terms. If neither degree nor \textit{nterms} are provided then the next non-zero term will be added to this t-series.

Parameters \textbf{order} : int, optional
The desired degree to extend the series to.

Returns None

\textbf{extend_to_t} (t, curve_tol=1e-08, rel_tol=0.0001)
Extend the series to accurately determine the y-values at \(t\).

Add terms to the t-series until the the regular place \((x(t), y(t))\) is within a particular tolerance of the curve that the Puiseux series is approximating.

Parameters \textbf{t} : complex

\textbf{eps} : double
The tolerance for the corresponding point to lie on the curve.

\textbf{curve_tol} : double
A relative tolerance parameter used to ensure that the point \((x(t_0), y(t_0))\) is on the same branch as the center of the Puiseux series.

Returns none

The PuiseuxTSeries is modified in-place.

\textbf{extend_to_x} (x, curve_tol=1e-08, rel_tol=0.01)
Extend the series to accurately determine the y-values at \(x\).

Add terms to the t-series until the the regular place \((x, y)\) is within a particular tolerance of the curve that the Puiseux series is approximating.

Parameters \textbf{x} : complex

\textbf{curve_tol} : double
The tolerance for the corresponding point to lie on the curve.

\textbf{rel_tol} : double
A relative tolerance parameter used to ensure that the point \((x(t_0), y(t_0))\) is on the same branch as the center of the Puiseux series.

Returns none

The PuiseuxTSeries is modified in-place.
Abelfunctions Documentation, Release 0.1.0

nterms ()
Returns the number of non-zero computed terms.

Parameters None
Returns int
terms_from_yseries (yseries)
Efficiently extract the terms from the y-part of the series.

Parameter yseries
Returns int

Polynomial arithmetic is efficient in Sympy. Write the y-part of the Puiseux series as

\[ y_P(t) = \frac{1}{c^d} p(t) \]

where \( p \) is a polynomial in \( t \). We can efficiently extract the coefficients of \( p \) and perform the appropriate
transformation using the coefficient above.

Returns terms
A list of terms (n,beta) of the y-part of the series.

xseries (all_conjugates=True)
Returns the corresponding x-series.

Parameters all_conjugates : bool
(default: True) If True, returns all conjugates x-representations of this Puiseux t-series.
If False, only returns one representative.

Returns list
List of PuiseuxXSeries representations of this PuiseuxTSeries.

class abelfunctions.puiseux.PuiseuxXSeries (f, x, y, x0, obj, order=None, ramification_index=None)
Bases: object

A Puiseux x-series centered at \( x = x_0 \).

A Puiseux series \( p(x) \) centered at \( x = x_0 \) is a series of the form

\[ p(x) = \sum_{h=0}^{\infty} \alpha_h (x - x_0)^{n_h/e}, \]

The primary reference for the notation and computational method of these Puiseux series is D. Duval.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>(sympy.Expr)</td>
</tr>
<tr>
<td>x</td>
<td>(sympy.Symbol)</td>
</tr>
<tr>
<td>y</td>
<td>(sympy.Symbol)</td>
</tr>
<tr>
<td>x0</td>
<td>(complex) The center of the Puiseux series expansion.</td>
</tr>
<tr>
<td>terms</td>
<td>(tuple) A list of exponent-coefficient pairs.</td>
</tr>
<tr>
<td>order</td>
<td>(sympy.Rational) The order of the Puiseux series expansion.</td>
</tr>
<tr>
<td>ramification_index</td>
<td>(sympy.Rational) The ramification index ( e ).</td>
</tr>
</tbody>
</table>

Methods
eval(x)  Symbolic evaluation of the Puiseux series.

evalf(x[, n])  Numerically evaluate the Puiseux series.

valuation()  Returns the valuation of this Puiseux series.

as_sympy_expr()  Returns the Puiseux series as a sympy expression.

__terms_from_sympy_const__(expr, order)
Returns terms of constant expression.

__terms_from_sympy_expression__(expr, order=None)
Compute series terms in (x-alpha) of a sympy expression.

**Note:** Sympy’s `series()` function behaves differently depending on whether `n` is specified. If `n` is given then the series is indeed in \((x - a)\) but is written in \(x\) so no shifting is necessary. However, without `n` each term is given as a power of \((x - a)\) so shifting (or alternate parsing) is needed. Very weird.

**Parameters**  
expr : sympy.Expr
order : int

  (Default: 5) The desired order of the series approximation of `expr`.

  Returns tuple of 2-tuples

__terms_from_sympy_generic__(expr, order)
Returns terms from a generic sympy expression.

The slowest method. Uses `sympy.series`.

__terms_from_sympy_polynomial__(expr, order)
Returns terms from polynomial expression.

__terms_from_sympy_rational__(expr, order)
Returns terms from rational expression.

Common and potentially slow situation. This method first separates the numerator and denominator. The denominator is written as

.. math

\[d(x) = cx^l \tilde{d}(x)\]

where \(\tilde{d}(0) = 1\). This is so a fast Taylor series calculation can be done on \(\tilde{d}\). The factor of \(cx^l\) is introduced back into the resulting series.

This approach is similar to the `abelfunctions.differentials.fast_expand()` approach using in localizing `Differential` objects.

as_sympy_expr()
Returns the Puiseux series as a sympy expression.

  Parameters None

  Returns sympy.Expr

eval(x)
Symbolic evaluation of the Puiseux series.

  Parameters x : complex

  Returns complex
evalf$(x, n=8)$
Numerically evaluate the Puiseux series.

Parameters $x$ : complex
$n$ : int
(Default: 8) Number of digits of accuracy.

Returns complex

initialize_terms$(obj, order=None)$
Initialize the terms of the Puiseux series from the object $obj$.

A PuiseuxXSeries can be initialized form a tuple, list, dictionary, or Sympy Expression object. See __init__() for more information.

If $obj$ is a tuple or list each element is a 2-tuple whose first element is the exponent and second element is the coefficient of that term. If $obj$ is a dictionary then the keys are the exponents and the values are the coefficients. If $obj$ is a Sympy Expression then the powers and coefficients are determined using sympy.lseries.

Parameters $obj$ : tuple, list, dict, or Sympify-able expression
$order$ : int, optional

Returns tuple of two-tuples

valuation() 
Returns the valuation of this Puiseux series.

The valuation of a Puiseux x-series is degree of the lowest-order non-zero term. Specifically, if
\[ y(x) = \sum_{h=0}^{\infty} \alpha_h (x - x_0)^{(n_h/e)} \]
then the valuation of $y$ is $n_0/e$.

Parameters None

Returns sympy.Rational

abelfunctions.puiseux.almost_monicize$(f, x, y)$
Transform $f$ to an “almost monic” polynomial.

Perform a sequence of substitutions of the form
\[ f \mapsto x^d f(x, y/x) \]
such that $l(0) \neq 0$ where $l = l(x)$ is the leading order coefficient of $f$.

Parameters $f, x, y$ : sympy.Expr
An algebraic curve in $x$ and $y$.

Returns $g, transform$
A new, almost monic polynomial $g$ and a polynomial transform such that $y \mapsto y/transform$.

abelfunctions.puiseux.bezout$(q, m)$
Returns $u, v$ such that $uq + mv = 1$.

Parameters $q, m$ : integer
Two coprime integers with $q > 0$. 

Returns tuple of integers

\texttt{abelfunctions.puiseux.generalized_polygon_side(side)}

Returns the generalization of a side on the Newton polygon.

A generalized Newton polygon is one where every side has slope no less than -1.

**Parameters** side : sympy.Segment

**Returns** side

\texttt{abelfunctions.puiseux.newton_data(H, x, y, exceptional=False)}

Determines the “newton data” associated with each side of the polygon.

For each side $\Delta$ of the Newton polygon of $H$ we associate the data $(q, m, l, \phi)$ where

$$
\phi(\Delta)(t) = \sum_{(i,j) \in \Delta} a_{ij} t^{(i-i_0)/q}
$$

Here, $a_{ij} x^i y_j$ is a term in the polynomial $H$ and $i_0$ is the smallest value of $i$ belonging to the polygon side $\Delta$.

**Parameters** H : sympy.Poly

Polynomial in $x$ and $y$.

**Returns** list

A list of the tuples $(q, m, l, \phi)$.

\texttt{abelfunctions.puiseux.newton_iteration(G, t, y, n)}

Returns a truncated series $y = y(t)$ satisfying

$$
G(t, y(t)) \equiv 0 \mod t^n
$$

where $n = \text{ceil}\{\log_2{n}\}$). Based on the algorithm in [XXX].

**Parameters** G : sympy.Poly

A polynomial in $t$ and $y$.

n : int

Requested degree of the series expansion.

**Notes**

This algorithm returns the series up to order $2^n > n$. Any choice of order below $2^n$ will return the same series.

\texttt{abelfunctions.puiseux.newton_iteration_step(phi, phiprime, g, s, p, t, y)}

Perform a single step of the newton iteration algorithm.

**Parameters** phi, phiprime : sympy.Poly

Equation and its $y$-derivative.

g, s : sympy.Poly

Current solution and inverse (conjugate) modulo $p$.

p : sympy.Poly

The current modulus. That is, $g$ is the Taylor series solution to $\phi(t, g) = 0$ modulo $p$.

\texttt{t, y} : sympy.Symbol

5.2. Module Reference 39
Dependent and independent variables, respectively.

**Returns**

\( g_{\text{next}}, s_{\text{next}}, p_{\text{next}} \)

`.abelfunctions.puiseux.newton_polygon(H, x, y, additional_points=[])`

Computes the Newton polygon of \( H \).

It’s assumed that the first generator of \( H \) here is the “dependent variable”. For example, if \( H = H(x,y) \) and we are aiming to compute a \( y \)-covering of the complex \( x \)-sphere then each monomial of \( H \) is of the form

\[ a_{ij} x^i y^j. \]

**Parameters**

- \( H \) : sympy.Poly
  Polynomial in \( x \) and \( y \).
- \( x \) : sympy.Symbol
  Dependent variable.
- \( y \) : sympy.Symbol
  Independent variable.

**Returns**

list

Returns a list where each element is a list, representing a side of the polygon, which in turn contains tuples representing the points on the side.

`.abelfunctions.puiseux.newton_polygon_exceptional(H, x, y)`

Computes the exceptional Newton polygon of \( H \).

`.abelfunctions.puiseux.puiseux(f, x, y, alpha, beta=None, t=t, order=None, exact=True, parameter=True)`

Singular parts of the Puiseux series above \( x = \alpha \).

**Parameters**

- \( f \) : sympy.Expr
  A plane algebraic curve in \( x \) and \( y \).
- \( alpha \) : complex
  The \( x \)-point over which to compute the Puiseux series of \( f \).
- \( beta \) : complex
  (Optional) The \( y \)-point at which to compute the Puiseux series.
- \( t \) : sympy.Symbol
  (Optional) Variable used in the Puiseux series expansions.
- \( order \) : int
  (Default: \( None \)) If provided, returns Puiseux series expansions up to the specified order.
- \( exact \) : boolean
  (Default: \( True \)) If False, coerce results into numerical coefficients.

**Returns**

list of PuiseuxTSeries

`.abelfunctions.puiseux.puiseux_rational(H, x, y, recurse=False)`

Puiseux data for the curve \( H \) above \( (x,y) = (0,0) \).

Given a polynomial \( H = H(x,y) \) \( \text{puiseux_rational()} \) returns the singular parts of all of the Puiseux series centered at \( x = 0, y = 0 \).
Parameters \( H \): sympy.Poly

A plane curve in \( x \) and \( y \).

\texttt{recurse} : boolean

(Default: True) A flag used internally to keep track of which term in the singular expansion is being computed.

\textbf{Returns} list of \((G,P,Q)\)

List of tuples where \(P\) and \(Q\) are the \(x\)- and \(y\)-parts of the Puiseux series, respectively, and \(G\) is a polynomial used in \texttt{newton_iteration()} to generate additional terms in the \(y\)-series.

\texttt{abelfunctions.puiseux.transform_newton_polynomial}(H, x, y, q, m, l, xi)

Recenters a Newton polynomial at a given singular term.

