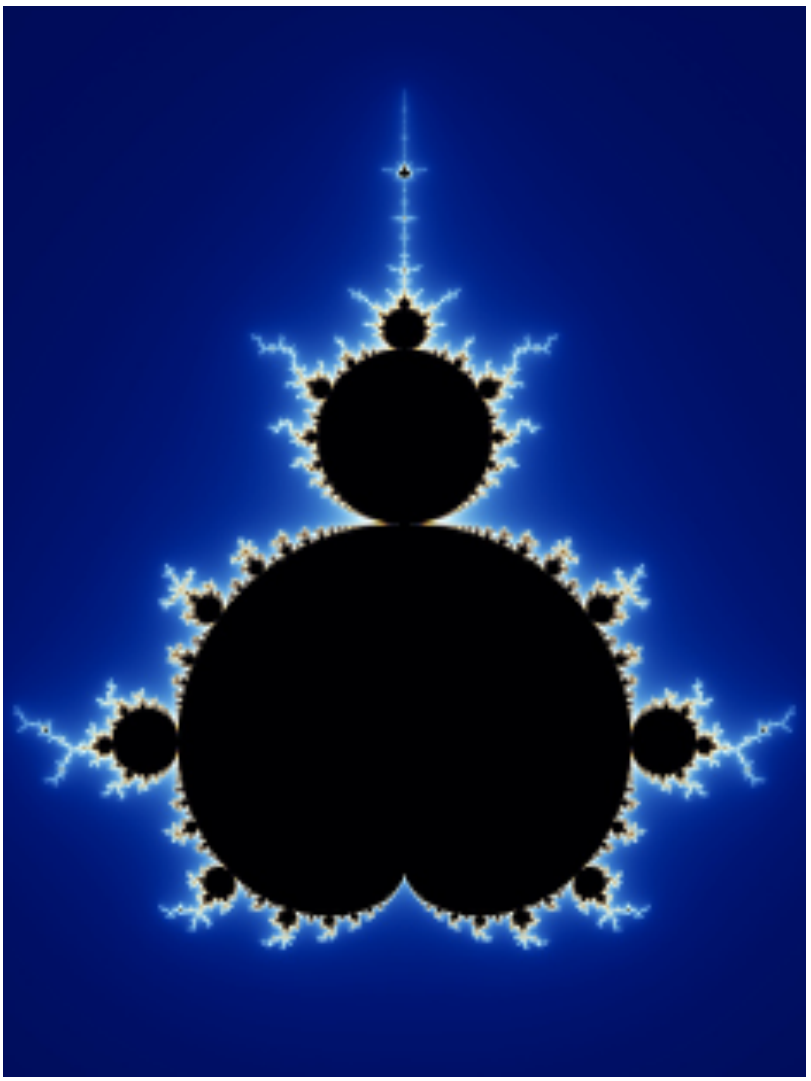
# Mandelbrot Sets
# Math Research Paper



# 1. Purpose and Content

The purpose of this paper is to introduce the reader into the magical mathematical phenomenon of fractals. We will be focusing primarily on the Mandelbrot sets (M-Set) and its visualisation. I will be showing the following content: explanation on the Mandelbrot equation, relation to the Julia and other sets, a way of visualising these sets through excel, how to make a simple python program that renders them and their relation to the real world and other sets.

# 2. Introduction

The discovery of the geometry of Mandelbrot covers a broad and diverse area of scientific research, from cosmology and dynamics to the day-to-day area of finance. The Mandelbrot set is a collection of coordinates within the complex plane. It was initially a derived set from another common set, the Julia sets, that were studied by French mathematicians Pierre Fatou and Gaston Julia in the early 20th century. But they're not the same as they are two different sets. Due to the lack of computer advancements in the 20th century, it was practically impossible to study the structure of the Julia set. Which led to its abandonment until 1961, when a Jewish-Polish mathematician named Benoit Mandelbrot began his research at IBM on reducing the white noise that disturbed the transmission on telephony lines. He observed that the disturbance came in burst, and at times there are lots of disturbances and at times there are none. So, he began to ponder whether or not he could create a model to explain this disturbance.

Mandelbrot took a unique way to approach this, he chose to visualise the model. The results showed a consecutive self-similarity at all levels. He later on called this fractals. There are many versions of a fractal, but one thing in common is that they all have a self-similarity at all levels of the set. It means that no matter how much you iterate the set, it will keep on producing patterns of itself. The fractal that Mandelbrot related to the telephony disturbance was called Cantor dust fractal. Which was generated through the common concept of all fractal the Iterated Function System (IFS). IFS is also commonly known as a recursive function, it means that the values will keep on repeating using the values from before. By repeating this process

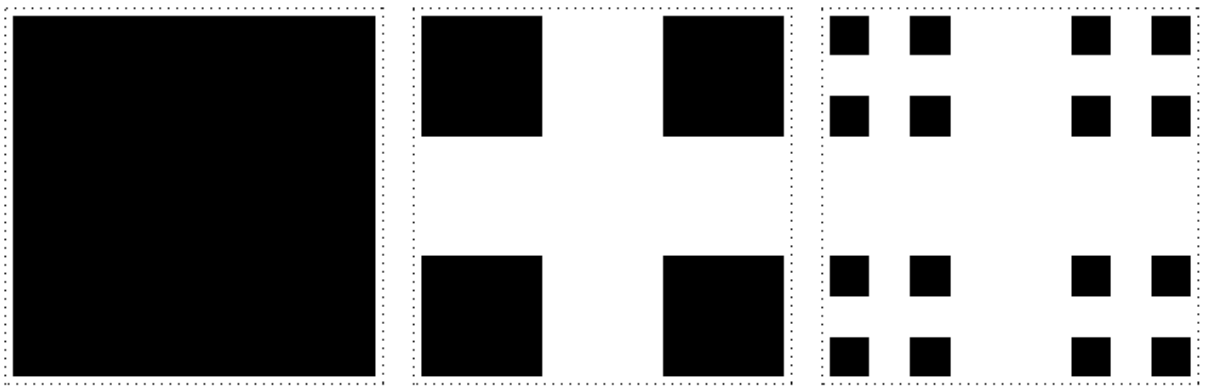


*Figure 1: The cantor dust fractal is visualise using a single black box as its initial value, each iteration splits it to four equal parts*

a cantor dust fractal was generated through splitting a single black square into four equal parts repeatedly. In context, the black dots represent the short burst periods where there is a disturbance in the line, and the more iterations the more frequent it will become

## 3. Mandelbrot set

Ecstatic by his finding of the Cantor dust, Benoit Mandelbrot continued his research, and further delved himself into the world of fractals. Eventually coming across the work of Pierre Fatou and Gaston Julia. With the advancement in technology available during his time he was able to visualise the sets and later on discovered that it was also a fractal, but with greater detail. It was later on called the Mandelbrot fractal or the Mandelbrot set. He later on wrote down his research and findings into his book.

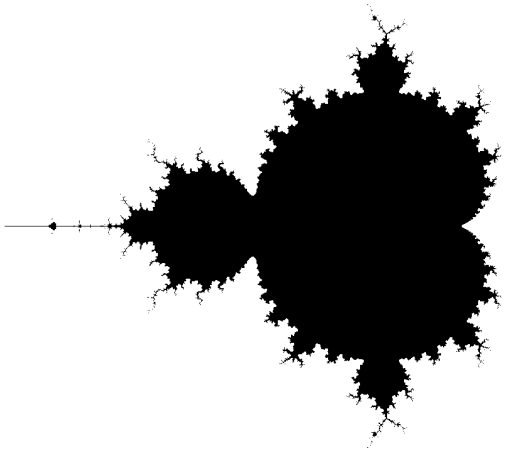


*Figure 2: The Mandelbrot set that can be obtained through* $Z_{n+1} = (Z_n)^2 + c$ *and is visualised with black and white coloring.*

## 4. Relationship of Mandelbrot and Julia Sets

Mandelbrot and Julia sets are in a way connected to each other. As mentioned above, we know that the mandelbrot set was actually a set that was derived from the julia set during Benoit's research. These two sets are also both fractals, other than that both of these sets were created using the same recursive formula. While the mandelbrot set is created using different values for *c* and an initial value of $Z_0 = 0$, the julia set is created using a fixed value of *c* and the $Z$values are varied. The complex number for both can be chosen freely. In mandelbrot we use the formula to see whether the *c* value belongs to the set or not. In julia sets, to determine whether the point z belongs to the julia set, iterate the recursive formula $Z_{n+1} = Z_n + c$ in the same manner as the mandelbrot set.

