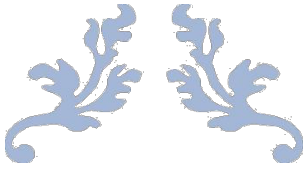
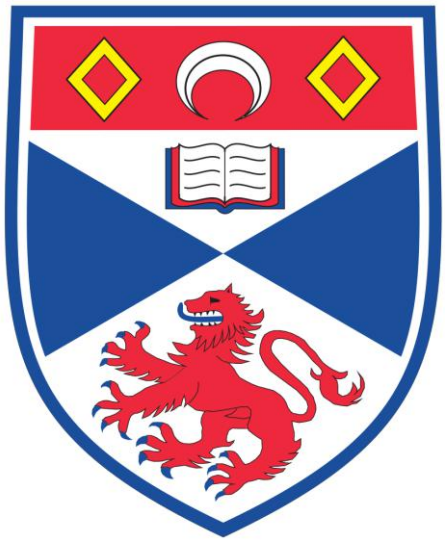


Assignment 4:

GUI – Mandelbrot Set Fractal Explorer

CS5001 – Object Oriented Modelling Design & Programming



170009629

NOVEMBER 24th, 2017

UNIVERSITY OF ST ANDREWS

i. Table of Contents

1 Introduction

A major area of Computer Science that is growing rapidly is Data Visualisation. This involves the creative ability of creating new and unique ways of representing what can often be very complex datasets. This is frequently done with interesting colour mappings to highlight interesting information.

Closely related to the Data Visualisation is the use of Graphical User Interfaces (GUIs), however these are not restricted to representing complicated data. The importance of GUIs is realised by noting the sheer quantity of devices around us that use these today. They are not only constrained to the world of advanced computing, and can even be found in these current days in cars, children's toys, and even fridge-freezers. They are made to create a clear and enjoyable environment for a user to interact with a given program, and are therefore very important to those interested in Human Computer Interaction (HCI).

This report will cover the design approach taken for the development of a GUI that is used for navigating and exploring the space of the Mandelbrot Set, which can be used to generate some very interesting fractal shapes. The main objective of this practical was not simply to achieve a working piece of software with fluid exploration, but also to focus on the setup of the GUI software, ensuring that the code way divided properly, and appropriately accessible. The aim was to allow the model to be detachable from the view/controller, and therefore not be dependant or on anything else. This way the model could be detached and have a completely different UI stitched on to it, but still be able to provide the same functionality as before, resulting in a very versatile model of code.

2 List of Parts

This section will simply list all the parts of the coursework that were attempted in this submission.

2.1 Part 1 – Basic Requirements

- The program is able to display the Mandelbrot set in black and white.
- The zoom feature has been implemented by means of mouse dragging and zooming to the user's selected area (by recalculating the Mandelbrot set with new bounds).
- The user is able to choose the number of Max-Iterations to improve precision.
- The user is able to reinstate prior points in the program's time by:
    - Resetting the display to its original state (default values).
    - Undoing and action (or several) to return to a previous state.
    - Redoing an action to move back forwards after previously undoing.
- Note undoing, redoing, and resetting, will not only return the view (zoom position), But also the coloured mapping that was used at the time.

2.2 Further Extensions

- Different coloured mappings were implemented.
- The user is able to switch between colour maps of their choice by clicking buttons.
- The user can save their position to a file, and later (even after exiting and re-entering the program) can reload their save point, including their position, zoom factor, choice of colour.
- The user is also able to save a picture instance of their current Mandelbrot set view to an image file (both PNG and JPEG are supported).
- When selecting the zoom area, a clear rectangle is shown of the area that will be zoomed into upon mouse release.
- A toggle feature was included to allow switching between zoom types. These types, mean either keeping the aspect ratio, or allowing the user complete control.
- For the non-restricted zoom (disregarding aspect ratio), the click and drag can be done in any direction (not only top-left to bottom-right).

