MTH6138

IMPLEMENTING ITERATED FUNCTION SYSTEMS IN PYTHON

Student: Habib Wahab 160355141
Supervisor: Dr Wolfram Just
ah17012@qmul.ac.uk

Contents

1 Introduction

1.1 Fractals

Fractals first began to gain traction in the 1970s, primarily due to Benoit Mandelbrot's work. Simply, a fractal is a geometric object which is similar to itself. What this means visually, is that if you zoom into said object, it will look similar to the original shape. We find that fractals occur both in nature, such as snowflakes but can also be generated with the use of computers.

One of the first examples of using computers to generate a fractal was done by Mandelbrot and, his method was used to visually represent the Mandelbrot set as shown below in fig. 1[1].

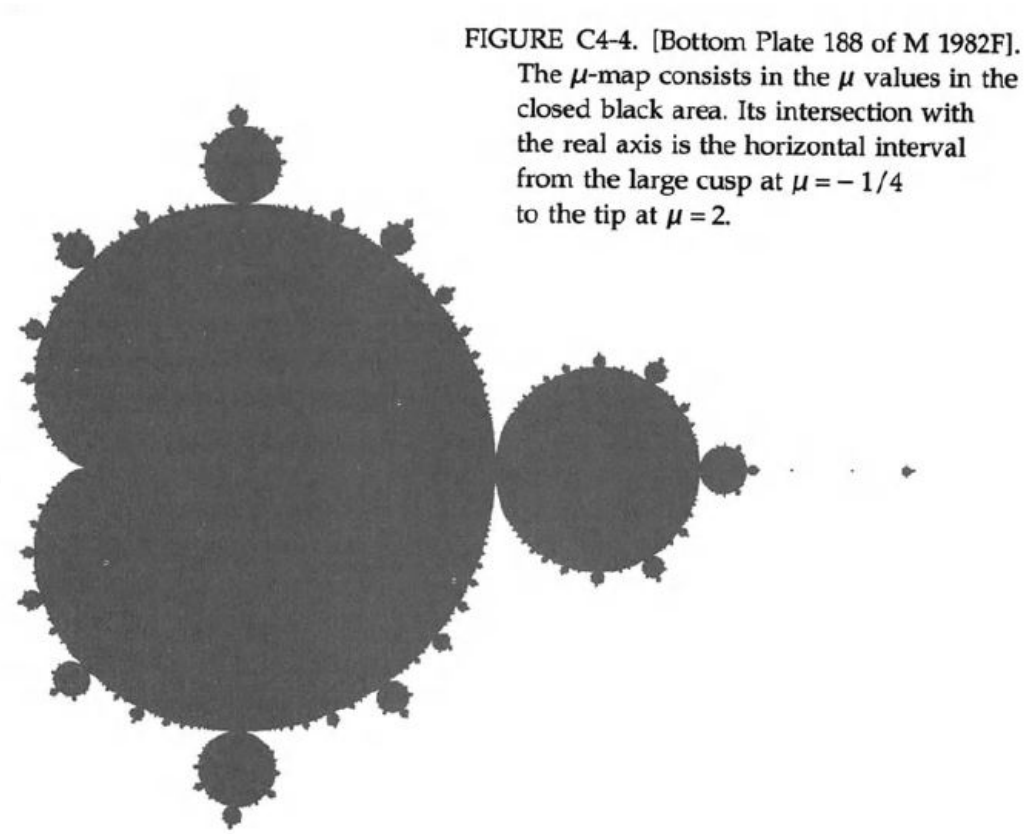

Figure 1: The Mandelbrot Set

[1]Mandelbrot, B., 2004. Fractals and Chaos. 1st ed. New York: Springer-Verlag. p56

We are able to observe that the various components that make up this image, are similar to each other, or it can be said that the Mandelbrot set contains smaller Mandelbrot sets, which leads us to discuss the concept of self-similarity.

1.2 Self-Similarity

An object that is self-similar can be said to be either exactly or approximately similar to itself. One important observation to make is that self-similarity does not have to occur on the same scale, what I mean by this is that a fractal can have self-similar parts which are smaller than the overall fractal. We saw this in the representation of the Mandelbrot set in 1.1 and, another example used to best demonstrate this is the Koch Snowflake (fig. 2)[2].

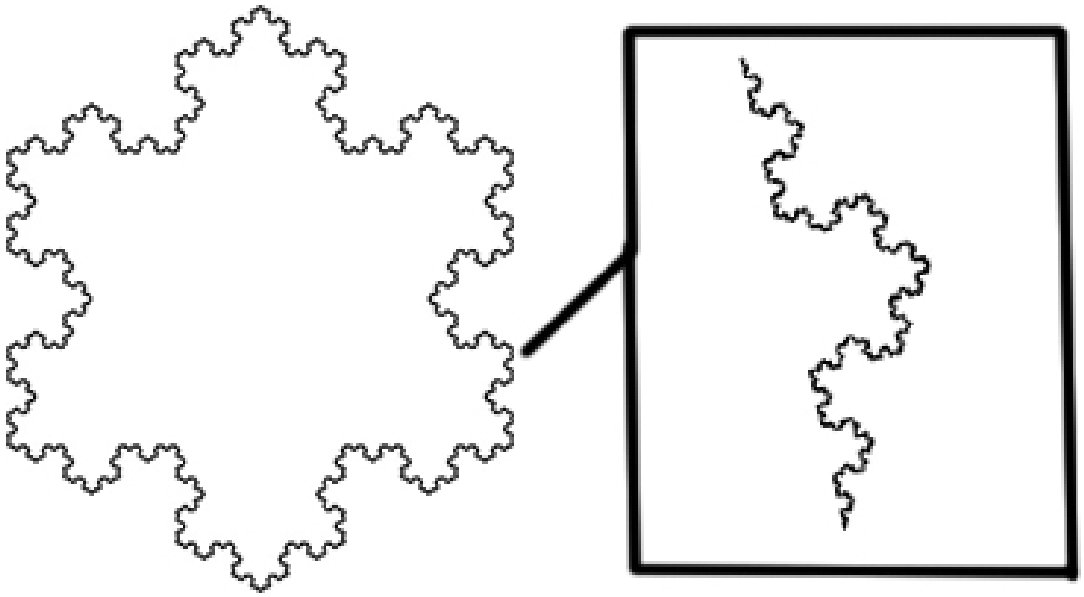

Figure 2: Koch Snowflake

When magnified, the Koch snowflake displays infinite self-similarity - we can zoom into the fractal and observe that the shape does not change. More formally, the Koch Snowflake is scale-invariant. An object that has scale invariance shows no changes to its shape or properties when changing its scale by a certain amount. Fractals that are constructed through iterated function systems are often scale invariant given the large number of iterations used in their generation.

