

## CREATING SOFTWARE FOR VISUALIZING KLEINIAN GROUPS

Yasushi Yamashita

*Department of Information and Computer Sciences  
Nara Women's University  
Kita-uoya-nishi-machi, Nara, 630-8506, Japan  
yamasita@ics.nara-wu.ac.jp*

We consider a simple class of Kleinian groups called once punctured torus groups. In this note, we will show how to create a computer program from scratch that can visualize fundamental sets and limit sets of the groups. No knowledge of computer programming is assumed.

### 1. Introduction

This note is based on six lectures given by the author at the summer school of “Geometry, Topology and Dynamics of Character Varieties” at Institute for Mathematical Sciences, National University of Singapore in the summer of 2010. The aim of the lecture was to show how to create a computer program from scratch that can visualize fundamental sets and limit sets of punctured torus groups. The main audience were graduate students and no knowledge of computer programming was assumed.

In low-dimensional topology and Kleinian group theory, computation and experiment become very important. There are many amazing software, and using them is not only helpful for study but fun. The author recommends [6] for this. In the lecture, we created our own computer program, starting with how to install the programming language *Python*. After introducing the basic grammar and rules of this programming language, we will make a program that can visualize the deformation of hyperbolic structures on a punctured torus and the limit set. This is a very simple version of M. Wada's software OPTi [13]. The size of our program is about 300 lines long. It is a simple and small program, but the user can deform the group by moving the mouse and save the picture in PostScript format.

The author hopes that this note helps the reader become familiar with the computer programming.

The paper is organized as follows. In §2 we begin by showing how to install the Python language system into the computer. Then, the programming language is introduced informally using many examples. In §3 we give background material for punctured torus groups and describe the algorithm which will be used in our program. In §4 we explain our program *OptPy* in detail. In §5 we list the source code of our program. In §6 we describe how to install and use it.

## 2. Python

This section gives a brief informal introduction to Python programming language. We refer to [11] for the in-depth introduction. It is a general purpose computer programming language. It is relatively easy to learn, and runs on Windows, Mac OS X, Linux. Also, it is free to use. The official website of Python is <http://www.python.org/>.

Python is already used in several software for hyperbolic geometry and related fields. The user interface to SnapPea was written by M. Culler and N. Dunfield in Python [4]. Its command-line interface is essentially Python's interactive shell that we will use later. See <http://www.math.uic.edu/~t3m/> for other Python libraries by the same authors. *Regina* by B. Burton is another example [3]. It is software for normal surface theory and has the ability to write and run scripts in Python so that it can perform repetitive tasks over large sets of data, such as SnapPea's census.

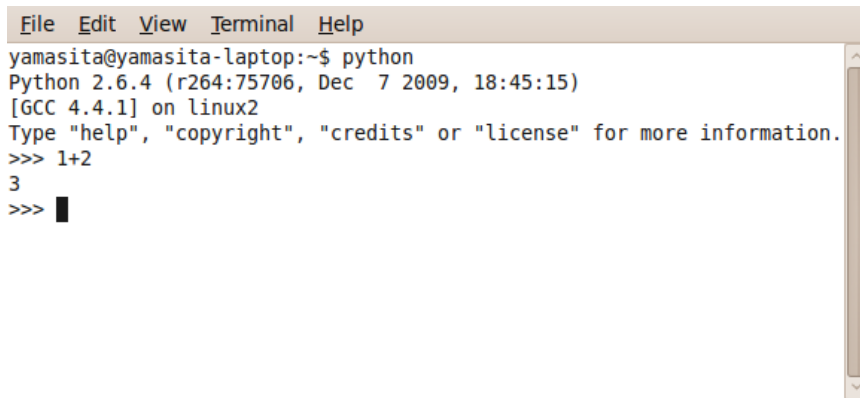
As mentioned in the introduction, we will begin by installing Python.

### 2.1. Python setup

This part gives information on the setup of the Python environment on different platforms.

**Mac OS X or Linux/Unix** Python comes pre-installed on most Linux distributions and Mac OS X. Start Python interactive shell with the command `python` in your terminal. Python will display version information and a prompt. See Figure 1. The prompt for Python interactive shell is `>>>` (and `...` for multi-line construct). After this prompt, you can type your Python program. In Figure 1, it is `1+2` and the result is `3`.

You can use your favorite editor to write a longer program. For example, emacs and vim support Python. To run the program, save the list into a



```
File Edit View Terminal Help
yamasita@yamasita-laptop:~$ python
Python 2.6.4 (r264:75706, Dec 7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+2
3
>>> █
```

Fig. 1. Python on Linux terminal.

file called, say, “sample.py”, and type

```
% python sample.py
```

in the terminal. Then you can see the result of the computation.

**Windows** You can download Python from the official web site

<http://www.python.org/download/>.

Choose **Python 2.X.X Windows Installer** in this page. (As of this note writing, the version number is 2.7.1. From now on, we write 2.7.1 instead of 2.X.X.) A file named `python-2.7.1.msi` will be saved. Then double click this file. The setup program will start and Python will be installed in the Windows system.

Next, we need an editor. In the lecture at the summer school, we used PyScripter [12]. This is an integrated development environment — a package consisting of editor, compiler and debugger. You can download it from <http://code.google.com/p/pyscripter/>. See Figure 2 for a screen shot. The upper right window is the editor. At the bottom window, you can run your Python program.

To test a short program, the interactive shell *IDLE*, which comes with the above Python 2.7.1 Windows Installer, is useful. You can start IDLE by **Start** → **Python 2.7** → **IDLE (Python GUI)**. See Figure 3. The prompt for Python interactive shell is `>>>` (and `...` for multi-line construct). After this prompt, you can type your Python program. In the Fig-

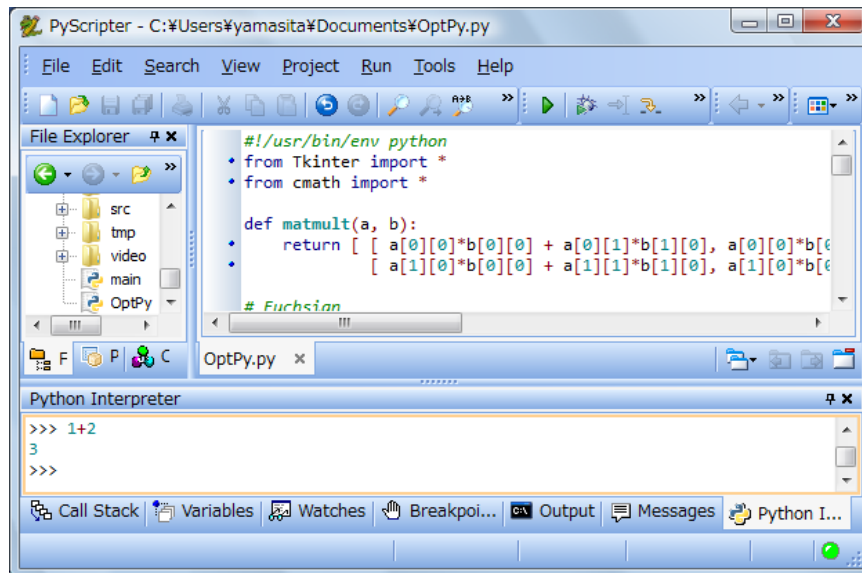


Fig. 2. PyScripter (IDE for Python).