## 5. Elaboration and Interpretation

As mentioned in the introduction, the Mandelbrot set is a set of points in the complex plane. A plane that is two-dimensional consisting of a vertical axis and a horizontal axis, with imaginary on its vertical and real on its horizontal. A simple way to represent a point is by using the complex number $c \in \mathbb{C}$written in the form of $c = a + bi$, where$a, b \in \mathbb{R}$ and $i = \sqrt{-1}$. When the set is visualised through the use of repeated IFS of the mandelbrot function it will show figure 2. The general function of the Mandelbrot set can be interpreted as

$$Z_{n+1} = (Z_n)^2 + c, \; where \; Z_0 = 0$$

This function shows an IFS or Iterated Function System with the recursive function of $Z_{n+1} = (Z_n)^2 + c$, which was explained in the introduction, and the initial value of $Z_0 = 0$. This function will follow the recursive set of.

$$\begin{aligned} Z_{n+1} &= (Z_n)^2 + c \\ Z_0 &= 0 \\ Z_1 &= Z^2 + c \\ &= c \\ Z_2 &= Z_1{}^2 + c \\ &= (a + bi)(a + bi) + c \\ &= a^2 + abi + abi + b^2 i^2 + c \\ &= a^2 + 2abi - b^2 + c \\ &= a^2 - b^2 + 2abi + c \end{aligned}$$

The process above will be repeated until the absolute of the complex number will be above the threshold. Something to be emphasised upon is that a point $c$ can only belong to a Mandelbrot set if it remains bounded after repeatedly calculating the formula in feedback loops. As an example, lets take $c = 1 - 1i$, it will go on repeatedly, like:

$$\begin{aligned} Z_0 &= 0 \\ Z_1 &= 0 + 1 - 1i \\ &= 1 - 1i \\ Z_2 &= (1 - 1i)^2 + 1 - 1i \\ &= 1 - 3i \end{aligned}$$

$$Z_3 = (1 - 3i)^2 + 1 - 1i$$
$$= -7 + 7i$$

and so on as long as the absolute value is still below 2. Although, it is at times impossible without the help of technology to determine whether it will go to infinity or will it stop at a certain iteration. This is why in a practical setting, the function feedback loop is run a maximum number of iterations, say *n* times. Furthermore, if the point still remains within the boundary of radius 2 we can consider the point that it belongs to the set. But why 2? It is possible to show but it will be time consuming, but generally if $|Z_n| \geq 2$, it will eventually escape towards infinity. This is commonly known as the bailout radius. A simple python code can be created to represent the recursive function.

```
import numpy #Import module numpy
MAX_ITERATION = 10 #Declare variable for maximum number of iteration, 10

def mandelbrot(c): #Define function mandelbrot with value c
   z = 0 #Declare variable z, this is complex number with initial value 0
   n = 0 #Declare variable n, this is iteration number with initial value 0
   while abs(z) <= 2 and n < MAX_ITERATION: #While the absolute value of z is still less
or equal to 2 and n is still below the number of maximum iteration

       z = z*z + c #Square z with itself, in this case z*z and add c
       n += 1 #Add 1 to n, in the loop to show number of iteration
   return n #Return n to the original function

if __name__ == "__main__": #It shows that after global variables are declared it will
move to this if statement first, this is because the def mandelbrot(c) have yet to
declare any value in its initial stage

   for a in numpy.arange(-2, 2, 0.5): #Change value of a with numbers in the range of -2
and 2 that is divisible by 0.5 (This is the imaginary part of the real number)

       for b in numpy.arange(-2, 2, 0.5): #Change value of b with numbers in the range
of -2 and 2 that is divisible by 0.5 (This is the imaginary part of the complex number)

           c = complex(a,b) #Change c value to a complex number with values of a and b

           print(c, mandelbrot(c)) #Display the values in the console, c represents the
complex number and mandelbrot(c) shows after how many iteration will it escape the
radius 2
```

*Figure 3: This is a simple python code that will show the complex number and how many times will it take for it to escape*

Moreover, the modulus of a complex number is its distance to 0 when we draw a right triangle in the complex plane. In python, this is obtained using the function *abs(z)* where *z* is a complex number. We will assume that the recursive function $Z_n$ is unbounded if the

modulus is greater than 2. In figure 3, anything that is in color dark grey is a detailed explanation on how the code works. To put it simply, the code will first declare the global variables of maximum iterations, and it will create a function of mandelbrot(c) that will be used to show how many iterations will it take for the absolute value of the complex number to escape the bailout radius, then it will proceed to next complex number if the variable n reaches 10 or the absolute value of *z* has already reached 2. The second part of the function, shows the basic mandelbrot function. In the main if statement, it will declare values of a and *b* in the 2 radius with values that are divisible by 0.5, as if it is changed to 0.01 it will produce a lot of values. As a and b are not in complex already it will use the function *complex()* and change variable *c* into a complex number. It will print the values out into the console. This process will be repeated over and over until all possible complex combinations within the range are achieved. An example of the print out values can be seen in figure 4.

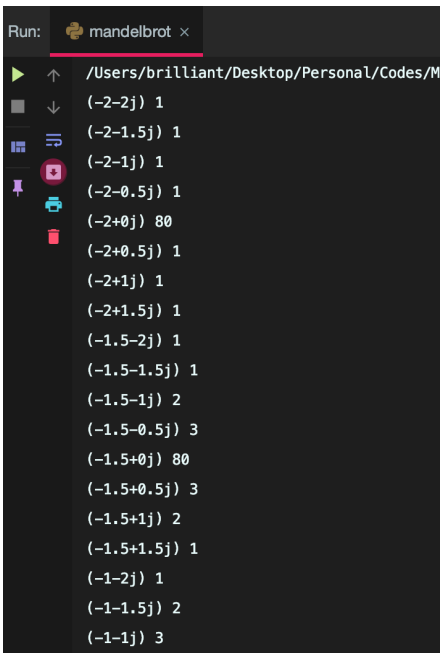



*Figure 4: Shows the print out values of the code above. Code referred to Condingame, citation below.*

## 5.1 Spreadsheet Plotting and Interpretation

The first method that I would like to discuss on how to visualise a Mandelbrot set is through a spreadsheet. Although spreadsheets are by far not the most reliable program to visualise a mandelbrot set, due to its frequent lag and unresponsiveness, nonetheless it gets the job done. This is essentially a starting point for us to set up a repetitive calculation, without doing it manually. It will also help us understand the concept before we implement another python code that can visualise the Mandelbrot, but for now just think of a single cell as a single pixel.

| Im/Re | 2,000 | 1,950 | 1,900 | 1,850 | 1,800 | 1,750 |
|---|---|---|---|---|---|---|
| 2,000 | 2+2i | 1,95+2i | 1,9+2i | 1,85+2i | 1,8+2i | 1,75+2i |
| 1,950 | 2+1,95i | 1,95+1,95i | 1,9+1,95i | 1,85+1,95i | 1,8+1,95i | 1,75+1,95i |
| 1,900 | 2+1,9i | 1,95+1,9i | 1,9+1,9i | 1,85+1,9i | 1,8+1,9i | 1,75+1,9i |
| 1,850 | 2+1,85i | 1,95+1,85i | 1,9+1,85i | 1,85+1,85i | 1,8+1,85i | 1,75+1,85i |
| 1,800 | 2+1,8i | 1,95+1,8i | 1,9+1,8i | 1,85+1,8i | 1,8+1,8i | 1,75+1,8i |
| 1,750 | 2+1,75i | 1,95+1,75i | 1,9+1,75i | 1,85+1,75i | 1,8+1,75i | 1,75+1,75i |
| 1,700 | 2+1,7i | 1,95+1,7i | 1,9+1,7i | 1,85+1,7i | 1,8+1,7i | 1,75+1,7i |
| 1,650 | 2+1,65i | 1,95+1,65i | 1,9+1,65i | 1,85+1,65i | 1,8+1,65i | 1,75+1,65i |
| 1,600 | 2+1,6i | 1,95+1,6i | 1,9+1,6i | 1,85+1,6i | 1,8+1,6i | 1,75+1,6i |
| 1,550 | 2+1,55i | 1,95+1,55i | 1,9+1,55i | 1,85+1,55i | 1,8+1,55i | 1,75+1,55i |

