Bear Reference Manual

For Version 0.9.4

David Dumas (daviddumas@gmail.com)

4 June 2005

Abstract

This manual is for Bear (version 0.9.4, 4 June 2005), a free software package that performs Bers slice computations and discreteness testing for punctured torus groups.

This reference manual is incomplete, and represents our first attempt at documentation beyond the brief comments in the source, the 'README' file, and the manual page.

Contents

Chapter 1

Command-Line Parameters

Bear accepts the following command-line options, which are also documented on the manual page:

- `-V`
  `--version`
  Print the version number and exit.

- `-h`
  `--help`
  Print a brief summary of the command-line options and exit.

- `-i`
  `--interactive`
  Enter interactive mode after processing the command-line. This is the default unless an input filename is provided.

- `-e <string>`
  `--exec <string>`
  Execute ¡string¿ as if it were entered into the interpreter during and interactive session. This option may be given more than once, in which case the statements will be executed in the order given.

- `-l <library-name>`
  `--library <library-name>`
  Load the Lua library ¡library-name¿ before executing any scripts or entering interactive mode.

- `-v`
  `--verbose`
  Provide verbose output on stderr about what is happening. This also enables time-to-completion estimation during calculations.

- `-q`
  `--quiet`

Be completely quiet. Suppresses even fatal errors and all Lua i/o except that which is specifically requested (e.g. the ‘print’ command). Exception: when in interactive mode, the prompt will still be displayed.

Chapter 2

Run-time Modules

2.1 Introduction

Bear interprets its input using Lua (`www.lua.org`), a procedural scripting language with a simple `C`-like syntax. What follows is a micro-introduction Lua as it is used to control Bear and a listing of the parameters and functions that Bear provides. The Lua documentation, which is available from the web page, provides much more detailed information.

A typical Bear script might calculate holonomy representations, test for discreteness, and write some of the resulting data to a file. This is accomplished by manipulating the values of certain parameters and then calling functions that perform the actual calculations and I/O operations.

The functionality of Bear is divided into several *modules*, which are like `C` structures or (instances of) `C++` classes. Each module is a single data structure (a Lua *table*, actually) that contains both control parameters and functions that perform operations depending on those parameters.

Parameters can be assigned values using a `C`-like syntax:

```
modulename.parameter = value;
```

Note that the semicolon at the end of the line is optional. The parameter value can be a literal of the appropriate type, as in these examples:

```
module.numberparam = 867530.9;
module.stringparam1 = 'foo in single quotes';
module.stringparam2 = ``bar in double quotes'';
```

Values can also be constructed via more complicated expressions, possibly using other variables; for example, the following will give `module.b` the value 205.80:

```
m = 2; d = 5;
module.b = (m*100) + d + 80/100
```

For the construction of strings there is the *concatenation operator* "..":

```
firstword = 'foo'; secondword = 'bar';
module.foospacebar = firstword .. ' ' .. secondword;
```

The core functionality of Bear is provided by the functions of the various modules; such functions are called as follows:

```
module.function(param1,param2,param3);
module.otherfunction();
```

Lua also has an extensive runtime library, providing many of the basic mathematical operations, string formatting functions, file input and output, and operating system interfaces common in other high-level languages. We will not detail these here, except for an example showing the usefulness of library functions for Bear computation scripts:

```
index = 15;                -- this is a comment, started by two hyphens
basefilename = 'output';
filename = basefilename .. string.format('\%03d',index) .. '.dat';
print(filename);           -- will print 'output015.dat'
```

2.2 `bear` - The global runtime options module

2.2.1 Purpose

The `bear` module allows certain global options to be changed at runtime, and allows a script to query certain configuration parameters (e.g. the version number). When the value of a parameter in this module is changed, it takes effect immediately.

2.2.2 Parameters

- `bear.verbose`
  If set to a nonzero value, produce verbose output. (See the command-line option `-v` or `--verbose`.)

- `bear.quiet`
  If set to a nonzero value, suppress all output. (See the command-line option `-q` or `--quiet`.)

- `bear.interactive`
  If set to a nonzero value, enter interactive mode after processing of scripts named as command-line parameters. This option has no effect if set interactively. (See the command-line option `-i` or `--interactive`.)

- `bear.version` *(read-only)*
  A string containing the Bear version number.

- `bear.build_hostname`*(read-only)*
  The name of the host on which Bear was compiled.