3 Design and Justification

3.1 Project Breakdown and Class Relationships

3.1.1 package main

- **class Main** – The main class is not used greatly in this program. It is only really used as an entry point, where the initial setup can be invoked through the instantiation of the model and delegate. It also calls the initial calculation of the Mandelbrot set so that there is a starting image on the screen.

3.1.2 package guiDelegate

- **class Delegate** – In this project the Delegate class handles the bulk of the delegate work such as many controller tasks like calling the model's recalculate/update the Mandelbrot set. It is used to set up the base JFrame object, initialise the basic GUI, and is able to update the repainting of the GUI through one of its methods. The last key aspect this class is that is listens to the JFrame waiting for any mouse presses, releases, and even drags (for zoom purposes).

- **class Display –** The Display class acts partly as a midway point between the Delegate (focused on controlling) and the Canvas(used as a view/interaction), and can be used to communicate either way. For example, it prepare the buffered image that the canvas displays, and invokes the delegates repaint method when necessary.
  The most important thing about this class is that it receives the model in its construction function. With this, it can add itself to the model's observer list, and wait to be updated the moment something happens. This meant the update method had to be overridden. This will usually just be used for retrieve the newly mapped colour image of the Mandelbrot set, and then request a repaint on the canvas.

- **class Canvas –** This class is used to add the coloured image so that the user can see, and also draw any other components (such as the zooming rectangle/square). The other important part of this class is all the buttons that it has attached. It has listeners for each of these buttons, and each act according to the individual button's requirement. Many of the listeners will invoke methods in the MethodsForListeners class in the model package. This is simply to get the model to update anytime the view has been changed or interacted with. The class extends JPanel in order to perform as required.

3.1.3 package model

- **class Model –** In this package, all the inner workings of the project happen. That means all of the hard calculations, the raw data that represents the model, and the methods to update the model as a response to the user (or not) before updating the view to render the changes. This class in particular handles a lot of that, but is mainly used to hold all the fields that characterise the model. It has a great number of getters and setters to keep the fields private, and uses its constructor to set the default values. It also has one other important job (the most important of them all) that can be broken into a few

methods. The first is to update the calculations for the Mandelbrot set, however to do this, the correct mapping from co-ordinates to the bounded real/imaginary numbers had to be performed before calling the method (from the following class) to actually work out the results. This calculation is essentially how it is able to zoom. After the array containing the results has been returned, it then uses a method to loop through every pixel, one at a time. For each pixel, a personally designed mapping method is called to colour the pixel appropriately according to its iteration result. Once this has been completed, the coloured image is then ready to be retrieved using its getter method. To end the operations of this class, it is critical that the "setChanged()" and "notifyObservers()" methods are called, as the Model is extending observable, and the observers will not know when to update without these calls.

- **class MandelbrotCalculator –** This method was provided along with the practical. It was written by Jon Lewis (as indicated in the Javadoc), and is used to helpfully calculate the Mandelbrot set for the frame resolution, and provided it in a 2D array. This was extremely useful, as it allowed less time to be spent getting stuck with the maths leaving more time for the GUI side of the software (however, it is still important to understand the underlying mathematics before using classes/libraries that utilise it heavily).

- **class MethodsForListeners –** This class was designed specifically to receive method calls from the view/controller. Although it says "for listeners" in the class name, it is designed in a way that it could receive these calls from any kind of UI (giving the model more flexibility and detachability). These methods are used predominantly to update the state of the model, and include functionalities such as undo, redo, reset, save, load, and toggle between the fixed aspect ratio and the free-zoom.