*Figure 5: Shows a brief image of the grid*

So the following will explain how I will visualise the Mandelbrot using the complex number function in excel. First I set up a grid of complex numbers. Refer to figure 5, with the real coefficient ranges from 2 to -2, and the imaginary coefficient ranges from 2 to -2. Both with intervals of 0.05, or in terms of the code in Figure 3 this is the divisible value. The entire inner grid are later on filled with the function of:

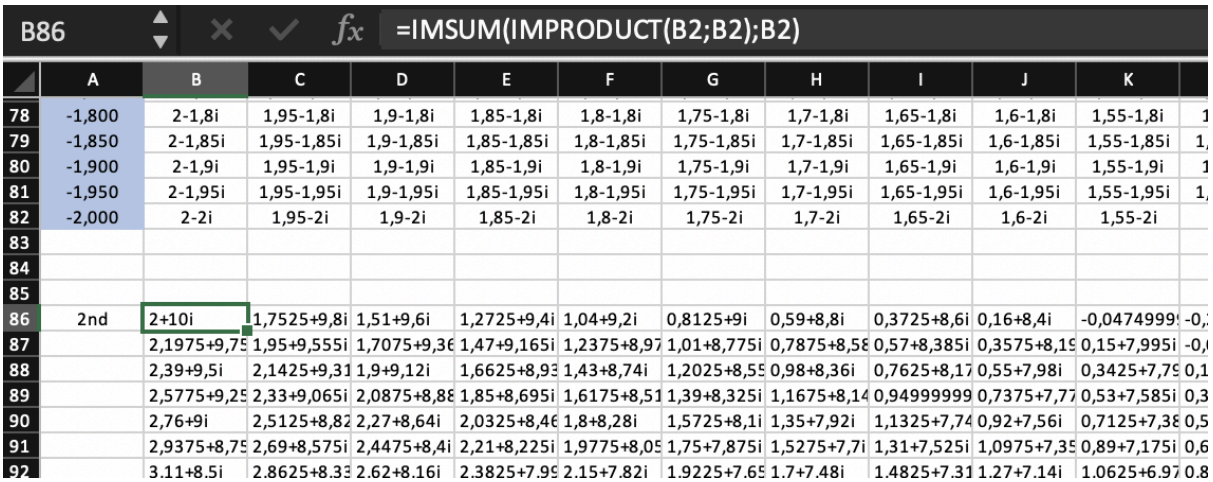
$$= COMPLEX(B\$1; \$A2)$$

This function is a simple function that will take the two coefficients and change it into a complex number. This is also the main purpose of using a spreadsheet as it has an autofill drag function which means that we can fill the grid by simply dragging the initial function of $= COMPLEX(B\$1; \$A2)$to the other grids. Keep in mind that example B1 is a reference to cell B and 1. One thing that may be obvious is the ***$*** symbol, this shows that in the first cell's case as I drag it across the alphabet will change but 1 will remain frozen, thus its $B\$1$. If I only want the value of the numbers to change but not the alphabet, as I go downwards I will change the position of the $ into the alphabet thus, $\$A2$. For different iterations the data will be referred from the previous set, and is calculated using a different function. Keep in mind that there is no square IMFUNCTION in the spreadsheet, therefore a broken down version of it was used. So, for the $(Z_n)^2$part will be calculated using the following function. This way it can be seen that for the next iteration the value of $Z_n$will refer to the previous iteration, but the constant c will always refer to the very first combination of complex numbers.

$$= IMSUM(IMPRODUCT(B2; B2); B2)$$

The interpretation for the following function is, first it will get the previous iteration and get the complex number that is on the same position on the grid, and multiply it by itself. So, for iteration 2 the complex number in B2 will be multiplied by B2, this is why it's called the

broken down version of the square. The IMPRODUCT will later on be added using IMSUM that represents *c* which is the initial value on the same point on the first grid.