- `bear.build_date` *(read-only)*
  The date on which Bear was compiled.

- `bear.build_flags` *(read-only)*
  The flags passed to the C compiler when Bear was compiled.

- `bear.build_compiler_version` *(read-only)*
  The vendor and version number of the C compiler used to compile Bear.

2.3 `bers` - The Bers Slice Module

2.3.1 Purpose

The `bers` module computes holonomy groups of complex projective structures on a punctured torus. When combined with a discreteness test, this allows one to draw pictures of Bers slices of punctured tori, which is the main purpose of Bear.

The punctured torus is specified by means of its commensurable punctured sphere $\hat{\mathbb{C}} - \{0, 1, \lambda, \infty\}$; the holonomy group is then computed as the linear monodromy of the differential equation

$$u''(z) + \frac{1}{2}\phi(z)u(z) = 0$$

where the quadratic differential $\phi(z)$ is given by

$$\phi(z) = \frac{1}{2z^2} + \frac{(\lambda - 1)^2}{2(z - \lambda^2)(z - 1)^2} + \frac{c}{z(z-1)(z-\lambda)}.$$

Here c is a complex-valued parameter that specifies the complex projective structure. The module iterates through values of c in a square grid with a specified center and size.

2.3.2 Features

The heart of the Bers module is an ordinary differential equation solver that is applied to the Schwarzian equation associated to a quadratic differential. All of the ODE solvers from the GNU Scientific Library (GSL) are available, including popular methods like Runge-Kutta (classical, and with higher-order error estimates) and the Bulirsch-Stoer implicit solver.

If a FORTRAN compiler is available when Bear is built, the LSODE solver is also included (since version 0.9.4), and its use is recommended for high performance in holonomy calculations of moderate complexity.

Both the `bers` and `bers2` modules support a modular integration contour API (since version 0.9.3), allowing several types of integration contours (splines, piecewise linear, ellipses, etc.). The various integration contours have different speed/accuracy characteristics that become more apparent near the extremes of the moduli space of punctured tori.

Older versions of the `bers` module (before 0.9.3) supported an adaptive precision modification algorithm that attempted to improve the accuracy of the holonomy calculation in cases where the Schwarzian equation integrand is highly oscillatory. This feature has been removed because its implementation was never very successful. Different schemes for the improvement of ODE solver performance in the oscillatory regime may be considered for future development.

2.3.3 Parameters

- `bers.lambda.real`
  `bers.lambda.imag`
  The real and imaginary parts of the modular parameter λ such that $\hat{\mathbb{C}} - \{0, 1, \lambda, \infty\}$ is commensurable to the conformal structure on the punctured torus.

- `bers.center.real`
  `bers.center.imag`
  The real and imaginary parts of the center of the grid.

- `bers.radius`
  The radius (i.e. half of the side length) of the square bounding the grid.

- `bers.size`
  The number of points in each row and column of the grid.

- `bers.precision`
  The ODE solver will attempt to maintain an absolute error less than `bers.precision` when integrating the Schwarzian differential equation.

- `bers.relprecision`
  Applies to the LSODE solver only. LSODE will attempt to maintain a relative (fractional) error of less than `bers.relprecision` when integrating the Schwarzian differential equation. New in Bear 0.9.4.

- `bers.method`
  The name of the ODE integration method to be used. Currently, the following methods from GSL are supported:

  - `rk2`
    Embedded 2nd order Runge-Kutta with 3rd order error estimate. (**recommended**)
  - `rk4`
    4th order (classical) Runge-Kutta.

  - `rkf45`
    Embedded 4th order Runge-Kutta-Fehlberg method with 5th order error estimate. This method is a good general-purpose integrator.
  - `rkck`
    Embedded 4th order Runge-Kutta Cash-Karp method with 5th order error estimate.
  - `rk8pd`
    Embedded 8th order Runge-Kutta Prince-Dormand method with 9th order error estimate. (**recommended**)
  - `rk2imp`
    Implicit 2nd order Runge-Kutta at Gaussian points
  - `rk4imp`
    Implicit 4th order Runge-Kutta at Gaussian points
  - `gear1`
    M=1 implicit Gear method (**recommended**)
  - `gear2`
    M=2 implicit Gear method
  - `bsimp`
    Bulirsch-Stoer method

  The following method from ODEPACK is available if a FORTRAN compiler can be found when Bear is built:

  - `lsode`
    LSODE, a basic but highly effective FORTRAN ODE solver developed at Lawrence Livermore National Laboratory. Internally it uses a modification of the Adams algorithm.