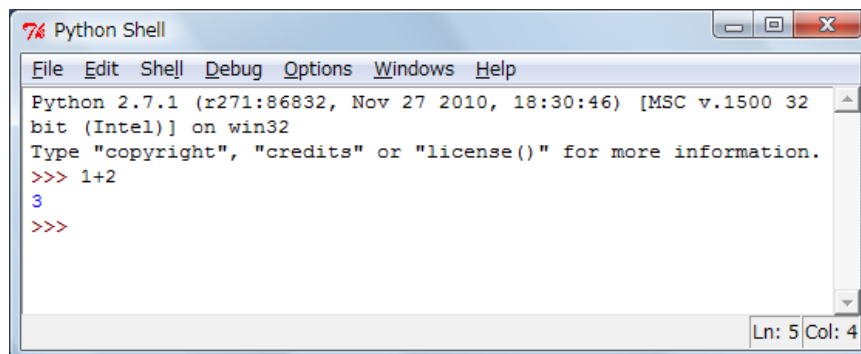


Fig. 3. IDLE (Python interactive shell).

ure 3, it is `1+2`, and the result is 3. Otherwise, you can use the bottom window of PyScripter.

**Remark 2.1.** In the summer school, we used the library *SciPy*. But, since some Mac OS X users have problems installing SciPy, we will use only the standard library which comes with python-2.7.1 and not SciPy.

## 2.2. Basic data types and operations

We describe the first steps of Python programming by showing the examples. The shaded boxes are what you see in the interactive shell. (Recall that `>>>` and `...` are the prompt.)

First of all, you can use it as a calculator.

```
>>> 1 + 2
3
>>> 3.4 * 5.6 - 7.8
11.239999999999998
```

**Remark 2.2.** The second result looks strange. It must be 11.24. This is called *representation error*. The input and output were in decimal (base 10) representation, but computers use binary (base 2) representation. The conversion between them causes this type of error.

In Python, the power  $2^{10}$  is written as `2 ** 10`. The complex number  $1.2 + 3.4\sqrt{-1}$  is written as `1.2+3.4j`. Note that `j` (not `i`) is used for  $\sqrt{-1}$ .

```
>>> 2 ** 10
1024
>>> (1.2 + 3.4j) * (5.6 + 7.8j)
(-19.800000000000001+28.399999999999999j)
```

You can assign a value to a variable. To get the real and imaginary part, type `.real` and `.imag` after the name of the variable. The function `abs` returns the absolute value of the number.

```
>>> a = 12.3 - 45.6j
>>> a
(12.300000000000001-45.600000000000001j)
>>> a.real
12.300000000000001
>>> a.imag
-45.600000000000001
>>> abs(a)
47.229757568719322
```

For later purpose, we want to use *cmath module*. Cmath module consists of many functions for complex numbers. To use it, type `from cmath import *`. The function `sqrt` returns the square root of the parameter and it is defined in cmath module. (The example below is continued from the previous shaded box.)

```
>>> from cmath import *
>>> sqrt(a)
(5.4557198227511341-4.179100236218277j)
```

To express a character or a string, use single-quotations.

```
>>> c = 'a'
>>> c
'a'
>>> s = 'abc'
>>> s
'abc'
```

Python contains so called *compound data types*. They are used to group together other values. Our first example is *tuple*. A tuple consists of a number of values separated by commas. Each component object can be accessed by offset (index).

```
>>> point = (300, 400)    # a tuple
>>> point[0]
300
>>> point[1]
400
```

A list is a series of objects in square brackets, separated by commas. The usage is almost similar to tuples.

```
>>> a = [2, 3, 5, 7, 11, 13]
>>> a[0]
2
>>> a[1] = 4
>>> a
[2, 4, 5, 7, 11, 13]
```

For the difference between tuple and list, see [11].

### 2.3. Control flow

There are three control flow statements in Python - `if`, `for` and `while`.

**If statement** The `if` statement is used for conditional execution. The syntax is as follows:

```
if <condition 1>:
    statements A
elif <condition 2>:
    statements B
elif <condition 3>:
    statements C
else:
    statements D
```

There can be zero or more `elif` parts, and the `else` part is optional. The body (statements) must be indented. If `condition 1` is satisfied, `statements A` is executed, and so on.

```
>>> x = 3
>>> if x > 0:
...     print "x is positive"
...     print x
... elif x == 0:
...     print "x is zero"
... else:
...     print "x is negative"
...     print x
...
x is positive
3
>>> tr = 2+3j
>>> if tr == 2 or tr == -2:
...     print "parabolic"
... elif tr.imag == 0 and (-2 < tr.real and tr.real < 2):
...     print "elliptic"
... else:
...     print "loxodromic"
...
loxodromic
```

The standard comparison operators are `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to). Note that `=` is used for assignment and `==` for equality.

**For statements** The `for` statement is used for sequential looping. The `for` statement iterates over the items of a list, in the order that they appear in the list. The syntax is as follows:

```
for <variable(s)> in <list>:
    statements
```

Each item of `list` is assigned to the `variable(s)` in turn, and then `statements` is executed.

```
>>> for x in [0, 1, 2, 3]:
...     print x
...
0
1
2
3
>>> for i, j, k in [(0,1,2), (1,2,0), (2,0,1)]:
...     print "i=", i
...     print "j=", j
...     print "k=", k
...
i= 0
j= 1
k= 2
i= 1
j= 2
k= 0
i= 2
j= 0
k= 1
```

The built-in function `range()` generates a list of integers.

```
>>> range(-3, 3)
[-3, -2, -1, 0, 1, 2]
>>> for i in range(-3, 3):
```



```
...     print i
...
-3
-2
-1
0
1
2
```

**While statements** The `while` statement is used for conditional looping. The syntax is as follows:

```
while <condition>:
    statements
```

The `while` statement repeatedly executes the block of statements as long as the condition is true.

```
>>> x = -3
>>> while x < 3:
...     print x
...     x = x+1
...
-3
-2
-1
0
1
2
```

If you want to interrupt a `for` or a `while` loop, you can use the statement `break`.

```
>>> x = -3
>>> while x < 3:
...     print x
...     if x == 0:
...         break
...     x = x+1
...
```

```
-3  
-2  
-1  
0
```

## 2.4. Function

We have seen several built-in functions, like `abs()`, `sqrt()`. To create our own function, write as follows:

```
def <function name> ( <parameters> ):  
    statement 1  
    statement 2  
    ...  
    return <expression>
```

The definition of the function starts with the key word `def`. The parameters passed to the function must be written within the parentheses. Note that the first line ends with a colon (:). The body (`statement 1`, `statement 2`, ...) of the function must be indented. The line `return <expression>` is optional. As an example, let us define a Möbius function  $T(z) = (2z + 3)/(4z + 5)$ . The name of the function is `T` and the name of the parameter is `z`. To call this function with parameter 7, type `T(7.0)`, like built-in functions.

```
>>> def T(z):  
...     w = (2*z+3)/(4*z+5)  
...     return w  
...  
>>> T(7.0)  
0.51515151515151514
```

The next example decides whether the input is prime number or not.

```
>>> def isprime(n):  
...     i = 2  
...     while i*i <= n:
```

```
...         if n % i == 0:
...             return False
...         i = i+1
...     return True
...
```

Here are more examples of functions.

```
>>> def norm(z):
...     return z.real*z.real + z.imag*z.imag
...
>>> def nextstx(st, x, ac = 1):
...     st = st + x[0]/(x[1]*x[2])
...     x = (x[1]*x[2] - x[0], x[2], x[1])
...     return st, x
...
>>> a,b = nextstx(2, (3,4,5), 6)
>>> a
2
>>> b
(17, 5, 4)
```

The second function `nextstx` returns two values (`st`, `x`). When it is called like `a,b = nextstx(2, (3,4,5), 6)`, the value of `st` in this function is assigned to `a` and the value of `x` is assigned to `b`. This function has the *default parameter value* for the third parameter `ac`. If you type `nextstx(2, (3,4,5))` and the third argument is omitted, `ac` becomes 1.