3.2 Undo and Redo

To implement the undo and redo requirements, it was decided to utilise Stacks, as their "first-in, last-out" nature presented them to be the most appropriate data structure that. A very small stack was kept for both the 'past' and 'future' of each parameter. This meant they could be very simply all be pushed/popped together, causing no confusions. They could then be used for essentially an unlimited record of states (or more the parameters for how to return to the states), for both undo's and redo's. This method was chosen for its simplicity over storing them in one place and needing to know a sequence. It was also heavily considered to just store the multiple states in objects which would arguably have been neater, however it was determined to be more trouble that its worth due to creating a new class with fields that are essentially repeating the fields already stored in variables. However, if there were more than just these few methods involved, then the object route would be taken instead.

One neat part to these features was the very simple ability to keep all the colour decisions saved with the undo/redo states. This way, when undoing/redoing, it would not simply change position, but remember the colours aswell.

3.3 Reset, Save and Load

This part was adapted from the previous section. As all the parameters were always being stored, there was very little additional code needed to record the 'save' parameters to a text file (including the colour mapping), and reload by reading it in line by line. This was another reason what this storage structure was smoother to implement than using state objects for

everything, as there was no serialisation needed for the saving to files, as the variables were only doubles, and not complex objects.

It was decided to also give the user the ability to save not only the file to reload later, but also the screen image, if the particularly like the pattern that they had found. The image saving formats that were decided on for the software to support were PNG and JPEG, however, it was found that the PNGs provided much better quality. After saving, these can be located in the project folder (outside of src and bin).

The reset was implemented by resetting ALL values to the default ones. There was the decision to retain the stacks which would allow the ability to undo back past the reset (and this is how it was originally), but it was decided against in the end. This is mainly because the requirement was interpreted to be analogous to performing a factory reset on e.g. a phone. In this case it would be wiped clean, returning completely to the original defaults.

3.4 Colour Mapping

The colours were mapped using the most familiar way, and that was the RGB scale. With each element being between 0 and 255, it was not too hard to map a colour to the pixels when the range of iterations were known as well as the number of iterations for each pixel. There were then simply a few conditionals added to allow the user to switch between cases. As part of the mapping some offsets were used to achieve a more pleasant colour map, however due to the trial and error of the offsetting there are what appear to be magic numbers in the RGB parameters, which seemed unavoidable but understandable as it is related to the colour values.

4 Testing Approaches

The testing was not done in a formal approach (such as JUnit tests and such), however small self-checks were performed every few lines of new code. This was done to ensure it was all still working as intended.

These included many printlns, ensuring code blocks were being entered as desired, and that such parameters as coordinates were mapped correctly and zooming to the correct places.

5 Problems Encountered

There were many small problems along the way that caused a great deal of debugging. This was mainly due to being new to the model- delegate and model-view-controller approaches of organising code.

Other problems often came from the mapping. One particular problem that still persists is the rectangle that is drawn when dragging a zoom. When allowing the free-drag, the rectangle can be dragged in any direction and works perfectly. However, the fixed ratio only showed the square well when dragging downwards. Although the zoom still functions when dragging up, it was hard to show the square correctly, as it doesn't follow the mouse exactly (because of the fixed ratio). For this reason it was set to only show when dragging downwards (although the user can still drag up without the rectangle if the like and it should still work.

One other problem that was encountered was that in some cases the model refused to set changed and notify observers, even the code on either side of these lines worked fine, and these lines did work fine in other cases. After a long time of debugging, it was decided in the end to bounce the method call through a class that was known to work in activating the observable to update the observers. This luckily allowed the code to be neater, as this class and method from then on became the only place that would call the model's update Mandelbrot calculation method, which kept everything funnelled through this one access point.