- `bers.blocking`
  An integer flag controlling *blocking mode*. If nonzero, the Bers slice window will be computed in a way that allows several Bers slice images to be glued together without duplication of the boundary pixels. Specifically, the window will be set to the upper-left `sizexsize` subset of a `(size + 1)x(size + 1)` grid with the specified center and offset.

- `bers.contour_type`
  Type of Schwarzian equation integration contour. The integration API supports any system of piecewise C^2 contours. Currently the following types are available:

  - `spline_ellipse`
    Closed cubic splines approximating ellipses. This was the only contour type available before version 0.9.3.

- `ellipse`
  Exact ellipses. This is sometimes a bit slower than `spline_ellipse` but is guaranteed to enclose the right set of singular points, even for extreme values of the parameters (e.g. $|\lambda| < 10^{-6}$).
- `piecewise_linear`
  Rectangular contours with horizontal and vertical edges.

2.3.4 Obsoleted parameters

These parameters were available in previous versions of Bear, but have since been removed. It is not expected that they found widespread use.

- `bers.adapt-steps`
  Removed in Bear 0.9.3
- `bers.markov-threshold`
  Removed in Bear 0.9.3
- `bers.adapt-divisor`
  Removed in Bear 0.9.3
- `bers.adapt-persist`
  Removed in Bear 0.9.3

2.3.5 Functions

- `bers.run()`
  Compute the holonomy.

2.4 `bers2` - The New Bers Slice Module

2.4.1 Purpose

The `bers2` module performs the same function as the `bers` module (above).

2.4.2 Features

The `bers2` module is a rewrite of the `bers` module that allows the holonomy to be computed on a sparse grid and then inteprolated over a much finer mesh. When the Schwarzian equation can be solved accurately, the holonomy typically varies slowly enough that such interpolation gives nearly the same output with much less calculation. In cases where the Schwarzian equation is highly oscillatory, however, the `bers` module may be a better choice.

Both the `bers` and `bers2` modules support a modular integration contour API (since version 0.9.3), allowing several types of integration contours (splines, piecewise linear, ellipses, etc.). The various integration contours have different speed/accuracy characteristics that become more apparent near the extremes of the moduli space of punctured tori.

2.4.3 Parameters

Parameters unchanged from the `bers` module.

- `bers2.lambda.real`
  `bers2.lambda.imag`
  The real and imaginary parts of the modular parameter λ such that $\hat{\mathbb{C}} - \{0, 1, \lambda, \infty\}$ is commensurable to the conformal structure on the punctured torus.
- `bers2.center.real`
  `bers2.center.imag`
  The real and imaginary parts of the center of the grid.
- `bers2.radius`
  The radius (i.e. half of the side length) of the square bounding the grid.
- `bers2.size`
  The number of points in each row and column of the grid.
- `bers2.precision`
  The ODE solver will attempt to maintain an absolute error less than `bers2.precision` when integrating the Schwarzian differential equation.
- `bers2.relprecision`
  Applies to the LSODE solver only. LSODE will attempt to maintain a relative (fractional) error of less than `bers2.relprecision` when integrating the Schwarzian differential equation. New in Bear 0.9.4.
- `bers2.method`
  The name of the ODE integration method to be used. Currently, the following methods from GSL 1.4 are supported (*Note: these are exactly the methods that do not require the Jacobian of the system*):
  - `rk2`
    Embedded 2nd order Runge-Kutta with 3rd order error estimate.
  - `rk4`
    4th order (classical) Runge-Kutta.
  - `rkf45`
    Embedded 4th order Runge-Kutta-Fehlberg method with 5th order error estimate. This method is a good general-purpose integrator.
  - `rkck`
    Embedded 4th order Runge-Kutta Cash-Karp method with 5th order error estimate.
  - `rk8pd`
    Embedded 8th order Runge-Kutta Prince-Dormand method with 9th order error estimate. (**recommended**)

- `rk2imp`
  Implicit 2nd order Runge-Kutta at Gaussian points
- `rk4imp`
  Implicit 4th order Runge-Kutta at Gaussian points
- `gear1`
  M=1 implicit Gear method (**recommended**)
- `gear2`
  M=2 implicit Gear method
- `bsimp`
  Bulirsch-Stoer method

The following method from ODEPACK is available if a FORTRAN compiler can be found when Bear is built:

- `lsode`
  LSODE, a basic but highly effective FORTRAN ODE solver developed at Lawrence Livermore National Laboratory. Internally it uses a modification of the Adams algorithm.

