Investigating fractals

Abstract

In this report, an investigation was conducted on fractals, including generating one and their applications in art, scientific and financial fields. While generating a Mandelbrot fractal, we consider the error and the limitations imposed by the computing device and software. These results were generated and compared to the actual representation. Potential improvements in computing were also stated
Student 1 – Gurraunak Singh Bedi
Student 2 – Neeraj Nandan Akella
Student 3 – Baran Altun
Student 4 – Callum Connell
Student 5 – Sam Clarke
Student 6 – Shayan
Student 7 – Hajar Bouchouya

Introduction

Fractals are mathematical shapes that exhibit self-similarity at different scales. This means that as you zoom in on a fractal, you see patterns that resemble the whole object. Natural fractals like coastlines and clouds have infinitely complex perimeters yet finite areas due to the perimeter becoming exponentially bigger as the measuring resolution increases. One-way fractals can be mapped mathematically using the iterative Mandelbrot formula,

$$Z_{n+1} = Z^2 + C$$

where Z is set initially equal to zero, and C is a complex number. The first value produced will be C. Then for the next iteration of n = 1, the value produced will be C2 + C. The Mandelbrot set refers to the set of values of C which cause the iterative value to converge at a fixed value, i.e., not diverge to infinity

Calculating Fractal Dimension

The calculation for fractal dimension, first introduced by Benoit Mandelbrot, arises from a simple observation. By noticing that lines, squares, and cubes are self-similar some crucial insights can be made¬:

A line of length 2 units can be constructed by 2 1-unit lines
A square of length 2 units can be constructed by 4 (or 2^2) 1-unit^2 squares
A cube of length 2 units can be constructed by 8 (or 2^3) 1-unit^3 cubes

Therefore, the number of unit lengths/squares/cubes (N) it takes to form the original shape can be calculated by raising the side length of the original shape (S) by the dimension of the shape (D):

$$N = S^D$$

This equation can be used to calculate dimensions for self-similar fractals:

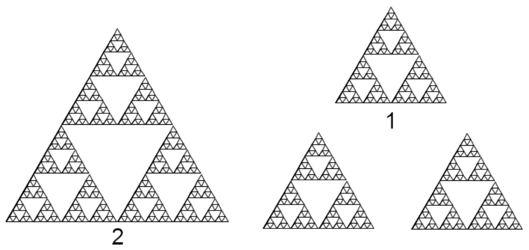


Figure 1 - triangular self-similar fractals

$$3 = 2^D$$

$$D = \log_2 3 = 1.585$$

Hence, fractals are ‘Shapes which have a non-integer dimension’.

The same idea can be extended to fractals which are not self-similar. Instead of counting how many unit shapes are required to form the original shape, we instead count the number of unit cells touched by the fractal like in the case of coastlines. This will be further discussed below.

Generating a Mandelbrot fractal

As stated above the Mandelbrot set is the set of all complex numbers where the function -:

$$f(z) = z^2 + c$$

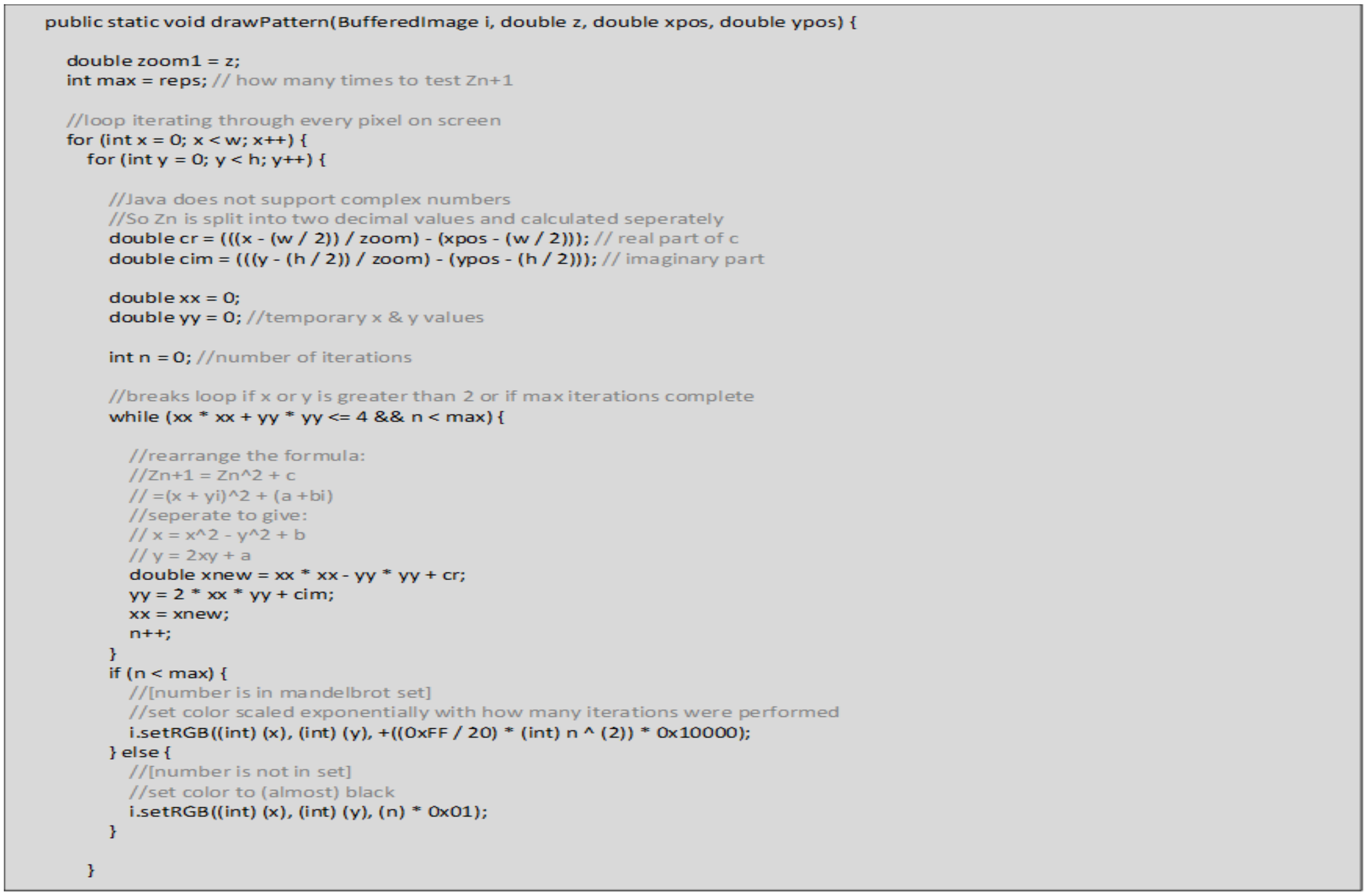
does not diverge to infinity when iterated (with $z_0 = 0$). By plotting this set graphically, with the x-axis coordinates representing the real component of *z* and the y-axis components representing the imaginary component, a fractal image can be generated, with an infinite amount of detail (limited by the computational power available).