[2]Trube, B., 2012. Fractals You Can Draw (The Koch Snowflake or Did It Really Snow In Cleveland In Late April?). [online] [BTW] : Ben Trube, Writer. Available at: ¡https://bentrubewriter.com/2012/04/24/fractals-you-can-draw-the-koch-snowflake/¿ [Accessed 31 March 2021].

2 Iterated Function Systems

Iterated function systems, or IFSs are a method used to construct fractals that are in most cases self-similar. The fractals are actually generated as the fixed attractor set of the IFS.[3] An attractor is said to be a set of numerical values, which a system has an inclination to evolve towards. In our case, the result of this evolution is the image we generate. In order to achieve this, an IFS is made up of functions, known as affine transformations.

2.1 Affine Transformations

An affine transformation, is a recursive function of the type:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

Each affine transformation will often yield a new attactor in the final image. The shape of the attractor is pre-determined through the choice of coefficients a through f, which also determine transformation itself. In order to generate a desired shape, a number of affine transformations may be used and, this method is what is referred to as an IFS.[4]

2.2 The Hutchinson Operator

The union of our affine transformations is known as the Hutchinson operator and, we use this operator to iteratively traverse through all possible movements in our set of transformations.

If we let $\{f_i : X \to X \mid 1 \leqslant i \leqslant N\}$ be an IFS, from a closed and bounded set X to itself, then the Hutchinson[5] operator H is defined over subsets $S \subset H$ as

$$H(S) = \bigcup_{i=1}^{N} f_i(S).$$

5

There are a few ways we can use the Hutchinson operator to generate our images, some more computationally expensive than others due to exponential scaling, which occurs when we iterate a large number of times. In order to avoid this, we will use the Hutchinson operator in the context of a chaos game, which is generally the most computationally feasible method to generate images with IFSs.

[3] Riddle, L., n.d. Iterated Function Systems. [online] Larryriddle.agnesscott.org. Available at: ¡https://larryriddle.agnesscott.org/ifs/ifs.htm¿ [Accessed 31 March 2021].

[4] Bradley, L., n.d. Iterated Function Systems - Chaos Fractals. [online] Stsci.edu. Available at: ¡https://www.stsci.edu/ lbradley/seminar/ifs.html¿ [Accessed 31 March 2021].

[5] Hutchinson, J. E.,Indiana University Mathematics Journal Vol. 30, No. 5 (1981), Indiana University Mathematics Department

2.3 Chaos Games

It should be made clear that in **2.2** we are iterating through sets however, the chaos games used for our implementation will iterate through points.

The algorithm[6] to do this was originally described by Barnsley in 1998 and, is as follows:

1. Pick a random point inside a regular polygon.
2. Draw the next point a fraction of the distance between the random point in step 1 and a random vertex of our polygon.
3. Repeat step 2. If we continue to do so, the result of this chaos game can sometimes result in a fractal.

More formally, if we begin from any point x_0 in the plane and then recursively define $x_k = \hat{f}_k(x_{k-1})$, where each $\hat{f}$ is chosen independently and with equal probability from (f_1,f_2,f_3). With probability one, the sequence of points $(x_k)_{k\geq 0}$ approaches and moves ergodically around and, increasingly closer to the attractor S. For this reason F is called an *iterated* function system.[7]

In order to implement this in python and, plot the points mentioned in step 2, we will define some functions, such that our initial point mentioned in step 1 can choose from them. The attractor will be the shape defined by the iteration through these functions and is what ultimately yields the final image. For our first implementation, I have chosen Sierpiński's triangle, as the transformations used are relatively simple.

For our second implementation we will use a similar method with some adjustments to generate the Barnsley fern. These adjustments are needed if instead $\hat{f}_k$ are selected from (f_1, f_2, f_3) with probabilities (p_1, p_2, p_3) respectively where each $p_i > 0$ and $p_1 + p_2 + p_3 = 1$. Then it can be said the same set S is determined by the sequence $(x_k)_{k\geq 0}$ but the points accumulate unevenly and the resulting attractor can be thought of as a greyscale image on S or probability distribution on S or as a measure. In this case $(f_1, f_2, f_3; p_1, p_2, p_3)$ is called an IFS with weights and, we will define these weighted probabilities in section **4**.[7]

[6] Weisstein, E., n.d. Chaos Game – from Wolfram MathWorld. [online] Mathworld.wolfram.com. Available at: ¡https://mathworld.wolfram.com/ChaosGame.html¿ [Accessed 2 April 2021].

[7] Barnsley, M. F., et al., 2008. V-variable fractals and superfractals p.4

3 Sierpiński's triangle

3.1 Background

Sierpiński's triangle, named after Waclaw Franciszek Sierpiński, is a fractal that is an equilateral triangle divided into smaller equilateral triangles.

It is also referred to as Sierpiński's gasket by Mandelbrot, which is his description due to an alternate method of construction (fig. 3)[8].