Parameters specific to the `bers2` module.

- `bers2.interp_type`
  Type of holonomy interpolation; this is used to extend the holonomy from a sparse grid where it is computed using the Schwarzian ODE to a dense grid that is used to generate the Bers slice. Currently, the following interpolation methods are supported (via GSL):
  - `linear`
    Bilinear interpolation.
  - `cubic`
    Bicubic interpolation. (default; recommended)
- `bers2.interp_delta`
  Threshold for automatic interpolation. The sparse grid is subject to repeated dyadic subdivision until exact holonomy values on the new grid points agree with the values interpolated there using the previous grid, up to a threshold of `bers2.interp_delta`. Setting this to zero effectively forces the use of the most dense grid allowed by `bers2.sample_max`.
- `bers2.sample_min`
  Initial size for the interpolation grid. **In the current implementation, the quotient `bers2.size/bers2.sample_min` must be a power of 2.**
- `bers2.sample_max`
  Maximum size for the interpolation grid. Dyadic refinement will stop when the grid reaches this size, even if the threshold `bers2.interp_delta` has not been satisfied. **In the current implementation, the `bers2.sample_max` must be of the form 2^k`bers2.sample_min` for a positive integer k.**

- `bers2.padding`
  Interpolation window padding control. In order to eliminate interpolation artifacts at the boundary of the grid, the holonomy grid is scaled by a factor of (1 + `bers2.padding`) about its center. Thus holonomy values are computed beyond the range in which the interpolating functions will be sampled. The default value of 0.1 is reasonable.

- `bers2.contour_type`
  Type of Schwarzian equation integration contour. The integration API supports any system of piecewise C^2 contours. Currently the following types are available:

  - `spline_ellipse`
    Closed cubic splines approximating ellipses. This contour type is (always) used by the `bers` module.
  - `ellipse`
    Exact ellipses. This is sometimes a bit slower than `spline_ellipse` but is guaranteed to enclose the right set of singular points, even for extreme values of the parameters (e.g. $|\lambda| < 10^{-6}$).
  - `piecewise_linear`
    Rectangular contours with horizontal and vertical edges.

2.4.4 Functions

- `bers2.run()`
  Compute the holonomy.

2.5 `linear` - The Linear Slice Module

2.5.1 Purpose

The `linear` module generates an array of Markov triples corresponding to a linear slice of the representation variety in which one trace is fixed and the other varies through a square grid of values in $\mathbb{C}$.

2.5.2 Parameters

- `linear.lambda.real`
  `linear.lambda.imag`
  The real and imaginary parts of the trace that remains fixed throughout the linear slice when the slope is zero.

- `linear.center.real`
  `linear.center.imag`
  The real and imaginary parts of the center of the grid (i.e. the center value of the trace that varies across the slice).

- `linear.slope.real`
  `linear.slope.imag`
  The real and imaginary parts of the complex 'slope' of the linear slice. If the slope is zero, then lambda is truly fixed; otherwise both traces vary but where the ratio of differences from the center point is equal to the slope.

- `linear.radius`
  The Euclidean norm of the maximum deformation of the trace in each of the real and imaginary axes. Also equal to half the side length of the bounding square of trace values.

- `linear.size`
  The number of points in each row and column of the grid.

2.5.3 Functions

- `linear.run()`
  Generate the linear slice.

2.6 `anosov` - The Anosov Slice Module

This module is experimental and may change substantially in future releases.

2.6.1 Purpose

The `anosov` module computes Markov triples on the unstable manifold associated to a fixed point of a pseudo-anosov mapping class on the representation variety. It is also possible to search for fixed points and stable/unstable eigenvalues, though this Newton-type iteration is not very predictable for initial guesses far from a fixed point.

2.6.2 Parameters

- `anosov.p`, `anosov.q`
  The pseudo-anosov mapping class is specified via the numerator `p` and denominator `q` of the slope of the image of a (0,1)-curve on the torus. Specifically, the mapping class is calculated via the Farey word $w_{p/q}(A, B)$ where A and B are the horizontal and vertical Dehn twists. (Note: not all choices of p and q correspond to pseudo-anosov mapping classes.)