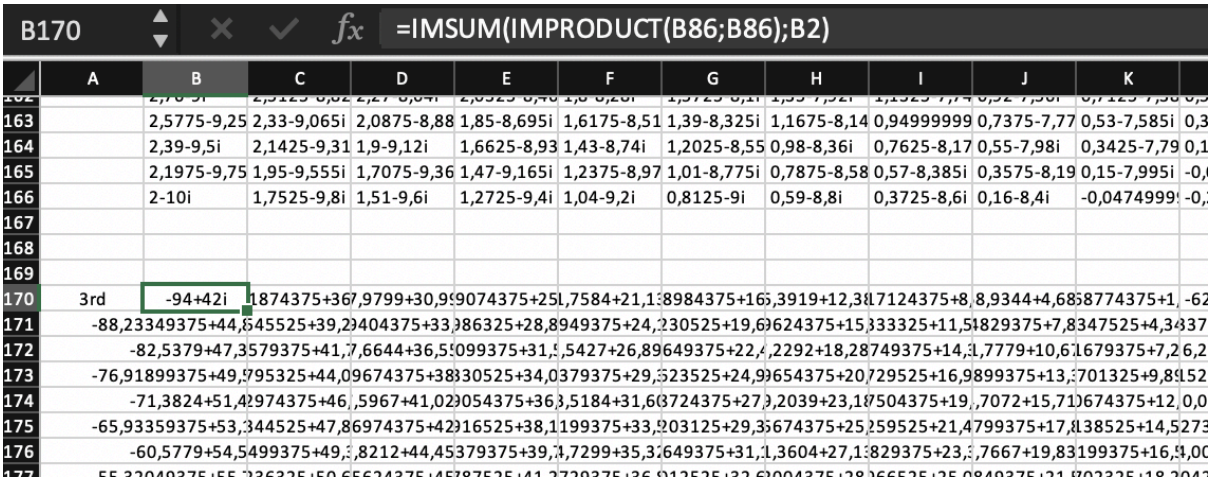

| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 78 | -1,800 | 2-1,8i | 1,95-1,8i | 1,9-1,8i | 1,85-1,8i | 1,8-1,8i | 1,75-1,8i | 1,7-1,8i | 1,65-1,8i | 1,6-1,8i | 1,55-1,8i |
| 79 | -1,850 | 2-1,85i | 1,95-1,85i | 1,9-1,85i | 1,85-1,85i | 1,8-1,85i | 1,75-1,85i | 1,7-1,85i | 1,65-1,85i | 1,6-1,85i | 1,55-1,85i |
| 80 | -1,900 | 2-1,9i | 1,95-1,9i | 1,9-1,9i | 1,85-1,9i | 1,8-1,9i | 1,75-1,9i | 1,7-1,9i | 1,65-1,9i | 1,6-1,9i | 1,55-1,9i |
| 81 | -1,950 | 2-1,95i | 1,95-1,95i | 1,9-1,95i | 1,85-1,95i | 1,8-1,95i | 1,75-1,95i | 1,7-1,95i | 1,65-1,95i | 1,6-1,95i | 1,55-1,95i |
| 82 | -2,000 | 2-2i | 1,95-2i | 1,9-2i | 1,85-2i | 1,8-2i | 1,75-2i | 1,7-2i | 1,65-2i | 1,6-2i | 1,55-2i |
| 83 | | | | | | | | | | | |
| 84 | | | | | | | | | | | |
| 85 | | | | | | | | | | | |
| 86 | 2nd | 2+10i | 1,7525+9,8i | 1,51+9,6i | 1,2725+9,4i | 1,04+9,2i | 0,8125+9i | 0,59+8,8i | 0,3725+8,6i | 0,16+8,4i | -0,0474999 |
| 87 | | 2,1975+9,75 | 1,95+9,555i | 1,7075+9,36 | 1,47+9,165i | 1,2375+8,97 | 1,01+8,775i | 0,7875+8,58 | 0,57+8,385i | 0,3575+8,19 | 0,15+7,995i |
| 88 | | 2,39+9,5i | 2,1425+9,31 | 1,9+9,12i | 1,6625+8,93 | 1,43+8,74i | 1,2025+8,55 | 0,98+8,36i | 0,7625+8,17 | 0,55+7,98i | 0,3425+7,79 |
| 89 | | 2,5775+9,25 | 2,33+9,065i | 2,0875+8,88 | 1,85+8,695i | 1,6175+8,51 | 1,39+8,325i | 1,1675+8,14 | 0,94999999 | 0,7375+7,77 | 0,53+7,585i |
| 90 | | 2,76+9i | 2,5125+8,82 | 2,27+8,64i | 2,0325+8,46 | 1,8+8,28i | 1,5725+8,1i | 1,35+7,92i | 1,1325+7,74 | 0,92+7,56i | 0,7125+7,38 |
| 91 | | 2,9375+8,75 | 2,69+8,575i | 2,4475+8,4i | 2,21+8,225i | 1,9775+8,05 | 1,75+7,875i | 1,5275+7,7i | 1,31+7,525i | 1,0975+7,35 | 0,89+7,175i |
| 92 | | 3,11+8,5i | 2,8625+8,33 | 2,62+8,16i | 2,3825+7,99 | 2,15+7,82i | 1,9225+7,65 | 1,7+7,48i | 1,4825+7,31 | 1,27+7,14i | 1,0625+6,97 |

*Figure 6: Shows the second iteration of the Mandelbrot set*

A similar process can be used to calculate for iteration 3 simply use the function:

$$= IMSUM(IMPRODUCT(B86; B86); B2)$$

See how the *B* values in the IMPRODUCT function changes, and if we refer to figure 6 we can see that cell *B86* is the first value of the second iteration. But keep in mind the IMSUM will still refer to the first iteration in this case B2. Refer to figure 7.




| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 163 | | 2,5775-9,25 | 2,33-9,065i | 2,0875-8,88 | 1,85-8,695i | 1,6175-8,51 | 1,39-8,325i | 1,1675-8,14 | 0,94999999 | 0,7375-7,77 | 0,53-7,585i |
| 164 | | 2,39-9,5i | 2,1425-9,31 | 1,9-9,12i | 1,6625-8,93 | 1,43-8,74i | 1,2025-8,55 | 0,98-8,36i | 0,7625-8,17 | 0,55-7,98i | 0,3425-7,79 |
| 165 | | 2,1975-9,75 | 1,95-9,555i | 1,7075-9,36 | 1,47-9,165i | 1,2375-8,97 | 1,01-8,775i | 0,7875-8,58 | 0,57-8,385i | 0,3575-8,19 | 0,15-7,995i |
| 166 | | 2-10i | 1,7525-9,8i | 1,51-9,6i | 1,2725-9,4i | 1,04-9,2i | 0,8125-9i | 0,59-8,8i | 0,3725-8,6i | 0,16-8,4i | -0,0474999 |
| 167 | | | | | | | | | | | |
| 168 | | | | | | | | | | | |
| 169 | | | | | | | | | | | |
| 170 | 3rd | -94+42i | 1874375+36 | 7,9799+30,99 | 9074375+25 | 1,7584+21,13 | 8984375+16 | 5,3919+12,38 | 17124375+8 | 8,9344+4,68 | 58774375+1 |
| 171 | | -88,23349375+44, | 645525+39,2 | 9404375+33 | 986325+28,8 | 949375+24, | 230525+19,6 | 9624375+15 | 333325+11,5 | 1829375+7,8 | 347525+4,34 |
| 172 | | -82,5379+47,3 | 579375+41, | 7,6644+36,5 | 099375+31, | 5427+26,89 | 649375+22, | 2292+18,28 | 749375+14, | 1,7779+10,6 | 1679375+7,2 |
| 173 | | -76,91899375+49, | 795325+44,0 | 9674375+38 | 330525+34,0 | 379375+29, | 523525+24,9 | 9654375+20 | 729525+16,9 | 899375+13, | 701325+9,89 |
| 174 | | -71,3824+51,4 | 2974375+46, | 5967+41,02 | 054375+36 | 3,5184+31,6 | 3724375+27 | 2039+23,18 | 7504375+19, | 7072+15,71 | 674375+12, |
| 175 | | -65,93359375+53, | 344525+47,8 | 6974375+42 | 916525+38,1 | 199375+33, | 203125+29,3 | 5674375+25 | 259525+21,4 | 799375+17,8 | 38525+14,52 |
| 176 | | -60,5779+54,5 | 499375+49, | 8212+44,45 | 379375+39, | 1,7299+35,3 | 649375+31, | 1,3604+27,13 | 829375+23, | 7667+19,83 | 199375+16, |
| 177 | | -55,32049375+55, | 236325+50, | 6624375+45 | 887525+41, | 2729375+36, | 912525+32,6 | 1004375+28, | 266525+25, | 0849375+21, | 702325+18,2 |

*Figure 7: Shows the third iteration of the Mandelbrot set*

This recursive process was repeated until I can see an outline of the Mandelbrot set using this function. Refer to figure 8 to get an idea on how many cells was used to do 10 iterations in an 81x81 grid. Every white skip line it shows that it is changing iterations.

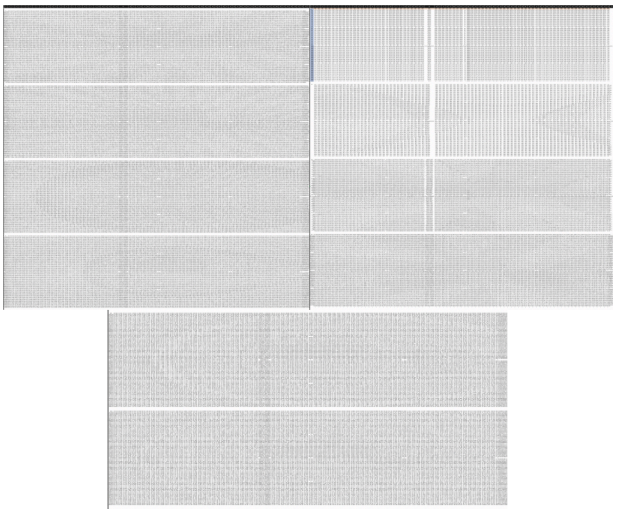


*Figure 8: Shows how many data points did it take to do 10 iterations in an 81x81 cell values*

To display the complex numbers from each iteration the following function can be used to display the Mandelbrot.

$$= IMABS(Iterations!B2)$$

As I am looking for the absolute value of the complex number in the cell the function *IMABS()* is used and in this case with reference to cell *B2*, *Iterations!* is also in the function as I am referring to the data from the tab named Iterations. As when I put everything into a single file it becomes very unresponsive. Referring to figure 9, it will show the absolute value calculations for iteration 1.