Given the Puiseux data \(x = \mu x^q, y = x^m(\beta + y)\) this function returns the polynomial

\[
\tilde{H} = H(\xi^v x^q, x^m(\xi^n + y))/x^l.
\]

where \(uq + mv = 1\).

\section*{5.2.9 Riemann Constant Vector \texttt{abelfunctions.riemann_constant_vector}}

Module for computing the Riemann constant vector \(K : X \to J(X),\)

\[
K(P) = (K_1(P), \ldots, K_g(P))
\]

\[
K_j(P) = \frac{1 + \Omega_{jj}}{2} - \sum_{k \neq j} \int A_j(P, Q) dQ
\]

where \(A = (A_1, \ldots, A_g)\) is the Abel map. (See \texttt{abelfunctions.abelmap.AbelMap Function}.) The Riemann constant vector is an essential ingredient to computing finite genus solutions to integrable systems as well as characterizing the theta divisor of a Riemann surface.

The algorithm for computing the RCV is based on the one described in [DPS]. It relies on observations from two theorems. The first theorem relates the RCV evaluated at a base place, \(K(P_0)\) to the Abel map evaluated at a canonical divisor:

\[
2K(P_0) \equiv -A(P_0, C).
\]

The second theorem relates the RCV to the Riemann theta function: for all effective, degree \(g - 1\) divisors \(D,\)

\[
\theta(K(P_0) + A(P_0, D), \Omega) = 0.
\]

\textbf{Functions}

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{canonical_divisor(X)}</td>
<td>Computes a canonical divisor on (X).</td>
</tr>
<tr>
<td>\texttt{compute_K0(X, epsilon1, epsilon2, C)}</td>
<td>Determine a base value of the Riemann Constant Vector.</td>
</tr>
<tr>
<td>\texttt{find_regular_places(X, n)}</td>
<td>Returns (n) regular places on (X).</td>
</tr>
<tr>
<td>\texttt{half_lattice_filter(half_lattice_vectors, J, ...)}</td>
<td>Filter out any incorrect half-lattice vectors.</td>
</tr>
<tr>
<td>\texttt{half_lattice_vector(X, C, epsilon1, epsilon2)}</td>
<td>Returns an appropriate half-lattice vector for the RCV.</td>
</tr>
<tr>
<td>\texttt{initialize_half_lattice_vectors(X)}</td>
<td>Generate a list of all half-lattice vectors.</td>
</tr>
<tr>
<td>\texttt{RiemannConstantVector(P[, epsilon1, epsilon2, C])}</td>
<td>Evaluate the Riemann constant vector at the place (P).</td>
</tr>
</tbody>
</table>

\section*{5.2. Module Reference}
References

Examples

We evaluate the Riemann constant vector at the base place of the genus four Riemann surface corresponding to the plane algebraic curve $f(x, y) = x^2y^3 - x^4 + 1$.

```python
>>> from sympy.abc import x, y
>>> from abelfunctions import RiemannSurface, RiemannConstantVector

>>> f = x**2*y**3 - x**4 + 1
>>> X = RiemannSurface(f, x, y)
>>> P0 = X.base_place()
>>> RiemannConstantVector(P0)
[ 0.8488+0.7203j -0.5941-0.1146j -0.7432+0.8913j -0.8189+1.1381j]
```

Contents

`abelfunctions.riemann_constant_vector.RiemannConstantVector(P, epsilon1=1e-06, epsilon2=1e-08, C=None)`

Evaluate the Riemann constant vector at the place $P$.

Internally, the value of the RCV at the base place $P_0$ of the Riemann surface containing $P$ is computed and stored. This is done because the value of the RCV at any other place can be later computed using shift with the Abel map.

**Parameters**

- $P$: Place
  - `epsilon1`: double, optional
    - Riemann theta tolerance used in the first pass. Default: $1e-6$.
  - `epsilon2`: double, optional
    - Tolerance used in all subsequent passes. Default: $1e-8$.
  - `C`: Divisor, optional
    - A canonical divisor on the Riemann surface. Computes one using `canonical_divisor()` if no such divisor is provided.

**Returns**

- $K$: array
  - The Riemann constant vector at $P$.

`abelfunctions.riemann_constant_vector.canonical_divisor(X)`

Computes a canonical divisor on $X$.

Selects a canonical divisor on $X$. Tries to select a place resulting in the easiest Abel map to compute by performing the following filters:

- minimize on number of distinct places: (more places results in more paths)
- prefer divisors containing only finite places: (paths to infinity need more testing)

**Parameters**

- $X$: RiemannSurface

**Returns**

- $C$: Divisor
Notes

It takes time to compute the valuation divisors of the Abelian differentials of the first kind, in the first place. There may be a way to rewrite this algorithm so that it picks a “local best” canonical divisor.

\texttt{abelfunctions.riemann\_constant\_vector.compute\_K0}(X, \epsilon_1, \epsilon_2, C)

Determine a base value of the Riemann Constant Vector.

Given a Riemann surface \( R^S \) and a canonical divisor \( C \) compute the Riemann Constant Vector at the base place.

**Parameters**

\( X \) : RiemannSurface

\( \epsilon_1, \epsilon_2 \) : double

Numerical accuracy thresholds. See \texttt{half\_lattice\_filter()}.

\( C \) : Divisor

A canonical divisor on the Riemann surface.

**Returns**

\( K_0 \) : array

The value of the RCV at the base place of the Riemann surface.

\texttt{abelfunctions.riemann\_constant\_vector.find\_regular\_places}(X, n)

Returns \( n \) regular places on \( X \).

This function begins at the x-origin and works “outward” along the real x-axis looking for points sufficiently far away from any discriminant points of the curve. At an appropriate point, the regular places lying above it are computed and added to a list.

We choose integral x-points because it tends to simplify the Puiseux computations.

**Parameters**

\( X \) : RiemannSurface

\( n \) : integer

**Returns**

\( \text{places} \) : list

\texttt{abelfunctions.riemann\_constant\_vector.half\_lattice\_filter}(half\_lattice\_vectors, J, C, D, \epsilon=1e-08)

Filter out any incorrect half-lattice vectors.

This function tests if

\[ h + A(P_0, D) - \frac{1}{2} A(P_0, C) \]

is a member of the theta divisor of the Riemann surface up to numerical accuracy \( \epsilon \) for each vector \( h \) appearing in \texttt{half\_lattice\_vectors}.

**Parameters**

\texttt{half\_lattice\_vectors} : list

A list of remaining half-lattice vectors to consider.

\( J \) : Jacobian

\( C \) : Divisor

A canonical divisor of the surface.

\( D \) : Divisor

An effective, degree g-1 divisor.

\( \epsilon \) : double

The tolerance to use when evaluating the Riemann theta function. Defaults to 1e-8.
Returns `half_lattice_vectors`: list
A filtered list of half-lattice vectors.

```
abelfunctions.riemann_constant_vector.half_lattice_vector(X, C, epsilon1, epsilon2)
```

Returns an appropriate half-lattice vector for the RCV.

**Parameters**
- `X`: RiemannSurface
- `C`: Divisor
  A canonical divisor on the Riemann surface.

**Returns**
- `h`: array

```
abelfunctions.riemann_constant_vector.initialize_half_lattice_vectors(X)
```

Generate a list of all half-lattice vectors.
There are $2^{2g}$ half lattice vectors to consider. This returns a list of all of them. A generator is not necessary due to the filtration process. (We need to check every single half-lattice vector.)

**Parameters**
- `X`: RiemannSurface

**Returns**
- `half_lattice_vectors`: list

```
abelfunctions.riemann_constant_vector.sum_partitions(n)
```

A generator of all length `n` tuples $(m_1, ..., m_n)$ such that

$$m_1 + \cdots + m_n = n,$$

where each $m_i \geq 0$. Used by `half_lattice_vector()` to generate a collection of effective degree $g-1$ divisors.

**Parameters**
- `n`: int

**Returns**
- `p`: generator

### 5.2.10 RiemannSurfaces

**Authors**

- Chris Swierczewski (January 2014)

```python
class abelfunctions.riemann_surface.RiemannSurface
    Bases: object
    
    A Riemann surface defined by a complex plane algebraic curve.
```

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f</code></td>
<td>(sympy.Expression) The algebraic curve representing the Riemann surface.</td>
</tr>
<tr>
<td><code>x,y</code></td>
<td>(sympy.Symbol) The dependent and independent variables, respectively.</td>
</tr>
</tbody>
</table>

```
based_lift()
```

Same as `base_sheets()`.

```
based_place()
```

Returns the base place of the Riemann surface.

The base place is the place from which all paths on the Riemann surface are constructed. The AbelMap begins integrating from the base place.
Abelfunctions Documentation, Release 0.1.0

Parameters None

Returns Place

**base_point()**
Returns the base x-point of the Riemann surface.

**base_sheets()**
Returns the base sheets of the Riemann surface.

The base sheets are the y-roots lying above the base point of the surface. The base place of the Riemann surface is given by the base x-point and the first element of the base sheets.

Parameters None

Returns list, complex

An ordered list of roots lying above the base point of the curve.

**closest_discriminant_point()**
Returns the closest discriminant point to a point x.

An often-used helper function by several components of `RiemannSurface`.

Parameters x : complex

A complex x-point.

exact : boolean

If True, returns a sympy.Expr representing the discriminant point exactly. Otherwise, returns a numerical approximation.

Returns complex or sympy.Expr

The discriminant point, either exact or numerical.

**discriminant_points()**
Returns the discriminant points of the underlying curve.

A discriminant point x = b is an x-point where at least one y-root lying above has multiplicity greater than one. A PuiseuxTSeries is required to represent a place on the Riemann surface whose x-projection is a discriminant point. These kinds of places are of type `DiscriminantPlace`.

**Note:** The ordering of the discriminant points is important for the purposes of computing the monodromy group, which is done in the `RiemannSurfacePathFactory` attribute, `PathFactory`.

Parameters exact : boolean

If True, returns symbolic discriminant points. Otherwise, returns a numerical approximation. Both are cached for performance.

Returns list

A list of the discriminant points of the underlying curve.

**holomorphic_differentials()**
Returns a basis of holomorphic differentials on the surface.

Parameters None

Returns list, HolomorphicDifferential

**holomorphic_oneforms()**
Alias for `holomorphic_differentials()`.
**integrate()**
Integrate the differential \( \omega \) over the path \( \gamma \).

**Parameters**
- \( \omega \): Differential
  A differential defined on the Riemann surface.
- \( \gamma \): RiemannSurfacePathPrimitive
  A continuous path on the Riemann surface.

**Returns**
- complex
  The integral of \( \omega \) on \( \gamma \).

**lift()**
List the x-point \( x \) to the fibre of y-roots.

Basically, computes the y-roots of \( f(x, y) = 0 \) for the given \( x \).

**Note:** The y-roots are given in no particular order. Be careful when using these to construct RiemannSurfacePath objects.

**Parameters**
- \( x \): complex

**Returns**
- list, complex

**monodromy_group()**
Returns the monodromy group of the underlying curve.

The monodromy group is represented by a list of four items:
- *base_point* - a point in the complex x-plane where every monodromy path begins and ends,
- *base_sheets* - the y-roots of the curve lying above *base_point*,
- *branch_points* - the branch points of the curve,
- *permutations* - the permutations of the base sheets corresponding to each branch point.

**path()**
Constructs a path to the place \( P \).

**Parameters**
- \( P \): Place

**Returns**
- RiemannSurfacePath

**period_matrix()**
Returns the period matrix of the Riemann surface.

The period matrix is obtained by integrating a basis of holomorphic one-forms over a first homology group basis.

**Parameters**
- None

**Returns**
- numpy.array
  A \( g \times 2g \) complex matrix of periods.

**riemann_matrix()**
Returns the Riemann matrix of the Riemann surface.
A Riemann matrix of the surface is obtained by normalizing the chosen basis of holomorphic differentials.

\[ \tau = [A \ B] = [I \ \Omega] \]

**Parameters** None  
**Returns** numpy.array  
A \( g \times g \) Riemann matrix corresponding to the Riemann surface.

**show_paths()**  
Plots all of the monodromy paths of the curve.  
**Parameters** ax : matplotlib.Axes  
The figure axes on which to plot the paths.  
**Returns** None

### 5.2.11 Riemann Surface Paths abelfunctions.riemann_surface_path

A framework for computing places along paths on Riemann surfaces.