In order to program this, the formula used was:

$$z_{n+1} = z_n^2 + c$$

And iterated through, starting with a value of zero. In order to test whether or not the value for *z* remains bounded, the iteration is capped at a certain point '*N*'. If the program loops N times, then it is assumed that the complex number $x + yi$ is part of the Mandelbrot set. These coordinates can then be used to fill a pixel array that can be rendered on screen, and a pattern can be generated by shading in pixels depending on whether or not they are in the Mandelbrot set. Values that do lie within the Mandelbrot set are shaded black, and values that are unbounded when inserted into equation 2 are shaded based on how many iterations it took to exceed a value of *at least* 2 (as -2 is the largest complex number in the set). By multiplying the $\frac{number\ of\ iterations}{\max iterations}$ by a constant RGB value (in this case 0x10000 to give a red shading) the Mandelbrot pattern can be seen

The software also needs to adjust the image to account for adjusting the position and 'magnification' set by the user. The final code for generating the image is as follows:



```
public static void drawPattern(BufferedImage i, double z, double xpos, double ypos) {

  double zoom1 = z;
  int max = reps; // how many times to test Zn+1

  //loop iterating through every pixel on screen
  for (int x = 0; x < w; x++) {
    for (int y = 0; y < h; y++) {

      //Java does not support complex numbers
      //So Zn is split into two decimal values and calculated seperately
      double cr = (((x - (w / 2)) / zoom) - (xpos - (w / 2))); // real part of c
      double cim = (((y - (h / 2)) / zoom) - (ypos - (h / 2))); // imaginary part

      double xx = 0;
      double yy = 0; //temporary x & y values

      int n = 0; //number of iterations

      //breaks loop if x or y is greater than 2 or if max iterations complete
      while (xx * xx + yy * yy <= 4 && n < max) {

        //rearrange the formula:
        //Zn+1 = Zn^2 + c
        // =(x + yi)^2 + (a +bi)
        //seperate to give:
        // x = x^2 - y^2 + b
        // y = 2xy + a
        double xnew = xx * xx - yy * yy + cr;
        yy = 2 * xx * yy + cim;
        xx = xnew;
        n++;
      }
      if (n < max) {
        //[number is in mandelbrot set]
        //set color scaled exponentially with how many iterations were performed
        i.setRGB((int) (x), (int) (y), +((0xFF / 20) * (int) n ^ (2)) * 0x10000);
      } else {
        //[number is not in set]
        //set color to (almost) black
        i.setRGB((int) (x), (int) (y), (n) * 0x01);
      }

    }
```

Figure 2 - Code used to generate an image of the Mandelbrot set

Where *'i'* is a '*BufferedImage*' class object that is used as a pixel array that can be drawn onto a screen. As shown in the above code, the formula from equation 2 must be split into its imaginary and real parts so that it can be evaluated for different values of x and y. The coordinate values are also subtracted by half of the size of the screen in each direction – this is to center the image around the center of the screen and so that when the image is magnified it scales around the center point. The coordinates must also be divided by the magnification in order to obtain the actual coordinates in space from the pixel coordinates of the screen, which allows the image to be scaled

Conclusion

In conclusion, it was found that by multiplying the $\frac{number\ of\ iterations}{\max iterations}$ by a constant RGB value (in this case 0x10000 to give a red shading) the Mandelbrot pattern can be seen.

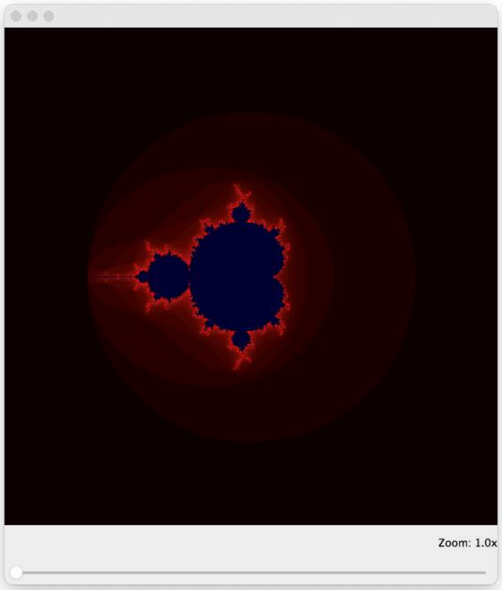


Figure 3 - Generated Mandelbrot set

This way of programming unfortunately does have a limit to its precision at higher magnifications to the limitation of the 64-bit double class, which begins to lose precision as the number increases significantly. The result is that at maximum magnification the pattern begins to lose definition and no longer produces recursive patterns

One improvement that could be made other than the precision is the performance – as the program must iterate a number of times for each pixel, it can be resource consuming especially at higher magnifications. The numbers that lie within the set (black in the images) take the longest as they must iterate the maximum number of times. To reduce some of this extra work, known values (i.e. the pixels that lie within the center of the Mandelbrot set) could be skipped and automatically marked as 'in the set'.