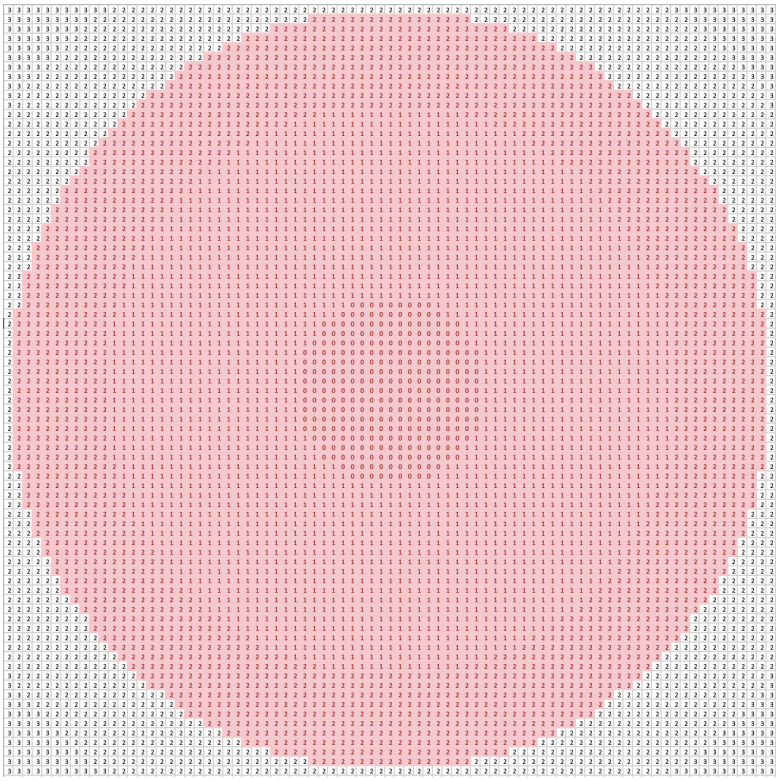


*Figure 9: The visualisation of iteration 1, zoom in to view the modulus values*

When we look closely into the grid, I intentionally made this into a square so we can see the actual shape, we can see that there are numbers ranging from 3 to 0, and later on in the following iterations we can even see some that go to infinity. While these are not the real absolute value of the corresponding complex number, they are rounded values to the first significant figure, as they still show the values that we need. Conditional formatting was also used in this grid, which can be seen from the red colored cells, these are the cells that have an absolute value of less than 2. Remember, as the Mandelbrot set can only be applied in the radius 2. If we keep on using this function to display the tenth iteration, the outlines of the part of the Mandelbrot set of radius 2 can begin to be seen. Refer to figure 10.

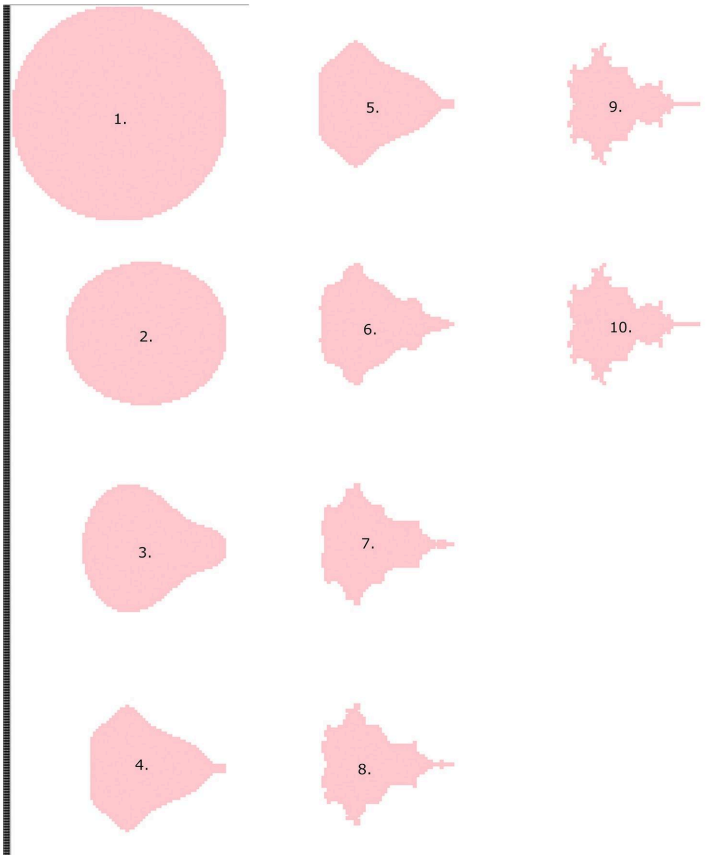



*Figure 10: Shows the Mandelbrot of all 10 iterations using datas from the spreadsheet*

## 5.2. Python Plotting, Interpretation, and Escape Time Algorithm

Another method on plotting a Mandelbrot function is by using a python code, unlike the first python code this will use the escape time algorithm, which is a common algorithm that is used to colour the aura of the set, or simply the number of iteration layers. Refer to figure 11 for example.

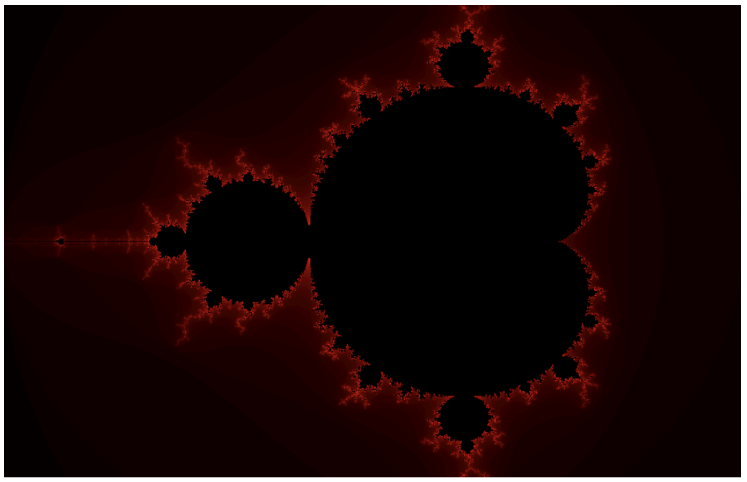


*Figure 11: The layers of the set colored using the escape algorithm, the inner black shape is the actual Mandelbrot set.*

The escape time algorithm is based on the value of *n*, which as I have stated before, the number of iterations before the *Z* value reaches the bailout radius. This is a really simple way to visualise the different layers or iterations in a Mandelbrot set. One of the ways of defining the colouring function is by using a piecewise function.

$$c(n)=\begin{cases}0 & n=max\_iter \\ 255 - n/max\_iter + 1 & n < max\_iter\end{cases}$$

In this case the function, *c(n)* will return the red component in the RGB-colour palette. Referring to figure 12, it can be seen that the value of only the red component ranges from 0 to 255, that is why when iteration *n* is still less then max iteration it will keep on changing the value of the red component until it stops. But, keep in mind that it will run firstly run the first statement, if n is equal to max_iter it will by default change the red component to 0, or black. It is also possible to approach this by making the iteration layers with multiple colors, simply by creating a function for every color component in the RGB. In this case, $c_r(n), c_g(n)$ and $c_b(n)$ where $c_r(n)$ returns the value of the red component, $c_g(n)$ the green component and $c_b(n)$ the blue component. This concept will be implemented in the python code later.