The classes in this module follow the composite design pattern [R1] with RiemannSurfacePathPrimitive acting as the “component” and RiemannSurfacePath acting as the “composite”. The classes RiemannSurfacePathLine and RiemannSurfacePathArc define particular types of paths.

**Classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RiemannSurfacePathPrimitive</td>
<td>Primitive Riemann surface path object.</td>
</tr>
<tr>
<td>RiemannSurfacePath</td>
<td>A continuous, piecewise differentiable path on a Riemann surface.</td>
</tr>
<tr>
<td>RiemannSurfacePathLine</td>
<td>A Riemann surface path for which the x-part of the path is a line segment.</td>
</tr>
<tr>
<td>RiemannSurfacePathArc</td>
<td>A Riemann surface path for which the x-part of the path is an arc.</td>
</tr>
</tbody>
</table>

**References**

**Examples**

**Contents**

**class abelfunctions.riemann_surface_path.RiemannSurfacePath**

**Bases:** abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive

A continuous, piecewise differentiable path on a Riemann surface.

RiemannSurfacePath is a composite of RiemannSurfacePathPrimitive objects. This path is parameterized for \( t \in [0, 1] \).

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_x</td>
<td>Return the x-part of the path at ( t \in [0, 1] ).</td>
</tr>
<tr>
<td>get_dxdt</td>
<td>Return the derivative of the x-part of the path at ( t \in [0, 1] ).</td>
</tr>
<tr>
<td>get_y</td>
<td>Return the y-fibre of the path at ( t \in [0, 1] ).</td>
</tr>
</tbody>
</table>
**evaluate()**
Evaluates \(\omega\) along the path at \(N\) uniform points.

**Note:** Right now it doesn’t matter what the values in \(t\) are. This function will simply turn \(t\) into a bunch of uniformly distributed points between 0 and 1.

**Parameters**
- \(\omega\) : Differential
- \(t\) : double[:,:]
  - An array of \(t\) between 0 and 1.

**Returns**
- complex[:,:]
  - The differential \(\omega\) evaluated along the path at \(N\) points.

**get_dxdt()**
Return the derivative of the x-part of the path at \(t \in [0, 1]\).

**Parameters**
- \(t\) : double

**Returns**
- complex

**Notes**
This RiemannSurfacePath is parameterized for \(t \in [0, 1]\). However, internally, each segment is separately parameterized for \(t \in [0, 1]\). This routine performs an appropriate scaling.

**Warning:** Riemann surface paths are only piecewise differentiable and therefore may have discontinuous derivatives at the boundaries. Therefore, it may be more useful to perform segment-wise operations instead of operations on the whole of this object.

**get_x()**
Return the x-part of the path at \(t \in [0, 1]\).

**Parameters**
- \(t\) : double

**Returns**
- complex

**Notes**
This RiemannSurfacePath is parameterized for \(t \in [0, 1]\). However, internally, each segment is separately parameterized for \(t \in [0, 1]\). This routine performs an appropriate scaling.

**get_y()**
Return the y-fibre of the path at \(t \in [0, 1]\).

**Parameters**
- \(t\) : double

**Returns**
- complex[:,:]

**Notes**
This RiemannSurfacePath is parameterized for \(t \in [0, 1]\). However, internally, each segment is separately parameterized for \(t \in [0, 1]\). This routine performs an appropriate scaling.
integrate()  
Integrate \( \omega \) on the path using its analytic continuator.

Delegates integration to the path's AnalyticContinuator. The strategy for analytic continuation depends on if this path terminates at a discriminant point of the curve.

Parameters \( \omega \) : Differential

Returns complex

class abelfunctions.riemann_surface_path.RiemannSurfacePathArc
Bases: abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive

A Riemann surface path for which the x-part of the path is an arc.

Attributes

<table>
<thead>
<tr>
<th>R</th>
<th>(complex) The radius of the semicircle. (Complex type for coercion performance.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>(complex) The center of the semicircle.</td>
</tr>
<tr>
<td>theta</td>
<td>(complex) The starting angle (in radians) on the semicircle. Usually 0 or ( \pi ). (Complex type for coercion performance.)</td>
</tr>
<tr>
<td>dtheta</td>
<td>(complex) The number of radians to travel where the sign of ( d\theta ) indicates direction. The absolute value of ( d\theta ) is equal to the arc length.</td>
</tr>
</tbody>
</table>

class abelfunctions.riemann_surface_path.RiemannSurfacePathLine
Bases: abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive

A Riemann surface path for which the x-part of the path is a line segment.

Attributes

| z0, z1 | (complex) The starting and ending points of the complex x-line. |

class abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive
Bases: object

Primitive Riemann surface path object.

Defines basic, primitive functionality for Riemann surface paths. Each path primitive is parameterized from \( t = 0 \) to \( t = 1 \).

Attributes

| segments | The individual RiemannSurfacePathPrimitive objects that make up this object. |

| RS       | (RiemannSurface) The Riemann surface on which this path primitive is defined. |
| AC       | (AnalyticContinuator) The mechanism this path uses for performing analytic continuation. An appropriate choice of analytic continuator, in part, depends on the proximity of the path to a singular point or branch point of the curve. |
| x0       | (complex) Starting x-value of the path. |
| y0       | (complex[]) Starting y-fibre of the path. |
**get_x**
Return the x-part of the path at \( t \in [0, 1] \).

**get_dxdt**
Return the derivative of the x-part of the path at \( t \in [0, 1] \).

**analytically_continue**
Analytically continue the fibre \( y_i \) from \( x_i \) to \( x_{i+1} \).

**get_y**
Return the y-fibre of the path at \( t \in [0, 1] \).

**plot_x**
Plot the x-part of the path in the complex x-plane.

**plot_y**
Plot the y-part of the path in the complex y-plane.

**plot3d_x**
Plot the x-part of the path in the complex x-plane with the parameter \( t \in [0, 1] \) along the perpendicular axis.

**plot3d_y**
Plot the y-part of the path in the complex y-plane with the parameter \( t \in [0, 1] \) along the perpendicular axis.

---

**_initialize_checkpoints()**
Analytically continue along the entire path recording y-values at evenly spaced points. We cache the y-values at various evenly-spaced points \( t \in [0, 1] \) so one doesn’t have to analytically continue from \( t = 0 \) every time.

- **Parameters** None
- **Returns** None

**_set_str_noncycle()**
Generates the string representation of self in the case when self is not a cycle.

**analytically_continue()**
Analytically continue the fibre \( y_i \) from \( x_i \) to \( x_{i+1} \).

- **Parameters**
  - \( xi \): complex
  - \( yi \): complex[]
    The current x,y-fibre pair.
  - \( x_{i+1} \): complex
    The target complex x-point.
- **Returns** complex[
  The fibre above \( x_{i+1} \).

**evaluate()**
Evaluates \( \omega \) along the path at \( t \) between 0 and 1.

- **Parameters**
  - \( \omega \): Differential
  - \( t \): double[]
    An array of \( t \) between 0 and 1.
- **Returns** complex[
  The differential \( \omega \) evaluated along the path at each of the points in \( t \).

**get_dxdt()**
Return the derivative of the x-part of the path at \( t \in [0, 1] \).

- **Parameters**
  - \( t \): double
- **Returns** complex
get_x()  
Return the x-part of the path at $t \in [0, 1]$.

Parameters t : double

Returns complex

get_y()  
Return the y-fibre of the path at $t \in [0, 1]$.

Parameters t : double

Returns complex[:]

integrate()  
Integrate $\omega$ on the path using its analytic continuator.

Delegates integration to the path’s AnalyticContinuator. The strategy for analytic continuation depends on if this path terminates at a discriminant point of the curve.

Parameters omega : Differential

Returns complex

plot3d_x()  
Plot the x-part of the path in the complex x-plane with the parameter $t \in [0, 1]$ along the perpendicular axis.

Additional arguments and keywords are passed to matplotlib.pyplot.plot.

Parameters N : int

The number of interpolating points used to plot.

t0 : double

Starting t-value in [0,1].

t1 : double

Ending t-value in [0,1].

Returns matplotlib lines array.

plot3d_y()  
Plot the y-part of the path in the complex y-plane with the parameter $t \in [0, 1]$ along the perpendicular axis.

Additional arguments and keywords are passed to matplotlib.pyplot.plot.

Parameters N : int

The number of interpolating points used to plot.

t0 : double

Starting t-value in [0,1].

t1 : double

Ending t-value in [0,1].

Returns matplotlib lines array.

plot_x()  
Plot the x-part of the path in the complex x-plane.

Additional arguments and keywords are passed to matplotlib.pyplot.plot.
Parameters

N : int
The number of interpolating points used to plot.

t0 : double
Starting t-value in [0,1].

t1 : double
Ending t-value in [0,1].

Returns matplotlib lines array.

plot_y()
Plot the y-part of the path in the complex y-plane.

Additional arguments and keywords are passed to matplotlib.pyplot.plot.

Parameters

N : int
The number of interpolating points used to plot.

t0 : double
Starting t-value in [0,1].

t1 : double
Ending t-value in [0,1].

Returns matplotlib lines array.

segments
The individual RiemannSurfacePathPrimitive objects that make up this object.

Every RiemannSurfacePathPrimitive object contains a list of “path segments”. In the case when this list of length one the list contains only the object itself.

When the number of path segments is greater than one, the object should be coerced to a RiemannSurfacePath object. Each element is a primitive representing an arc or straight line path in the complex x-plane.

set_analytic_continuator()
Select and appropriate analytic continuator for this path.

If either the starting or ending place is at a discriminant point of the underlying curve then use an analytic continuation method that can distinguish between places lying above discriminant points. Otherwise, use a fast numerical method.

Parameters none

Returns AnalyticContinuator

An AnalyticContinuator which defines how to analytically continue y-roots along a path.

Notes

Currently, this method will only look at the endpoint of the path to determine if it is epsilon close to a discriminant point of the curve. (Meaning that the places lying above the point can only be distinguished by Puiseux series.) A future update should determine if the path crosses through or passes near enough to the discriminant point.
class \texttt{abelfunctions.riemann\_surface\_path.RiemannSurfacePathRay}

\textbf{Bases:} \texttt{abelfunctions.riemann\_surface\_path.RiemannSurfacePathPrimitive}

A Riemann surface path for which the \textit{x}-part goes to infinity.

Given a starting point $x_0$ the \textit{x}-path $\gamma_x : [0, 1] \to \mathbb{C}_x$ going to infinity is the one that travels radially outward from the origin $x = 0$ given by the equation

$$\gamma_x(t) = \frac{x_0}{1 - t}$$

### 5.2.12 Riemann Surface Path Factory

This module implements the \texttt{RiemannSurfacePathFactory} class, a class for generating \texttt{RiemannSurfacePath} objects from various kinds of input data.

#### Classes

\texttt{RiemannSurfacePathFactory}

#### Functions

#### Authors

- Chris Swierczewski (January 2013)

class \texttt{abelfunctions.riemann\_surface\_path\_factory.RiemannSurfacePathFactory}(\texttt{RS}, \texttt{kappa=0.4})

\textbf{Bases:} \texttt{object}

Factory class for constructing paths on the Riemann surface.

##### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{RS}</td>
<td>(RiemannSurface) The Riemann surface.</td>
</tr>
<tr>
<td>\texttt{XPF}</td>
<td>(XPathFactory) A factory object for determining how to navigate the complex x-plane. How to construct paths avoiding, encircling, and approaching discriminant points.</td>
</tr>
<tr>
<td>\texttt{YPF}</td>
<td>(YPathFactory) A factory object for determining how to navigate the complex y-plane. Computes the branching structure and homology of the surface. Defines how to swap sheets.</td>
</tr>
<tr>
<td>_base_sheets</td>
<td>(list) The ordered sheets of the surface at the base point.</td>
</tr>
<tr>
<td>_monodromy_group</td>
<td>(list) The monodromy group of the curve. Used to compute the homology.</td>
</tr>
</tbody>
</table>

##### Methods

- \texttt{discriminant\_points\([\text{exact}]\)} \hspace{1cm} \textit{Same as \texttt{RiemannSurface.discriminant\_points}().}
- \texttt{closest\_discriminant\_point}(x[, exact]) \hspace{1cm} \textit{Same as \texttt{RiemannSurface.closest\_discriminant\_point}().}
- \texttt{path\_to\_place}(P) \hspace{1cm} Returns a path to a specified place \textit{P} on the Riemann surface.
- \texttt{monodromy\_group}() \hspace{1cm} Returns the monodromy group of the algebraic curve defining the Riemann surface.
- \texttt{monodromy\_path}(bi) \hspace{1cm} Returns the monodromy path around the discriminant point \textit{bi}.  