Applications of Fractals

The applications of fractals are diverse. Some of the applications discussed in this investigation are -:

1. Measuring coastlines
2. Financial market hypotheses
3. Fractal Antennas
4. Fractal in art

Measuring coastlines

As stated above, one example of a fractal is a coastline. The measured length grows as you zoom in, revealing additional details. We refer to this phenomenon as the coastline paradox. The box-counting dimension D method is determined by covering the coastline with boxes of a certain size s and counting the number of boxes N needed to cover the coastline at that scale. The relationship between s and N is:

$$\log(N) = -Dlog(s) + constant$$

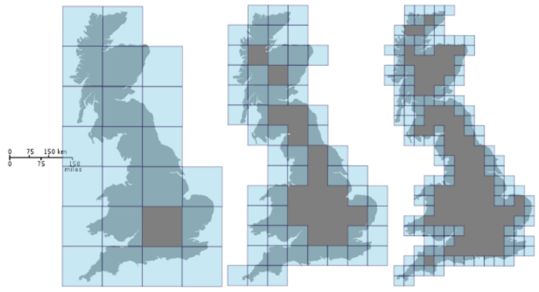


Figure 4 - estimating the coastline of britain

As we take finer and finer grids, each scaled down by a scale factor S, the number of unit cells touched (N) increases by a factor of S^D. Hence, the Fractal Dimension can be found by simply taking the slope of a graph of these two variables:

$$D = \frac{\Delta \log N}{\Delta \log S}$$

Fractal Market Hypothesis

FMH asserts that time series data of stock market prices exhibits properties similar to fractals. It attributes these properties to varying time horizons (length of investment) and information among investors. FMH is a theory about how heightened market uncertainty can lead to sudden market crises and crashes. This theory is based on the idea that there are two different types of investors, short-term and long-term investors. Short-term investors base their investment choices from technical information and analysis while long-term investors base their investment choices from more fundamental information. This creates similarly shaped investment returns and therefore similarly shaped market directions. Hence the idea of a fractal market. But when long-term investors become short-term investors this fractal shape collapses and so too does the market. This can be seen in the image below of a heatmap graph of investment period over year:

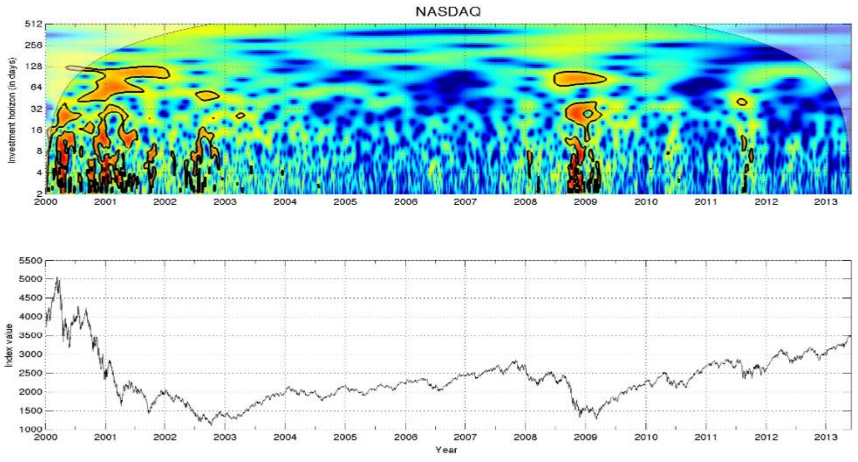


Figure 5 - heatmap graph of investment period

Fractal Antennas

Fractal antennas take advantage of fractals shapes and dimensions to enhance performance of previous technology on communication systems. The repeating nature of fractal shapes at different sizes allow resonance at multiple frequencies which allow coverage of broader frequency. Fractal antennas take advantage of length which broadens frequencies as the longer length allows more resonance at different frequencies so the antenna can retrieve data in a broader spectrum. Fractals complex natures allow the miniaturisation of the antennas as the self-similar structure allows for more compact antenna's while maintaining or even improving signal strength. Uses of fractal antennas stretch from commercial uses for communication via mobile phones and WIFI but also within military and defence departments as the improved covering and reduced interference play crucial roles within military communication. Enhanced performance due to fractals allow practical uses for medical instruments such as biosensors and imaging applications. There are different types of fractal antennas such as the Koch antenna that is made from the Koch snowflake which as the name intends, looks like a snowflake. A Koch Snowflake (Koch Curve) is generated using an iterative function system. The length of a Koch curve can be given by:

$$L_n = 3L_o\left(\frac{4}{3}\right)^n$$

Where Ln is the length at the nth iteration. Using this equation, as n tends to infinity, Ln also tends to infinity. This shows us that the perimeter/length of a Koch snowflake is infinite however the area is finite. Koch antennas are mainly used for Ultra-High-Frequencies due to their self-similar nature and it is advantageous compared to non-fractal antennas in limited spaces as it can be used at compact sizes with great efficiency.

Fractals in Art

Fractals having infinitely repeating patterns make them a prime candidate to be used in art. Fractal-like patterns often occur within nature itself. Even though these patterns aren't real fractals they can be modelled by fractals to create some beautiful visuals.

A simple example of this is the Pythagorean tree. When you think about it a tree starts from a trunk which then splits into large branches which each split into smaller branches and eventually to the smallest visible part, the leaves. This is perfect for fractal modelling. Another famous example is the Koch snowflake. This works as snowflakes are always symmetrical (due to the ice crystals in a snowflake reflecting the water molecule arrangement in ice) and each 'arm' of a snowflake also branches, similar to a tree.