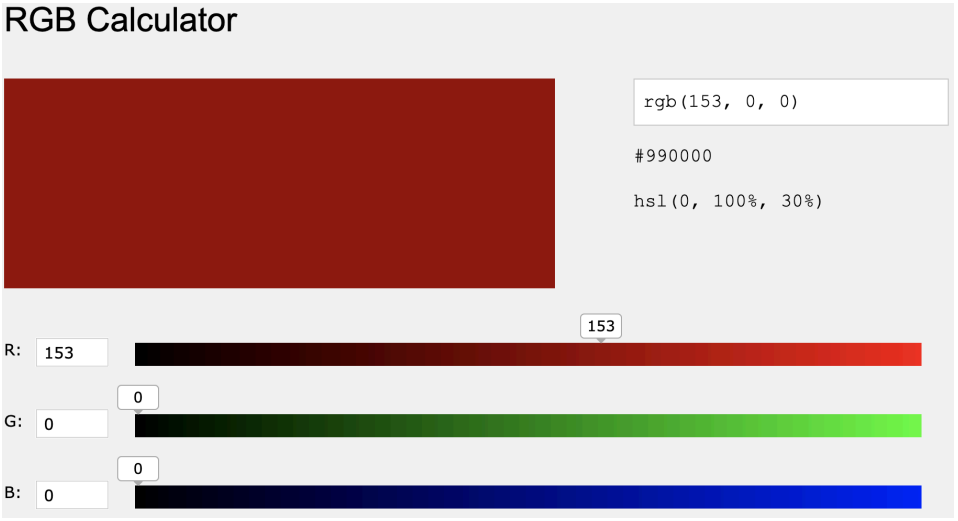



*Figure 12: It shows how only the red component can be varied*

Plotting the mandelbrot set through python code is relatively simple, as it is just a code that goes through a number of loops. First we would code it to iterate over all the pixels of our image or canvas in this case, then we want to convert the coordinate of the pixel into the complex number plane. Essentially think of this pixel as a cell in the spreadsheet the concept is the same. Then, call the function *mandelbrot(c)*, if *mandelbrot(c)* returns *MAX_ITERATION,* plot a black pixel, otherwise the pixel color will depend on the number of iterations.

```
from PIL import Image, ImageDraw #From module PIL import Image and ImageDraw

RE_AXIS = 100 #Declaring the axis width or canvas width
IM_AXIS = 100 #Declaring the axis height or canvas height

RE_START = -2 #Declaring the start of the Real axis
RE_END = 2 #Declaring the end of the Real axis
IM_START = -2 #Declaring the start of the Imaginary axis
IM_END = 2 #Declaring the end of the Real axis

MAX_ITERATION = 10 #Declaring the maximum number of iterations, go to im = image.new first for
initial startup

def mandelbrot(c): #Creating function mandelbrot with value c
   z = 0 #Declare variable z to initial 0, this will be the complex number
   n = 0 #Declare variable n to initial 0, this will be the number of iterations

   while abs(z) <= 2 and n < MAX_ITERATION: #While the absolute value of z is less or equal to two
and n is still less than the max iteration

       z = z*z + c #Mandelbrot recursive function, z is complex and c is the initial complex
       n += 1 #Change the value of n by +1 for every loop of the function
 return n #Return value of n if absolute value is not 2 or max iteration not reached repeat

im = Image.new('HSV', (RE_AXIS, IM_AXIS), (0, 0, 0)) #Declare variable im with value Image.new HSV
is the color mode, (RE_AXIS, IM_AXIS) declares the width and the height in pixels and the last
declares the initial color which is black
draw = ImageDraw.Draw(im) #Draw the plane with the parameters above
```

```
for x in range(0, RE_AXIS): #Insert x value from 0 to RE_AXIS, this is the values of real
coefficients in terms of pixels

   for y in range(0, IM_AXIS): #Insert y value from 0 to IM_AXIS, this is the values of real
coefficients in terms of pixels

       c = complex(RE_START + (x / RE_AXIS) * (RE_END - RE_START), IM_START + (y / IM_AXIS) *
(IM_END - IM_START)) #Insert value c with complex, with real value of pixel chosen divided by total
RE_AXIS, and multiplied by R(E_END - Re_Start) this will give us the total points in terms of the
complex plane. This value will be added to RE_START as we are starting from the top. This process is
also repeated for the imaginary part. But with imaginary parameters

       m = mandelbrot(c) #Declare variable m with the value of mandelbrot(c), or iterations
       hue = int(255 * m / MAX_ITERATION) #Declare variable hue with integer value of 255 total
number of colors and multiplied by the iteration for a complex number divided by the maximum
iteration this will correspond to a color. This is the implementation of the ESCAPE TIME ALGORITHM.
       saturation = 255 #Saturation will remain 255 which is default as we are not changing the
saturation, we only want the color
       value = 255 if m < MAX_ITERATION else 0 #Declare variable value with value of 255 default it
will shows color if m is still less than MAX_ITERATION, else it will return 0 which is black
       draw.point([x, y], (hue, saturation, value)) #It will draw the point in the complex plane
with pixel coordinates of the used x and y, with the color HSV

im.convert('RGB').show() #It will convert the colors and present it into a new tab
```

*Figure 13: This is a python code that will calculate the complex number in terms of pixels and translate it into pixel grid drawing with colors to show the different iterations. Modified code by Condingame, citation below.*

Text in dark grey above is commentary. With the following code above, the RGB colors are changed to HSV colors. As unlike RGB, in HSV we can easily change the color by only modifying the hue component. An example of the mathematical concept can be seen in the following. Variables used in this example will refer to the code above.

**Pixel chosen x = 0 and y = 1**

1. $Declare\ global\ variables$
2. $Take\ value\ x\ =\ 0,\ and\ value\ y\ =\ 1$
3. $Change\ c\ to\ complex\ =\ (-\ 2\ +\ ((0\ \div\ 1000)(2\ --\ 2))\ ,-\ 2\ +\ ((1\ \div\ 1000)(2\ --\ 2)))$
4. $c\ =\ \ -\ 2\ -\ 1.996i$
5. $Go\ to\ mandelbrot(c)\ declare\ z\ =\ 0,\ and\ n\ =\ 0$
6. $abs(z)\ =\ modulus\ =\ |\ -\ 2\ -\ 1.996i|$
7. $abs(z)\ =\ modulus\ =\ 2.8256$
8. $abs(z)\ >\ 2\ return\ n\ =\ 1$
9. $hue\ =\ (255\ \times\ (1\ \div\ 10))\ ,\ saturation\ =\ 255,\ value\ =\ 255$
10. $Draw\ pixel\ in\ grid\ with\ corresponding\ color$

**Pixel chosen x = 400 and y = 101**

1. $Declare\ global\ variables$
2. $Take\ value\ x\ =\ 400,\ and\ value\ y\ =\ 101$

3. $Change\ c\ to\ complex\ =\ (-\ 2\ +\ ((400\ \div\ 1000)(2\ --\ 2))\ ,-\ 2\ +\ ((101\ \div\ 1000)(2\ --$
4. $c\ =\ \ -\ 0.3999999999999999\ -\ 1.6i$
5. $Go\ to\ mandelbrot(c)\ declare\ z\ =\ 0,\ and\ n\ =\ 0$
6. $abs(z)\ =\ modulus\ =\ |\ -\ 0.3999999999999999\ -\ 1.6i|$
7. $abs(z)\ =\ modulus\ =\ 1.64536$
8. $abs(z)\ <\ 2\ return\ n\ =\ 1$
9. $z\ =(-\ 0.399999999\ -\ 1.596i)(-\ 0.399999999\ -\ 1.596i)\ +\ (-\ 0.399999999\ -\ 1.596$
10. $z\ =(-\ 2.787216\ -\ 0.319204i)$
11. $n\ =\ 2$
12. $abs(z)\ =\ modulus\ =-\ 2.787216\ -\ 0.319204i|$
13. $abs(z)\ =\ modulus\ =\ 2.80543$
14. $abs(z)\ >\ 2\ return\ n\ =\ 2$
15. $hue\ =\ (255\ \times\ (2\ \div\ 10))\ ,\ saturation\ =\ 255,\ value\ =\ 255$
16. $Draw\ pixel\ in\ grid\ corresponding\ color$

This recursive process will keep on repeating until all pixels within the specified parameters are calculated. Visualising the model it will show a model that is closely similar to the excel set just with colors and in more detail. Figure 14 will show the computer generated Mandelbrot set in a 100x100 pixel with *MAX_ITERATIONS* 10 iterations, similar iteration specifications are also used in the excel but rather than pixel it uses cells. From both sets of python code and spreadsheet we can see that they both produced similar diagrams. With the right most image verifying by overlapping both images.