- `anosov.fx`, `anosov.fy`, `anosov.fz`
  A Markov triple fixed by the mapping class. (This can be specified directly or computed via the `find_fixed` function.)

- `anosov.sx`, `anosov.sy`, `anosov.sz`
  The stable eigenvector of the mapping class acting on the tangent space at the fixed point. (This can be specified directly or computed via the `find_fixed` function.)

- `anosov.ux`, `anosov.uy`, `anosov.uz`
  The unstable eigenvector of the mapping class acting on the tangent space at the fixed point. (This can be specified directly or computed via the `find_fixed` function.)

2.6.3 Functions

- `anosov.find_fixed`
  Attempt to find a pseudo-anosov fixed point starting from the triple (`anosov.fx`, `anosov.fy`, `anosov.fz`). If successful, reset these parameters and the eigenvectors accordingly.

- `anosov.run`
  Compute the holonomy for the unstable manifold.

2.7 `bowditch` – The Bowditch Discreteness Algorithm Module

2.7.1 Purpose

The `bowditch` module takes as input the Markov triples representing holonomy representations of a family of projective structures on punctured tori and attempts to determine which among these representations have discrete image. It does by searching for a *finite attractor* within the tree of generating triples for the holonomy group. Such an attractor is defined by the property that every edge of the tree points toward it, where each edge is given an orientation pointing toward the smaller trace. (Note that the endpoints of an edge correspond to triples that differ only in one generator.)

Bowditch has shown that if such an attractor exists, the set of all edges that satisfy a certain inequality is such an attractor; furthermore, this set is *always* connected. Thus the discreteness algorithm proceeds by searching for edges in the Bowditch subtree and declaring the representation discrete if the search terminates (within a certain time limit).

The inequality that defines the Bowditch locus involves a real parameter $t > 0$, and the corresponding subtree grows as t is increased.

2.7.2 Parameters

- `bowditch.maxsinkdepth`
  Search at most `bowditch.maxsinkdepth` steps for a sink in the tree of Markov triples. Larger values will result in fewer "undecided" points.

A large value here has little effect on resource consumption.

- `bowditch.maxdepth`
  Search to depth at most `bowditch.maxdepth` in the tree of Markov triples in an attempt to find a finite attracting subtree. Larger values will result in fewer "undecided" points.

  A large value here will greatly increase the stack space used by the program, and can lead to stack overflow.

- `bowditch.maxtrace`
  If a trace larger than `bowditch.maxtrace` in absolute value is encountered during the tree search, declare the associated representation "undecided".

- `bowditch.bowditch_threshold`
  Use the threshold $t =$`bowditch.bowditch_threshold` in looking for Bowditch's attracting subtree of Markov triples. This means that every egde in the subtree must originate or terminate in a trace of absolute value less than $3 + t$.

- `bowditch.jorgensen_threshold`
  Test for traces smaller than `bowditch.jorgensen_threshold` rather than applying the actual Jørgensen inequality. As long as `bowditch.jorgensen_threshold` < 1.0, this is a sufficient criterion for indiscreteness.

  Values slightly smaller than 1.0 may make the discreteness algorithm more tolerant of numerical error.

- `bowditch.extended`
  This parameter determines what data will be computed and stored each time the discreteness algorithm is run. If set to zero, only 'basic' discreteness data is computed, which consists of the following fields:

  - `depth` - the maximum depth reached in the search through the tree of Markov triples.
  - `result` - the result of the discreteness test (1 for discrete, -1 for indiscrete, other values indicate errors)

  If `bowditch.extended` is set to a nonzero value, additional (time-consuming) computations are performed and the following additional fields are available:

  - `sinks` - the number of sinks found during the search through the Markov tree (which is equal to the total number of sinks if the representation is discrete).
  - `nodes` - the total number of nodes examined during the search through the Markov tree (which is the size of the finite attractor if the representation is discrete).

2.7.3 Functions

- `bowditch.run()`
  Start discreteness testing.

2.8 `holonomy` – The Holonomy I/O Module

2.8.1 Purpose

The `holonomy` module writes holonomy data (Markov triples) and associated metadata to HDF5 data files.