## 2.5. Global variable and local variable

In this subsection, we consider the variable's visibility within a program. In Python, there are two kinds of variables — *global variables* and *local variables*. Global variables are accessible inside and outside of functions. Local variables are only accessible inside the function. If a variable is assigned a value outside the functions, it is a global variable. If a variable is assigned a value in a function, it is a local variable.

```
>>> x = 10          # this "x" is global
>>> def f():
...     print x     # get the value of the global variable "x"
...
>>> f()
10
>>> def g():
...     x = 20      # this "x" is local
...     print x
...
>>> g()
20
>>> x              # this "x" is global
10
```

The variable `x` in the function `g()` above is local, because its value is set in this function, even though the global variable with the same name `x` already exists. To assign a global variable a value from within a function, `global` statement is needed.

```
>>> x = 10          # this "x" is global
>>> def f():
...     global x    # global statement
...     x = 20      # this "x" is global
...     print x
...
>>> x
10
>>> f()
20
>>> x              # the value of the global variable "x" is changed.
20
```

## 2.6. GUI

To open a window and draw lines and circles on it, we use *Canvas* in *Tkinter* module. In the following example, we

- (1) declare that we will use 'Tkinter module',

- (2) create a canvas with width 400 pixel and height 250 pixel, and assigned it to a variable named `c`.
- (3) draw a line segment from (100, 50) to (300, 200) on `c`, and
- (4) draw an oval in a square with upper left=(200,30) and lower right = (300, 130).

```
>>> from Tkinter import *
>>> c = Canvas(width=400, height=250)
>>> c.create_line(100, 50, 300, 200)
>>> c.create_oval(200, 30, 300, 130)
>>> c.focus_set()
>>> c.pack()
>>> c.mainloop()
```

See Figure 4 for the result of running this example. Note that, in the co-

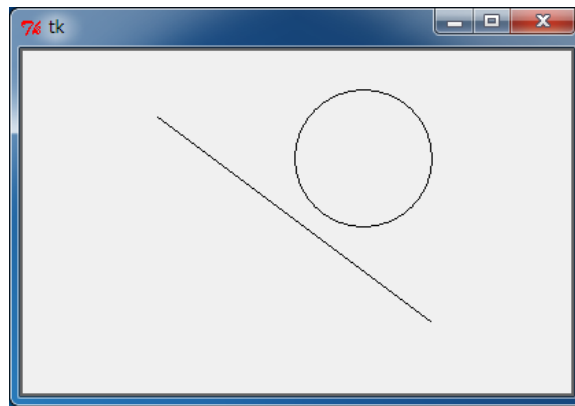


Fig. 4. TkInter.

ordinate system of `Canvas`, (0,0) is the upper left corner. The one unit in this coordinate system is pixel on the screen.

Next, we want to capture the mouse move in our canvas. In the example below, when the (left) mouse button (`Button-1`) is clicked, the function `handler_b1` is called with one parameter which describes this event.

We first define the function `handler_b1`. This is a usual function and takes one parameter named `event` as input and prints `event.x` and

`event.y`. Here, `print` is a built-in function. Connecting *mouse press event* and `handler_b1` is done by the line `c.bind("<Button-1>", handler_b1)`.

```
>>> def handler_b1(event):
...     print "mouse is at (", event.x, ",", event.y, ")"
...     return
...
>>> c = Canvas(width=800, height=600)
>>> c.bind("<Button-1>", handler_b1)
'44804936handler_b1'
>>> c.focus_set()
>>> c.pack()
>>> c.mainloop()
mouse is at ( 166 , 127 )
mouse is at ( 109 , 138 )
```

After `c.mainloop()` is called, the Python system takes care of this canvas, waiting mouse event. When the left mouse button is pressed, the Python system calls `handler_b1` with one parameter putting `x` and `y` coordinate of the mouse in it.

**Remark 2.3.** Some of the data types can contain a number of data. Recall that complex number type contains `real` and `imag`, so that you can write like `z.imag` and `z.imag` if a complex value is assigned to the variable `z`. The event type used in the previous example contains `x` and `y`. The Canvas type contains even more — functions, like `bind()`, `create_line()`, etc. They are called *classes*. and provide the features of *Object Oriented Programming*. See [11].

## 2.7. An example

The following is a sample python program using functions (`create_line`, `paint`, `handler.button`), GUI (`canvas`), global variables (`p1`, `p2`, `canvas`), and an if statement.

```
1: #!/usr/bin/env python
2: from cmath import *
3: from Tkinter import *
4: p1 = (333, 300)
5: p2 = (466, 300)
6: canvas = Canvas(width=800, height=600)
7:
```

```

8: def create_line(z1, z2, color="black"):
9:     x1 = z1.real*400.0+200.0
10:    y1 = -(z1.imag*400.0)+300.0
11:    x2 = z2.real*400.0+200.0
12:    y2 = -(z2.imag*400.0)+300.0
13:    canvas.create_line(x1, y1, x2, y2, fill=color)
14:    canvas.create_oval(x1-10, y1-10, x1+10, y1+10, fill=color)
15:    canvas.create_oval(x2-10, y2-10, x2+10, y2+10, fill=color)
16:
17: def paint():
18:     canvas.delete(ALL)
19:     q1 = (p1[0] - 200.0)/400.0 - (p1[1] - 300.0)/400.0*(0+1j)
20:     q2 = (p2[0] - 200.0)/400.0 - (p2[1] - 300.0)/400.0*(0+1j)
21:     create_line(0, q1, "red")
22:     create_line(q1, q2, "red")
23:     create_line(q2, 1, "red")
24:
25: def handler_button (event):
26:     global p1, p2
27:     if ((p1[0]-event.x)**2 + (p1[1]-event.y)**2 <
28:         (p2[0]-event.x)**2 + (p2[1]-event.y)**2):
29:         p1 = (event.x, event.y)
30:     else:
31:         p2 = (event.x, event.y)
32:     paint()
33:
34: canvas.bind("<Button-1>", handler_button)
35: canvas.bind("<B1-Motion>", handler_button)
36: canvas.focus_set()
37: canvas.pack(expand=YES, fill=BOTH)
38: paint()
39: canvas.mainloop()

```

To run the program, save the list into a file called, say, “sample.py”, and double click the icon of the file (Windows) or type

```
% python sample.py
```

in the terminal (Mac OS X or Linux. Here, % is a prompt). Then a window appears (Figure 5 left). The user can move the middle vertices of degree two by mouse (Figure 5 right).

The overall structure of the program is the same as our final program *OptPy* in §5. It defines global variables, defines functions, calls several `canvas...` to set up the canvas, and stays in `canvas.mainloop()` forever. When the mouse is clicked or moved, `handler_button` is called. It

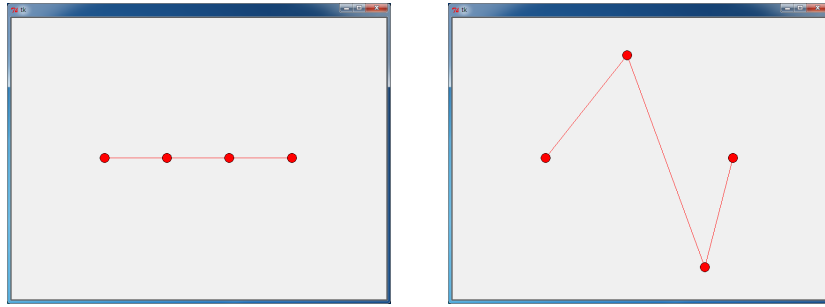


Fig. 5. Left: The initial window. Right: After some mouse movement.

change the value of the global variable `p1` or `p2` (location of the vertices) and calls `paint`. Using the value of the global variables, `paint` repaints the window.

### 3. Once Punctured Torus Group

As mentioned in the introduction, we will make a very simple version of Wada's OPTi. In this section, we review T. Jørgensen's theory on once punctured torus groups. See [1] and [7] for detail.