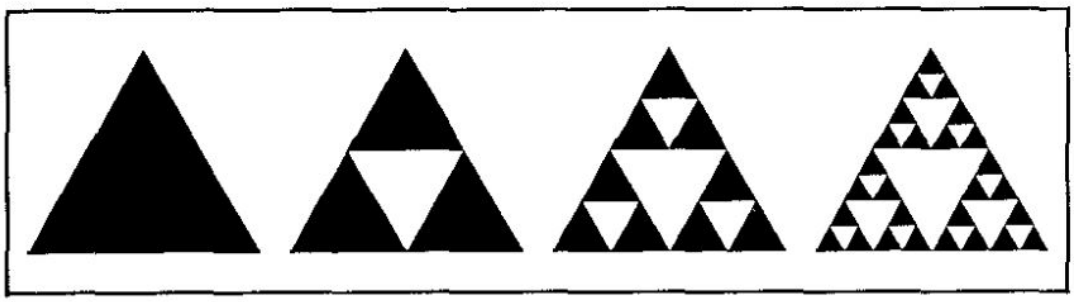

Figure 3: Construction of Sierpiński's gasket

This method of construction relies upon cutting out holes from the original triangle and is as follows:

1. Start with an equilateral triangle.
2. Dissect this triangle into four congruent parts, each part similar to the original triangle.
3. Delete the central part as shown in the second step of fig. 3.
4. Dissect the three remaining similar triangles and delete their central part.
5. From the remaining 27 congruent triangles, dissect each and then remove their centers.

Repeating this process a sufficient number of times yields what is known as Sierpiński's triangle shown in fig. 4 below.

[8]Mandelbrot, B., 2004. Fractals and Chaos. 1st ed. New York: Springer-Verlag. p142

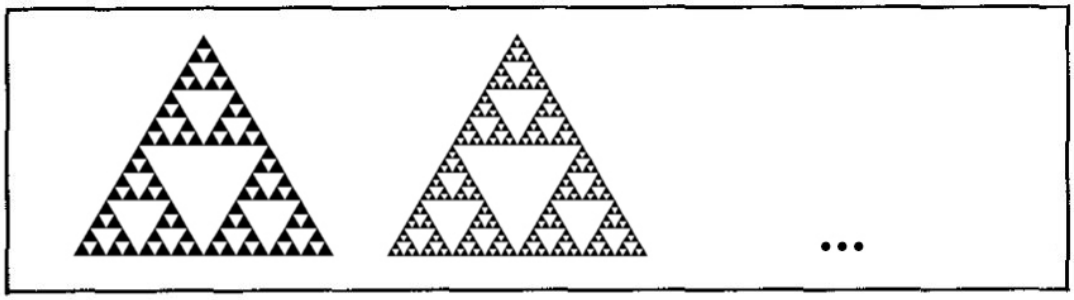

Figure 4: Construction of Sierpiński's gasket

In **1.2**, we discussed the concept of self-similarity which is apparent when looking at the finalised construction of Sierpiński's triangle as shown in fig. 4 - the overall triangle is made up of smaller similar triangles of varying scales.

Although this method of construction is feasible, to generate Sierpiński's triangle in python, it is more computationally efficient to use a chaos game as described in **2.3**, which leads us to define our IFS.

3.2 The IFS Method

In order to define this IFS, we must carefully choose our functions such that the attractor defined by these is Sierpiński's triangle. We must also ensure that with each iteration, our newly generated points are stored so that they can be plotted.

1. Define a triangle with three points acting as its vertices and, call them A, B and C.
2. Choose a random point P, that lies inside our triangle.
3. Create another point midway between P and a randomly chosen vertex. In order to do this, we define three functions, each having an equal chance of being chosen by our point:

   $f_1(P) = \frac{A+P}{2}$, $f_2(P) = \frac{B+P}{2}$ and $f_3(P) = \frac{C+P}{2}$

   for $P \in \mathbb{R}^2$
4. Repeat step 3, storing each newly created point.

Given we choose a suitable number of iterations, the result of storing and plotting each point created in step 3 should yield Sierpiński's triangle.

3.3 Implementation in Python

As a preface, we will be using functions from the *matplotlib* and *random* libraries. *Matplotlib* will enable us to visually represent our points by using functions from the *pyplot* module, whilst the *randint* function from *random* will allow our points to choose from the three functions we define. Importation of *matplotlib* and *random* is as follows:

```
1 from random import randint
2 import matplotlib.pyplot as plt
```

We then create a set which will store our points, initially containing only our first point. This point will choose between our three functions and is also the point used in our first iteration.

```
1 x, y = [0], [0]
```

We then set the number of iterations we wish to execute and, create a variable which for each iteration can take the value of 1, 2 or 3 using *randint*.

```
1 for n in range(100000):
2     transformation = randint(1,3)
```

We now define our three functions within some if statements. With each iteration, if the transformation variable is given a value of 1, we will apply function 1 to transform our point and so forth with functions 2 and 3. These functions will return the midpoint between the point in the iteration and a chosen vertex as defined in our IFS method. We will also add each point created in every iteration to the set of points we initially created (lines 13 and 14) by appending to our initial set after each iteration.

```
1     if transformation == 1:
2         x[n] = (x[n] - 3) / 2
3         y[n] = (y[n]) / 2
4
5     elif transformation == 2:
6         x[n] = (x[n] + 3) / 2
7         y[n] = (y[n]) / 2
8
9     else:
10        x[n] = (x[n]) / 2
11        y[n] = (y[n] + 3) / 2
12
13    x.append(x[n])
14    y.append(y[n])
```

Finally, we use the plot function from *matplotlib* to plot all the points that have been created and stored in our set of points. We also use some colouring in lines 2 and 3 below to make the image more presentable.

```
1 plt.title('Sierpinski Triangle')
2 plt.scatter(x, y, s=0.5, edgecolor='Blue')
3 plt.scatter(x, y, s=0.001, edgecolor='White')
```

