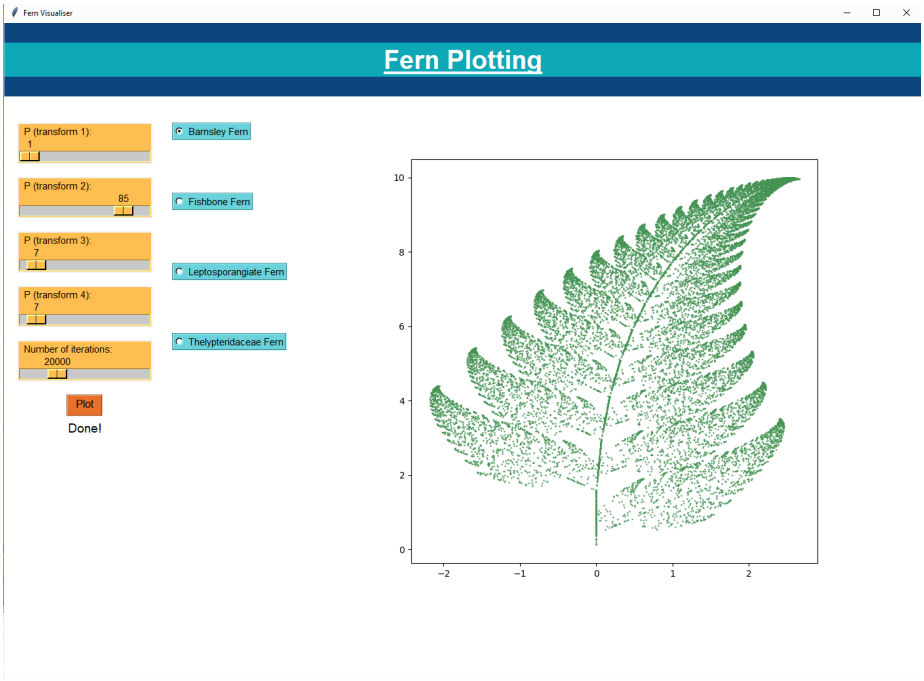
Fern Visualiser - CSSE1001 Assignment 3

Minh Trang Nguyen - 45270532

October 2018

1 Introduction



Fractal is a term in mathematics used to describe a recursive, infinitely self-similar set. Fractal structures can be found everywhere in nature due to having a repeating pattern at many levels of scale. Because of the repetitive nature, many fractals can be generated recursively by a computer.

In this application, we explore the Barnsley Fern and its mutated versions, the Thelypteridaceae fern, the Fishbone fern and the Leptosporangiate fern. The Barnsley fern is one of many fractals that can be generated mathematically and is reproducible at any magnification.

The Barnsley Fern was named after a British Mathematician Michael Barnsley who discovered that self-similarities occur in real-life plants, trees and leaves similar to fractals. As an example, the Barnsley Fern and the Thelypteridaceae Fern are meant to represent these 2 real ferns:

(a) Black Speenwort fern (b) Thelypteridaceae fern

Figure 1: The real-life versions of Barnsley fern and Thelypteridaceae fern respectively