2.8.2 Parameters

- `holonomy.filename`
  A string containing the filename (and optional internal path) to which the holonomy data will be written. For example,

  ```
  holonomy.filename = 'output.h5:mydataset'
  ```

  will write to a dataset named 'mydataset' in an HDF5 file called 'output.h5'; similarly,

  ```
  holonomy.filename = 'output.h5:outer/inner/mydataset'
  ```

  will write to a dataset named 'mydataset' contained in a group 'inner', which is in turn contained in a group 'outer' in an HDF5 file called 'output.h5'.

- `holonomy.file_mode`
  A string indicating the desired handling of existing output files; the possible values are:

  - 'create' – if the file already exists, generate an error.
  - 'truncate' – if the file already exists, truncate it and proceed.
  - 'update' – if the file already exists, open it and append data; dataset name conflicts will produce errors.

2.8.3 Functions

- `holonomy.write( data )`
  Use the current holonomy output settings to write holonomy data from the module `data` (e.g. `bers` or `linear`) to a file.

2.9 `discreteness` – The Discreteness Data I/O Module

2.9.1 Purpose

The `discreteness` module writes discreteness data and associated metadata to HDF5 data files.

2.9.2 Parameters

- `discreteness.filename`
  A string containing the filename (and optional internal path) to which the holonomy data will be written. For example,

  ```
  holonomy.filename = 'output.h5:mydiscdata'
  ```

  will write to a dataset named 'mydiscdata' in an HDF5 file called 'output.h5'; similarly,

  ```
  holonomy.filename = 'output.h5:outer/inner/mydiscdata'
  ```

  will write to a dataset named 'mydiscdata' contained in a group 'inner', which is in turn contained in a group 'outer' in an HDF5 file called 'output.h5'.

- `discreteness.file_mode`
  A string indicating the desired handling of existing output files; the possible values are:

  - 'create' – if the file already exists, generate an error.
  - 'truncate' – if the file already exists, truncate it and proceed.
  - 'update' – if the file already exists, open it and append data; dataset name conflicts will produce errors.

- `discreteness.output_method`
  A string indicating what discreteness data should be written to the output file. The first major distinction is that between *scalar* output modes, which result in an array of floating-point values, and *compound* output modes, which result in arrays of compound data types.

  The following scalar modes are supported:

  - 'scalar_depth' - if discrete, output value is `depth`; if indiscrete, output value is `(-1) * depth`.
  - 'scalar_nodes' - if discrete, output value is `nodes`; if indiscrete, output value is `(-1) * depth`. *For this output mode, extended discreteness testing is required.*

- 'scalar_sinks' - if discrete, output value is `sinks`; if indiscrete, output value is `(-1) * depth`.*For this output mode, extended discreteness testing is required.*
- 'scalar_mask' - if discrete, output value is 1; if indiscrete, output value is -1.

The following compound modes are supported:

- 'basic' - output all basic discreteness data.
- 'extended' - output all extended discreteness data. *For this output mode, extended discreteness testing is required.*

2.9.3 Functions

- `discreteness.write( data )`
  Use the current discreteness output settings to write discreteness data from the module `data` (e.g. `bowditch`) to a file.

Chapter 3

Mathematical Details

3.1 Parameterization of quadratic differentials

Bers slices for punctured tori are computed by numerically solving the Schwarzian differential equation

$$u" + \frac{1}{2}\phi u = 0$$

on the four-times punctured Riemann sphere that is commensurable with the punctured torus in question. The parameter λ specifies the punctured torus by means of the cross ratio of the four punctures of the commensurable punctured sphere.

In this way the space of holomorphic quadratic differentials on the punctured torus is identified with a complex affine subspace of the space of meromorphic quadratic differentials on the Riemann sphere. For the punctured torus with parameter λ, the affine space consists of differentials with poles of order 2 at each of the four punctures that are symmetric under the Mo"bius transformations that preserve the set of punctures.

Such a meromorphic differential can be expressed in the form

$$\phi(z) = \frac{1}{2z^2} + \frac{(\lambda-1)^2}{2(z-\lambda^2)(z-1)^2} + \frac{c}{z(z-1)(z-\lambda)}$$

for some complex number C. Note that $c = 0$ does not correspond to the Fuchsian uniformization of the punctured torus; the value of C with this property is called the "accessory parameter", and is difficult to calculate in general.