Putting this all together, our finalised code is as follows.

```
1  from random import randint
2  import matplotlib.pyplot as plt
3
4  x, y = [0], [0]
5
6  for n in range(100000):
7      transformation = randint(1,3)
8
9      f transformation == 1:
10         x[n] = (x[n] - 3) / 2
11         y[n] = (y[n]) / 2
12
13     elif transformation == 2:
14         x[n] = (x[n] + 3) / 2
15         y[n] = (y[n]) / 2
16
17     else:
18         x[n] = (x[n]) / 2
19         y[n] = (y[n] + 3) / 2
20
21     x.append(x[n])
22     y.append(y[n])
23
24 plt.title('Sierpinski Triangle')
25 plt.scatter(x, y, s=0.5, edgecolor='Blue')
26 plt.scatter(x, y, s=0.001, edgecolor='White')
```

Executing this gives us the following image:

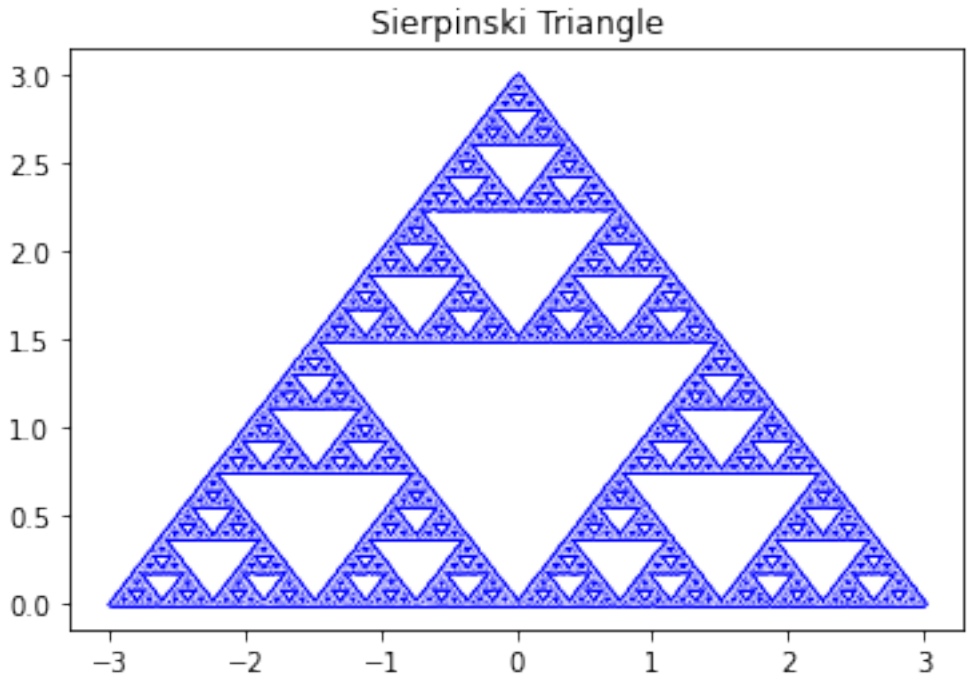

Figure 5: IFS Sierpiński's triangle

Our IFS has successfully generated Sierpiński's triangle. We have an equilateral triangle that has been divided into many smaller, similar equilateral triangles. Our method of construction relied on the points converging to the attractor defined by our functions, as opposed to repeatedly cutting holes.

The code itself reached completion relatively quickly, although this is dependent on the processing power of the machine. I chose 100000 iterations and, was able to generate the image in less than 10 seconds, which shows our choice of using chaos games was computationally efficient. Of course we can reduce the number of iterations to make the process faster - at the cost of clarity in the final image. However, I found 100000 iterations to be suitable for both speed and clarity.

4 Barnsley Fern

4.1 Background

The Barnsley fern is a fractal (depicted in fig. 6)[9] that can be generated through an IFS and was first described by Michael Barnsley in "Fractals Everywhere"[10].

This fractal resembles the naturally occurring Black Spleenwort Fern and, is often an example used to demonstrate a self-similar set which shows that its generation is reproducible at any given reduction or magnification.

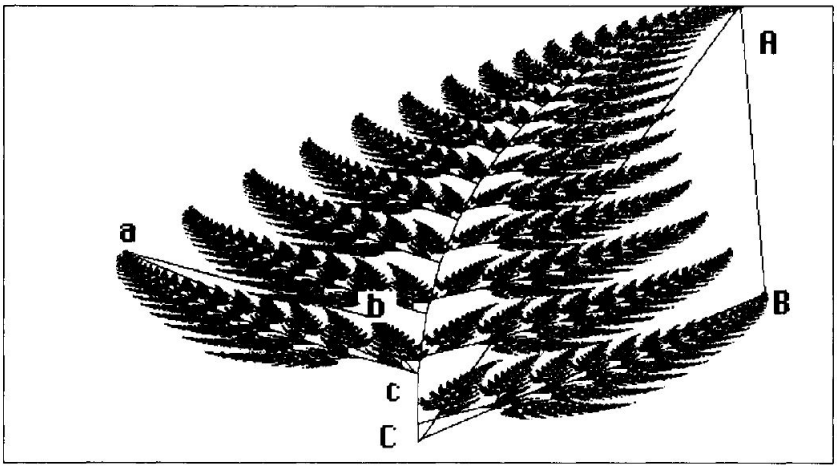

Figure 6: Black Spleenwort Fern

In theory, Barnsley's fern can be plotted by hand however, the number of iterations needed to actually generate the fractal can be in the tens of thousands so it is suitable to use a computer. There are a few popular methods to do so, either by iterating through sets or iterating through points. Fig. 6 shows how sets of triangles are iterated to produce the Barnsley fern. A transformation takes the triangle ABC to triangle abc and, this is repeated until the Fern is generated.

However, in **2.2** it was mentioned that iterating through sets can be computationally inefficient and, that is why for our IFS method we will iterate through points instead as described in **2.3**. As long as we ensure the transformations given by Barnsley are correct in our implementation - utilising his matrix of constants, we will produce a Black Spleenwort fern. The difference here versus our Sierpiński's triangle IFS is that the algorithm we used in **3.2** can be said to be a random IFS that acts as a generalisation of the chaos game, whereas here we will use a deterministic IFS algorithm that makes use of weighted probabilities, which we will now define.