5.2. Module Reference
### Table 5.23 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>a_cycles()</em></td>
<td>Returns the a-cycles on the Riemann surface.</td>
</tr>
<tr>
<td><em>b_cycles()</em></td>
<td>Returns the b-cycles on the Riemann surface.</td>
</tr>
<tr>
<td><em>c_cycles()</em></td>
<td>Returns the c-cycles of the Riemann surface and the linear combination</td>
</tr>
<tr>
<td><em>show_paths([ax])</em></td>
<td>Plots all of the monodromy paths of the curve.</td>
</tr>
<tr>
<td><em>RiemannSurfacePath_from_cycle(cycle)</em></td>
<td>Constructs a RiemannSurfacePath object from a list of x-path data.</td>
</tr>
<tr>
<td><em>RiemannSurfacePath_from_xpath(xpath[, x0, y0])</em></td>
<td>Constructs a RiemannSurfacePath object from x-path data.</td>
</tr>
<tr>
<td><em>_path_to_discriminant_place(P)</em></td>
<td>Returns a path to a discriminant place on the surface.</td>
</tr>
<tr>
<td><em>_path_to_regular_place(P)</em></td>
<td>Returns a path to a regular place on the surface.</td>
</tr>
</tbody>
</table>

#### RiemannSurfacePath_from_cycle(cycle)
Constructs a RiemannSurfacePath object from a list of x-path data.

**Parameters**

- **xpath** : list
  A list of tuples defining the x-path.

#### RiemannSurfacePath_from_xpath(xpath[, x0=None, y0=None])
Constructs a RiemannSurfacePath object from x-path data.

**Parameters**

- **xpath** : list
  A list of tuples defining the xpath.
- **x0** : complex (default *self.base_point*())
  The starting x-point of the path.
- **y0** : complex list (default *self.base_sheets*())
  The starting ordering of the y-sheets.

**Returns**

A path on the Riemann surface with the prescribed x-path.

#### _path_near_discriminant_point(P)
Returns a path to a place *P* where the x-coordinate is a discriminant point on the surface.

#### _path_to_discriminant_place(P)
Returns a path to a discriminant place on the surface.

A “discriminant” place *P* is a place on the Riemann surface where Puiseux series are required to determine the x- and y-projections of the place.

**Parameters**

- **P** : Place

**Returns**

RiemannSurfacePath

#### _path_to_infinite_place(P)
Returns a path to a place at an infinity of the surface.

A place at infinity is one where the x-projection of the place is the point *x = infty* of the complex Riemann sphere. An infinite place is a type of discriminant place.

**Parameters**

- **P** : Place

**Returns**

RiemannSurfacePath
_path_to_regular_place(\(P\))

Returns a path to a regular place on the surface.

A “regular” place \(P\) is a place on the Riemann surface where the x-projection of the place is not a discriminant point of the curve.

**Parameters**  
\(P\) : Place

**Returns**  
\(\text{RiemannSurfacePath}\)

\(a\_cycles()\)

Returns the a-cycles on the Riemann surface.

**Returns**  
\(\text{list}, \text{RiemannSurfacePath}\)

A list of the a-cycles of the Riemann surface as \(\text{RiemannSurfacePath}\) objects.

\(b\_cycles()\)

Returns the b-cycles on the Riemann surface.

**Returns**  
\(\text{list}, \text{RiemannSurfacePath}\)

A list of the b-cycles of the Riemann surface as \(\text{RiemannSurfacePath}\) objects.

\(c\_cycles()\)

Returns the c-cycles of the Riemann surface and the linear combination matrix defining the a- and b-cycles from the c-cycles.

The a- and b- cycles of the Riemann surface are formed from linear combinations of the c-cycles. These linear combinations are obtained from the :py:linear_combinations() method.

**Note:** It may be computationally more efficient to integrate over the (necessary) c-cycles and take linear combinations of the results than to integrate over the a- and b-cycles separately. Sometimes the column rank of the linear combination matrix (that is, the number of c-cycles used to construct a- and b-cycles) is lower than the size of the homology group and sometimes the c-cycles are simpler and shorter than the homology cycles.

**Returns**  
\(\text{list}, \text{RiemannSurfacePath}\)

A list of the c-cycles of the Riemann surface as \(\text{RiemannSurfacePath}\) objects.

\(closest\_discriminant\_point(x, exact=True)\)

Same as \(\text{RiemannSurface.closest_discriminant_point()}\).

\(discriminant\_points(exact=False)\)

Same as \(\text{RiemannSurface.discriminant_points()}\).

\(monodromy\_group()\)

Returns the monodromy group of the algebraic curve defining the Riemann surface.

**Returns**  
\(\text{list}\)

A list containing the base point, base sheets,

\(monodromy\_path(bi)\)

Returns the monodromy path around the discriminant point \(bi\).

**Parameters**  
\(bi\) : complex

A discriminant point of the curve.

\(nrots\) : optional, integer

The number of times to rotate around \(bi\). (Default 1).
Returns RiemannSurfacePath
The path encircling the branch point $b_i$.

path_to_place ($P$)
Returns a path to a specified place $P$ on the Riemann surface.

Parameters $P$ : complex tuple
A place on the Riemann surface.

show_paths ($ax=None$, *args, **kwds)
Plots all of the monodromy paths of the curve.

Parameters $ax$ : matplotlib.Axes
The figure axes on which to plot the paths.

Returns None

5.2.13 Riemann Theta

The `abelfunctions.riemann_theta` module contains several submodules for computing the Riemann theta function. The primary submodule is `abelfunctions.riemann_theta.riemann_theta`, containing the function RiemannTheta itself.

**Riemann Theta Function `abelfunctions.riemann_theta.riemann_theta`**

The primary module for computing the Riemann theta function.

$$\theta(z, \Omega) = \sum_{n \in \mathbb{Z}} e^{2\pi i \left( \frac{1}{2} n \cdot \Omega n + n \cdot z \right)}$$

**Classes**

RiemannThetaFunction The Riemann theta function.

**Functions**

oscillatory_part Compute the oscillatory part of the Riemann theta function.

**References**

**Examples**

**Contents**

class abelfunctions.riemann_theta.riemann_theta.RiemannThetaFunction
Bases: object

The Riemann theta function.

This class is globally instantiated as RiemannTheta.
Attributes

| none |

Methods

| exponential_part | Returns the exponential part of the Riemann theta function. |
| oscillatory_part | Compute the oscillatory part of the Riemann theta function. |
| oscillatory_part_gradient | Returns the oscillatory part of the gradient of Riemann theta. |
| gradient | Returns the gradient of Riemann theta. |
| oscillatory_part_hessian | Returns the oscillatory part of the Hessian of Riemann theta. |
| eval | Returns the value of the Riemann theta function at \( z \) and \( \Omega \). |

**eval()**

Returns the value of the Riemann theta function at \( z \) and \( \Omega \).

In many applications it’s preferred to use \texttt{exponential_part()} and \texttt{oscillatory_part()} due to the double-exponential growth of theta in the directions of the columns of \( \Omega \).

**Parameters**

\begin{itemize}
  \item \texttt{z : complex[:]}  
    A complex row-vector or list of row-vectors at which to evaluate the Riemann theta function.
  \item \texttt{Omega : complex[:,:,:]}  
    A Riemann matrix.
  \item **kwds : keywords**  
    See \texttt{exponential_part()} and \texttt{oscillatory_part()} for optional keywords.
\end{itemize}

**Returns**

array

The value of the Riemann theta function at each \( g \)-component vector appearing in \( z \).

**exponential_part()**

Returns the exponential part of the Riemann theta function.

This function is “vectorized” over \( z \). By default, each row of \( z \) is interpreted as a separate input vector to the Riemann theta function.

**Parameters**

\begin{itemize}
  \item \texttt{z : complex[:]}  
    A complex row-vector or list of row-vectors at which to evaluate the Riemann theta function.
  \item \texttt{Omega : complex[:,:,:]}  
    A Riemann matrix.
  \item \texttt{axis : int}  
    (Default: \texttt{1}) If multiple \( z \)-vectors are given in the form of a two-dimensional array, specify over which axis to compute the Riemann theta function. By default, each row of \( z \) is interpreted as an input vector.
\end{itemize}

**Returns**

array
The value of the exponential part of the Riemann theta function at each $g$-component vector appearing in $z$.

**gradient** ()

Returns the gradient of Riemann theta.

The gradient of $\theta((z_1, \ldots, z_g), \Omega)$ is defined to be

$$\theta(z, \Omega) = (\ldots, \partial_{z_i} \theta(z, \Omega), \ldots)$$

where $\partial_{z_i}$ denotes the directional derivative in the $[\ldots, 0, 1, 0, \ldots]$ direction. (The “1” occurs in the $i$th position.)

**Parameters** (see :meth:`oscillatory_part`)

**Returns** array

If a single $z$-argument is given then returns a 1-dimensional Numpy array representing the gradient. If multiple $z$-arguments are given then returns a 2-dimensional Numpy array of gradients where each gradient is listed along $axis$.

**See also:**

*oscillatory_part_gradient*

**Notes**

This could be made shorter by using `eval()` but we choose not to for performance reasons. (We avoid having to compute the exponential part multiple times.)

**oscillatory_part** ()

Compute the oscillatory part of the Riemann theta function.

The oscillatory part of the Riemann theta function is the infinite summation left over after factoring out the double-exponential growth.

This function is “vectorized” over $z$ in order to take advantage of the uniform approximation theorem.

**Parameters**

- $z$ : complex[:]
  A complex row-vector or list of row-vectors at which to evaluate the Riemann theta function.

- $\Omega$ : complex[::]
  A Riemann matrix.

- $\epsilon$ : double
  (Default: $1e-8$) The desired numerical accuracy.

- derivs : list of lists
  (Default: []) A directional derivative given as a list of lists.

- $\text{accuracy\_radius}$ : double
  (Default: 5.) The radius from the $g$-dimensional origin where the requested accuracy of the Riemann theta is guaranteed when computing derivatives. Not used if no derivatives of theta are requested.

- $\text{axis}$ : int
(Default: \( I \)) If multiple \( z \)-vectors are given in the form of a two-dimensional array, specify over which axis to compute the Riemann theta function. By default, each row of \( z \) is interpreted as an input vector.

**Returns** array

The value of the Riemann theta function at each \( g \)-component vector appearing in \( z \).

**oscillatory_part_gradient()**

Returns the oscillatory part of the gradient of Riemann theta.

A helper function for \( \text{gradient()} \). Useful in its own right for detecting if the gradient vanishes since the exponential part of Riemann theta never does as well as for controlling exponential growth in rational functions of theta.

**Parameters** (see :meth:`oscillatory_part`)

**Returns** array

If a single \( z \)-argument is given then returns a 1-dimensional Numpy array representing the gradient. If multiple \( z \)-arguments are given then returns a 2-dimensional Numpy array of gradients where each gradient is listed along \( axis \).

**oscillatory_part_hessian()**

Returns the oscillatory part of the Hessian of Riemann theta.

A helper function for \( \text{hessian()} \). Useful in its own right for detecting if the Hessian vanishes since the exponential part of Riemann theta never does as well as for controlling exponential growth in rational functions of theta.

**Parameters** (see :meth:`oscillatory_part`)

**Returns** array

If a single \( z \)-argument is given then returns a 2-dimensional Numpy array representing the Hessian. If multiple \( z \)-arguments are given then returns a 3-dimensional Numpy array of Hessians where each Hessian is indexed by the 0th coordinate.

**5.2.14 Singularities abelfunctions.singularities**

A module for computing the singular points of a complex plane algebraic curve including their multiplicities, branching numbers, multiplicities, and delta invariants.