#### 3.1. The deformation space of once-punctured torus groups

Let  $T$  be a once punctured torus. Its fundamental group  $\pi_1(T)$  is the free group of rank two  $F_2 = \langle X, Y \rangle$ , generated by two elements  $X$  and  $Y$  corresponding to simple closed curves on  $T$  with geometric intersection number one. The  $\mathrm{SL}(2, \mathbb{C})$  character variety  $\mathcal{X} = \mathrm{Hom}(\pi_1(T), \mathrm{SL}(2, \mathbb{C})) // \mathrm{SL}(2, \mathbb{C})$  is identified with  $\mathbb{C}^3$ . The identification is given by

$$\iota : [\rho] \mapsto (x, y, z) = (\mathrm{tr} \rho(X), \mathrm{tr} \rho(Y), \mathrm{tr} \rho(XY)).$$

A  $\mathrm{SL}(2, \mathbb{C})$  representation  $\rho$  of  $\pi_1(T)$  is called *type preserving* if  $\rho(XYX^{-1}Y^{-1}) = -2$ . The set of characters which corresponds to type preserving representations is

$$\mathcal{X}_{-2} = \{(x, y, z) \in \mathcal{X} \mid x^2 + y^2 + z^2 = xyz\}.$$

$\mathcal{X}_{-2}$  is the *deformation space* of hyperbolic structures on once punctured torus.

We want to see the deformation by visualizing a fundamental domain of the corresponding representation. We refer to [7] for the pioneering work.



Later, it is studied by several people. See [1], [5] and [10] for example. As mentioned before, Wada made nice computer software named OPTi [13] which visualizes the fundamental domain. The user can deform the representation interactively by moving the mouse, and it shows the limit set, too. In the next section, we will create a very simplified version of Wada's OPTi by using Python introduced in the previous section.

### 3.2. Ford domain

We describe a type of fundamental domain, called Ford domain, which we will visualize.

For  $T(z) = (az + b)/(cz + d)$  with  $T(\infty) \neq \infty$  (that is  $c \neq 0$ ), the *isometric circle*  $Ic(T)$  of  $T$  is

$$Ic(T) := \{z \in \mathbb{C} \mid |T'(z)| = 1\} = \{z \in \mathbb{C} \mid |cz + d| = 1\}$$

and the *isometric hemisphere*  $Ih(T)$  of  $T$  is the hyperplane of the upper half-space  $\mathbb{H}^3$  bounded by  $Ic(T)$ . The center is  $-d/c$  and the radius is  $1/|c|$ .  $Eh(T) \subset \mathbb{H}^3$  denotes the closure of the exterior of the  $Ih(T)$ . Let  $\Gamma$  be a non-elementary Kleinian group such that the stabilizer  $\Gamma_\infty$  of  $\infty$  consists of parabolic transformations. The (*extended*) *Ford domain*  $F(\Gamma)$  of  $\Gamma$  is defined as follows:

$$F(\Gamma) = \bigcap \{Eh(A) \mid A \in \Gamma - \Gamma_\infty\}.$$

Then,  $F(\Gamma)$  is a “fundamental polyhedron modulo  $\Gamma_\infty$ ”.

### 3.3. Jørgensen's normalization

Let  $(x, y, z)$  be an element of  $\mathcal{X}_{-2}$  with  $y \neq 0$ . Let  $\rho$  be the representation  $\rho : \pi_1(T) \rightarrow \mathrm{SL}(2, \mathbb{C})$  defined by the following:

$$\rho(X) = \begin{pmatrix} x - z/y & x/y^2 \\ x & z/y \end{pmatrix}, \quad \rho(XY) = \begin{pmatrix} y & -1/y \\ y & 0 \end{pmatrix}, \quad (3.1)$$

$$\rho(Y) = \begin{pmatrix} z - x/y & -z/y^2 \\ -z & x/y \end{pmatrix}. \quad (3.2)$$

Then,  $\rho$  is type preserving and  $\iota([\rho]) = (x, y, z)$ . It is easy to see that  $\rho(XYX^{-1}Y^{-1}) = \begin{pmatrix} -1 & -2 \\ -0 & -1 \end{pmatrix}$ . Let  $C(T)$  and  $R(T)$  denote the center and the radius of  $Ic(\rho(T))$  respectively. Then, by direct calculation, we have

that

$$\begin{aligned}
 C(Y^{-1}) &= -y/xz - z/xy, & R(Y^{-1}) &= 1/|z| \\
 C(X) &= -z/xy, & R(X) &= 1/|x| \\
 C(XY) &= 0, & R(XY) &= 1/|y| \\
 C(Y) &= x/yz, & R(Y) &= 1/|z| \\
 C(X^{-1}) &= x/yz + y/zx, & R(X^{-1}) &= 1/|x| \\
 C((XY)^{-1}) &= 1 & R((XY)^{-1}) &= 1/|y|.
 \end{aligned}$$

Define  $a_1 = x/yz$ ,  $a_2 = y/zx$  and  $a_3 = z/xy$ . Then we have  $a_1 + a_2 + a_3 = 1$ . Since  $\rho(XYX^{-1}Y^{-1})(z) = z + 2$ , the above calculation shows that there are isometric hemispheres with (center, radius):

$$(n, 1/|y|), (n + a_1, 1/|z|), (n + a_1 + a_2, 1/|x|) \quad (\text{for any } n \in \mathbb{Z}). \quad (3.3)$$

The circles in (3.4) are a part of the set of all isometric circles of  $\Gamma - \Gamma_\infty$  that we (may) want to draw.

We will consider the “broken line” (or “polyline”) given by connecting the consecutive centers of (3.4). That is, the union of line segments

$$\cup_{n \in \mathbb{Z}} \{L(n, n + a_1), L(n + a_1, n + a_1 + a_2), L(n + a_1 + a_2, n + 1)\}, \quad (3.4)$$

where  $L(z_1, z_2)$  is the line segment from  $z_1$  to  $z_2$  in  $\mathbb{C}$ . See “red broken line” in Figure 6.

Now, define “moves” from  $(x, y, z)$  as

$$(x_1, y_1, z_1) := (yz - x, z, y) \quad (3.5)$$

$$(x_2, y_2, z_2) := (z, zx - y, x) \quad (3.6)$$

$$(x_3, y_3, z_3) := (y, x, xy - z) \quad (3.7)$$

and

$$a_{i,1} := x_i/y_i z_i, \quad a_{i,2} := y_i/z_i x_i, \quad a_{i,3} := z_i/x_i y_i \quad (i = 1, 2, 3).$$

Then  $(x_i, y_i, z_i)$  is also in  $\mathcal{X}_{-2}$  and we have  $a_{i,1} + a_{i,2} + a_{i,3} = 1$ . For  $i = 1, 2, 3$ , there are isometric hemispheres with (center, radius):

$$(s_i + n, 1/|y_i|), (s_i + n + a_{i,1}, 1/|z_i|), (s_i + n + a_{i,1} + a_{i,2}, 1/|x_i|), \quad (3.8)$$

for any  $n \in \mathbb{Z}$ , where  $s_1 = a_1$ ,  $s_2 = a_1 + a_{1,2}$ ,  $s_3 = -a_3$ . Also, we will consider the union of line segments connecting these centers:

$$\begin{aligned}
 \cup_{n \in \mathbb{Z}} \{ & L(s_i + n, s_i + n + a_{i,1}), L(s_i + n + a_{i,1}, s_i + n + a_{i,1} + a_{i,2}), \\
 & L(s_i + n + a_{i,1} + a_{i,2}, s_i + n + 1)\}. \quad (3.9)
 \end{aligned}$$

For  $(x_i, y_i, z_i)$ , we can do the same “move” again, and this gives another infinite family of isometric hemispheres.

There is an action of the outer automorphism group  $\text{Out}(F_2)$  of  $F_2$  on  $\mathcal{X} \cong \mathbb{C}^3$  and  $\mathcal{X}_{-2}$  is invariant under this action.  $\text{Out}(F_2)$  can be generated by three elements which induce polynomial automorphisms (3.5), (3.6), (3.7).