[9]Barnsley, M. F., 1993. Fractals Everywhere. 2nd ed. s.l.:Academic Press p.102
[10]Barnsley, M. F., 1993. Fractals Everywhere. 2nd ed. s.l.:Academic Press

4.2 The IFS Method

In **2.3** it was mentioned we would need some adjustments to our IFS algorithm to generate the Barnsley fern and these come in the form of weighted probabilities which are attached to the the transformations used in our IFS as illustrated in fig. 7.[11] It should be noted that these transformations are affine transformations of the form described in **2.1**

Transformation	Translation	Probability
$\begin{pmatrix} +0.00 & +0.00 \\ +0.00 & +0.16 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$	1%
$\begin{pmatrix} +0.85 & +0.04 \\ -0.04 & +0.85 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$	85%
$\begin{pmatrix} +0.20 & -0.26 \\ +0.23 & +0.22 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$	7%
$\begin{pmatrix} -0.15 & +0.28 \\ +0.26 & +0.24 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0.44 \end{pmatrix}$	7%

Figure 7: Transformations with weighted probabilities

We include these probabilities to speed up convergence to the attractor in what is known as a deterministic IFS algorithm. The reason this is more efficient is beyond the scope of this paper, but relies on a proof that the similarity dimension agrees with the Hausdorff dimension and, can be explored in theorem **9.3** of the first edition of *Falconer's Fractal Geometry*. With these weighted probabilities in mind, our IFS method is as follows:

1. We define our first point at the origin where $x_1 = 0$ and $y_1 = 0$.
2. The next point and successive points are iteratively computed by applying one of the four transformations shown in fig. 7.
   a) the first transformation is chosen 1% of the time and generates our stem.
   b) the second transformation is chosen 85% of the time and generated copies of the stem and divided leaves to create the fern.
   c) the third transformation is chosen 7% of the time and generates the leaves to the lower left.
   d) the fourth transformation is chosen 7% of the time and generates the leaves to the lower right.

[11] Bradley, L., n.d. Iterated Function Systems - Chaos Fractals. [online] Stsci.edu.Available at: ¡https://www.stsci.edu/ lbradley/seminar/ifs.html¿

4.3 Implementation in Python

Again, we will be using functions from the *matplotlib* and *random* libraries. The uses of these are discussed in **3.3**.

Similarly to **3.3**, we create a set that stores points, initially only containing our first point, which will choose between our affine transformations to begin our iterations. We then choose the number of iterations we wish to execute and use *randint* to give each transformation a weighted probability.

```
from random import randint
import matplotlib.pyplot as plt

x, y = [0], [0]

for n in range(100000):
    transformation = randint(1, 100)
```

The difference here, from **3.3** is that we will have numbers 1 to 100 versus 1 to 3, to assign our transformation variable to with each iteration. This is so we can choose a range of values as opposed to a single value for our transformations to be chosen from and these will act as our weighted probabilities.

For example, the second transformation in fig.7 is chosen 85% of the time, so we use an if statement such that if the randomly assigned value to our transformation variable is between 2 and 86, this is the transformation that will be chosen. We use this template and the coefficients given by Barnsley to define our affine transformations, ensuring the ranges of the transformation variable are set correctly so they represent the correct weighted probabilities. It is to be noted these transformations are of the form described in **2.1**.

```
 if transformation == 1:
        x.append(0)
        y.append(0.16 * (y[n]))

    if 2 <= transformation <= 86:
        x.append(0.85 * (x[n]) + 0.04 * (y[n]))
        y.append(-0.04 * (x[n]) + 0.85 * (y[n]) + 1.6)

    if 87 <= transformation <= 93:
        x.append(0.2 * (x[n]) -0.26 * (y[n]))
        y.append(0.23 * (x[n]) + 0.22 * (y[n]) + 1.6)

    if 94 <= transformation <= 100:
        x.append(-0.15 * (x[n]) + 0.28 * (y[n]))
        y.append(0.26 * (x[n]) + 0.24 * (y[n]) + 0.44)
```

Lines 14 and 15 (above) will append the point generated with each iteration to our set that stores points. Our final step is to actually plot these points with *matplotlib* and and again we can add some colour to make the output more presentable.

```
1 plt.title('Barnsley Fern')
2 plt.scatter(x, y, s = 0.5, edgecolor='mediumseagreen')
3 plt.scatter(x, y, s = 0.001, edgecolor='darkgreen')
```

Putting this all together, our finalised code is as follows.

```
1  from random import randint
2  import matplotlib.pyplot as plt
3
4  x, y = [0], [0]
5
6  for n in range(100000):
7      transformation = randint(1, 100)
8
9      if transformation == 1:
10         x.append(0)
11         y.append(0.16 * (y[n]))
12
13     if 2 <= transformation <= 86:
14         x.append(0.85 * (x[n]) + 0.04 * (y[n]))
15         y.append(-0.04 * (x[n]) + 0.85 * (y[n]) + 1.6)
16
17     if 87 <= transformation <= 93:
18         x.append(0.2 * (x[n]) -0.26 * (y[n]))
19         y.append(0.23 * (x[n]) + 0.22 * (y[n]) + 1.6)
20
21     if 94 <= transformation <= 100:
22         x.append(-0.15 * (x[n]) + 0.28 * (y[n]))
23         y.append(0.26 * (x[n]) + 0.24 * (y[n]) + 0.44)
24
25 plt.title('Barnsley Fern')
26 plt.scatter(x, y, s = 0.5, edgecolor='mediumseagreen')
27 plt.scatter(x, y, s = 0.001, edgecolor='darkgreen')
```

Executing this gives us the following image:

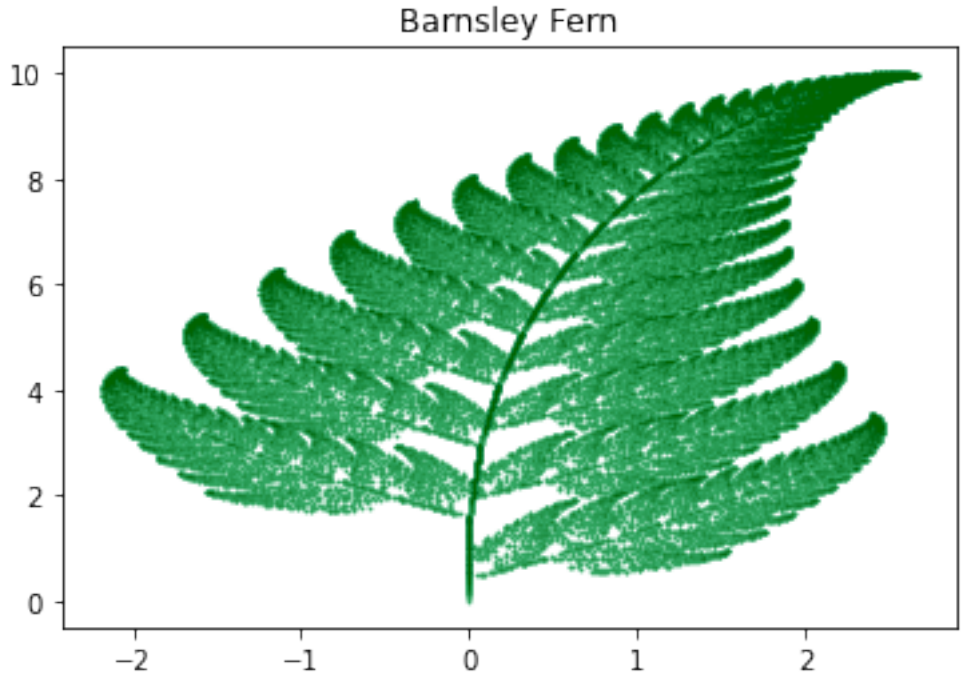

Figure 8: IFS Barnsley Fern

Again we can see that using 100000 iterations was sufficient to produce a clear image of the Barnsley fern. By using the weighted probabilities in a deterministic IFS algorithm, we used a comparatively more efficient process than in the random IFS used for our generation of Sierpiński's triangle. The weighted probabilities allowed us to vary the occurrence of our points choosing a given affine transformation, as some are needed more so than others which ultimately sped up convergence to the attractor.

4.4 Mutant Ferns

Through changing the coefficients of our four affine transformations, we are able to observe some interesting changes to the Barnsley fern.

Let

w	a	b	c	d	e	f	p
f_1	0	0	0	0.25	0	−0.4	0.02
f_2	0.95	0.005	−0.005	0.93	−0.002	0.5	0.84
f_3	0.035	−0.2	0.16	0.04	−0.09	0.02	0.07
f_4	−0.04	0.2	0.16	0.04	0.083	0.12	0.07

Figure 9: Varied coefficients and weighted probabilities

be the new coefficients used in our four affine transformations and, let p be the weighted probabilities attached to each of these transformations. It is to be noted that our weighted probabilities here are different from the ones used in **4.2** and **4.3**. Using the same code for our Barnsley fern implementation with this alteration yields:

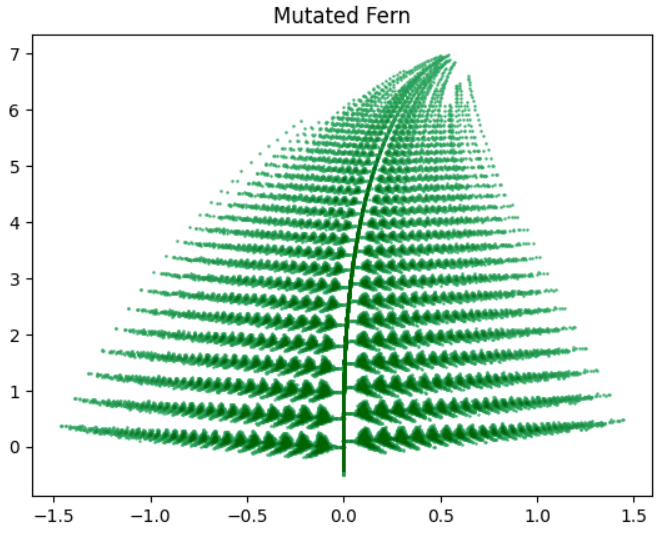

Figure 10: Mutated Fern

This is an example of a V-variable[8] fractal and, Barnsley actually described collections of these and their probability distributions to be known as a superfractal[12].

[12]Barnsley, M. F., et al., 2008. V-variable fractals and superfractals

4.5 V-Variable Fractals and Superfractals

In his paper about V-variable fractals and superfractals, Barnsley discusses how IFSs can provide models for naturally occurring plants, leaves and ferns by virtue of self-similarity that natural branching structures exhibit. He then goes on to say that these natural structures also show randomness from one level to the next and that no two of these are exactly alike.

The construction of V-variable fractals is also detailed in this paper, using the model situation of two IFSs F and G with fixed points A_1, A_2 and A_3 but with contraction ratios $\frac{1}{3}$ as opposed to $\frac{1}{2}$.