6 Results

Figure 1 - Initial simple design (notice lack of buttons)

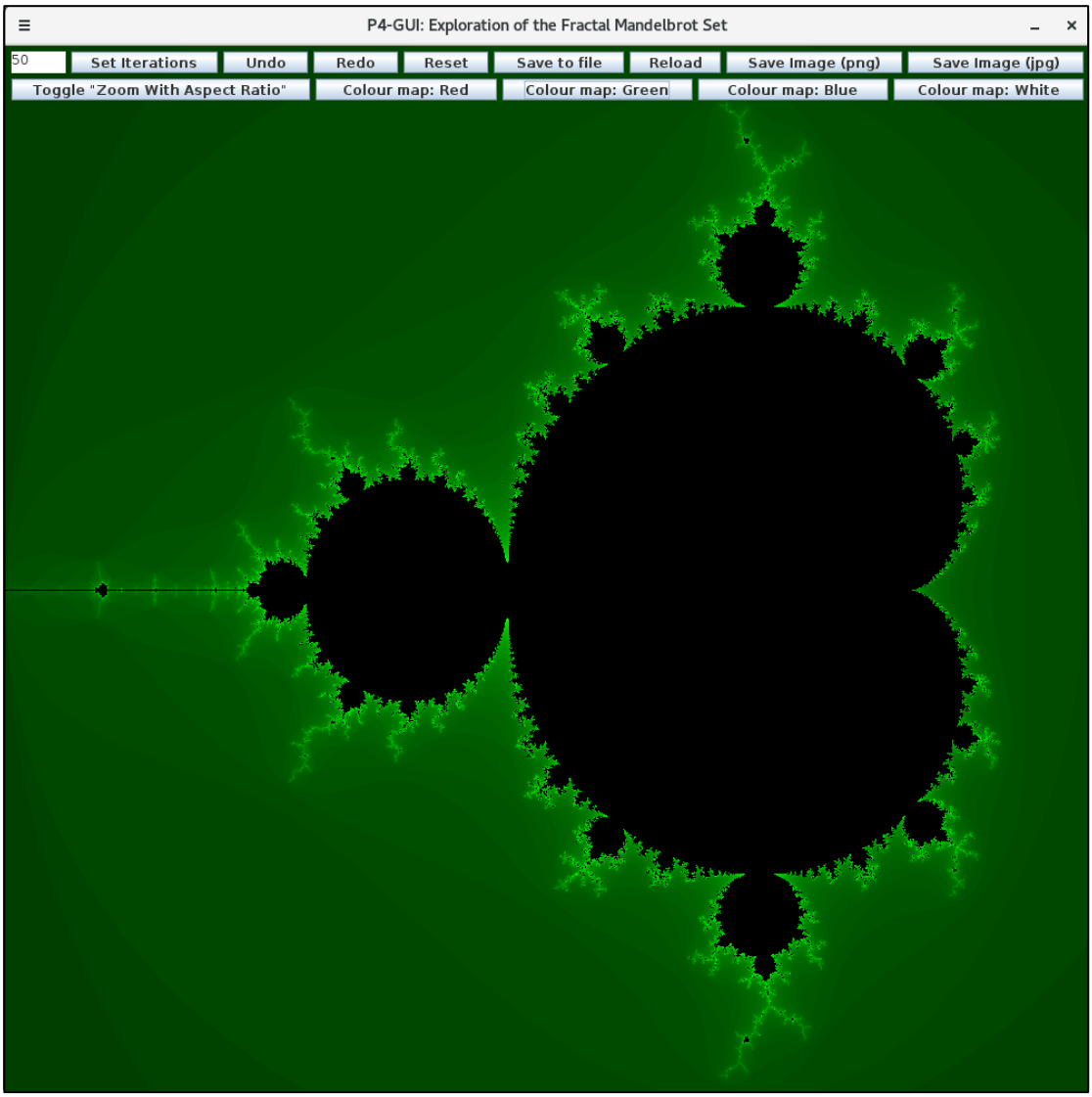


Figure 2 - All functionalities now included - new colour mapping

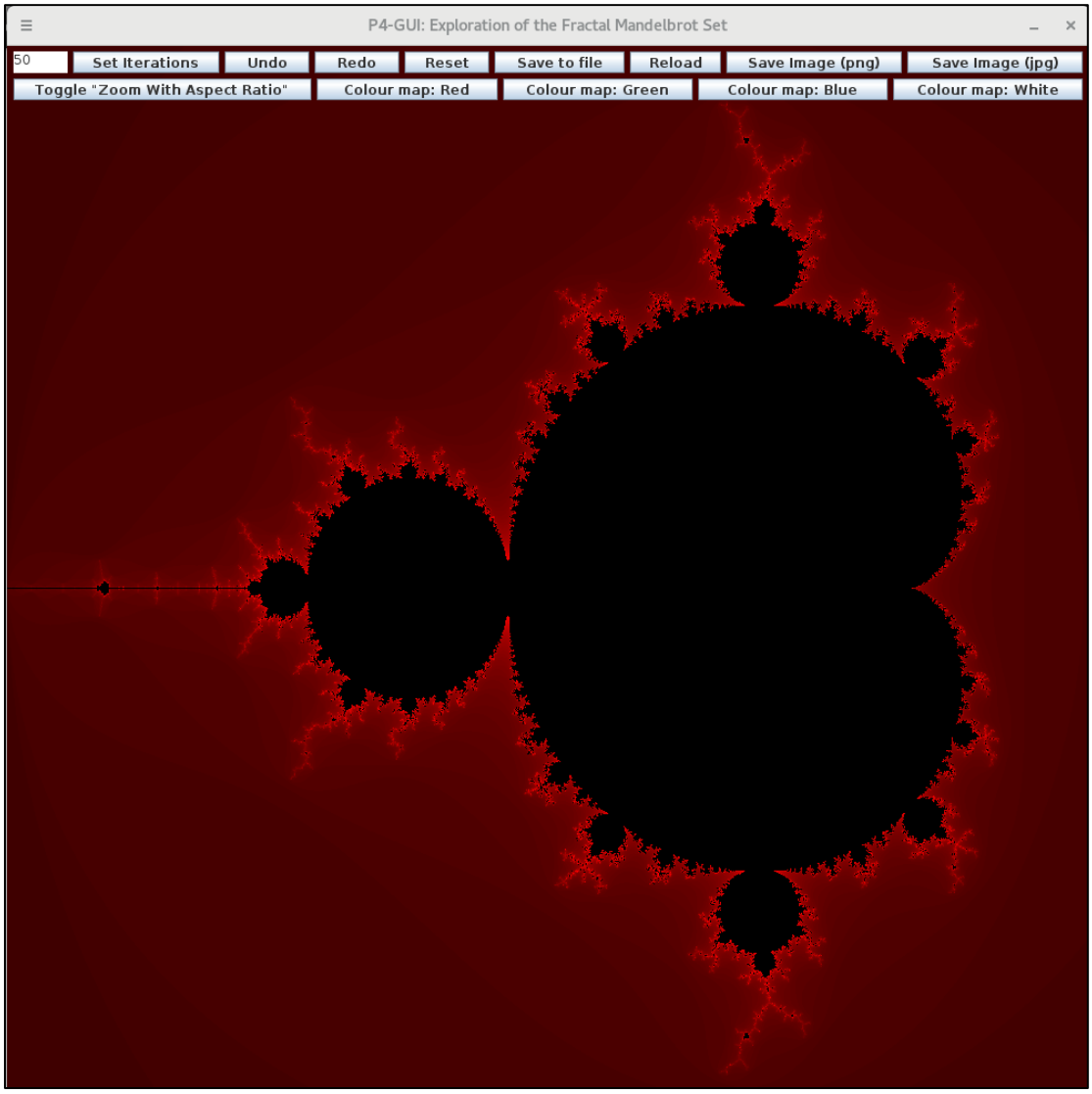


Figure 3 - Red mapping - all buttons

7

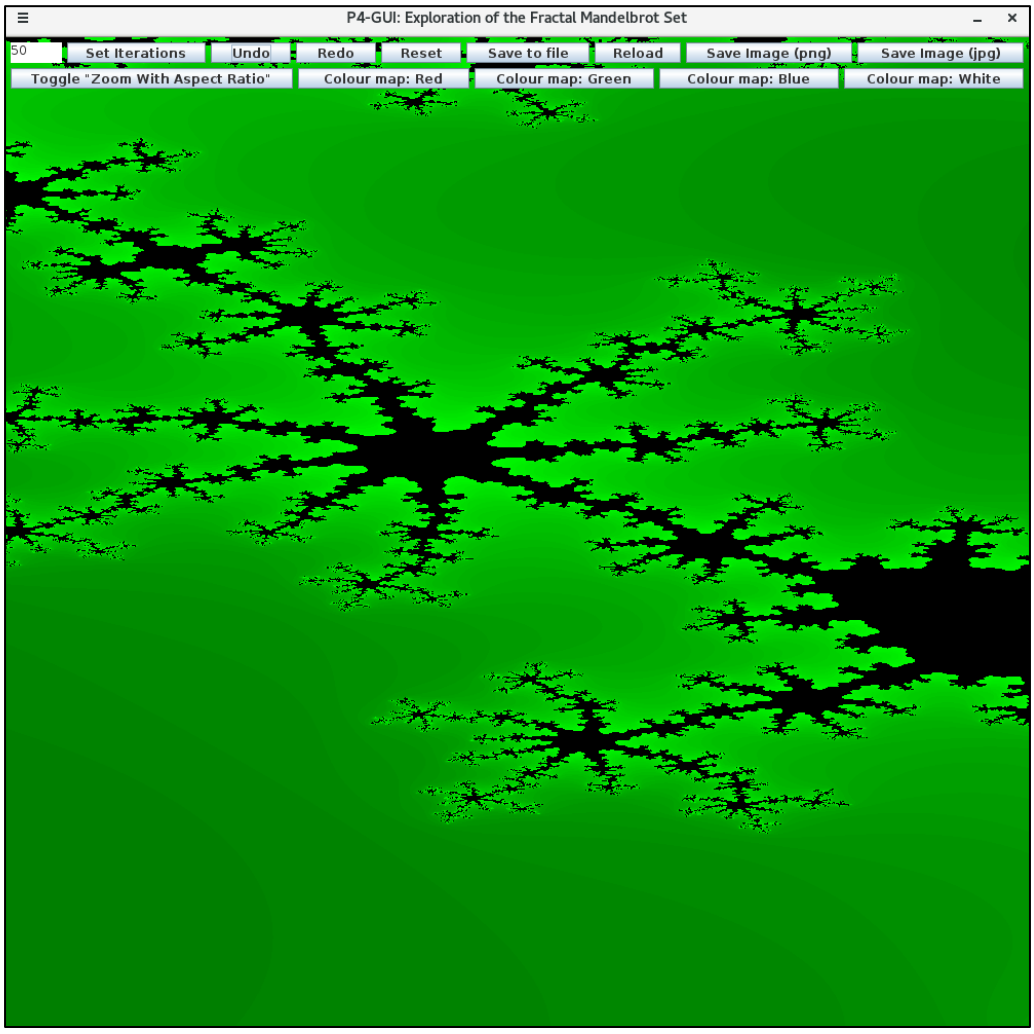


Figure 4 - When zoomed at a poor aspect ratio