Let  $\mathcal{F}$  be the Farey tessellation (triangulation) of  $\mathbb{H}^2$ . Each vertex corresponds to an isotopy class of simple nonperipheral curves on  $T$  and parameterized by  $\widehat{\mathbb{Q}} := \mathbb{Q} \cup \{1/0\}$ . The Markoff map  $\phi_{(x,y,z)} : \widehat{\mathbb{Q}} \rightarrow \mathbb{C}$  associated with  $(x, y, z)$  is defined by the following conditions:

$$\begin{aligned} \phi_{(x,y,z)}(1/0) &= x, & \phi_{(x,y,z)}(0/1) &= y, & \phi_{(x,y,z)}(1/1) &= z, \\ \phi_{(x,y,z)}(p) \times \phi_{(x,y,z)}(q) &= \phi_{(x,y,z)}(r) + \phi_{(x,y,z)}(s), & (p, q, r, s) &\in \widehat{\mathbb{Q}} \end{aligned}$$

where  $p, q, r$  and  $p, q, s$  correspond to a pair of adjacent triangles in  $\mathcal{F}$ . If  $p, q, r \in \widehat{\mathbb{Q}}$  correspond to a triangle in  $\mathcal{F}$ , then the triple of complex numbers  $(\phi_{(x,y,z)}(p), \phi_{(x,y,z)}(q), \phi_{(x,y,z)}(r))$  is an orbit of  $(x, y, z)$  under the action of  $\text{Out}(F_2)$ . For each triangle  $p, q, r$  in  $\mathcal{F}$ , an infinite family of isometric circles and an infinite broken line is defined using induction ((3.3), (3.4), (3.8), (3.9)). By abusing notation, we write  $C(\phi_{(x,y,z)}(p), \phi_{(x,y,z)}(q), \phi_{(x,y,z)}(r))$  for the circles and  $L(\phi_{(x,y,z)}(p), \phi_{(x,y,z)}(q), \phi_{(x,y,z)}(r))$  for the broken line.

### 3.4. Jørgensen’s method to construct the Ford domain

The definition of the Ford domain says that we need to consider all the elements in  $\Gamma - \Gamma_\infty$ . But, most of the isometric hemispheres are “hidden” under other isometric hemispheres. Jørgensen [7] gave an algorithm which tells us how to find the “visible” ones, using the normalization in the previous subsection. The works in [1], [8], [13] are based on this theory. We outline this algorithm.

The input is  $(a_1, a_2, a_3)$ . Then we calculate  $x = 1/\sqrt{a_2 a_3}$ ,  $y = 1/\sqrt{a_3 a_1}$ ,  $z = 1/\sqrt{a_1 a_2}$ .

**Remark 3.1.** We have to be careful which square root (for example  $\pm\sqrt{a_1 a_2}$ ) we choose. This is done in our program in §5.

**Step 1 (Go to “Sink”).** We consider three cases.

- If  $|yz - x| < |x|$ , change  $(x, y, z)$  to  $(yz - x, z, y)$ .
- If  $|zx - y| < |y|$ , change  $(x, y, z)$  to  $(z, zx - y, x)$ .
- If  $|xy - z| < |z|$ , change  $(x, y, z)$  to  $(y, x, xy - z)$ .

Then, do this step 1 again until all three will fail.

**Step 2 (Draw Isometric Circles).** Draw isometric circles for  $C(x, y, z)$ , as in (3.4) or (3.9).

We say that  $x$  (resp.  $y, z$ ) is *active* if  $|x| > |y + zi|$  (resp.  $|y| > |z + xi|$ ,  $|z| > |x + yi|$ ).

- (1) If  $x$  is active and  $y$  and  $z$  are not active,  
change  $(x, y, z)$  to  $(yz - x, z, y)$ .
- (2) If  $y$  is active and  $z$  and  $x$  are not active,  
change  $(x, y, z)$  to  $(z, zx - y, x)$ .
- (3) If  $z$  is active and  $x$  and  $y$  are not active,  
change  $(x, y, z)$  to  $(y, x, xy - z)$ .
- (4) If all  $x, y$  and  $z$  are not active, go to next step
- (5) For the case where two of  $x, y, z$  is active and one is not active, there is an algorithm, but we omit here. (This is done in lines 68–81 in §5.)

Then, do this step 2 again until case (4) will happen.

Now, we set  $(x, y, z)$  as given at the end of step 1.

**Step 3 (Draw Isometric Circles).** Draw isometric circles for  $C(x, y, z)$  as in (3.4) or (3.9). This step is almost the same as step 2 except one sign in the following definition of active. We say that  $x$  (resp.  $y, z$ ) is *active* if  $|x| > |y - zi|$  (resp.  $|y| > |z - xi|$ ,  $|z| > |x - yi|$ ).

- (1) If  $x$  is active and  $y$  and  $z$  are not active,  
change  $(x, y, z)$  to  $(yz - x, z, y)$ .
- (2) If  $y$  is active and  $z$  and  $x$  are not active,  
change  $(x, y, z)$  to  $(z, zx - y, x)$ .
- (3) If  $z$  is active and  $x$  and  $y$  are not active,  
change  $(x, y, z)$  to  $(y, x, xy - z)$ .
- (4) If all  $x, y$  and  $z$  are not active, finish the process.
- (5) For the case where two of  $x, y, z$  is active and one is not active, there is an algorithm, but we omit here. (This is done in lines 68–81 in §5.)

Then, do this step 3 again until case (4) will happen.

**Remark 3.2.** (1) By Jørgensen’s inequality, if the absolute value of one of  $x, y, z$  becomes smaller than 1, we can stop our process and say that “indiscrete”.

(2) Jørgensen’s theory [7] tells us that if the group is quasi-fuchsian, then there exists a consecutive sequence of triangles  $S = \{s_1, s_2, \dots, s_m\}$  in the Farey tessellation  $\mathcal{F}$  such that the corresponding family of isometric hemi-

spheres is equal to the boundary of the extended Ford domain  $F(\rho(F_2))$  of the group  $\rho(F_2)$  (Jørgensen's normalization), and the corresponding family of broken lines is the dual of the images of the boundaries (edges) of the faces of  $F(\rho(F_2))$  projected to the ideal boundary  $\mathbb{C}$ . In **Step 1**, we try to find some triangle  $t_i \in S$ . In **Step 2** and **Step 3**, we have found triangles  $\{t_{i-1}, \dots, t_1\}$  and  $\{t_{i+1}, \dots, t_m\}$ .

(3) If the infinite broken line  $L(x, y, z)$  has a self intersection, then we stop our process. Typically, this happens when the group is indiscrete. But, this may happen even if the group is discrete, because the calculations in **Step 1** is not enough to find a triangle  $t_i \in S$  mentioned above. The sink may not be unique in  $\mathcal{F}$ . Moreover, we don't know that  $S$  always contains a sink. In order to avoid this, we must change the step 1 using Bowditch's arguments. In [2], section 3, the notion  $T(t)$  was introduced to show that  $\Phi_Q$  (a subset of  $\mathcal{X}$ ) is open.  $T(t)$  is a connected subtree of the dual graph of  $\mathcal{F}$ . If  $\rho(F_2)$  is quasi-fuchsian, then  $[\rho]$  is in  $\Phi_Q$  and, in this case, it was proved that  $T(t)$  is finite and there is an algorithm which finds all the vertices in  $T(t)$ . We can find a triangle in  $S$  (starting point of **Step 2** and **Step 3**) from  $T(t)$ .

### 3.5. Limit set

The limit set  $\Lambda$  of a Kleinian group  $\Gamma$  is the set of all accumulation points of the orbits  $\Gamma v$  for any  $v \in \mathbb{H}^3$ .  $\Lambda$  is a closed subset of the ideal boundary  $\widehat{\mathbb{C}}$ . If our group  $\rho(F_2)$  is a quasi-fuchsian Kleinian group,  $\Lambda$  is a Jordan curve.

For the calculation, we will use matrices  $\rho(X)$  and  $\rho(Y)$  in (3.1, 3.2) and their inverses, Fix some point  $v \in \mathbb{H}^3$ . We will plot  $(\rho(W))(v)$  for sufficiently long reduced words  $W$  over  $\{X, X^{-1}, Y, Y^{-1}\}$ .