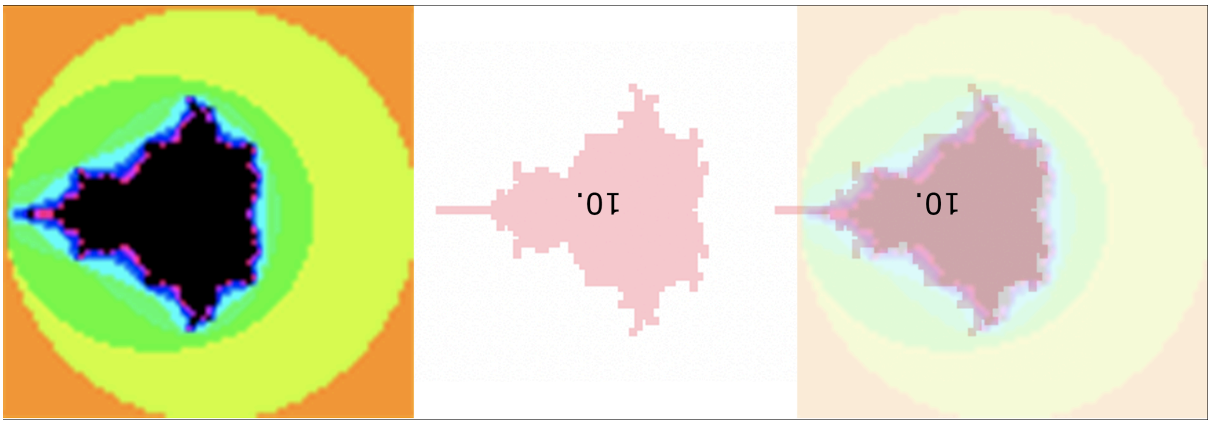


*Figure 14: Left image shows the computer generated set of 100x100 with 10 iterations, center image shows the set generated from the spreadsheet, right image shows both sets overlapping to show similarity.*

Multiple parameters and iteration values can also be used to produce different mandelbrot sets with different precision qualities. In figure 15, we can see the difference of the pixel count, the width and height, and how it can possibly affect the mandelbrot set that is generated. In terms of spreadsheet, this increase in pixel count means the smaller the increments if we translate it into the complex plane. From 100x100 mandelbrot, we can see that the image is very pixelated, the bottom part of the figure also shows the 3x zoom of

the set, with the 100x100 pixel becoming extremely visible. As the pixel count goes up so will the image quality or the increments of the complex number. So in terms of the complex numbers with the example of $1 + 1i$, the coefficient for the next complex number will be $1 + 1.001i$, if we only vary the imaginary part, in this case the increment is by 0.001. As mentioned before with the increase in increments so will pixel counts, thus the detail of the set.

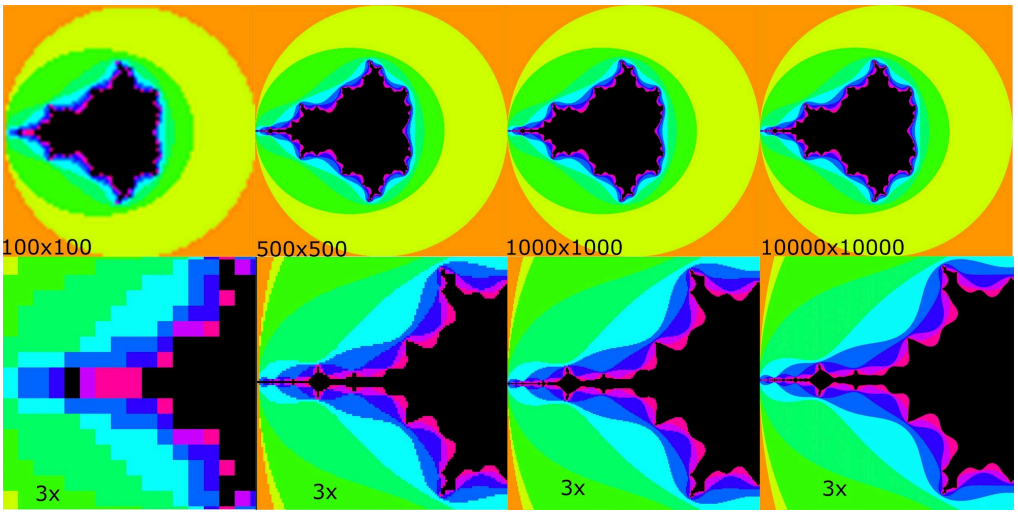



*Figure 15: The upper sets are original size with different canvas sizes, lower sets are the zoomed 3x of the upper sets.*

Another way to observe the change in data is by increasing its number of iterations. In the basic function of $Z_{n+1} = (Z_n)^2 + c$, we are referring to the *n* value. In the terms of the code the value of *MAX_ITERATION* was changed to visualise these changes. Keep in mind that all these sets are made with 1000x1000 pixel grid. As seen in figure 16, I began from 10 iterations, which shows distinctive color, almost all the shades available by the HSV are displayed in the set. But it only shows 10 different layers of the mandelbrot set, and when we zoom in we can actually see the end of the iteration. In the 50 iteration model, we can see there are still layers outside the actual mandelbrot but they're HSV colors are of a similar shade, which means that through the escape time algorithm they return similar *n* values. Although when we zoom in we can still see the end of the set. For the 100 iterations, this is where it gets interesting, the color of the radius 2 complex plane is starting to blend in with the complex number values that escaped with first iterations, which is essentially the 2x2 complex plane grid. The lighter colors are beginning to converge into the actual mandelbrot set leaving the outer region with only a few hue values. For the last one, the 1000 iterations model we can actually see the entirety of the outer plane is purely blood red, we can no longer see the radius 2 complex plane circle, this is due to the fact that most of the colors are used in a wider range of iterations. We can think of this like intervals if we are looking at 10 iterations it will mean that we are looking at 10 different shades of color, same with 50 and 100. But, in the case of 1000 iterations, the hue values that consist of only 255 values cannot accommodate all its corresponding colors, they eventually begin to

minimize their intervals to the point of decimal places. For example, if the outer region of 100 iteration is *hue = 255* and the radius 2 circle have a *hue = 254*, they can still accommodate all the values with the same *n* with their own colors. But in the case of 1000 iterations, they need to go into the decimal places, so in this case the radius 2 is actually there but the shade of the hue value of the outer region and the radius 2 is very minimal that we won't even see the difference. Another interesting point with the 1000 iterations is that when we zoom in we can still see the recursive function repeating over itself without showing any visible end to it, this is due to the fact that the more *n* values the more layers there will be in the model.