So far, these have been relatively simple examples which can be recreated using a pen and paper. However, with the advancement of computational power, people have created much more complex and abstract art using the idea of fractals and having infinitely recursive patterns. These art pieces focus less on their roots in reality, and more on the abstract beauty created by fractals

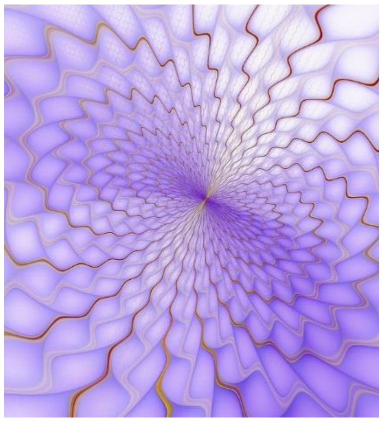


Figure 6 - piece by Matthias Hauser called 'Purple and golden fractal explosion'

References

Fractals - Fractal Dimension
https://www.cs.cornell.edu/courses/cs212/1998sp/handouts/Fractals/similar.html
Accessed 28/11/2023

Fractals are typically not self-similar – YouTube
https://www.youtube.com/watch?v=gB9n2gHsHN4
Accessed 29/11/2023

The Fractal Market Hypothesis: An Overview by Edgar E Peters – Fractal Market Cycles and Regimes
https://www.edgarepeters.com/blog-2-2/blog-post-title-one-sf2m5
Accessed 27/11/2023

The Fractal Market Hypothesis and its implications for the stability of financial markets – CEPR
https://cepr.org/voxeu/columns/fractal-market-hypothesis-and-its-implications-stability-financial-markets
Accessed 27/11/2023

Pythagoras tree (fractal) – GeoGebra
https://www.geogebra.org/m/fwebuvfr
Accessed 29/11/2023

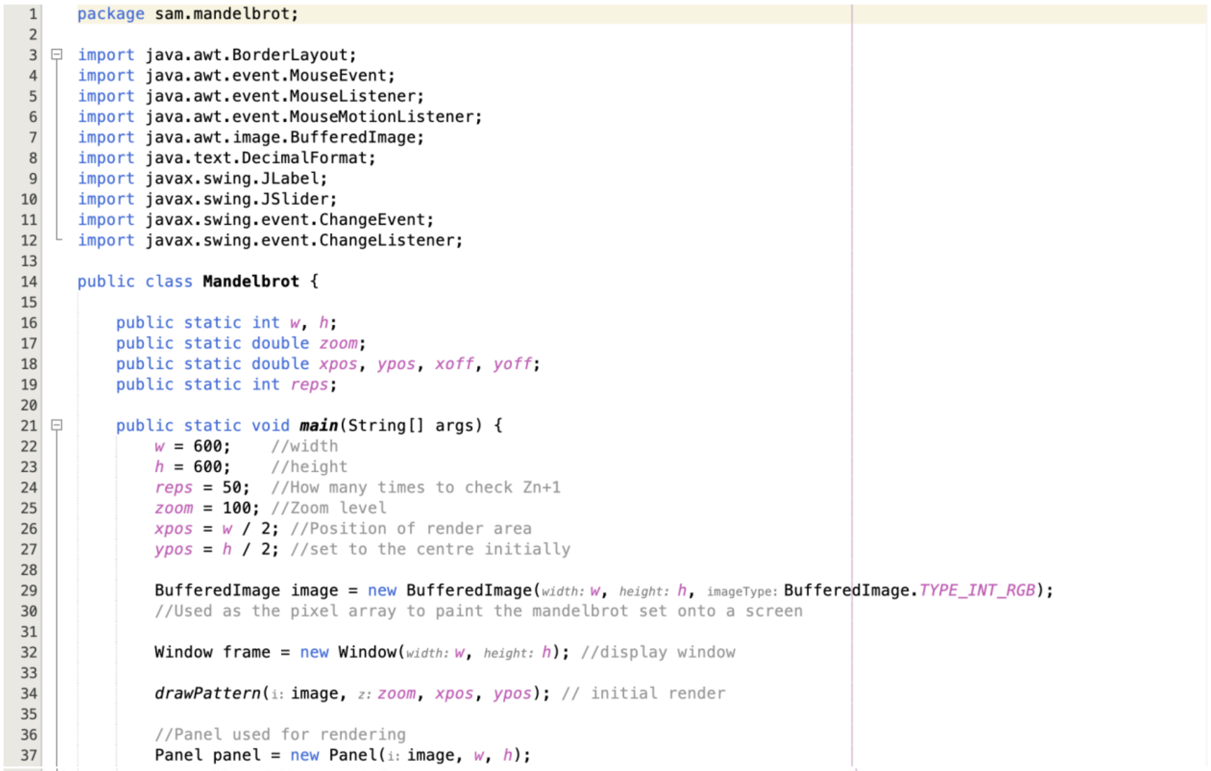
The Boar
https://theboar.org/2022/12/the-science-of-snowflakes/
Accessed 29/11/2023

Koch Snowflake – GeoGebra
https://www.geogebra.org/m/NP6kmyjF
Accessed 29/11/2023