## 4. OptPy

In this section, we will take a closer look at our example Python program. We named it "OptPy"—Once Puncture Torus Python. The program is listed in the next section. The reader can download the program from [14]. See Figure 6 for a screen shot. The circles are isometric circles and the blue points are the limit set.

At line 33, we define `canvas` with size  $800 \times 600$  pixel. (See 2.6.) When started, we consider that this canvas corresponds to  $[-0.5, 1.5] \times [-0.75, 0.75]$  in x-y coordinate. This is realized by the variables on lines 29–31 and two functions `create_line` (l. 87) and `create_circle` (l. 95). `create_line` takes two complex numbers `z1` and `z2` as inputs and draw

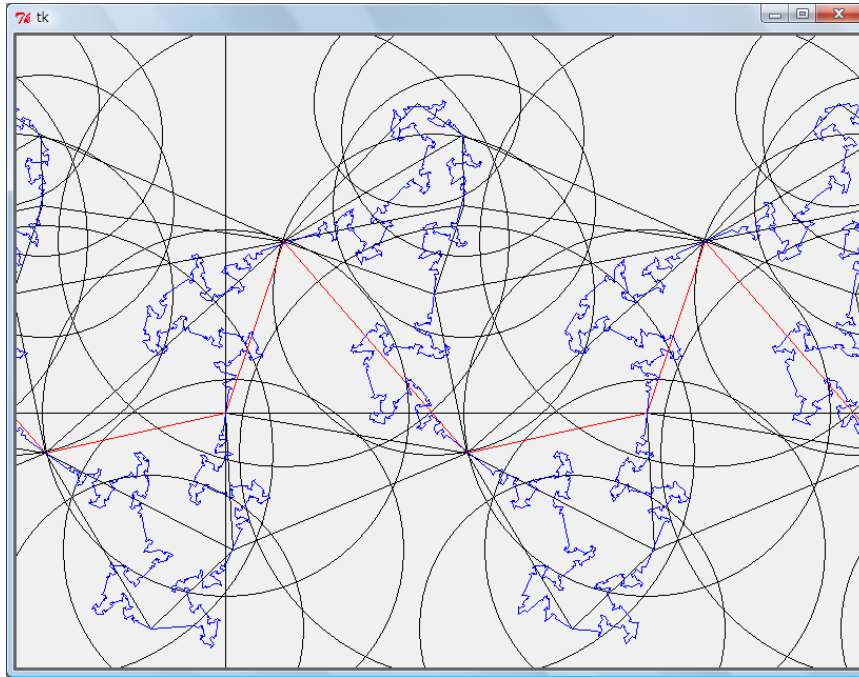


Fig. 6. OptPy.

the line segment connecting them. The arguments of `create_circle` are a complex number `center` and a real number `radius`.

By clicking or moving the mouse, the user can change  $a_1 = x/yz$  and  $a_1 + a_2 = x/yz + y/zx$  which results in the deformation of the group. The original position of these points are specified at line 13 and 14 in pixel coordinates. `p1` is the end point corresponding to  $a_1$  and `p2` is the end point corresponding to  $a_1 + a_2$ . The comments on lines 16–27 give other (hopefully interesting) starting points.

`I` (l. 31) is used as  $\sqrt{-1}$  (l. 32).

`oldpt` (l. 33) and `draw_limitset` (l.35) is used for drawing limit set.

The function `norm` (l. 37) returns  $|x|^2$  of complex number `x`.

The function `transition1` (l. 41) is used for step 1. The argument `x` is assumed to be a tuple of three elements (triple) which corresponds to  $(x, y, z)$ . It returns

- $-2$  if the group is indiscrete
- $0$ , (resp.  $1, 2$ ) if  $x$  (resp.  $y, z$ ) is active.

- $-1$  if there is no active, that is step 1 is finished.

`transition2` (l. 53) is used for step 2 and 3. The argument `x` is assumed to be a tuple of three elements (triple) which corresponds to  $(x, y, z)$ . The next argument `direction` is set to be 1 when in step 2 and  $-1$  when in step 3. It returns

- $-2$  if the group is indiscrete.
- $-3$  if there is a self intersection.
- 0, (resp. 1, 2) if  $x$  (resp.  $y, z$ ) is active.
- $-1$  if there is no active, that is step 1 is finished.

`create_lines_and_circles` (l. 107) is used for step 2 and 3 to draw broken lines and isometric circles.

`next_stx` takes a triple `x` and returns the new triple if active = `ac`.

`limit_set` is used to draw limit set. It is done by fairly simple recursive calls. See [9] for more detail.

The function `paint` (l. 165) is the most important. This function draws everything.

- clears the canvas (l. 168).
- draws axis (l. 170–171).
- calculate the triple `x` ( $= (x, y, z)$ ) from `p1` and `p2` (l. 173–180).
- step 1 (l. 182–194).
- recalculate `x` and `st` (l. 196–204).
- step 2 (l. 209–216).
- step 3 (l. 218–228).
- draw limit set (l. 233–240).

After the line 243 are the handlers. See §2.6. Each handlers change some global variable and calls `paint()`.

## 5. Program List

```
1: #!/usr/bin/env python
2: # 31 Jan, 2011, Yasushi Yamashita
3:
4: from Tkinter import *
5: from cmath import *
6:
7: def matmult(a, b):
8:     return [ [ a[0][0]*b[0][0] + a[0][1]*b[1][0],
9:               a[0][0]*b[0][1] + a[0][1]*b[1][1] ],
```

```

10:          [ a[1][0]*b[0][0] + a[1][1]*b[1][0],
11:            a[1][0]*b[0][1] + a[1][1]*b[1][1] ] ]
12:
13: # Fuchsian
14: p1 = (333, 300)
15: p2 = (466, 300)
16: # singly cusped
17: # p1 = (333.3333, 300)
18: # p2 = (466.6667, 150.9290)
19: # doubly cusped
20: # p1 = (400, 100)
21: # p2 = (500, 200)
22: # singly degenerate
23: # p1 = (279.3056, 149.2277)
24: # p2 = (490.6948, 92.1941)
25: # doubly degenerate
26: # p1 = (400, 184.5300)
27: # p2 = (400, 415.4700)
28:
29: origin_x = 200.0
30: origin_y = 300.0
31: scale    = 400.0
32: I = 0+1j   # square root of -1
33: oldpt = 100 # used for drawing limit set
34: canvas = Canvas(width=800, height=600)
35: draw_limitset = True
36:
37: def norm(x):
38:     return x.real*x.real + x.imag*x.imag
39:
40: # Step 1 (Subsection 3.4)
41: def transition1(x):
42:     if norm(x[0]) < 1 or norm(x[1]) < 1 or norm(x[2]) < 1:
43:         return -2
44:     elif norm(x[0]) > norm(x[1]*x[2]-x[0]):
45:         return 0
46:     elif norm(x[1]) > norm(x[2]*x[0]-x[1]):
47:         return 1
48:     elif norm(x[2]) > norm(x[0]*x[1]-x[2]):
49:         return 2
50:     else:
51:         return -1
52:
53: # Step 2(direction=1), 3(direction=-1) (Subsection 3.4)
54: def transition2(x, direction=1):
55:     # if not discrete