Each singularity \( P = (x, y, z) \) on the projectivization of the curve has associated with it a three-tuple \((m, \delta, r)\) representing the multiplicity, delta invariant, and branching number of the multiplicity.

**Functions**

**singularities**

See RiemannThetaFunction.oscillatory_part() for information on the arguments.
Examples

Contents

abelfunctions.singularities._branching_number(P)
Computes the branching number of the singularity at \( u_0, v_0 \).

The branching number is simply the number of distinct branches (i.e. non-interacting branches) at the place. In parametric form, this is simply the number of Puiseux series at the place.

Parameters P : list
A list of PuiseuxTSeries of \( g = g(u, v) \) centered at some \((u_0, v_0)\) where \( g \) is a complex affine algebraic curve.

Returns int
The branching number of the singularity \((u_0, v_0)\).

abelfunctions.singularities._delta_invariant(P)
Computes the delta invariant of the singularity at \( u_0, v_0 \).

Parameters P : list
A list of PuiseuxTSeries of \( g = g(u, v) \) centered at some \((u_0, v_0)\) where \( g \) is a complex affine algebraic curve.

Returns int
The delta invariant of the singularity \((u_0, v_0)\).

abelfunctions.singularities._multiplicity(P)
Computes the multiplicity of the singularity at \( u_0, v_0 \).

The singularity is given on an affine plane along with the curve recentered at the point.

For each (parametric) Puiseux series

\[ P_j = \{ x = x(t), y = y(t) \} \]

at \((1 : \alpha : \beta)\) the contribution from : math : \( P_j \) to the multiplicity is the minimum of the degree of \( x \) and the degree of \( y \).

Parameters P : list
A list of PuiseuxTSeries of \( g = g(u, v) \) centered at some \((u_0, v_0)\) where \( g \) is a complex affine algebraic curve.

Returns int
The multiplicity of the singularity \((u_0, v_0)\).

abelfunctions.singularities._transform(f, x, y, z, singular_pt)
Recents the affine curve \( f \) at a singular point.

Returns \((g, u, v, u0, v0)\) where \( g = g(u, v) \) is the transformed polynomial and \( u0, v0 \) is the projection of \([\alpha : \beta : \gamma]\) on the appropriate affine plane.

If the singular point \([\gamma : \alpha : \beta]\) is on the line at infinity (i.e. \( \gamma = 0 \)) then make the appropriate transformation to the curve \( f(x, y) = 0 \) so we can compute Puiseux series at the point.

Parameters f : sympy.Expr
  x : sympy.Symbol
  y : sympy.Symbol
z : sympy.Symbol

singular_pt : list

A projective singular point of the curve.

Returns sympy.Expr, sympy.Symbol, sympy.Symbol, complex, complex

Returns the transformed curve, $g(u, v)$ along with the variable $u, v$ and the centers $u_0, v_0$.

Examples

For example, let $F(x, y, z) = 0$ be the homogenized curve. If $beta \neq 0$ then the transformation is

$$g(u, v) = F(u, beta, v), \quad u0 = \alpha, \quad v0 = \gamma.$$
This information is used to resolve singularities for the purposes of computing a Riemann surface. The singularities are resolved using Puiseux series.

**Parameters**
- \( f \) : sympy.Expr
- \( x \) : sympy.Symbol
- \( y \) : sympy.Symbol

**Returns**
- list
  - A list of the singularities, both finite and infinite, along with their multiplicity, delta invariant, and branching number information.

### 5.2.15 Utilities

Various utilities and tools used by parts of the Abelfunctions library.

**Cache**

**Cache**

**abelfunctions.utilities.cache**

Module defining cached function decorators.

The decorator `cached_function()` works with instance methods as well. It relies on the decorator module for forwarding function signatures and documentation. (Results in poorer caching performance than some other decorator designs but doesn’t break the reference documentation.)

#### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cached_function(obj)</code></td>
<td>Decorator for argument and keyword caching.</td>
</tr>
<tr>
<td><code>cached_function_fast(obj)</code></td>
<td>Decorator for argument and keyword caching.</td>
</tr>
<tr>
<td><code>cached_method(obj)</code></td>
<td>Decorator for argument and keyword caching.</td>
</tr>
</tbody>
</table>

#### Examples

**Contents**

**abelfunctions.utilities.cache.cached_function(obj)**

Decorator for argument and keyword caching.

This memoizing decorator caches over arguments as well as keywords. Including keywords does come at a large performance cost but is completely general.

The use of `decorator.decorator` ensures that function signatures, documentation, etc. pass along to the decorated function.

**Parameters**
- \( obj \) : function object

**Returns**
- `decorated` : function or instancemethod

**abelfunctions.utilities.cache.cached_function_fast(obj)**

Decorator for argument and keyword caching.

This memoizing decorator caches over arguments as well as keywords. Including keywords does come at a large performance cost but is completely general.
The use of `decorator.decorator` ensures that function signatures, documentation, etc. pass along to the decorated function.

**Parameters** `obj` : function object

**Returns** `decorated` : function or instancemethod

```python
abelfunctions.utilities.cache.cached_method(obj)
```

Decorator for argument and keyword caching.

This memoizing decorator caches over arguments as well as keywords. Including keywords does come at a large performance cost but is completely general.

The use of `decorator.decorator` ensures that function signatures, documentation, etc. pass along to the decorated function.

**Parameters** `obj` : function object

**Returns** `decorated` : function or instancemethod

## Permutations

A simple implementation of permutations on \( n \) elements.

### Authors

- Chris Swierczewski (Feb 2014)

```python
class abelfunctions.utilities.permutations.Permutation(l)
    Bases: object

    A permutation on \( n \) elements.
```

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_identity()</code></td>
<td>Returns <code>True</code> if the Permutation is the identity.</td>
</tr>
<tr>
<td><code>index(j)</code></td>
<td>Representing the Permutation in “map” notation, a list where ( i ) is mapped to ( j = lst[i] ), returns ( i ). That is, the preimage of ( j ).</td>
</tr>
<tr>
<td><code>action(a)</code></td>
<td>Returns the permutation of an iterable ( a ) under the action of the permutation.</td>
</tr>
<tr>
<td><code>inverse()</code></td>
<td>Returns the inverse of the Permutation.</td>
</tr>
</tbody>
</table>

```python
_p._cycles_from_list(l)  
Create a list of cycles from a permutation list.
```

```python
_p._list_from_cycles(cycles) 
Create a permutation list \( i o l[i] \) from a cycle notation list.
```

### Examples

```python
>>> p = Permutation([[0, 1], [2], [3]])
>>> p._list
[1, 0, 2]
```

```python
>>> q = Permutation([[2, 4], [1, 3]])
>>> q._list
[2, 3, 0, 1]
```
**action**($a$)  
Returns the action of the permutation on an iterable.

**Examples**

```python
>>> p = Permutation([0,3,1,2])
>>> p.action(['a','b','c','d'])
['a', 'd', 'b', 'c']
```

**index**($j$)  
If $p(i) = j$, returns $i$.

**inverse**()  
Returns the inverse permutation.

**is_identity**()  
Returns True if permutation is the identity.

**abelfunctions.utilities.permutations.matching_permutation**($a$, $b$)  
Returns the permutation $p$ mapping the elements of $a$ to the elements of $b$.

This function returns a Permutation $p$ such that $b \sim p\text{.action}(a)$ or, equivalently, $\text{norm}(b - p\text{.action}(a))$ is small. The elements of $a$ and $b$ need not be exactly the same but close enough to each other that it’s unambiguous which elements match.

**Parameters**  
$a$, $b$ : iterable  
Lists of approximately the same elements.

**Returns**  
Permutation  
A Permutation $p$ such that $\text{norm}(b - p\text{.action}(a))$ is small.

**Examples**

If the two lists contain the same elements then matching_permutation simply returns permutation defining the rearrangement.

```python
>>> a = [6, -5, 9]
>>> b = [9, 6, -5]
>>> p = matching_permutation(a,b); p
[2, 0, 1]
```

The function matching_permutation will attempt to find such a permutation even if the elements of the two lists are not exactly the same.

```python
>>> a = [1.1, 7.2, -3.9]
>>> b = [-4, 1, 7]
>>> p = matching_permutation(a,b); p
[2, 0, 1]
>>> p.action(a)
[-3.9, 1.1, 7.2]
```

**RootOf Tools abelfunctions.utilities.rootoftools**

Various tools (on top of what Sympy provides) for dealing with RootOf objects.
Sympy’s `RootOf` construct still lacks some features. Until the functionality in this module is incorporated into Sympy itself it will be defined here. (Since it’s very necessary for the computations in this software.) The primary motivation behind writing this functionality come from:

- Sympy not preserving the `radicals=False` specification in `RootOf` construction.
- The lack of a simplification function: expressions involving powers of `RootOf` can be rewritten such that all powers larger than the degree of the defining polynomial can be rewritten in terms of lower order terms. This is useful for managing expression swell.

Functions

```python
def all_roots(f, gen, multiple=True, radicals=False, tol=1e-15):
    """Return a list of real and complex roots with multiplicities."

    Modified from `sympy.all_roots()` to accept polynomials with a `RootOf` type appearing in the coefficients.

    Parameters:
    - `f`: polynomial
    - `gen`: Sympy symbol
      The dependent variable of the polynomial.
    Returns:
    - `roots`: list
      The roots of `f` as `RootOf` objects.
```

References

Examples

```python
>>> from sympy import RootOf
>>> from sympy.abc import x
>>> from abelfunctions.utilities import all_roots

>>> a = RootOf(y**2 + 1, 0, radicals=False)

>>> f = x**2 + a*x - 1

>>> all_roots(f, x)
[RootOf(_x**4 - _x**2 + 1, 1), RootOf(_x**4 - _x**2 + 1, 3)]
```
```python
>>> map(lambda z: f.evalf(subs={x:z}), all_roots(f,x))
[0.e-126 + 0.e-128*I, 0.e-126 - 0.e-128*I]
```

**abelfunctions.utilities.rootoftools.rootofsimp(expr)**

Simplifies expressions containing RootOf types.

Reduces expressions containing RootOf types modulo their defining polynomials. In particular, a root \( r \) of a polynomial \( p(x) = a_n x^n + ... + a_0 \) satisfies \( p(r) = 0 \) by definition. Therefore, any power of \( r \) greater than \( n \) can be rewritten in terms of smaller powers. In general, any polynomial or rational expression of \( r \) can be reduced modulo its defining polynomial \( p \).

**rootofsimp()** rewrites the expression as a numerator-denominator pair, computes the modular inverse of the denominator, and returns the product of this inverse with the numerator modulo the defining polynomial of the root. This is done for each RootOf appearing in the expression.

**Parameters**

- **expr**: Sympy expression

**Returns**

- **expr**: Sympy expression

The input expression reduced modulo the defining polynomials of the RootOfs appearing within.

**Examples**

```python
>>> from sympy import rootofsimp, RootOf
>>> from sympy.abc import x,y
>>> r = RootOf(x**4 - x - 1, 0)
>>> rootofsimp(r**4)
RootOf(x**4 - x - 1, 0) + 1
>>> rootofsimp(r**8)
2*RootOf(x**4 - x - 1, 0) + RootOf(x**4 - x - 1, 0)**2 + 1
>>> rootofsimp(1/r)
-l + RootOf(x**4 - x - 1, 0)**3
```

### 5.2.16 X-Path Factory abelfunctions.xpath_factory

Module for computing the monodromy group of the set of discriminant points of a complex plane algebraic curve.

**class abelfunctions.xpath_factory.XPathFactory(RS, base_point=None, kappa=0.6)**

**Attributes**

- **base_point()**: Returns the base point of the monodromy group.
RS | (RiemannSurface)  
---|-------------------
kappa | (double (default: 3/5)) A scaling factor between 0.5 and 1.0 used to modify the radius of the bounding circles.

**Methods**

- **discriminant_points**([exact]) Returns the ordered discriminant points of the curve.
- **xpath_to_discriminant_point**(bi) Returns the xpath leading to the discriminant point bi.
- **xpath_monodromy_path**(bi[, nrots]) Returns the xpath starting from the base point, going around the discriminant point bi nrots times, and returning to the base x-point.
- **xpath_to_point**(x) Returns an xpath to an arbitrary point x.
- **xpath_reverse**(xpath) Reverses an x-path.
- **show_paths**(ax=None) Plots all of the monodromy paths of the curve.

