CS5001 Practical 4

Mandelbrot Set Explorer

Design

This GUI's architecture was based on the Model-Delegate using the examples provided (SimpleFXGUIExample) as a reference point. The Mandelbrot model which was used to compute the Mandelbrot set values was also provided by the lecturers.

The program developed can display the Mandelbrot set and it allows the user to change the maximum number of iterations where Z remains bounded. Other common functions of an image visualizer were implemented such as Zoom and Pan. Zooming in on an area of the image was possible when the user clicked and dragged across the canvas. A square is drawn to show the user the area that would be zoomed in. The panning method implemented allowed the user to move through the image by a given amount and in any direction. The ability to show the scale of magnification was also provided for the user. This scale was based on the initial default parameters provided by the Mandelbrot model.

Redoing and undoing certain actions were implemented up to a maximum of 30 actions. However, any action performed after an undo request would disable and clear the information regarding redo actions. Only certain actions were recorded for the undo/redo functionality. These were Zooming, Panning, recalculating the Set and changing color.

The program can display the Mandelbrot in three different color schemes (white, red and blue). Black pixels represented where the Z remained bounded in the set. A brightness gradient was also applied to the model where the lowest brightness levels represented the lowest values of the set. The opposite applied to locations with the highest brightness.

Additionally, the program had file loading and saving properties. A set could be stored as a text file anywhere to be visualized and explored later using the same program. An image on the other hand could only be exported as a PNG but not reloaded into the program.

This program fulfills (or attempts) to fulfill all the basic requirements and all the enhancements apart from the Zoom Animation.

Implementation

The GUI was built on a Model-Delegate architecture. The model controlled the class MandelbrotCalculator which was responsible for computing the values of the set. This class was provided by the lecturers. The Mandelbrot set require a list of parameters which could be changed by the user. The delegate would request data from the model would be received when the *dataReady* event was triggered. This data would be used to display the Mandelbrot set using the function *drawCanvas* to iterate through the 2D array and set the color of every individual pixel in the canvas using *PixelWriter*. If the user decided to show the magnification factor by ticking the *magnification* check box, the magnification factor would be refreshed displayed. A snapshot of the canvas would then be saved as a Writable Image to be used during the Panning and Zooming operations. The maximum iteration was fetched in the of the method in order to ensure that the maximum iteration value displayed on the *InputField* on the view was most current value in the model.

The user had two options with regards to actions triggered by a mouse drag. The default Zoom and alternatively Panning. These two modes were mutually exclusive and thus, the user could only select a single option because they were displayed as radio buttons. Pressing zoom would deselect pan and vice versa. These two features required mouse listeners which were defined in *setupMouseListener* in the *MandelGuiDelegate* class. Mouse presses simply stored the starting X and Y coordinates (xPressed,yPressed) and the mouse drag listener was simply responsible for either drawing a square in the canvas if the *currentDragMode* was *ZOOM* or drawing a line if *PAN mode was selected.* In order to maintain the ratio of the image where it would be zoomed in, the user could only select squares since the ratio would then be preserved (canvas was 1:1 ratio). This was implemented by taking the highest distance (I.e. xReleased – xPressed) between the X and Y coordinates and setting it to be the new height and width of the rectangle drawn (effectively a square). The zoom function would only work if the user dragged the mouse from top left to bottom right. This was implemented to simplify the logic. It is important to note that because a snapshot of the canvas was saved as a *WritableImage,* the image could quickly be redrawn on the canvas as well as the line or square without lagging.

The mouse released listener would then be responsible for informing the model of the changes if a valid mouse drag was detected and it would rescale the real and imaginary ranges according to new X and Y values. The action of Zooming in used *calculateMapping* which required the new starting X and Y coordinates (where the mouse was first pressed) and the length of the square to determine the other two bounds. The model would then be updated, and the view refreshed once the data was ready. Panning on the other hand, only required the amount of change in both the X and Y direction in order to remap the current real and imaginary range. Once this changed was added, both the model and the view were refreshed.