2 User Manual

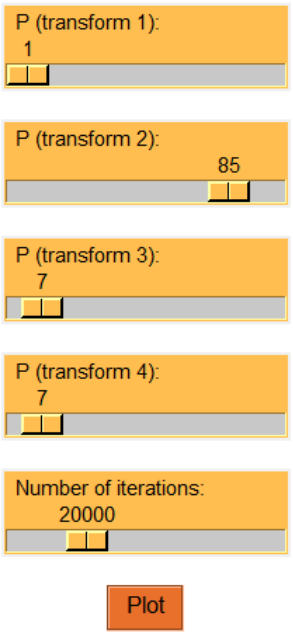


Figure 2: Plotting parameters

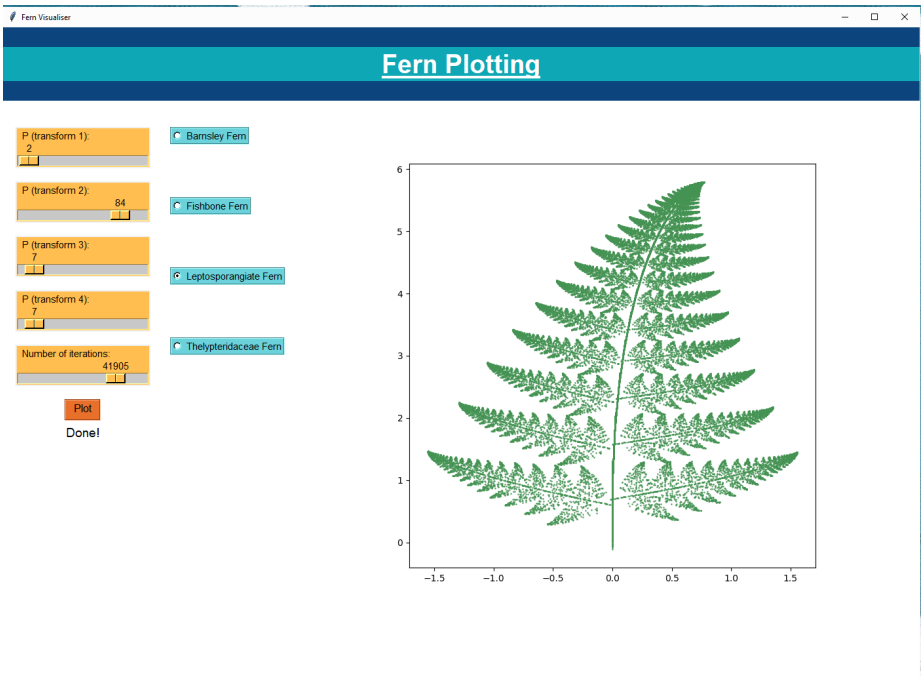
These 5 sliders in 2 control the relative probability of the four affine transformations and the number of iterations needed for plotting. The fractal ferns are created by applying one of these four affine transformations repeatedly which are chosen at random, starting at the origin

The default parameters in this application are set to produce the true Barnsley Fern. Users can explore further by adjusting these parameters which are matrix coefficients to see how the pattern changes. Additionally, the higher the number of iterations, the more detailed the fractals get, however, the plotting time also increases (although, not significantly). For best results, the default number of iterations was set to be from 10000 at the minimum and 50000 at the maximum.

The 4 option buttons bellow represent the 4 types of ferns which users can choose for the plotting.

Figure 3: Fern options

An example of how the Leptosporangiate fern should look like with its coefficients is shown below:



3 Approach

The Python libraries needed for the plotting application are:

- NumPy - for matrix manipulation
- Matplotlib Figure: for displaying plots on a tkinter canvas
- tkinter - main GUI library
- random - for generating random variables

The four affine transformations of the Barnsley fern are described as follows:

$$f_1(x, y) = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

In a true Barnsley Fern, the first transformation rule draws the stem and is chosen 1% of the time.

$$f_2(x, y) = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

The second transformation rule creates copies of the stem and the bottom fronds to make the complete fern and is chosen 85% of the time.

$$f_3(x,y) = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

$$f_4(x,y) = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.44 \end{bmatrix}$$

The third and fourth transformations respectively draw the fronds on the left and on the right and are equally chosen at 7% of the time. Using NumPy, these rules are represented by this block of code:

```
# the rules for 4 affine transformations
transform1 = lambda p: np.matrix([[0.00, 0.00], [0.00, 0.16]]) * p
transform2 = lambda p: np.matrix([[0.85, 0.04], [-0.04, 0.85]]) * p + np.matrix([[0.00], [1.60]])
transform3 = lambda p: np.matrix([[0.20, -0.26], [0.23, 0.22]]) * p + np.matrix([[0.00], [1.60]])
transform4 = lambda p: np.matrix([[-0.15, 0.28], [0.26, 0.24]]) * p + np.matrix([[0.00], [0.44]])
```

To be able to plot the Barnsley fern correctly, the first point must be drawn at the origin ($x_0 = 0$, $y_0 = 0$) then the new points are computed randomly by applying one of the four transformations.

```
    # initialise the plot with a matrix of zeros for faster iteration
    pts = np.zeros((iteration, 2))
    # accumulated probabilities
    accumulated = input1 + input2
    second_accumulated = accumulated + input3

    # plot according to given inputs
    for i in range(iteration):
        probability = random.random()
        if probability <= input1:
            p = transform1(p)
        elif probability <= accumulated:
            p = transform2(p)
        elif probability <= second_accumulated:
            p = transform3(p)
        else:  # 0.07
            p = transform4(p)

        # flattern the matrix to be an 1D 1xiteration matrix for plotting
        pts[i] = p.A1
```