- **_compute_radii**(kappa=0.6) Returns the radii of the bounding circles.
  
  **Parameters** kappa : double  
  
  A scaling factor between 0.5 and 1.0. kappa = 1.0 means that the bounding circles are taken to be as large as possible without overlapping.

- **base_point**() Returns the base point of the monodromy group. Same point as the x-coordinate of the base place on the Riemann surface.

- **discriminant_points**(exact=False) Returns the ordered discriminant points of the curve.
  
  The discriminant points are ordered by increasing argument with the base point. Exact (sympy.Expr) or numerical approximations to these discriminant points can be retrieved by setting the exact keyword.
  
  **Parameters** exact : boolean  
  
  (Default: False) If true, returns

  **Returns** list  
  
  The discriminant points of the algebraic curve.

- **radius**(bi) Returns the radius of the bounding circle around bi.
  
  **Parameters** bi : complex  
  
  A discriminant point of the algebraic curve.

- **show_paths**(ax=None, *args, **kwds) Plots all of the monodromy paths of the curve.
  
  **Parameters** ax : matplotlib.Axes  
  
  The figure axes on which to plot the paths.

  **Returns** None

- **xpath_around_discriminant_point**(bi, nrots=1) Returns the xpath consisting of nrots rotations around the bounding circle of discriminant point bi.
  
  The sign of nrots indicates the sign of the direction.
**Parameters**

- **bi**: complex
  
  A discriminant point of the curve.

- **nrots**: integer (default $I$)
  
  A number of rotations around this discriminant point.

---

**xpath_around_infinity** ($nrots=I$)

Returns the xpath starting at the base point, going around infinity $nrots$ times, and returning to the base point.

This path is sure to not only encircle all of the discriminant points but also stay sufficiently outside the bounding circles of the points.

**Parameters**

- **nrots**: integer, (default $I$)
  
  The number of rotations around infinity.

**Returns**

RiemannSurfacePath

The xpath encircling infinity.

---

**xpath_monodromy_path** ($bi, nrots=I$)

Returns the xpath starting from the base point, going around the discriminant point $bi$ $nrots$ times, and returning to the base x-point.

The sign of $nrots$ indicates the sign of the direction.

**Parameters**

- **bi**: complex
  
  A discriminant point.

- **nrots**: integer (default $I$)
  
  A number of rotations around this discriminant point.

**Returns**

list

An x-path representing the monodromy path with $nrots$ rotations about the discriminant point $bi$.

---

**xpath_reverse** ($xpath$)

Reverses an x-path.

Useful for building the return path from a discriminant point to the base point.

**Parameters**

- **xpath**: list
  
  A list of tuples defining lines and semicircles in the complex x-plane.

**Returns**

list

A list of tuples defining lines and semicircles in the complex x-plane representing the reverse of $xpath$.

---

**xpath_to_discriminant_point** ($bi$)

Returns the xpath leading to the discriminant point $bi$.

**Parameters**

- **bi**: complex
  
  A discriminant point of the curve.

---

**xpath_to_point** ($x$)

Returns an xpath to an arbitrary point $x$.

**Parameters**

- **x**: complex
A point on the complex x-sphere.

class abelfunctions.xpath_factory.XPathFactoryAbel(RS, base_point=None, kappa=0.6)

Bases: abelfunctions.xpath_factory.XPathFactory

An XPathFactory designed for constructing paths used in the Abel map. Based on the technique of Deconinck, van Hoeij, and Patterson [R2].

References

[R2]

Attributes

See XPathFactory.

Methods

xpath_to_discriminant_point(bi)  Returns the xpath leading to the discriminant point bi.

xpath_to_point(x)  Returns an xpath to an arbitrary point x.

xpath_build_avoiding_path(z0, z1)  Returns an xpath to z from a (default: base point) avoiding discriminant points as necessary.

xpath_circle_discriminant_point _avoiding_arc(w0, w1, bi)

Returns the arc (radius, center, starting_theta, dtheta), from the points w0 and w1 on the bounding circle around bi.

The arc is constructed in such a way so that the monodromy properties of the path are conserved.

_intersection_points(z0, z1, bi)

Returns the complex points w0,w1 where the line from z0 to z1 intersects the bounding circle around bi.

_intersects_discpt(z0, z1, bi)

Returns True if the line from z0 to z1 intersects the bounding circle around the discriminant point bi.

xpath_around_discriminant_point(bi, nrots=1)

Returns the xpath consisting of nrots rotations around the bounding circle of discriminant point bi.

The sign of nrots indicates the sign of the direction.

Parameters

bi : complex

A discriminant point of the curve.

nrots : integer (default 1)

A number of rotations around this discriminant point.

xpath_build_avoiding_path(z0, z1)

Returns an xpath to z from a (default: base point) avoiding discriminant points as necessary.

xpath_to_discriminant_point(bi)

Returns the xpath leading to the discriminant point bi.

Parameters

bi : complex
A discriminant point of the curve.

**xpath_to_point**(x)

Returns an xpath to an arbitrary point x.

**Note:** Implement this.

**Parameters** x : complex

A point on the complex x-sphere.

## 5.2.17 Y-Path Factory `abelfunctions.ypath_factory`

This module defines the y-skeleton of the Riemann surface. That is, a means of not only computing the a- and b-cycles of the first homology group of the Riemann surface but also mechanisms for travelling from one sheet to another on the Riemann surface.

**class** `abelfunctions.ypath_factory.YPathFactory(RS, monodromy_group)`

**Bases:** object

Defines the basic y-path structure of the Riemann surface.

In particular, this class offers methods for determining which y-paths, given by a list of branch points in the complex x-plane and rotation numbers, to take order to define homology basis cycles as well as sheet switching paths.

**Note:** This class is a light wrapper around legacy code. This legacy code should eventually be made part of this class. What’s implemented here is a temporary hack.

**Attributes**

<table>
<thead>
<tr>
<th>RS</th>
<th>(Riemann Surface)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>(networkx.Graph) A graph encoding the y-skeleton of the Riemann surface.</td>
</tr>
</tbody>
</table>

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_cycles()</code></td>
<td>Returns the y-paths of the a-cycles of the Riemann surface.</td>
</tr>
<tr>
<td><code>b_cycles()</code></td>
<td>Returns the y-paths of the b-cycles of the Riemann surface.</td>
</tr>
<tr>
<td><code>c_cycles()</code></td>
<td>Returns the y-paths of the c-cycles of the Riemann surface and the linear combination matrix defining the a- and b-cycles.</td>
</tr>
<tr>
<td><code>y_path_sheet_swap</code></td>
<td></td>
</tr>
<tr>
<td><code>_node</code>(value)</td>
<td>Converts value to its associated node on the y-skeleton self.C.</td>
</tr>
<tr>
<td><code>_nodes</code>(ypath)</td>
<td>Converts a ypath from node data to value data.</td>
</tr>
<tr>
<td><code>_trim_ypath</code>(ypath)</td>
<td>Trims off the sheet data from ypath.</td>
</tr>
</tbody>
</table>

Given a ypath in (..., sheet, (branch_point, rotations), ...) form return the same path but with only the
branch points and rotations information.

_value (node)

  Gets the value associated with node on the y-skeleton self.C.

_values (ypath, rotations=False)

  Converts a ypath from value data to node data.

  See self._value(). This method can return the rotation information, as opposed to the permutation, as an option.

  **Note:** In order to provide rotation data the ypath must contain a starting and ending sheet. Also, it is assumed that the ypath is starting at / closer to the base place and ending further away.

  **Parameters** ypath : list

  A list of nodes on the y-skeleton self.C.

_a_cycles()

  Returns the y-paths of the a-cycles of the Riemann surface.

  **Returns** list

  A list of the a-cycles of the Riemann surface as RiemannSurfacePath objects.

_b_cycles()

  Returns the y-paths of the b-cycles of the Riemann surface.

_base_node()

  Returns the root node of the y-skeleton.

_c_cycles()

  Returns the y-paths of the c-cycles of the Riemann surface and the linear combination matrix defining the a- and b-cycles from the c-cycles.

  The a- and b- cycles of the Riemann surface are formed from linear combinations of the c-cycles. These linear combinations are obtained from the :py:linear_combinations() method.

  **Note:** It may be computationally more efficient to integrate over the (necessary) c-cycles and take linear combinations of the results than to integrate over the a- and b-cycles separately. Sometimes the column rank of the linear combination matrix (that is, the number of c-cycles used to construct a- and b-cycles) is lower than the size of the homology group and sometimes the c-cycles are simpler and shorter than the homology cycles.

/homology()

  Computes the first homology group of the Riemann surface.

  **Returns** a_cycles, b_cycles, c_cycles, linear_combinations

  y-paths corresponding to the a-, b-, and c-cycles and a matrix giving the linear combination of c-cycles for each a- and b-cycle.

  **Note:** Move compress cycle to RiemannSurfacePathFactory?

  **Note:** Delete legacy behavior of including sheet numbers in y-paths.

/plot()

  Plots the y-skeleton of the Riemann surface.

5.2. Module Reference
**ypath_from_base_to_sheet**(sheet)
Returns a ypath from the base sheet of the Riemann surface to sheet.

**Parameters**
- **sheet**: int
  The index of the target sheet.

**ypath_from_sheet_to_base**(sheet)
Returns a ypath from sheet to the base sheet of the Riemann surface.

**Note**: This is simply a reversal of *ypath_from_base_to_sheet()*.

**Parameters**
- **sheet**: int
  The index of the target sheet.

**ypath_values_reverse**(ypath)
Returns a ypath representing the reverse of ypath.

Reversing a ypath means not only visiting the branch points in reverse order but also rotating about them in reverse.

**Note**: Only accepts trimmed ypaths in \((bpt, nrots)\) notation.

**abelfunctions.ypath_factory.compress_cycle**(cycle, tretkoff_graph)
Given a cycle, the Tretkoff graph, and the monodromy graph, return a shortened equivalent cycle.

**abelfunctions.ypath_factory.compute_ab_cycles**(c_cycles, linear_combinations, g, tretkoff_graph)
Returns the a- and b-cycles of the Riemann surface given the intermediate ‘c-cycles’ and linear combinations matrix.

**Input**:
- **c_cycles**
- **linear_combinations**: output of the Frobenius transform of the

**abelfunctions.ypath_factory.compute_c_cycles**(tretkoff_graph, final_edges)
Returns the c-cycles of the Riemann surface.

**Input**:
- **C**: the Tretkoff graph
- **final_edges**: a list of the final edges of the Tretkoff graph

**Output**:
A list of the form

\[[s_0, (b_{i_0}, n_{i_0}), s_1, (b_{i_1}, n_{i_1}), ...]\]

where “s_k” is a sheet number, “b_{i_k}” is the \(i_k\)'th branch point, and “n_{i_k}” is the number of times and direction to go about branch point “b_{i_k}”.

**abelfunctions.ypath_factory.final_edges**(C)
Returns a list of final edges from the homology graph.

The final edges are those that define the c-cycles on the Riemann surface. Note that the edges returned are such that the nodes of the edge are _both_ final nodes.

The final edges are ordered such that the sheet number appears first in the edge.
Abelfunctions Documentation, Release 0.1.0

Input:

• homology graph

Output:

• list of (ordered) tuples representing the final edges

`abelfunctions.ypath_factory.find_cycle(pi, j)`

Returns the cycle (as a list) of the permutation pi containing j.

The ordering of a cycle is important for the homology functions since cycles are used to index dictionaries. For example, although “(0 7 4)” and “(7 4 0)” are the same cycle, this function outputs the cycles with the smallest element of the cycle first.

`abelfunctions.ypath_factory.frobenius_transform(A, g)`

Perform the Frobenius transform on the matrix A.

This procedure brings any intersection matrix a to its canonical form b by a transformation alpha * a * transpose(alpha)=b. If 2g=rank(a) and d is the size of the square matrix a, then b has d-2g null rows and d-2g null columns. These are moved to the lower right corner. On its diagonal, b has 2 gxg null blocks. Above the diagonal is a gxg identity block. Below the diagonal is a gxg negative identity block. The output of the procedure is the transformation matrix alpha.