3.2 Computation of monodromy

Bear uses the GNU Scientific Library to numerically integrate the Schwarzian differential equation. Starting with a modular parameter λ and a complex number c specifying a quadratic differential, Bear produces a "Markov triple" (x, y, z) of traces that define the holonomy group via the following steps:

1. Compute contours.
   Ellipses encircling $\{0, \lambda\}$ and $\{1, \lambda\}$ are computed and then replaced with approximating periodic cubic splines (to avoid costly computation of trigonometric functions). Currently, the number of points used for the splines is fixed at compile-time, with a default of 16.

2. Integrate ODE.
   The linear second-order holomorphic ODE is converted to a first-order four-dimensional real system, which is then integrated along the contours computed above. GSL uses one of several standard integration algorithms, and attempts to maintain an absolute error of at most ε by adjusting the step size. This produces a pair of monodromy transformations M_1 and M_2.

3. Convert traces.
   The monodromy transformations correspond to the squares of generating holonomy elements because of the commensurability relationship between the punctured torus and the sphere. The traces of generators A, B, and AB are then computed from the matrix entries of M_1 and M_2 using the $SL_2(C)$ trace identities.

3.3 The discreteness algorithm

The discreteness algorithm used by Bear is based on the work of Brian Bowditch as described in his paper *Markoff triples and quasifuchsian Groups* (Preprint, University of Southampton). In this paper a certain class of representations of punctured torus groups into $PSL_2(C)$ is defined, and it is conjectured that this locus is exactly the set of quasifuchsian representations. It is also shown that the set of quasifuchsian representations is a connected component of this locus, and in particular that the boundary of the "Bowditch locus" contains the boundary of the quasifuchsian locus.

Bear proceeds as though the Bowditch conjecture holds, labeling a representation discrete (quasifuchsian) if it can be shown to lie in the Bowditch locus. On the other hand, the Jørgensen inequality is used as a sufficient condition for indiscreteness, so it is possible that some indiscrete representations belonging to the Bowditch locus are excluded. The table below summarizes the situation; unless limited by the number of computations allowed for a given representation, Bear reports the status of representations as follows:

	Bowditch	Non-Bowditch
Jørgensen holds	quasifuchsian	uncertain
Jørgensen violated	undefined	indiscrete

while conjecturally the reality is as follows:

	Bowditch	Non-Bowditch
Jørgensen holds	quasifuchsian	(nowhere dense set)
Jørgensen violated	(never occurs)	indiscrete

The Bowditch algorithm makes use of the infinite trivalent tree of Markov triples associated to the initial triple of traces (x, y, z) of generators A, B, and AB. In this tree, the three neighbors of a given triple are related as follows:

Triple	Neighbors
(x,y,z)	(x,y,xy-z)
	(x,xz-y,z)
	(yz-x,y,z)

Note that in each case, a triple differs from its neighbor by a single entry; the edges of the tree are then oriented so that the edge in which z is replaced by w, which we call the (z, w)-edge, is oriented toward whichever of the two has smaller absolute value. If $|z| = |w|$ then the edge can be oriented arbitrarily with affecting the algorithm.

A representation lies in the Bowditch locus if it has a "finite attractor", that is, a finite connected subtree Ω of markov triples such that every other edge in the tree points toward Ω. Bowditch shows that it is possible to determine whether or not a representation has this property using a simple algorithm; he defines a subtree $\tau(\varepsilon)$ where the (z, w)-edge belongs to $\tau(\varepsilon)$ if and only if the traces satisfy one of the following conditions:

1. $|z| < 3 + \varepsilon$ and $|w| < H(z)$
2. $|w| < 3 + \varepsilon$ and $|z| < H(w)$

Here H is a complicated function that has certain properties which imply that $\tau(\varepsilon)$ is always connected. Bowditch then shows that if a representation has a finite attractor, then for any positive ε, $\tau(\varepsilon)$ is a finite attractor.

Thus the algorithm proceeds as follows:

1. Check the Markov triple against the J/orgensen inequality, which states that each trace must have absolute value greater than or equal to one. As other Markov triples are generated, they are also checked, and if a violation is ever discovered, the representation is 'indiscrete'.

2. Follow oriented edges until a sink is found. (If no sink is found, then call the representation 'undecided'.)

3. Recursively search the tree starting from this sink, enumerating all edges that satisfy the Bowditch inequalities.

4. If only finitely many edges are found, the representation is (conjecturally) 'quasifuchsian'.

5. If the search proceeds for too long (exceeds a set depth, numerical overflow, etc.), then the representation is 'undecided'.