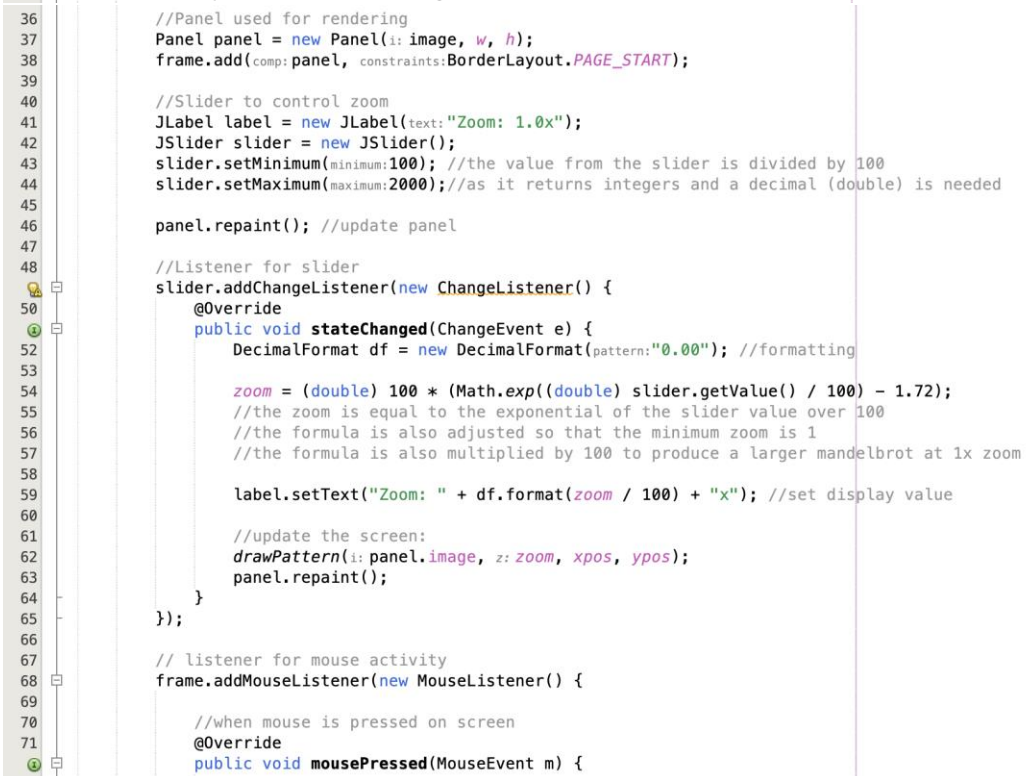
Purple and golden Fractal Explosion by Matthias Hauser – Matthias Hauser-Website
https://matthias-hauser.pixels.com/featured/purple-and-golden-fractal-explosion-matthias-hauser.html
Accessed 29/11/2023

Source code -:

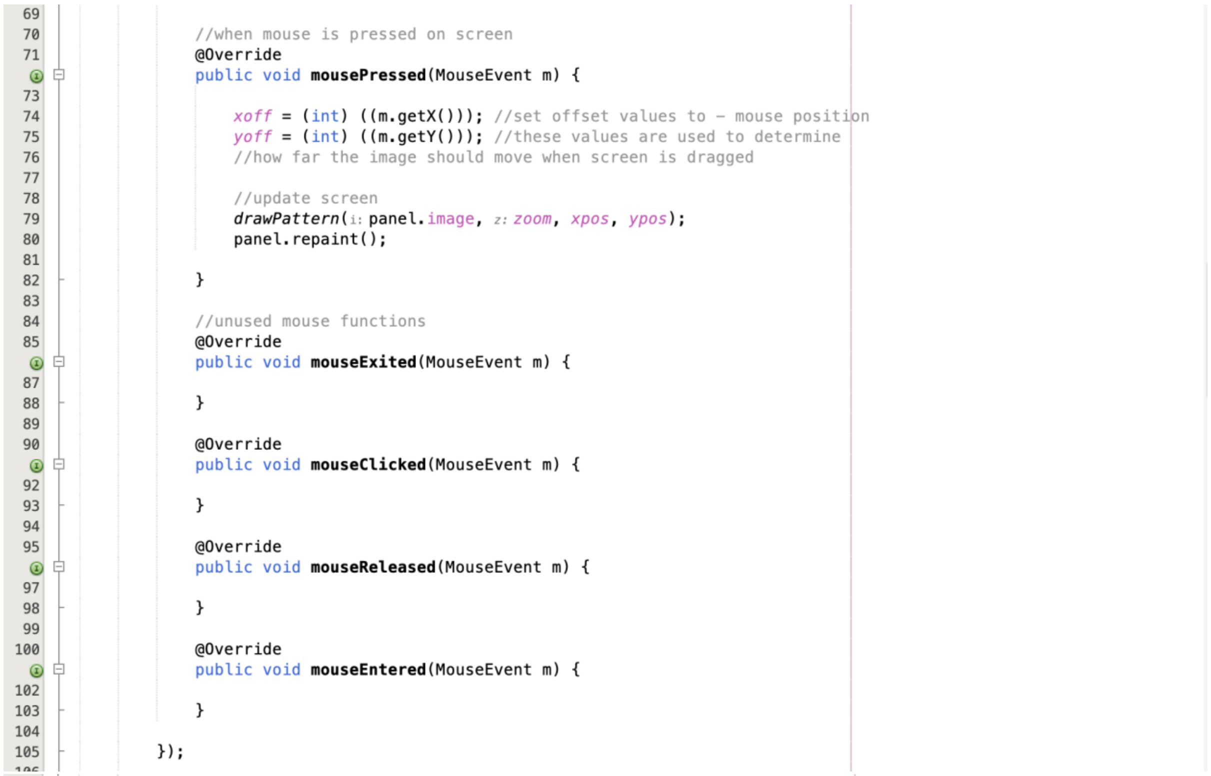
1. Main class



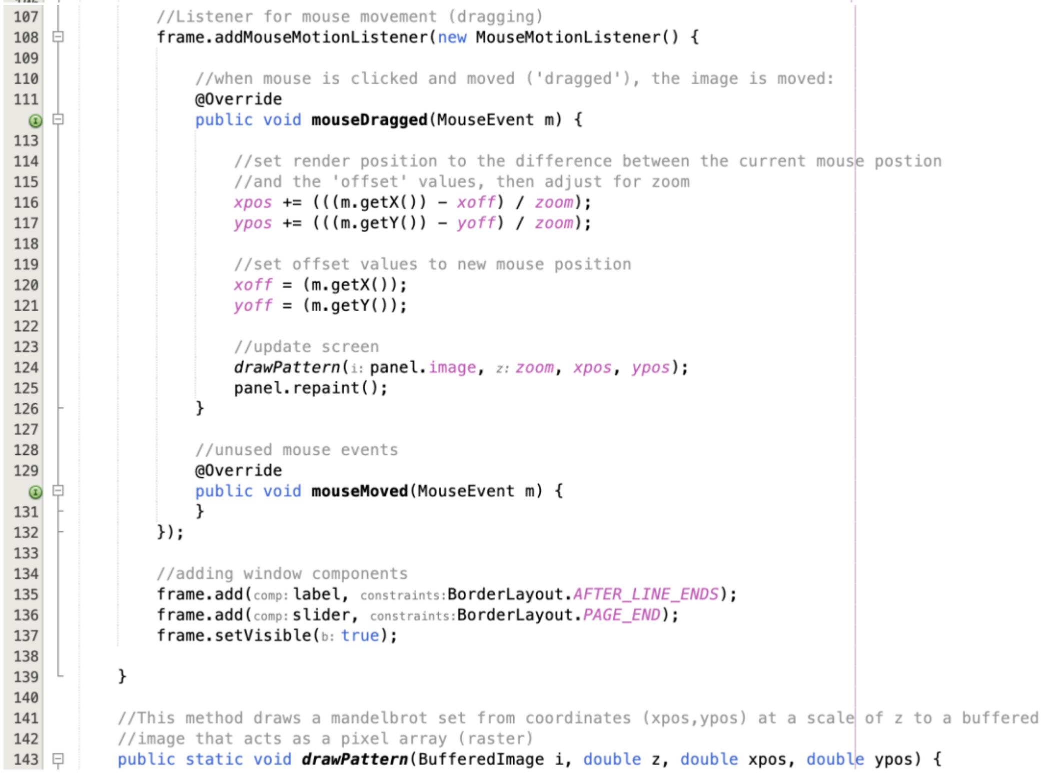
```java
package sam.mandelbrot;

import java.awt.BorderLayout;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.image.BufferedImage;
import java.text.DecimalFormat;
import javax.swing.JLabel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class Mandelbrot {

    public static int w, h;
    public static double zoom;
    public static double xpos, ypos, xoff, yoff;
    public static int reps;

    public static void main(String[] args) {
        w = 600;    //width
        h = 600;    //height
        reps = 50;  //How many times to check Zn+1
        zoom = 100; //Zoom level
        xpos = w / 2; //Position of render area
        ypos = h / 2; //set to the centre initially

        BufferedImage image = new BufferedImage(width: w, height: h, imageType: BufferedImage.TYPE_INT_RGB);
        //Used as the pixel array to paint the mandelbrot set onto a screen

        Window frame = new Window(width: w, height: h); //display window

        drawPattern(i: image, z: zoom, xpos, ypos); // initial render

        //Panel used for rendering
        Panel panel = new Panel(i: image, w, h);
```



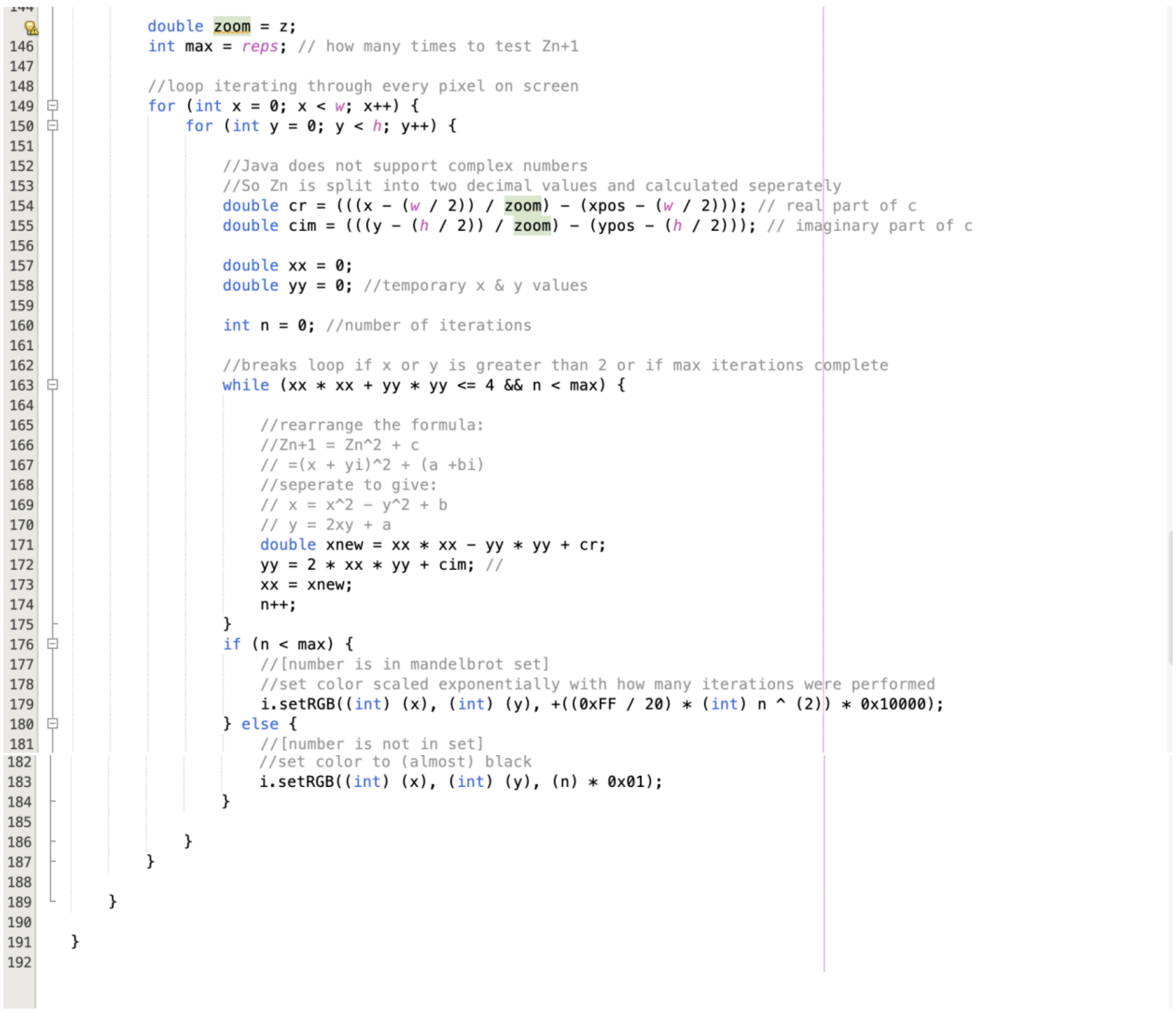
```java
        //Panel used for rendering
        Panel panel = new Panel(i: image, w, h);
        frame.add(comp: panel, constraints:BorderLayout.PAGE_START);

        //Slider to control zoom
        JLabel label = new JLabel(text:"Zoom: 1.0x");
        JSlider slider = new JSlider();
        slider.setMinimum(minimum:100); //the value from the slider is divided by 100
        slider.setMaximum(maximum:2000);//as it returns integers and a decimal (double) is needed

        panel.repaint(); //update panel

        //Listener for slider
        slider.addChangeListener(new ChangeListener() {
            @Override
            public void stateChanged(ChangeEvent e) {
                DecimalFormat df = new DecimalFormat(pattern:"0.00"); //formatting

                zoom = (double) 100 * (Math.exp((double) slider.getValue() / 100) - 1.72);
                //the zoom is equal to the exponential of the slider value over 100
                //the formula is also adjusted so that the minimum zoom is 1
                //the formula is also multiplied by 100 to produce a larger mandelbrot at 1x zoom

                label.setText("Zoom: " + df.format(zoom / 100) + "x"); //set display value

                //update the screen:
                drawPattern(i: panel.image, z: zoom, xpos, ypos);
                panel.repaint();
            }
        });

        // listener for mouse activity
        frame.addMouseListener(new MouseListener() {

            //when mouse is pressed on screen
            @Override
            public void mousePressed(MouseEvent m) {
```



```
            //when mouse is pressed on screen
            @Override
            public void mousePressed(MouseEvent m) {

                xoff = (int) ((m.getX())); //set offset values to - mouse position
                yoff = (int) ((m.getY())); //these values are used to determine
                //how far the image should move when screen is dragged

                //update screen
                drawPattern(i: panel.image, z: zoom, xpos, ypos);
                panel.repaint();

            }

            //unused mouse functions
            @Override
            public void mouseExited(MouseEvent m) {

            }

            @Override
            public void mouseClicked(MouseEvent m) {

            }

            @Override
            public void mouseReleased(MouseEvent m) {

            }

            @Override
            public void mouseEntered(MouseEvent m) {

            }

        });
```