`abelfunctions.ypath_factory.intersection_matrix(final_edges, g)`

Returns the intersection matrix from a list of final edges.

Compute the intersection matrix of the c-cycles from the Tretkoff graph and final edge data output by `tretkoff_graph()`.

Input:

• C: (networkx.Graph) Tretkoff graph
• final_edges: each edge corresponds to a c-cycle on the Riemann surface
• g: the expected genus of the riemann surface as given by singularities.genus()

`abelfunctions.ypath_factory.reorder_cycle(c, j=None)`

Reorder a cycle.

Returns a cycle (as a list) with the element “j” occurring first. If “j” isn’t provided then assume sorting by the smallest element.

`abelfunctions.ypath_factory.reverse_cycle(cycle)`

Returns the reversed cycle. Note that rotation numbers around branch points are correctly computed.

`abelfunctions.ypath_factory.smallest(l)`

Returns a cycle with the smallest sheet appearing first.

The cycles of the homology are written with their smallest sheet number first. This function finds the smallest sheet number in the cycle `l = (sheet, branch point, sheet, branch point, ...)`

`abelfunctions.ypath_factory.tretkoff_graph(monodromy_group)`

Construct the Tretkoff graph from a monodromy group.

There are two types of nodes:

• sheets: (integer) these occur on the even levels
• branch places: (complex, permutation) the first elements is the projection of the place in the complex x-plane. the second element is a cycle appearing in the monodromy element. (places above a branch point are in 1-1 correspondence with the cycles of the permutation) these occur on the odd levels
Abelfunctions is the primary result of Chris Swierczewski’s Ph.D. thesis. If you are someone who is interested in computing with Riemann surfaces, Abelain functions and algebraic curves then you may enjoy contributing to this project. The goal of this documentation is to get you started with adding to the software’s functionality.

6.1 Development Prerequisites

Abelfunctions uses the Git version control system to manage changes and contributions. We also use GitHub to host the Abelfunctions project.

6.2 Development Topics

The following articles describe how to contribute to specific aspects of Abelfunctions.

6.2.1 Writing Documentation

This document describes how to contribute to Abelfunctions’s documentation. The following packages are required to build the documentation, in addition to the usual Abelfunctions prerequisites:

- sphinx
- releases

Documentation Structure

The abelfunctions documentation is found in the doc directory located at the top of the file tree of the code repository. There are several files and subdirectories within but the most important one is the doc/source directory.

You’ll find several files written in the reStructuredText (reST) syntax. It’s a simple and easy to use syntax similar to wiki or markdown format. You can find a reST primer on the Sphinx website.

There are several subdirectories for specific types of documentation.

- doc/source/applications: contains a list of applications demonstrating how to use abelfunctions to solve various problems,
- doc/source/reference: auto-generated documentation of the abelfunctions API,
Building the Documentation

To build the documentation on your own machine, simply execute

```
$ cd abelfunctions/doc
$ make html
```

if you want to build the documentation website or

```
$ make pdf
```

for a .pdf version. Also note that the documentation is auto-updated at http://abelfunctions.cswiercz.info whenever a commit is pushed to the master branch of the GitHub repository https://github.com/cswiercz/abelfunctions.

Adding A New Page

Adding a new page to the documentation is easy. For example, let’s say we wish to add a new application to the Applications page.

1. Create a mynewapplication.rst document in /doc/source/applications. (Of course, using a more appropriate file name.)

2. Make this new document appear in the documentation by adding mynewapplication to the applications/index.rst table of contents tree (toctree) like so:

   ```
   .. toctree::
      :maxdepth: 1
      <other, already existing documents>
      mynewapplication
   ```

Rebuild the documentation to see a link to your new document appear in the table of contents and root page.

Source-Level Documentation

Every module of abelfunctions (the code itself) in abelfunctions should be documented using the numpydoc style guide. That is, every function, class, and module itself should have a docstring associated with it describing to both the user and the developer its purpose, use, and any other pertinent information. The numpydoc style in particular was chosen because its legibility in both the code itself and the auto-generated documentation webpage.

To add an abelfunctions module mymodule to this documentation

1. Add the text mymodule to the table of contents tree in doc/source/reference/index.rst as if you were adding a new page.

2. Add a file mymodule.rst to doc/source/reference with the text

   ```
   .. automodule:: abelfunctions.mymodule
      :members:
      :show-inheritance:
   ```

For a decent example of code documentation syntax and layout see abelfunctions/puiseux.py.

In brief, every function and class method should have a docstring with the following fields. The details are given in the numpydoc guide:
```python
def myfunc(a, b):
    r'''A brief description of the function that fits in one line.

    More in-depth description of the function and its behavior that
    can span multiple lines. You can include mathematical
    expressions :math:`f(x,y) = 0` and notes

    .. note::
        This is a note.

    Parameters
    ----------
    a : the type of a
        Description of `a` as a function parameter.
    b : the type of b
        Description of `b` as a function parameter.

    Returns
    ------
    the type of the return
    A description of the return value.

    Raises
    ------
    (optional) If this function is designed to raise errors, describe
    the errors and which conditions invoke them.

    Algorithm
    ---------
    (optional) Brief explanation of the algorithm used in this func.

    References
    ----------
    (optional) References for the source of the algorithm.
    '''
```

Classes should have docstrings with a similar structure (though without a Returns field, for instance) with the fields Attributes and Methods:

```python
class MyClass(object):
    r'''A brief description of the class that fits in one line.

    More in-depth description of the class and its purpose that can
    span multiple lines.

    Attributes
    ----------
    myattr : type of attribute
        Description of attribute

    Methods
    -------
    mymethod1
    mymethod2
    mymethod3
    '''
```
Again, consult the numpydoc style guide for more information on docstring syntax and structure.
Changelog

• #91: Complete reference section of documentation
  – Every file in Abelfunctions should now have a reference to it in the Reference section of the documentation.
• #108: Compute roots of polynomials with RootOf in coefficients
• : Initial version.
Abelfunctions is the Ph.D. thesis work of Chris Swierczewski. Several others have made key contributions as well.

- Chris Swierczewski: started project and project lead.
- Bernard Deconinck: Chris Swierczewski’s Ph.D. thesis advisor
- James Collins: Riemann theta functions. Siegel modular transformations.
- Daniel Shapero: software design advice.

License

Unless stated otherwise, all files in the Abelfunctions project, all images, and all documentation including this documentation are licensed using the new BSD license:

Copyright (c) 2012, Chris Swierczewski
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


[R1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994, pg. 163


abelfunctions.abelmap, 15
abelfunctions.analytic_continuation, 18
abelfunctions.analytic_continuation_smale, 21
abelfunctions.differentials, 22
abelfunctions.divisor, 27
abelfunctions.integralbasis, 29
abelfunctions.polynomials, 31
abelfunctions.puiseux, 32
abelfunctions.riemann_constant_vector, 41
abelfunctions.riemann_surface, 44
abelfunctions.riemann_surface_path, 47
abelfunctions.riemann_surface_path_factory, 53
abelfunctions.riemann_theta.riemann_theta, 56
abelfunctions.singularities, 59
abelfunctions.utilities.cache, 62
abelfunctions.utilities.permutations, 63
abelfunctions.utilities.rootoftools, 64
abelfunctions.xpath_factory, 66
abelfunctions.ypath_factory, 70
### Symbols

- `_avoiding_arc()` (abelfunctions.xpath_factory.XPathFactoryAbel method), 69
- `_branching_number()` (in module abelfunctions.singularities), 60
- `_compute_puiseux_series()` (abelfunctions.analytic_continuation.AnalyticContinuatorPuiseux method), 20
- `_compute_radii()` (abelfunctions.xpath_factory.XPathFactory method), 67
- `_cycles_from_list()` (abelfunctions.utilities.permutations.Permutation method), 63
- `_delta_invariant()` (in module abelfunctions.singularities), 60
- `_eval_primitive()` (abelfunctions.abelmap.AbelMap_Function method), 17
- `_find_necessary_xvalues()` (abelfunctions.differentials.Differential method), 25
- `_initialize_checkpoints()` (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 50
- `_initialize_extension()` (abelfunctions.puiseux.PuiseuxTSeries method), 34
- `_integral_basis_monic()` (in module abelfunctions.integralbasis), 30
- `_intersection_points()` (abelfunctions.xpath_factory.XPathFactoryAbel method), 69
- `_intersects_disc_pt()` (abelfunctions.xpath_factory.XPathFactoryAbel method), 69
- `_list_from_cycles()` (abelfunctions.utilities.permutations.Permutation method), 63
- `_multiplicity()` (in module abelfunctions.singularities), 60
- `_node()` (abelfunctions.ypath_factory.YPathFactory method), 70
- `_nodes()` (abelfunctions.ypath_factory.YPathFactory method), 70
- `_path_near_discriminant_point()` (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 54
- `_path_to_discriminant_place()` (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 54
- `_path_to_infinite_place()` (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 54
- `_path_to_regular_place()` (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 54
- `_set_str_noncycle()` (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 50
- `_terms_from_sympy_const()` (abelfunctions.puiseux.PuiseuxXSeries method), 37
- `_terms_from_sympy_expression()` (abelfunctions.puiseux.PuiseuxXSeries method), 37
- `_terms_from_sympy_generic()` (abelfunctions.puiseux.PuiseuxXSeries method), 37
- `_terms_from_sympy_polynomial()` (abelfunctions.puiseux.PuiseuxXSeries method), 37
- `_terms_from_sympy_rational()` (abelfunctions.puiseux.PuiseuxXSeries method), 37
- `_transform()` (in module abelfunctions.singularities), 60
- `_trim_ypath()` (abelfunctions.ypath_factory.YPathFactory method), 70
- `_value()` (abelfunctions.ypath_factory.YPathFactory method), 71
- `_values()` (abelfunctions.ypath_factory.YPathFactory method), 71

### A

- `a_cycles()` (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 55
Abelfunctions Documentation, Release 0.1.0

Index

A

a_cycles() (abelfunctions.ypath_factory.YPathFactory method), 71
abelfunctions.abelmap (module), 15
abelfunctions.analytic_continuation (module), 18
abelfunctions.analytic_continuation_smale (module), 21
abelfunctions.differentials (module), 22
abelfunctions.divisor (module), 27
abelfunctions.integralbasis (module), 29
abelfunctions.polynomials (module), 31
abelfunctions.puiseux (module), 32
abelfunctions.riemann_constant_vector (module), 41
abelfunctions.riemann_surface (module), 44
abelfunctions.riemann_surface_path (module), 47
abelfunctions.riemann_surface_path_factory (module), 53
abelfunctions.riemann_theta (module), 56
abelfunctions.singularities (module), 59
abelfunctions.utilities.cache (module), 62
abelfunctions.utilities.permutations (module), 63
abelfunctions.utilities.rootoftools (module), 64
abelfunctions.xpath_factory (module), 66
abelfunctions.ypath_factory (module), 70
AbelianDifferentialSecondKind (class in abelfunctions.differentials), 23
AbelMap_Fun (class in abelfunctions.abelmap), 16
action() (abelfunctions.utilities.permutations.Permutation method), 63
add_term() (abelfunctions.puiseux.PuiseuxTSeries method), 34
all_roots() (in module abelfunctions.utilities.rootoftools), 65
almost_monicize() (in module abelfunctions.puiseux), 38
analytically_continue() (abelfunctions.analytic_continuation.AnalyticContinuator method), 19
analytically_continue() (abelfunctions.analytic_continuation_smale.AnalyticContinuatorSmale method), 22
analytically_continue() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 50
AnalyticContinuator (class in abelfunctions.analytic_continuation), 19
AnalyticContinuatorPuiseux (class in abelfunctions.analytic_continuation), 20
AnalyticContinuatorSmale (class in abelfunctions.analytic_continuation_smale), 21
as_numer_denom() (abelfunctions.differentials.Differential method), 25
as_sympy_expr() (abelfunctions.differentials.Differential method), 25
as_sympy_expr() (abelfunctions.puiseux.PuiseuxXSeries method), 37