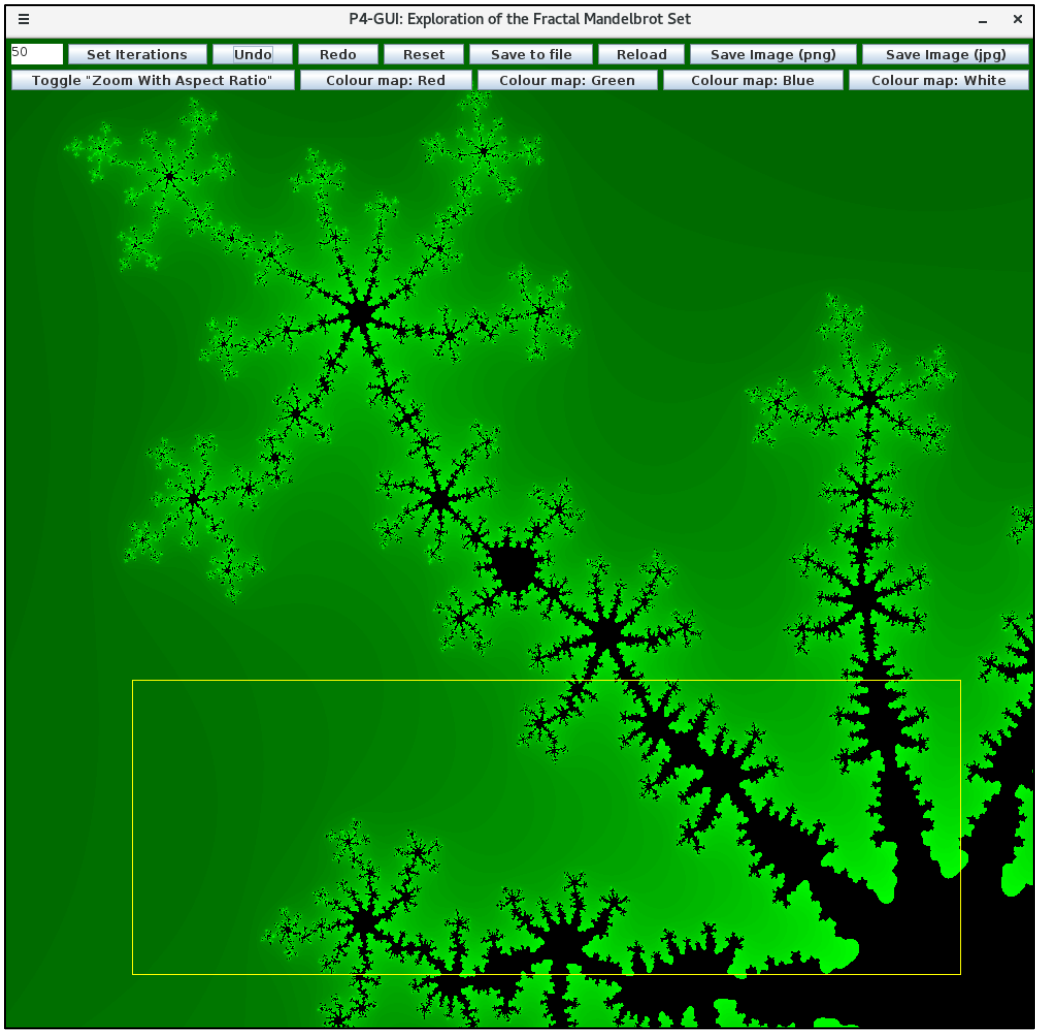


Figure 5 - Demo of the dragging then Aspect Ratio lock is NOT on

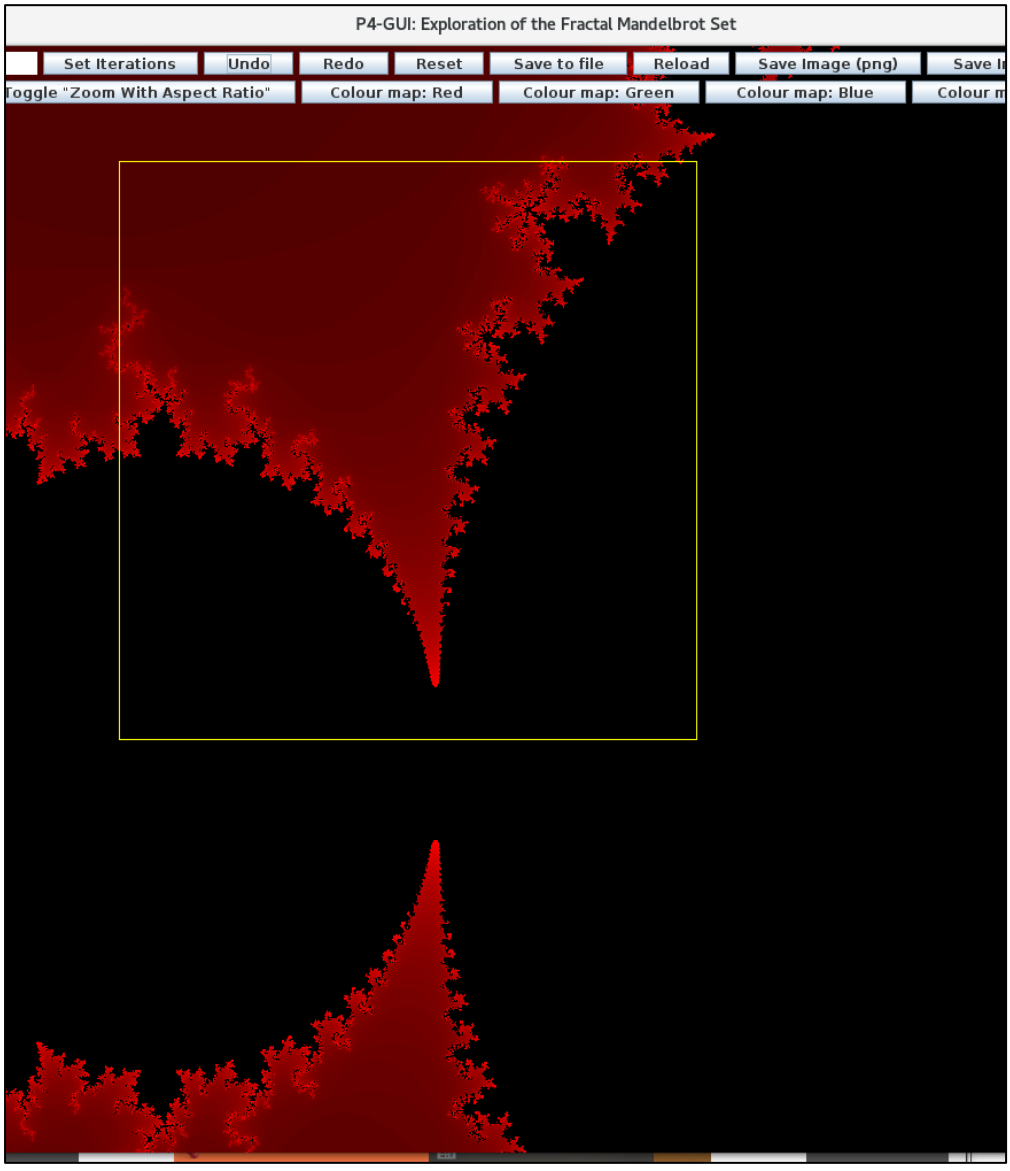


Figure 6 - Demo of the dragging then Aspect Ratio lock IS on

7 Evaluation

From the results, it can be seen that the software does act as requested. It follows the basic requirements, as well many of the enhancements. In addition, some further enhancements were also implemented, as discussed in the report. The program runs very well when operating the explorer, and seems very efficient, as it does not slow down very much until the max iterations is increased to high, or zoomed in on a black only area (as that performs more calculations that reach the max).

The one enhancement that was studied and thought about was the zoom animation, however there was not enough time to implement the ideas that were had.

On the other had, a particularly nice feature about this program is the ability to allow the user to alter between zoom types. As shown in figure 4, the view gets, squished when allowing the user to be free all the time, and some users may prefer to have the restriction that keeps the shape of the fractal image. To compare the two zoom types, see figures 5 and 6.

In conclusion, the practical seems successful in meeting the initial aims, and has proven to allow strong personal development in the implementation of GUI techniques.