The user was also able to change between a certain number of color schemes by pressing the change color button. This would not recalculate the Mandelbrot set and simply redraw the image using stored values from the previous calculation whilst applying a brightness scale for the colors red and blue. The brightness levels were determined by scaling the Mandelbrot value to a scale of 0 to 1 and using it as the brightness factor. The three primary colors (Red, Green and Blue) were then fetched from the color currently selected and the brightness factor applied to them. This was performed in the *drawCanvas* function whilst the program was iterating through the 2D array

The number of MaxIterations could be changed via a TextField in the tool bar. Only numbers could be inserted, and it was limited to the size of an Integer in the machine. If valid, the number of iterations would be set to the value inputted. However, the recalculation of the set would only occur when the user pressed the generate button.

The redo and undo functionalities of the program were added by storing the parameters required to display a particular set of data. The class *LogStruct* was created to store the *MandelbrotStruct* which contained the parameters required to compute the model. Additionally, the *colorSelected* was also stored to record any color changes. Whenever a Zoom, Pan, color change or model recalculation occurred these structures would be stored in an *ObjectStack* called *undoStack* which could be popped if the user decided to undo this action. This was performed in the addLog method in the MandelGuiDelegate. This worked in conjunction with the *redoStack* since the element popped in the *undoStack* would be pushed into the *redoStack* and vice versa. As the user pressed undo and redo, these two stacks would work together to maintain a history of the user actions. However, if the user decided to undo an action and then perform another action (zooming in), the redo information would be deleted, and the button disabled. Enabling and disabling these buttons would depend on the current stack size. However, whenever the reset button in the GUI's toolbar was pressed, both stacks would be cleared and both the buttons controlling these functionalities would be disabled.

Storing and loading data also used this structure. The information inside of it was stored in a text file which the user could select when they press the save option in the menu bar. A dialog would be displayed using the *FileChooser* class and if a valid path/file was selected, a method call to the static method *storeData* in the class *DataStorage* would manage the storing of these parameters in the correct format. This method would overwrite any file present (first delete then create new one) and using the *FileWriter* and the *BufferedWriter* classes, prepare and store the data in a text file. The opposite would occur for loading data. This would have a similar behavior to the save data functionality mentioned above, however the user had to click on the "Load" option in the menu bar and select an existing text file. Any unsupported files would throw an *Alert*. The parameters had to have a certain structure (attribute name and value) and using the static method *loadData* in the class *DataStorage*, the data could be loaded if correct. The file needed to have 7 lines. The data was read using the *readAllLines* in the *Files* class and stored into a list. The list was then iterated through to find the required parameters. Once this process was finished, the view and model would be reupdated accordingly. The third feature implemented related to data storage was the exporting of images. It made use of the same logic (menu bar listeners) for menu bars as mentioned before and it took a snapshot of the current canvas using the *snapshot* method of the *Canvas* class to produce a *WritableImage.* This image would then be passed to the static method *storeImage* where it would be converted to a *BufferedImage* using the method *fromFXImage* in the *SwingFxUtils* class. This *BufferedImage* could then be saved as a PNG image in the directory chosen by the user. Alert displays would be shown via the *showErrorMessage* method in *mandelGuiDelegate* class if errors in any of these three operations occurred.

Testing

An example of a data file stored, and an image saved using this GUI was also added to this submission in a folder called Tests

The following snapshots show some of the program's abilities:

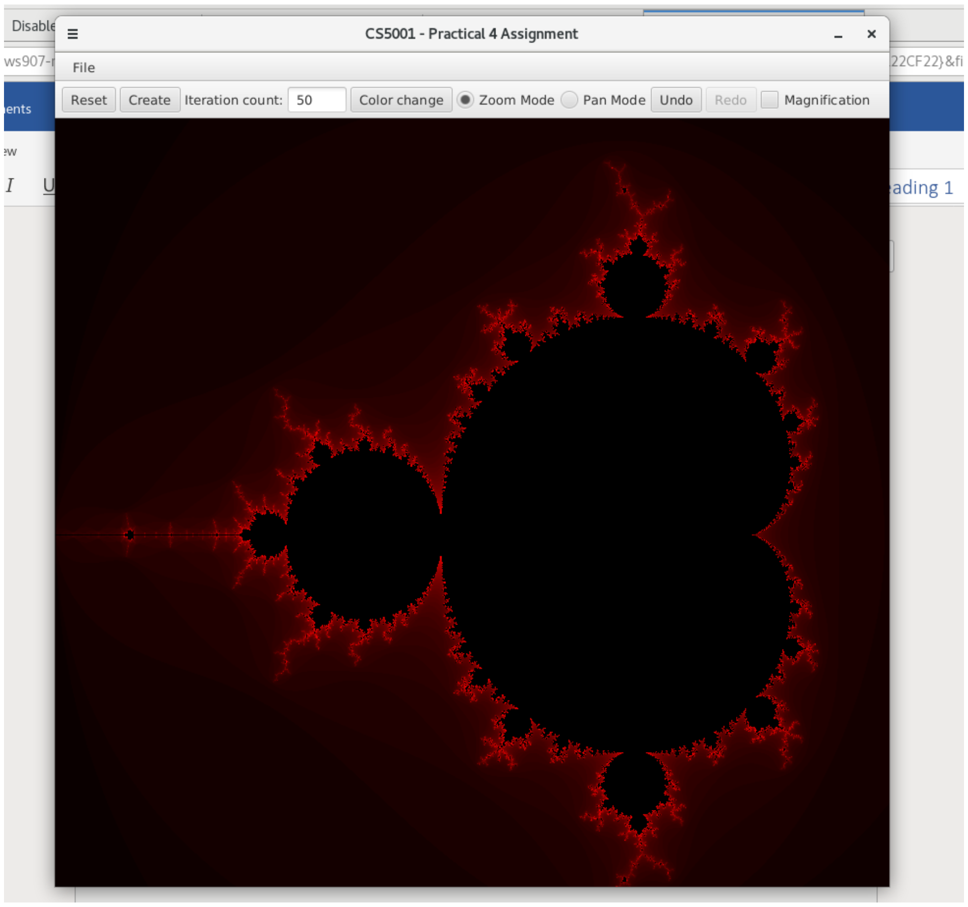
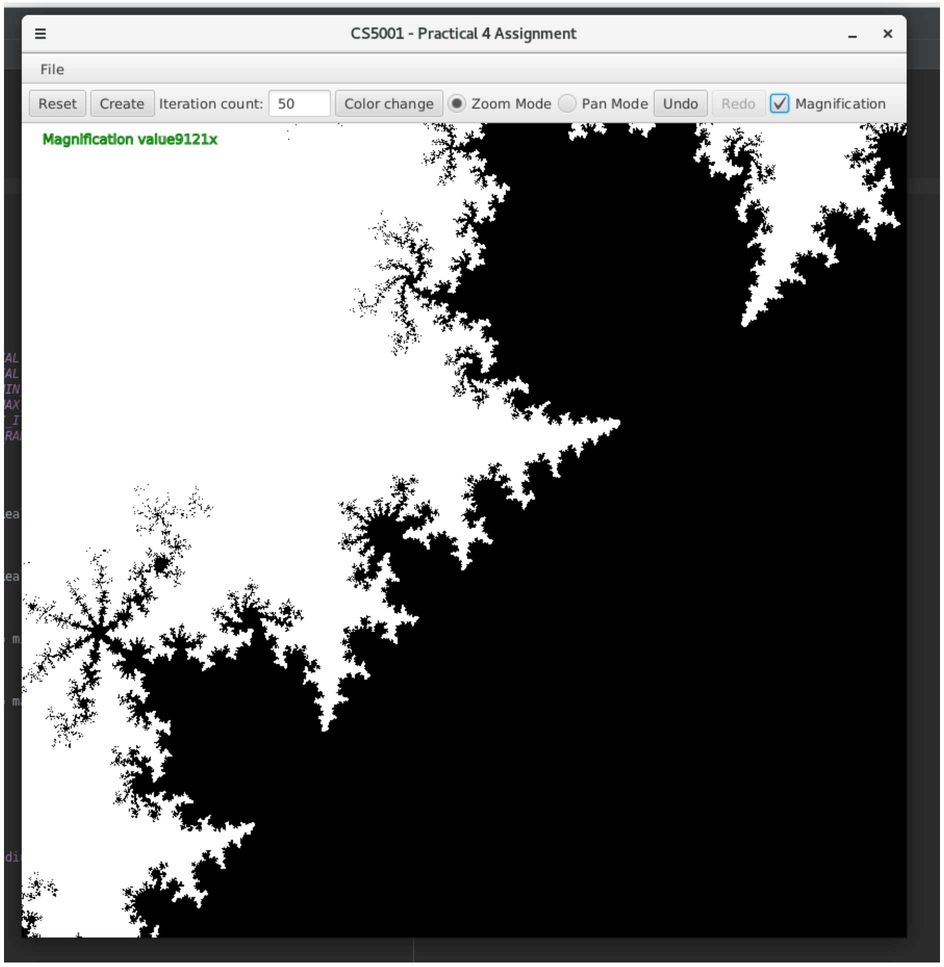