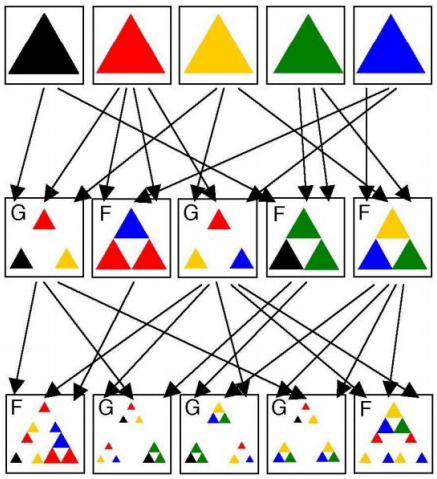

Figure 11: Construction of V-Variable Fractals

Fig. 11[13] shows a forward algorithm that generates families of V-variable fractals. We have level 1 at the top, followed by levels 2 and 3 in the construction of a possibly infinite sequence of 5-tuples of 5-variable Sierpiński triangles. From level 2 onward, F or G indicates which IFS is used and the input arrows indicate the buffers to which IFS was applied.[11]

The importance of these V-variable fractals is linked to the genetic makeup of the ferns they describe. Barnsley speculated that when a V-variable geometrical fractal model was found to have a good match to the geometry of a given plant, then there would be a relationship between the code trees and the information stored in the genes of the plant.[14]

[13]Barnsley, M. F., et al., 2008. V-variable fractals and superfractals p.5
[14]Barnsley, M. F., et al., 2008. V-variable fractals and superfractals p.2

5 Conclusion

5.1 Usage of Python

As shown by our outputs, the implementation of our IFSs were successful. The simple syntax of python makes it easy to define and use functions on our points, making use of libraries such as *matplotlib* and *random* to visually represent our points and allow them to choose between the functions we define. The ability to also use Jupyter notebook to adjust parameters for my transformations and immediately see the output made python attractive for the implementation of IFSs.

The code in sections **3.3** and **4.3** can be extended to produce other common fractals through alterations in the number and definitions of functions and when using a deterministic IFS algorithm, the weighted probabilities. For example, we can easily generate a Sierpiński pentagon, through the use of two additional functions (fig. 12)[15].

$$f_1(\mathbf{x}) = \begin{bmatrix} 0.382 & 0 \\ 0 & 0.382 \end{bmatrix} \mathbf{x}$$

$$f_2(\mathbf{x}) = \begin{bmatrix} 0.382 & 0 \\ 0 & 0.382 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0.618 \\ 0 \end{bmatrix}$$

$$f_3(\mathbf{x}) = \begin{bmatrix} 0.382 & 0 \\ 0 & 0.382 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0.809 \\ 0.588 \end{bmatrix}$$

$$f_4(\mathbf{x}) = \begin{bmatrix} 0.382 & 0 \\ 0 & 0.382 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0.309 \\ 0.951 \end{bmatrix}$$

$$f_5(\mathbf{x}) = \begin{bmatrix} 0.382 & 0 \\ 0 & 0.382 \end{bmatrix} \mathbf{x} + \begin{bmatrix} -0.191 \\ 0.588 \end{bmatrix}$$

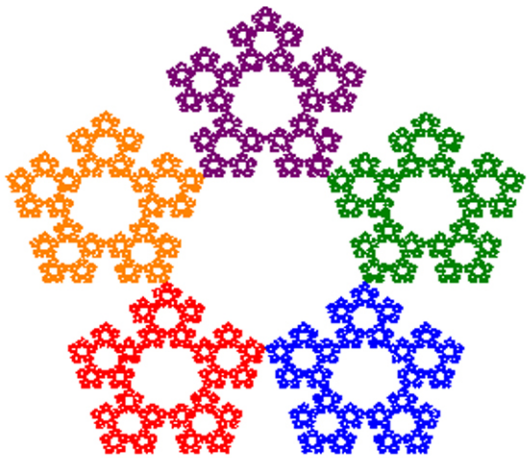

Figure 12: Sierpiński pentagon

5.2 Further applications

One way to extend the code I have written, would be to enable user input, coupling this with a front-end and some functionality to allow those studying IFSs to alter parameters as they see fit. This would need implementation of both random IFSs and deterministic IFSs such that the most common IFS fractals were able to be generated, in conjunction with allowing for users to transform the the visual representations of the fractals generated with their mouse.

[15] Riddle, L., n.d. Iterated Function Systems. [online] Larryriddle.agnesscott.org. Available at: ¡https://larryriddle.agnesscott.org/ifs/ifs.htm¿ [Accessed 31 March 2021].

In fact Larry Riddle, whom produced fig. 11, has an IFS construction kit on his website, which enables users to design and draw fractals by varying parameters of affine transformations without the need to code them. Each transformation can be scaled, rotated, stretched or skewed using a mouse and keyboard and, the user is given immediate visual representation on how the fractal changes as a transformation is modified. Either a random or deterministic IFS can be used.

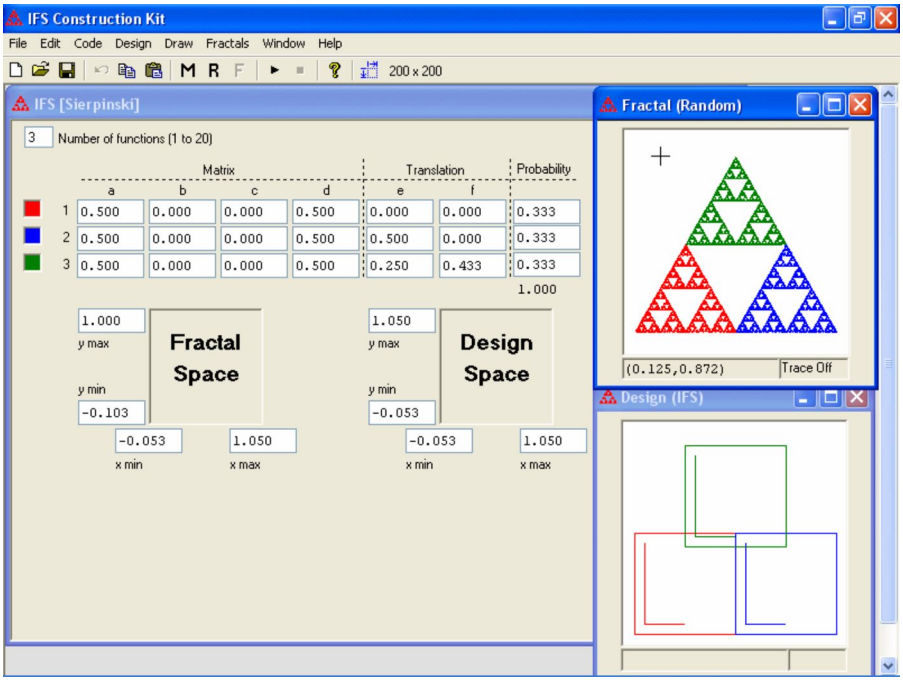

Figure 13: IFS Construction Kit

Here is a screenshot of the program, which shows a matrix representation of the affine transformations along with the resultant Sierpinski's triangle on the right hand side. We can see that we can input values into both the parameter and probability fields to alter the fractal (fig. 13)[16].