Additionally, the sliders will adjust automatically whenever an user chooses an option. This was done by collecting the value of each option, which was set up in tkinter and set the slider parameters to be the default for each fractal fern. Therefore, when an user does not remember the coefficients for the matrices, the plot will always plot the correct fern.

```
def choose(self):
    """
    (str) Adjust the default settings for scalers and returns the
    value of the option variable when an user clicks a radio button
    """
    value = self.option.get()
    if value == "Fishbone Fern":
        # set the default values of each slider
        self.scaler1.set(2)
        self.scaler2.set(84)
        self.scaler3.set(7)
        self.scaler4.set(7)
    elif value == "Barnsley Fern":
        self.scaler1.set(1)
        self.scaler2.set(85)
        self.scaler3.set(7)
        self.scaler4.set(7)
    elif value == "Thelypteridaceae Fern":
        self.scaler1.set(1)
        self.scaler2.set(93)
        self.scaler3.set(3)
        self.scaler4.set(3)
    else:
        self.scaler1.set(2)
        self.scaler2.set(84)
        self.scaler3.set(7)
        self.scaler4.set(7)
    return self.option.get()
```

By giving the users the flexibility of changing the coefficients, the users can explore how the plot transforms and not be limited to the original ferns alone. The Thelypteridaceae fern, the Fishbone fern and the Leptosporangiate fern were written in similar ways with different sets of coefficients for the four transformations.

After an user has already chosen a particular fern type, by clicking the "Plot" button, the button will call the method "plot fractal" which gets the returned value from the method "choose" to know which fractal fern to plot.

After knowing which fern needs to be plotted, the application collects the user's inputs and normalise the values in order to ensure the 4 probabilities for the 4 transformations add up to 1. The number of iterations stay the same. This was implemented in the method "get values" which is called by each fractal method at the start of the plotting process.

```
def get_values(self):
    """
    (flt, flt, flt, flt, int) Returns a sequence of numbers to be used for the plot
    """
    # get the chosen probabilities given by the user
    # divide by 100 for the probability format(betwwen 0 and 1)
    p1, p2, p3, p4, iteration = (
        self.scaler1.get() / 100,
        self.scaler2.get() / 100,
        self.scaler3.get() / 100,
        self.scaler4.get() / 100,
        self.scaler5.get(),
    )

    # calculate the total for the normalisation
    total = p1 + p2 + p3 + p4

    # normalise the given probabilities to ensure 4 probabilities will add up to 1
    normalised1, normalised2, normalised3, normalised4 = (
        p1 / total,
        p2 / total,
        p3 / total,
        p4 / total,
    )
    return normalised1, normalised2, normalised3, normalised4, iteration
```

Call values:

```
def barnsley(self):
    """
    (None) Plot the barnsley fractal
    """
    self.printing_var.set("Plotting...")
    # collect the given variables from the user
    input1, input2, input3, input4, iteration = self.get_values()

    # a 1x2 starting matrix
```

After having done all that, the application will check if there already exists a plot on the canvas (for example, the user has already been plotting using the application), remove it and replace it with a new plot every time the "Plot" button is pressed. This was done to avoid having multiple plots on the screen every time the "Plot" function is called.

Since Matplotlib has a library to support plotting on a tkinter widget, it was possible to draw the Matplotlib plot on a tkinter canvas. The canvas then draws a scatter plot of the given parameters with point size being $0.5mm$.

```
# if a plot already exist, destroy and create a new one
if self.widget:
    self.widget.destroy()

# create a scatter plot
fig = Figure(figsize=(8, 8))
a = fig.add_subplot(111)  # 1x1 plot, first subplot
# s is the size of the scatter point
a.scatter(*zip(*pts), s=0.5, color="#386b30", label="Barnsley Fern")

# draws the plot on a tkinter canvas
canvas = FigureCanvasTkAgg(fig, master=self.plot)
self.widget = canvas.get_tk_widget()
self.widget.pack(side=tk.TOP)
canvas.draw()

self.printing_var.set("Done!")
```

After, all that was done, the tkinter printing label will be set to "Done" and will be printed out along with the final result.