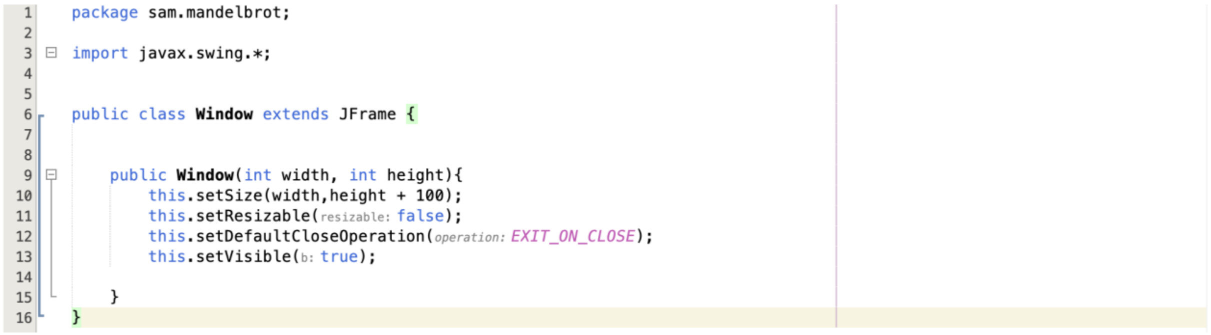


```
        //Listener for mouse movement (dragging)
        frame.addMouseMotionListener(new MouseMotionListener() {

            //when mouse is clicked and moved ('dragged'), the image is moved:
            @Override
            public void mouseDragged(MouseEvent m) {

                //set render position to the difference between the current mouse postion
                //and the 'offset' values, then adjust for zoom
                xpos += (((m.getX()) - xoff) / zoom);
                ypos += (((m.getY()) - yoff) / zoom);

                //set offset values to new mouse position
                xoff = (m.getX());
                yoff = (m.getY());

                //update screen
                drawPattern(i: panel.image, z: zoom, xpos, ypos);
                panel.repaint();
            }

            //unused mouse events
            @Override
            public void mouseMoved(MouseEvent m) {
            }
        });

        //adding window components
        frame.add(comp: label, constraints:BorderLayout.AFTER_LINE_ENDS);
        frame.add(comp: slider, constraints:BorderLayout.PAGE_END);
        frame.setVisible(b: true);

    }

    //This method draws a mandelbrot set from coordinates (xpos,ypos) at a scale of z to a buffered
    //image that acts as a pixel array (raster)
    public static void drawPattern(BufferedImage i, double z, double xpos, double ypos) {
```