Ultimately, the implementation of IFSs in python (which can be extended to other languages) is relatively straightforward as long as we ensure we model our chaos games correctly, using weighted probabilities when needed for fractals such as the Barnsley Fern. We can choose a significant number of iterations, without computational expense due to the efficient nature of chaos games which makes these implementations feasible for modern machines.

[16]https://larryriddle.agnesscott.org/ifskit/index.htm

6 References

1. Mandelbrot, B., 2004. Fractals and Chaos. 1st ed. New York: Springer-Verlag.

2. Barnsley, M. F., 1993. Fractals Everywhere. 2nd ed. s.l.:Academic Press.

3. Hutchinson, J. E.,Indiana University Mathematics Journal Vol. 30, No. 5 (1981), Indiana University Mathematics Department

4. Barnsley, M. F., et al., 2008. V-variable fractals and superfractals

5. Trube, B., 2012. Fractals You Can Draw (The Koch Snowflake or Did It Really Snow In Cleveland In Late April?). [online] [BTW] : Ben Trube, Writer. Available at: ¡https://bentrubewriter.com/2012/04/24/fractals-you-can-draw-the-koch-snowflake/¿ [Accessed 31 March
2021].

6. Riddle, L., n.d. Iterated Function Systems. [online] Larryriddle.agnesscott.org. Available at: ¡https://larryriddle.agnesscott.org/ifs/ifs.htm¿ [Accessed 31 March 2021].

7. Bradley, L., n.d. Iterated Function Systems - Chaos Fractals. [online] Stsci.edu. Available at: ¡https://www.stsci.edu/ lbradley/seminar/ifs.html¿ [Accessed 31 March 2021].

8. Schloss, J., 2020. Iterated Function Systems · Arcane Algorithm Archive. [online] Algorithm-archive.org. Available at: ¡https://www.algorithm-archive.org/contents/IFS/IFS.html¿ [Accessed 1 April 2021].

9. Weisstein, E., n.d. Chaos Game – from Wolfram MathWorld. [online] Mathworld.wolfram.com. Available at: ¡https://mathworld.wolfram.com/ChaosGame.html¿ [Accessed 2 April 2021].

7 Appendix

7.1 Sierpiński's triangle

```
1  from random import randint
2  import matplotlib.pyplot as plt
3
4  x, y = [0], [0]
5
6  for n in range(100000):
7      transformation = randint(1,3)
8
9      if transformation == 1:
10         x[n] = (x[n] - 3) / 2
11         y[n] = (y[n]) / 2
12
13     elif transformation == 2:
14         x[n] = (x[n] + 3) / 2
15         y[n] = (y[n]) / 2
16
17     else:
18         x[n] = (x[n]) / 2
19         y[n] = (y[n] + 3) / 2
20
21     x.append(x[n])
22     y.append(y[n])
23
24 plt.title('Sierpinski Triangle')
25 plt.scatter(x, y, s=0.5, edgecolor='Blue')
26 plt.scatter(x, y, s=0.001, edgecolor='White')
```

7.2 Barnsley Fern

```
from random import randint
import matplotlib.pyplot as plt

x, y = [0], [0]

for n in range(100000):
    transformation = randint(1, 100)

    if transformation == 1:
        x.append(0)
        y.append(0.16 * (y[n]))

    if 2 <= transformation <= 86:
        x.append(0.85 * (x[n]) + 0.04 * (y[n]))
        y.append(-0.04 * (x[n]) + 0.85 * (y[n]) + 1.6)

    if 87 <= transformation <= 93:
        x.append(0.2 * (x[n]) -0.26 * (y[n]))
        y.append(0.23 * (x[n]) + 0.22 * (y[n]) + 1.6)

    if 94 <= transformation <= 100:
        x.append(-0.15 * (x[n]) + 0.28 * (y[n]))
        y.append(0.26 * (x[n]) + 0.24 * (y[n]) + 0.44)

plt.title('Barnsley Fern')
plt.scatter(x, y, s = 0.5, edgecolor='mediumseagreen')
plt.scatter(x, y, s = 0.001, edgecolor='darkgreen')
```

7.3 Mutated Fern

```
from random import randint
import matplotlib.pyplot as plt

x, y = [0], [0]

for n in range(100000):
    transformation = randint(1, 100)

    if transformation <= 2:
        x.append(0)
        y.append(0.25 * (y[n]) - 0.4)

    if 3 <= transformation <= 84:
        x.append(0.95 * (x[n]) + 0.005 * (y[n]) - 0.002)
        y.append(-0.005 * (x[n]) + 0.93 * (y[n]) + 0.5)

    if 85 <= transformation <= 92:
        x.append(0.035 * (x[n]) - 0.2 * (y[n]) - 0.09)
        y.append(0.16 * (x[n]) + 0.04 * (y[n]) + 0.02)

    if 93 <= transformation <= 100:
        x.append(-0.04 * (x[n]) + 0.2 * (y[n]) + 0.083)
        y.append(0.16 * (x[n]) + 0.04 * (y[n]) + 0.12)

plt.title('Mutated Fern')
plt.scatter(x, y, s=0.5, edgecolor='mediumseagreen')
plt.scatter(x, y, s=0.001, edgecolor='darkgreen')
```