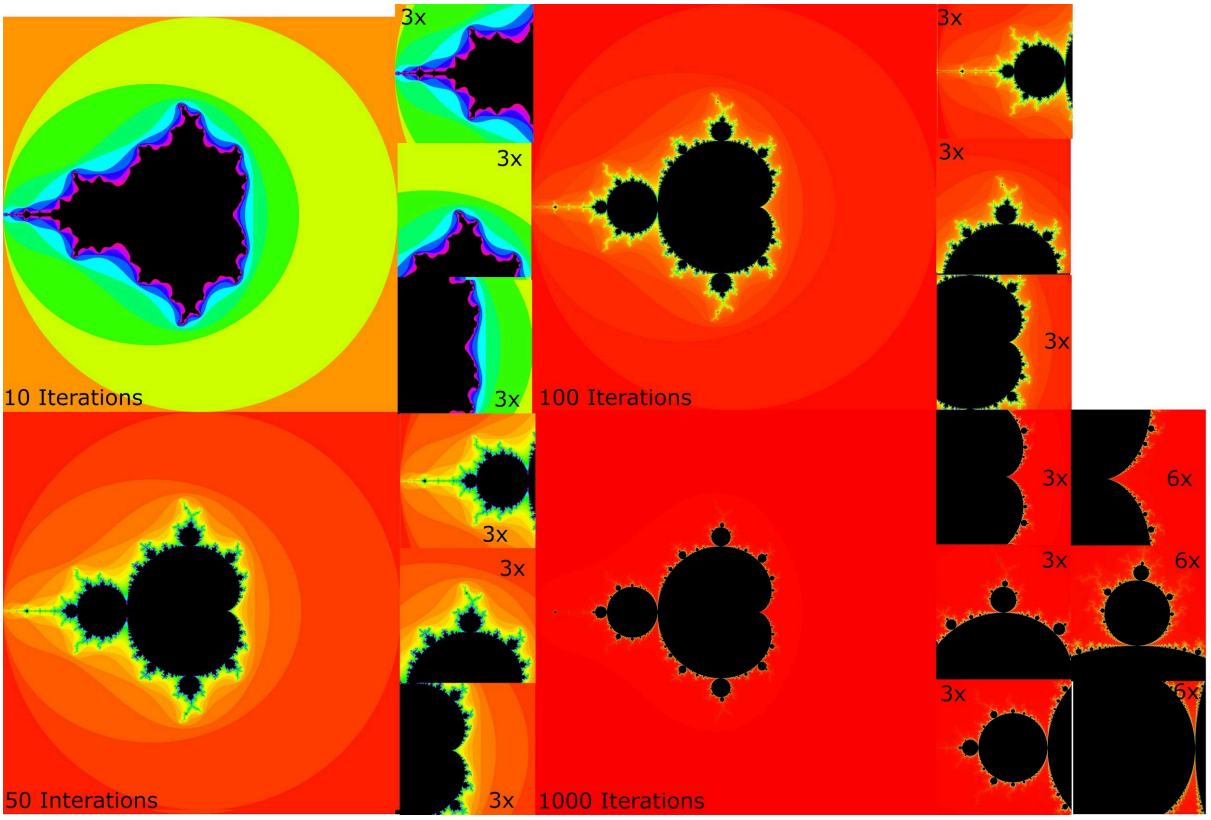



*Figure 16: This shows the four different iterations that I used to visualise the difference, 10, 50, 100, and 1000. Zoomed in images are also shown on each set's right.*

## 6. Chaotic Data Iterations

Chaos theory is simply the study of literal chaos, which implies that it is the study of irregularities in mathematics. In the mandelbrot fractal after we plug in the function *Z* with multiple complex numbers, the set is equal to all of the expressions that are generated. The plot below will describe using a time series, refer to figure 17, which shows the plots for a specific *c* value. To help demonstrate this we used the numbers -1.1, -1.3, -1.38 and -1.9, keep in mind these are not in the complex plane. The first three can be expressed as a normal mathematical function whose yield values are consistent. Whereas for c = -1.9 we

can't, the data are in disorder or irregular. In simple terms, it can be inferred that when c is -1.1, -1.3, -1.38 their function is deterministic, while when c = -1.9 the function is chaotic.


*Figure 17: Time series plot for a variety of of c values*

## 7. Application in Cryptography

Fractals are not only recursive patterns in a graphic model. The chaotic nature of fractals have made them incredibly popular in the fields of cryptography, where they are used for functions and encryption. Unlike matrix cryptography, where we used a common cipher to decrypt the message, and once that cipher is cracked, that message will be leaked. With a key exchange protocol based on the mandelbrot set it will be more secure. The concept is called fractal key exchange. In cryptography, a key exchange is the process of two parties, 'X' and 'Y', exchanging keys with each other allowing the use of a cryptographic algorithm. This is done through a key exchange protocol known as Dieffie-Hellman, first suggested by Whifield Diffie and Martin Hellman. A key exchange protocol that will be discussed is similar to Dieffie-Hellman The key exchange protocol has a distinctive property compared to other algorithms; even if 'Z' intercepted the traffic sent between 'X' and 'Y', 'Z' would still not be able to decrypt the message.

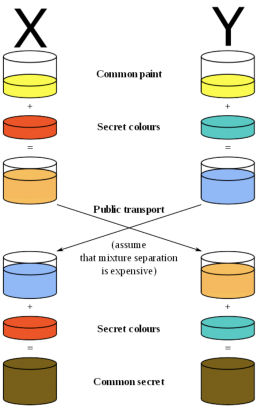



Figure 18: A diagram on how the protocol algorithm works

The algorithm can be represented by the following equivalence relation:

$$c^{n-k}q_n e = c^{k-x}q_k d \quad \forall x \in \mathbb{Z} \quad where\ c,\ d,\ e \in \mathbb{C}\ and\ a,\ b \in \mathbb{N}$$

Where $q_n \in \mathbb{C}$ is the result from running q in a feedback loop *n* times. $P_x$ and $P_y$ are public keys for 'X' and 'Y' respectively.

1. The algorithm begins with 'X' and 'Y' creating on a complex number *c* which belongs to the mandelbrot set and an integer x. c and x are public it would not matter if it is intercepted by 'Z'
2. 'X' will create a secret key consisting of the tuple, or a sequence of values, (a, e) where *a* > *x* and *e* will belong to the mandelbrot set.
3. 'Y' will create a secret key consisting of the tuple, or a sequence of values, (b, d) where *b* > *x* and *d* will belong to the mandelbrot set.
4. 'X' calculates $Z_a$ by iterating the formula $Z_{n+1} = Z_n c^2 e, \ where \ Z_0 = c$, *a* times and sends $P_x = Z_a e$ to 'Y'
5. 'Y' calculates $Z_b$ by iterating the formula $Z_{n+1} = Z_n c^2 d, \ where \ Z_0 = c$, *b* times and sends $P_y = Z_b d$ to 'X'
6. 'X' calculates the common secret the formula $c^{a-x} q_n e$ , by iterating the formula $q_{n+1} = q_n ce, where \ q_0 = P_y$ *a* times.
7. 'Y' calculates the common secret the formula $c^{b-x} q_n d$ , by iterating the formula $q_{n+1} = q_n cd, where \ q_0 = P_x$ *a* times.

The keyspace, the set of all possible permutations of a key, given a key with length n, is larger for the fractal key exchange algorithm than the keyspace for Dieffe-Helman developed. In the case of the fractal key exchange, unlike the Dieffe-Helman we can potentially end up with any key. The key exchange protocol also elaborates the relationship of the mandelbrot and julia set.

## 8. Conclusion

I hope with this paper the reader of this research paper can get a better understanding of the mandelbrot set. As elaborated in the paper, it has several real life practical applications, not only for computer graphics. In essence, the fractal is like all other patterns; it's a never ending cycle of repetition of itself, or even at times it can also be chaotic. There are still many practical applications of the mandelbrot set, or even the fractals as a whole is still open to a wide range of possibilities. One particular research that interests me is by Lori Gardi, where she did research on the mandelbrot set and how it can be related to the Quasi-Black hole, through mathematical and relativity theories. All in all it has been an interesting research and exploration into the world of maths.

## Work Cited Page

[redacted]


- https://www.semanticscholar.org/paper/New-Key-Exchange-Protocol-Based-on-Mandelbrot-and-Alia-Samsudin/2c1042a16eb034dcc286f78d6341bfe5ee164367
- https://www.ibm.com/ibm/history/ibm100/us/en/icons/fractal/
- https://www.codingame.com/playgrounds/2358/how-to-plot-the-mandelbrot-set/mandelbrot-set
- Mandelbrot understanding from Oxford book
- http://www.fundamentalfinance.com/excel/projects/mandelbrot-set-in-excel.php
- https://danbscott.ghost.io/mandelbrot-set-in-excel/
- http://www.tnellen.com/alt/chaos.html
- https://science.howstuffworks.com/math-concepts/chaos-theory6.htm
- http://www.alunw.freeuk.com/mandelbrotroom.html
- https://www.karlsims.com/julia.html
- https://www.researchgate.net/publication/277211340_New_Key_Exchange_Protocol_Based_on_Mandelbrot_and_Julia_Fractal_Sets
- https://www.researchgate.net/publication/270285889_THE_MANDELBROT_SET_FRACTAL_GEOMETRY_AND_BENOIT_MANDELBROT_-_The_Life_and_Work_of_a_Maverick_Mathematician
- https://www.researchgate.net/publication/264018922_Superior_Mandelbrot_Set
- https://www.kth.se/social/files/5504b42ff276543e4aa5f5a1/An_introduction_to_the_Mandelbrot_Set.pdf