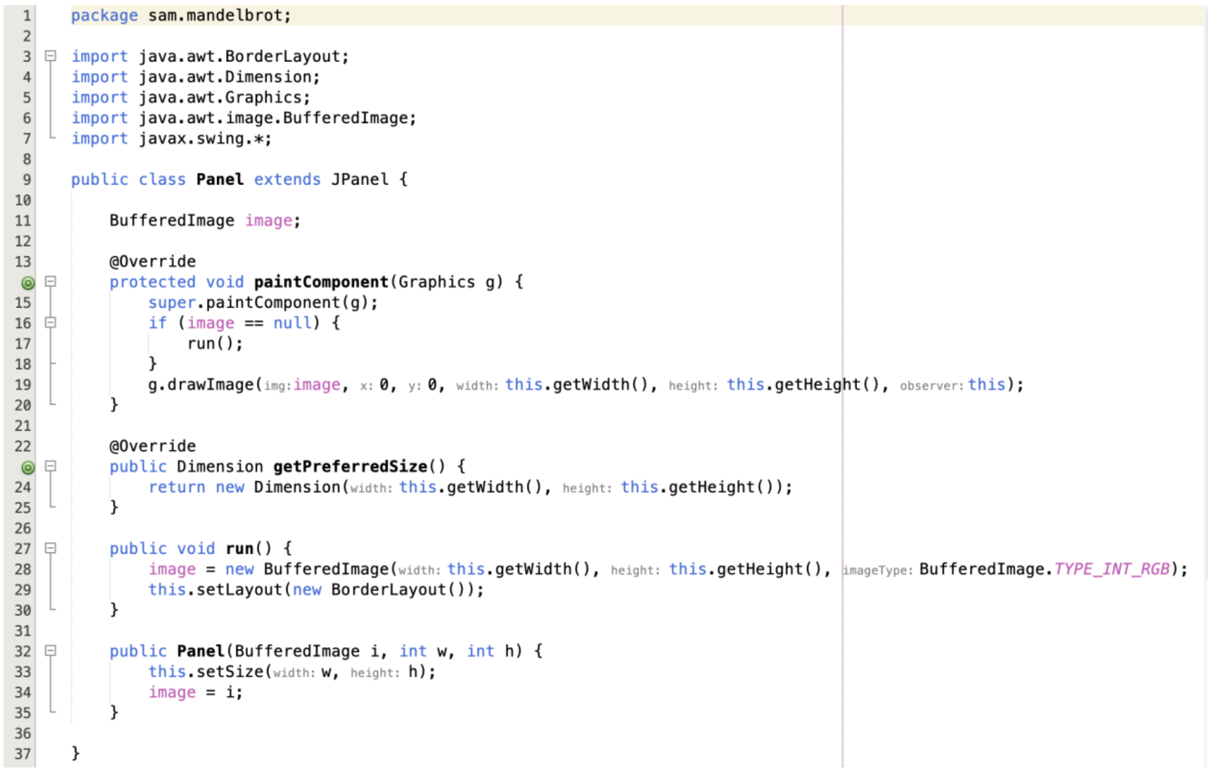
```
        double zoom = z;
        int max = reps; // how many times to test Zn+1

        //loop iterating through every pixel on screen
        for (int x = 0; x < w; x++) {
            for (int y = 0; y < h; y++) {

                //Java does not support complex numbers
                //So Zn is split into two decimal values and calculated seperately
                double cr = (((x - (w / 2)) / zoom) - (xpos - (w / 2))); // real part of c
                double cim = (((y - (h / 2)) / zoom) - (ypos - (h / 2))); // imaginary part of c

                double xx = 0;
                double yy = 0; //temporary x & y values

                int n = 0; //number of iterations

                //breaks loop if x or y is greater than 2 or if max iterations complete
                while (xx * xx + yy * yy <= 4 && n < max) {

                    //rearrange the formula:
                    //Zn+1 = Zn^2 + c
                    // =(x + yi)^2 + (a +bi)
                    //seperate to give:
                    // x = x^2 - y^2 + b
                    // y = 2xy + a
                    double xnew = xx * xx - yy * yy + cr;
                    yy = 2 * xx * yy + cim; //
                    xx = xnew;
                    n++;
                }
                if (n < max) {
                    //[number is in mandelbrot set]
                    //set color scaled exponentially with how many iterations were performed
                    i.setRGB((int) (x), (int) (y), +((0xFF / 20) * (int) n ^ (2)) * 0x10000);
                } else {
                    //[number is not in set]
                    //set color to (almost) black
                    i.setRGB((int) (x), (int) (y), (n) * 0x01);
                }

            }
        }

    }

}
```

2. Window



```
package sam.mandelbrot;

import javax.swing.*;


public class Window extends JFrame {


    public Window(int width, int height){
        this.setSize(width,height + 100);
        this.setResizable(resizable: false);
        this.setDefaultCloseOperation(operation: EXIT_ON_CLOSE);
        this.setVisible(b: true);

    }
}
```

3. Pane



```
package sam.mandelbrot;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import javax.swing.*;

public class Panel extends JPanel {

    BufferedImage image;

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (image == null) {
            run();
        }
        g.drawImage(img:image, x: 0, y: 0, width: this.getWidth(), height: this.getHeight(), observer: this);
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(width: this.getWidth(), height: this.getHeight());
    }

    public void run() {
        image = new BufferedImage(width: this.getWidth(), height: this.getHeight(), imageType: BufferedImage.TYPE_INT_RGB);
        this.setLayout(new BorderLayout());
    }

    public Panel(BufferedImage i, int w, int h) {
        this.setSize(width: w, height: h);
        image = i;
    }

}
```

Student contribution

Gurraunak Singh Bedi – My main contribution was skimming down the raw data provided and compiling it to produce a meaningful report.

Neeraj Nandan Akella – My main contribution was data and mathematical calculations for the dimensions of fractals.

Baran Altun - I was tasked with scientific applications of fractals and focused on the uses of fractals with antennas and described the nature of fractals and how they are used in antennas.

Callum Connell – I was given the task of finding out the financial market hypotheses, the trend it follows and its relation to fractals.

Sam Clarke – My contribution was writing down and testing the code for generating fractal

Shayan – I was tasked with finding the application of fractals in art and how different artworks depended on fractals

Hajar Bouchouya – My part in this report was finding data for the application of fractals in the measurement of coastlines and non-self-similar structures