B

b_cycles() (abelfunctions.ypath_factory.YPathFactory method), 55
b_cycles() (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 71
base_lift() (abelfunctions.riemann_surface.RiemannSurface method), 44
base_node() (abelfunctions.ypath_factory.YPathFactory method), 71
base_place() (abelfunctions.riemann_surface.RiemannSurface method), 44
base_point() (abelfunctions.riemann_surface.RiemannSurface method), 45
base_point() (abelfunctions.xpath_factory.XPathFactory method), 67
base_sheets() (abelfunctions.riemann_surface.RiemannSurface method), 45
bezout() (in module abelfunctions.puiseux), 38

C

c_cycles() (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 55
c_cycles() (abelfunctions.ypath_factory.YPathFactory method), 71
cached_function() (in module abelfunctions.utilities.cache), 62
cached_function_fast() (in module abelfunctions.utilities.cache), 62
cached_method() (in module abelfunctions.utilities.cache), 63
canonical_divisor() (in module abelfunctions.riemann_constant_vector), 42
centered_at_place() (abelfunctions.differentials.Differential method), 25
closest_discriminant_point() (abelfunctions.riemann_surface.RiemannSurface method), 45
closest_discriminant_point() (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 55
coerce_to_numerical() (abelfunctions.puiseux.PuiseuxXSeries method), 34
components() (abelfunctions.abelmap.Jacobian method), 18
compress_cycle() (in module abelfunctions.ypath_factory), 72
compute_ab_cycles() (in module abelfunctions.ypath_factory), 72
compute_bd() (in module abelfunctions.integralbasis), 30
compute_c_cycles() (in module abelfunctions.ypath_factory), 72
compute_expansion_bounds() (in module abelfunctions.integralbasis), 30
compute_K0() (in module abelfunctions.riemann_constant_vector), 43
compute_series_truncations() (in module abelfunctions.integralbasis), 31

differential (class in abelfunctions.differentials), 23
differentials() (in module abelfunctions.differentials), 26
discriminant_points() (abelfunctions.riemann_surface.RiemannSurface method), 45
discriminant_points() (abelfunctions.riemann_surface_path_factory.RiemannSurfacePath method), 55
discriminant_points() (abelfunctions.xpath_factory.XPathFactory method), 67

E

eval() (abelfunctions.abelmap.AbelMap_Function method), 17
eval() (abelfunctions.abelmap.Jacobian method), 18
eval() (abelfunctions.differentials.Differential method), 25
eval() (abelfunctions.puiseux.PuiseuxXSeries method), 37
eval() (abelfunctions.riemann_theta.riemann_theta.RiemannTheta_Function method), 57
eval_x() (abelfunctions.puiseux.PuiseuxTSeries method), 34
eval_y() (abelfunctions.puiseux.PuiseuxTSeries method), 34
evalf() (abelfunctions.puiseux.PuiseuxXSeries method), 37
evaluate() (abelfunctions.differentials.Differential method), 25
evaluate() (abelfunctions.riemann_surface_path.RiemannSurfacePath method), 48
evaluate() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 50
exponential_part() (abelfunctions.riemann_theta.riemann_theta.RiemannTheta_Function method), 57
extend() (abelfunctions.puiseux.PuiseuxTSeries method), 35
extend_to_t() (abelfunctions.puiseux.PuiseuxTSeries method), 35
extend_to_x() (abelfunctions.puiseux.PuiseuxTSeries method), 35

Index

F

fast_expand() (in module abelfunctions.differentials), 26
final_edges() (in module abelfunctions.ypath_factory), 72
find_cycle() (in module abelfunctions.ypath_factory), 73
find_regular_places() (in module abelfunctions.riemann_constant_vector), 43
fractional_part() (in module abelfunctions.abelmap), 18
frobenius_transform() (in module abelfunctions.ypath_factory), 73

G

generalized_polygon_side() (in module abelfunctions.puiseux), 39
genus() (in module abelfunctions.singularities), 61
gradient() (abelfunctions.riemann_theta.riemann_theta.RiemannTheta_Function method), 58

H

half_lattice_filter() (in module abelfunctions.riemann_constant_vector), 43
half_lattice_vector() (in module abelfunctions.riemann_constant_vector), 44
holomorphic_differentials() (abelfunctions.riemann_surface.RiemannSurface method), 45
holomorphic_oneforms() (abelfunctions.riemann_surface.RiemannSurface method), 45
homology() (abelfunctions.ypath_factory.YPathFactory method), 71

I

index() (abelfunctions.utilities.permutations.Permutation method), 64
initialize_half_lattice_vectors() (in module abelfunctions.riemann_constant_vector), 44
initialize_terms() (abelfunctions.puiseux.PuiseuxXSeries method), 38
Int() (in module abelfunctions.integralbasis), 29
integral_basis() (in module abelfunctions.integralbasis), 31
integrate() (abelfunctions.analytic_continuation.AnalyticContinuator method), 19
integrate() (abelfunctions.analytic_continuation.AnalyticContinuatorPuiseux method), 21
integrate() (abelfunctions.analytic_continuation_smale.AnalyticContinuatorSmale method), 22
integrate() (abelfunctions.riemann_surface.RiemannSurface method), 45
integrate() (abelfunctions.riemann_surface_path.RiemannSurfacePath method), 48
integrate() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 51
intersection_matrix() (in module abelfunctions.ypath_factory), 73
inverse() (abelfunctions.utilities.permutations.Permutation method), 64
is_identity() (abelfunctions.utilities.permutations.Permutation method), 64
is_zero_poly() (abelfunctions.polynomials.MultivariatePolynomial method), 32
J
Jacobian (class in abelfunctions.abelmap), 17
L
lift() (abelfunctions.riemann_surface.RiemannSurface method), 46
localize() (abelfunctions.differentials.Differential method), 26
M
matching_permutation() (in module abelfunctions.utilities.permutations), 64
mnuk_conditions() (in module abelfunctions.differentials), 27
monodromy_group() (abelfunctions.riemann_surface.RiemannSurface method), 46
monodromy_group() (abelfunctions.riemann_surface_path.RiemannSurfacePath method), 55
monodromy_path() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 55
MultivariatePolynomial (class in abelfunctions.polynomials), 32
N
newton_data() (in module abelfunctions.puiseux), 39
newton_iteration() (in module abelfunctions.puiseux), 39
newton_iteration_step() (in module abelfunctions.puiseux), 39
newton_polygon() (in module abelfunctions.puiseux), 40
newton_polygon_exceptional() (in module abelfunctions.puiseux), 40
oscillatory_part() (abelfunctions.riemann_theta.riemann_theta.RiemannTheta_Function method), 59
oscillatory_part() (in module abelfunctions.riemann_theta), 59
oscillatory_part_gradient() (abelfunctions.riemann_theta.riemann_theta.RiemannTheta_Function method), 59
oscillatory_part_hessian() (abelfunctions.riemann_theta.riemann_theta.RiemannTheta_Function method), 59
P
parameterize() (abelfunctions.analytic_continuation.AnalyticContinuator method), 20
parameterize() (abelfunctions.analytic_continuation.AnalyticContinuatorPuiseux method), 21
parameterize() (abelfunctions.analytic_continuation_smale.AnalyticContinuatorSmale method), 22
path() (abelfunctions.riemann_surface.RiemannSurface method), 46
path_to_place() (abelfunctions.riemann_surface_path.RiemannSurfacePathFactory method), 56
period_matrix() (abelfunctions.riemann_surface.RiemannSurface method), 46
Permutation (class in abelfunctions.utilities.permutations), 63
Place (class in abelfunctions.divisor), 28
plot() (abelfunctions.differentials.Differential method), 26
plot() (abelfunctions.ypath_factory.YPathFactory method), 71
plot3d_x() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 51
plot3d_y() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 51
plot_x() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 51
plot_y() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 52
puiseux() (in module abelfunctions.puiseux), 40
puiseux_rational() (in module abelfunctions.puiseux), 40
PuiseuxTSeries (class in abelfunctions.puiseux), 33
PuiseuxXSeries (class in abelfunctions.puiseux), 36

radius() (abelfunctions.xpath_factory.XPathFactory method), 67
RegularPlace (class in abelfunctions.divisor), 28
reorder_cycle() (in module abelfunctions.xpath_factory), 73
reverse_cycle() (in module abelfunctions.xpath_factory), 73
riemann_matrix() (abelfunctions.riemann_surface.RiemannSurface method), 46
RiemannConstantVector() (in module abelfunctions.riemann_constant_vector), 42
RiemannSurface (class in abelfunctions.riemann_surface), 44
RiemannSurfacePath (class in abelfunctions.riemann_surface_path), 47
RiemannSurfacePath_from_cycle() (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 54
RiemannSurfacePath_from_xpath() (abelfunctions.riemann_surface_path_factory.RiemannSurfacePathFactory method), 54
RiemannSurfacePathArc (class in abelfunctions.riemann_surface_path,RiemannSurfacePathFactory method), 49
RiemannSurfacePathFactory (class in abelfunctions.riemann_surface_path_factory), 53
RiemannSurfacePathLine (class in abelfunctions.riemann_surface_path), 49
RiemannSurfacePathPrimitive (class in abelfunctions.riemann_surface_path), 49
RiemannSurfacePathRay (class in abelfunctions.riemann_surface_path), 52
RiemannTheta_Function (class in abelfunctions.riemann_theta.riemann_theta), 56
rootofsimp() (in module abelfunctions.utilities.rootoftools), 66

segments (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive attribute), 52
set_analytic_continuator() (abelfunctions.riemann_surface_path.RiemannSurfacePathPrimitive method), 52
show_paths() (abelfunctions.riemann_surface.RiemannSurface method), 47
show_paths() (abelfunctions.riemann_surface_path.RiemannSurfacePathFactory method), 56
show_paths() (abelfunctions.xpath_factory.XPathFactory method), 67
singular_points_finite() (in module abelfunctions.singularities), 61
singular_points_infinite() (in module abelfunctions.singularities), 61
singularities() (in module abelfunctions.singularities), 61
smallest() (in module abelfunctions.xpath_factory), 73
solve_coefficient_system() (in module abelfunctions.integralbasis), 31
sum_partitions() (in module abelfunctions.riemann_constant_vector), 44

terms_from_yseries() (abelfunctions.puiseux.PuiseuxTSeries method), 36
transform_newton_polynomial() (in module abelfunctions.puiseux), 41
tretkoff_graph() (in module abelfunctions.xpath_factory), 73

UnivariatePolynomial (class in abelfunctions.polynomials), 32

valuation() (abelfunctions.divisor.Place method), 28
valuation() (abelfunctions.puiseux.PuiseuxXSeries method), 38
valuation_divisor() (abelfunctions.differentials.AbelianDifferentialSecondKind method), 23
valuation_divisor() (abelfunctions.differentials.Differential method), 26

xpath_around_discriminant_point() (abelfunctions.xpath_factory.XPathFactory method), 67
xpath_around_discriminant_point() (abelfunctions.xpath_factory.XPathFactoryAbel method), 69
xpath_around_infinity() (abelfunctions.xpath_factory.XPathFactory method), 68
xpath_build_avoiding_path() (abelfunctions.xpath_factory.XPathFactoryAbel method), 69
xpath_monodromy_path() (abelfunctions.xpath_factory.XPathFactory method), 68
xpath_reverse() (abelfunctions.xpath_factory.XPathFactory method), 68
Abelfunctions Documentation, Release 0.1.0

xpath_to_discriminant_point() (abelfunctions.xpath_factory.XPathFactory method), 68
xpath_to_discriminant_point() (abelfunctions.xpath_factory.XPathFactoryAbel method), 69
xpath_to_point() (abelfunctions.xpath_factory.XPathFactory method), 68
xpath_to_point() (abelfunctions.xpath_factory.XPathFactoryAbel method), 70
XPathFactory (class in abelfunctions.xpath_factory), 66
XPathFactoryAbel (class in abelfunctions.xpath_factory), 69
xseries() (abelfunctions.puiseux.PuiseuxTSeries method), 36

Y
ypath_from_base_to_sheet() (abelfunctions.ypath_factory.YPathFactory method), 71
ypath_from_sheet_to_base() (abelfunctions.ypath_factory.YPathFactory method), 72
ypath_values_reverse() (abelfunctions.ypath_factory.YPathFactory method), 72
YPathFactory (class in abelfunctions.ypath_factory), 70

Z
ZeroDivisor (class in abelfunctions.divisor), 29