```



```

56:  if norm(x[0]) < 1 or norm(x[1]) < 1 or norm(x[2]) < 1:
57:      return -2
58:  # if not simple
59:  if (((x[0]*x[0])/(x[2]*x[2])).imag *
60:      ((x[1]*x[1])/(x[0]*x[0])).imag > 0 and
61:      ((x[1]*x[1])/(x[0]*x[0])).imag *
62:      ((x[2]*x[2])/(x[1]*x[1])).imag > 0):
63:      return -3
64:  active = [ norm(x[0]) > norm(x[1] + x[2]*I*direction),
65:            norm(x[1]) > norm(x[2] + x[0]*I*direction),
66:            norm(x[2]) > norm(x[0] + x[1]*I*direction) ]
67:  for i,j,k in [(0,1,2), (1,2,0), (2,0,1)]:
68:      if active[i] and not active[j] and not active[k]:
69:          return i
70:      if active[i] and active[j] and not active[k]:
71:          al = ( norm(x[k])-norm(x[i])+norm(x[j]) )/2
72:          am = ( norm(x[j])-norm(x[i])+norm(x[k]) )/2
73:          ar = ( norm(x[k])-norm(x[j])+norm(x[i]) )/2
74:          bl = (x[j]*x[j])/(x[k]*x[k])
75:          br = (x[i]*x[i])/(x[k]*x[k])
76:          cl = -(bl/abs(bl))*al
77:          cr = norm(x[k]) + (br/abs(br))*ar
78:          dl = ( bl.real*(cl.real-am) + bl.imag*cl.imag )/bl.imag
79:          dr = ( br.real*(cr.real-am) + br.imag*cr.imag )/br.imag
80:          if dl*direction < dr*direction:
81:              return i
82:          else:
83:              return j
84:  return -1
85:
86: # draw line from z1 to z2 (z1, z2: complex number)
87: def create_line(z1, z2, color="black"):
88:     x1 = z1.real*scale+origin_x
89:     y1 = -(z1.imag*scale)+origin_y
90:     x2 = z2.real*scale+origin_x
91:     y2 = -(z2.imag*scale)+origin_y
92:     canvas.create_line(x1, y1, x2, y2, fill=color)
93:
94: # draw circle (center: complex number, radius: real number)
95: def create_circle(center, radius):
96:     x1 = center.real - radius
97:     y1 = center.imag - radius
98:     x2 = center.real + radius
99:     y2 = center.imag + radius
100:     x1 = x1*scale+origin_x
101:     y1 = -(y1*scale)+origin_y

```

184

Yasushi Yamashita

```

102:     x2 = x2*scale+origin_x
103:     y2 = -(y2*scale)+origin_y
104:     canvas.create_oval(x1, y1, x2, y2)
105:
106: # for Step 2 and 3
107: def create_lines_and_circles(st, x, color="black"):
108:     a = (x[0]/(x[1]*x[2]), x[1]/(x[2]*x[0]), x[2]/(x[0]*x[1]))
109:     for i in range(-4, 4):
110:         create_line(st+i, st+i+a[0], color)
111:         create_line(st+i+a[0], st+i+a[0]+a[1], color)
112:         create_line(st+i+a[0]+a[1], st+i+a[0]+a[1]+a[2], color)
113:         create_circle(st+i, abs(1/x[1]))
114:         create_circle(st+i+a[0], abs(1/x[2]))
115:         create_circle(st+i+a[0]+a[1], abs(1/x[0]))
116:
117: def nextstx(st, x, ac):
118:     if ac == 0:
119:         st = st + x[0]/(x[1]*x[2])
120:         x = (x[1]*x[2] - x[0], x[2], x[1])
121:     elif ac == 1:
122:         y = (x[2], x[2]*x[0] - x[1], x[0])
123:         st = st + x[0]/(x[1]*x[2]) + y[2]/(y[0]*y[1])
124:         x = y
125:     elif ac == 2:
126:         st = st - x[2]/(x[0]*x[1])
127:         x = (x[1], x[0], x[0]*x[1] - x[2])
128:     return st, x
129:
130: # draws the limit set using depth first search
131: def limitset(T, lastlabel, level):
132:     global oldpt
133:     if lastlabel == 'E':
134:         limitset(matmult(T, ma), 'a', level+1)
135:         limitset(matmult(T, mB), 'B', level+1)
136:         limitset(matmult(T, mA), 'A', level+1)
137:         limitset(matmult(T, mb), 'b', level+1)
138:     return
139:     if norm(T[1][1]) > 0.00001:
140:         newpt = T[0][1]/T[1][1]
141:     else:
142:         newpt = 100j
143:     if (level > 4 and norm(newpt - oldpt) < 0.0001) or level > 9:
144:         create_line(oldpt, newpt, "blue")
145:         oldpt = newpt
146:     return
147:     if lastlabel == 'a':

```

```

148:         limitset(matmult(T, mb), 'b', level+1)
149:         limitset(matmult(T, ma), 'a', level+1)
150:         limitset(matmult(T, mB), 'B', level+1)
151:     if lastlabel == 'B':
152:         limitset(matmult(T, ma), 'a', level+1)
153:         limitset(matmult(T, mB), 'B', level+1)
154:         limitset(matmult(T, mA), 'A', level+1)
155:     if lastlabel == 'A':
156:         limitset(matmult(T, mB), 'B', level+1)
157:         limitset(matmult(T, mA), 'A', level+1)
158:         limitset(matmult(T, mb), 'b', level+1)
159:     if lastlabel == 'b':
160:         limitset(matmult(T, mA), 'A', level+1)
161:         limitset(matmult(T, mb), 'b', level+1)
162:         limitset(matmult(T, ma), 'a', level+1)
163:
164: # This function draws everything
165: def paint():
166:     global ma, mA, mb, mB, oldpt
167:     # clears the canvas
168:     canvas.delete(ALL)
169:     # draws axis
170:     canvas.create_line(0, origin_y, 2000, origin_y)
171:     canvas.create_line(origin_x, 0, origin_x, 1000)
172:     # calculate the triple: x=(x,y,z)
173:     q1 = (p1[0] - origin_x)/scale - (p1[1] - origin_y)/scale*I
174:     q2 = (p2[0] - origin_x)/scale - (p2[1] - origin_y)/scale*I
175:     st = 0
176:     a = (q1, q2-q1, 1-q2)
177:     x = (1/sqrt(a[1]*a[2]), 1/sqrt(a[2]*a[0]), 1/sqrt(a[0]*a[1]))
178:     y = (x[0], x[1], -x[2])
179:     ax = (x[0]/(x[1]*x[2]), x[1]/(x[2]*x[0]), x[2]/(x[0]*x[1]))
180:     ay = (y[0]/(y[1]*y[2]), y[1]/(y[2]*y[0]), y[2]/(y[0]*y[1]))
181:     # Step 1
182:     if (norm(ax[0]-a[0]) + norm(ax[1]-a[1]) + norm(ax[2]-a[2]) >
183:         norm(ay[0]-a[0]) + norm(ay[1]-a[1]) + norm(ay[2]-a[2])):
184:         x = y
185:     discrete = True
186:     tmpx0 = x
187:     while True:
188:         ac = transition1(x)
189:         if ac < -1:
190:             discrete = False
191:         if ac < 0:
192:             break
193:         st, x = nextstx(st, x, ac)

```

```

194:         create_lines_and_circles(st, x, "green")
195:     # recalculate x and st
196:     a = (x[0]/(x[1]*x[2]), x[1]/(x[2]*x[0]), x[2]/(x[0]*x[1]))
197:     x = (x[0] if ((1/x[0])/a[1]).imag >= 0 else -x[0],
198:         x[1] if ((1/x[1])/a[2]).imag >= 0 else -x[1],
199:         x[2] if ((1/x[2])/a[0]).imag >= 0 else -x[2] )
200:     if (((x[0]*x[0])/(x[2]*x[2])).imag *
201:         ((x[1]*x[1])/(x[0]*x[0])).imag > 0 and
202:         ((x[1]*x[1])/(x[0]*x[0])).imag *
203:         ((x[2]*x[2])/(x[1]*x[1])).imag > 0):
204:         x = (x[0], x[1], -x[2])
205:     tmpx1 = x
206:     tmpst = st
207:     create_lines_and_circles(st, x, "black")
208:     # Step 2
209:     while True:
210:         ac = transition2(x, 1)
211:         if ac < -1:
212:             discrete = False
213:         if ac < 0:
214:             break
215:         st, x = nextstx(st, x, ac)
216:         create_lines_and_circles(st, x, "black")
217:     # Step 3
218:     x = tmpx1
219:     st = tmpst
220:     while True:
221:         ac = transition2(x, -1)
222:         if ac < -1:
223:             discrete = False
224:         if ac < 0:
225:             break
226:         st, x = nextstx(st, x, ac)
227:         create_lines_and_circles(st, x, "black")
228:     create_lines_and_circles(0, tmpx0, "red")
229:     print q1
230:     print q2
231:     print "discrete" if discrete else "indiscrete"
232:     # draw limit set
233:     x = tmpx0
234:     oldpt = 100
235:     ma = [[x[0]-x[2]/x[1], x[0]/(x[1]*x[1])], [x[0], x[2]/x[1]]]
236:     mA = [[x[2]/x[1], -x[0]/(x[1]*x[1])], [-x[0], x[0]-x[2]/x[1]]]
237:     mb = [[x[2]-x[0]/x[1], -x[2]/(x[1]*x[1])], [-x[2], x[0]/x[1]]]
238:     mB = [[x[0]/x[1], x[2]/(x[1]*x[1])], [x[2], x[2]-x[0]/x[1]]]
239:     if draw_limitset:

```

```
240:         limitset( [[1,0],[0,1]], 'E', 0 )
241:
242: # handlers (See section 2.6)
243: def handler_button (event):
244:     global p1, p2
245:     if ((p1[0]-event.x)*(p1[0]-event.x) +
246:         (p1[1]-event.y)*(p1[1]-event.y) <
247:         (p2[0]-event.x)*(p2[0]-event.x) +
248:         (p2[1]-event.y)*(p2[1]-event.y)):
249:         p1 = (event.x, event.y)
250:     else:
251:         p2 = (event.x, event.y)
252:     paint()
253:
254: def handler_left (event):
255:     global origin_x
256:     global p1
257:     global p2
258:     origin_x = origin_x + 20
259:     p1 = (p1[0]+20, p1[1])
260:     p2 = (p2[0]+20, p2[1])
261:     paint()
262:
263: def handler_right (event):
264:     global origin_x
265:     global p1
266:     global p2
267:     origin_x = origin_x - 20
268:     p1 = (p1[0]-20, p1[1])
269:     p2 = (p2[0]-20, p2[1])
270:     paint()
271:
272: def handler_up (event):
273:     global origin_y
274:     global p1
275:     global p2
276:     origin_y = origin_y + 20
277:     p1 = (p1[0], p1[1]+20)
278:     p2 = (p2[0], p2[1]+20)
279:     paint()
280:
281: def handler_down (event):
282:     global origin_y
283:     global p1
284:     global p2
285:     origin_y = origin_y - 20
```

188

*Yasushi Yamashita*

```
286:     p1 = (p1[0], p1[1]-20)
287:     p2 = (p2[0], p2[1]-20)
288:     paint()
289:
290: def handler_l (event):
291:     global draw_limitset
292:     draw_limitset = not draw_limitset
293:     paint()
294:
295: canvas.bind("<Button-1>", handler_button)
296: canvas.bind("<B1-Motion>", handler_button)
297: canvas.bind("<Left>", handler_left)
298: canvas.bind("<Right>", handler_right)
299: canvas.bind("<Up>", handler_up)
300: canvas.bind("<Down>", handler_down)
301: canvas.bind("l", handler_l)
302:
303: # This part is due to Ma Jia Jun.
304: # Type 's' to save the picture into a PostScript file.
305: import tkinterFileDialog
306: def saveps(event):
307:     filename = tkinterFileDialog.asksaveasfilename(
308:         defaultextension="ps",
309:         filetypes=[("PostScript", "*.ps")],
310:         initialfile="OptPy.ps",
311:         title="Save to PostScript")
312:     canvas.postscript(
313:         {"file":filename
314:         })
315: canvas.bind("s",saveps)
316:
317: canvas.focus_set()
318: canvas.pack(expand=YES, fill=BOTH)
319: paint()
320: canvas.mainloop()
```

## 6. Using OptPy

To use the program OptPy, create a file named `OptPy.py` with contents listed in the previous section. You can download this file from [14]. Double click this file, or type `python OptPy.py` in your terminal (Mac OS X or Linux). Then our program will start and a window appears. See Figure 6.

The initial broken line described at (3.4) is red. Other broken lines (3.9) calculated in steps 2 and 3 are black.

The user can change  $a_1$  and  $a_2$  by the mouse. Arrow keys can be used to translate the complex plane. The key ‘1’ toggles whether draw the limit set or not. The key ‘s’ saves the picture in PostScript format. The PostScript part is due to Ma Jia Jun (one of the audience). I would like to thank him for his contribution.

Let  $(x, y, z) \in \mathcal{X}_2$  and  $\phi_{(x,y,z)} : \widehat{\mathbb{Q}} \rightarrow \mathbb{C}$  be the associated Markoff map. (See subsection 3.3.) In [2] (Conjecture A), Bowditch conjectured that  $(x, y, z)$  corresponds to a quasifuchsian representation if and only if  $\phi_{(x,y,z)}^{-1}([-2, 2]) = \emptyset$  and the number of elements  $r \in \widehat{\mathbb{Q}}$  with  $|\phi_{(x,y,z)}(r)| \leq 2$  is finite. This conjecture is still open. Quasifuchsian Kleinian groups might be better understood using this type of software.

### Acknowledgements

I would like to express my thanks to Ser Peow Tan for organizing the program “Geometry, Topology and Dynamics of Character Varieties.” I was supported as a visiting senior research fellow by the Department of Mathematics, National University of Singapore during the program. I am grateful to the IT staff of the Department of Mathematics for setting up software in the computer room needed for the lectures that I gave. I would also like to thank all the participants in the lecture. Finally, I wish to express my thanks to the referee for helpful comments.

### References

1. H. Akiyoshi, M. Sakuma, M. Wada and Y. Yamashita, “Punctured torus groups and 2-bridge knot groups. I”, *Lecture Notes in Mathematics*, **1909**, Springer, Berlin, 2007.
2. B. H. Bowditch, “Markoff triples and quasi-Fuchsian groups”, *Proc. London Math. Soc.*, **77** (1998), 697–736.
3. B. Burton, “Regina”, a normal surface theory calculator,  
<http://regina.sourceforge.net/>.
4. M. Culler, N. M. Dunfield and J. R. Weeks, “SnapPy”, a computer program for studying the geometry and topology of 3-manifolds,  
<http://snappy.computop.org/>
5. F. Guéritaud, “Triangulated cores of punctured-torus groups”, *J. Differential Geom.*, **81** (2009), 91–142.
6. T. Hamada and KNOPPIX/Math committers, “KNOPPIX/Math”, a bootable live linux system with emphasis on mathematical software,  
<http://www.knoppix-math.org/>.
7. T. Jørgensen, “On pairs of once-punctured tori”, *London Math. Soc. Lecture Note Ser.*, **299** (2003), 183–207.

8. Y. Komori, T. Sugawa, M. Wada and Y. Yamashita, “Drawing Bers embeddings of the Teichmüller space of once-punctured tori”, *Experiment. Math.*, **15** (2006), 51–60.
9. D. Mumford, C. Series and D. Wright, “Indra’s pearls. The vision of Felix Klein”, Cambridge University Press, New York, 2002.
10. M. Lackenby, “The canonical decomposition of once-punctured torus bundles”, *Comment. Math. Helv.*, **78** (2003), 363–384.
11. M. Lutz, “Learning Python”, O’Reilly Media, Sebastopol CA, 2009.
12. K. Vlahos, “PyScripter”, IDE for Python,  
<http://code.google.com/p/pyscripter/>
13. M. Wada, “OPTi’s algorithm for discreteness determination”, *Experiment. Math.*, **15** (2006), 61–66.
14. Y. Yamashita, “OptPy”, Once punctured torus software using Python,  
<http://vivaldi.ics.nara-wu.ac.jp/~yamasita/0ptPy/>.