Figure 1.) The figure above shows the program’s ability to display the Mandelbrot set using a red color and simultaneously applying the brightness scaling as previously described.



Magnification value9121x

Figure 2.) The figure above shows the magnification scale of the current zoomed region.

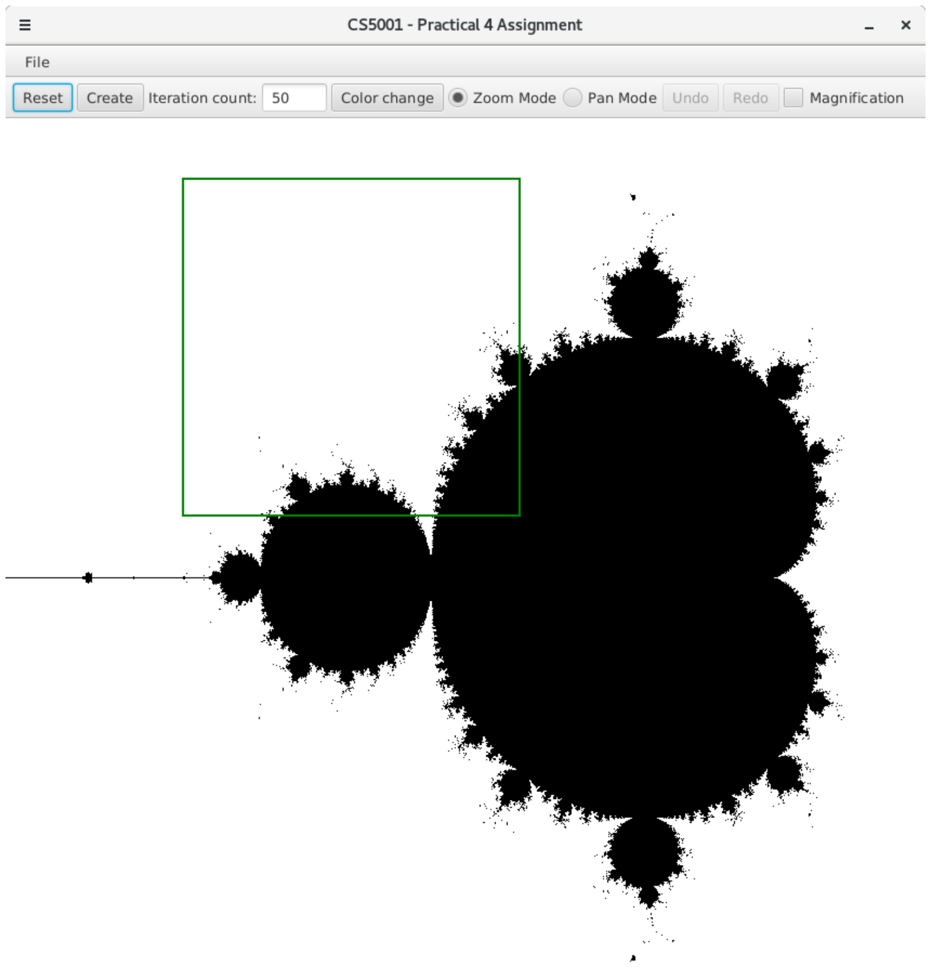


Figure 3.) This figure displays the zoom ability. A green square is drawn as the mouse (not picture) is dragged across the image.

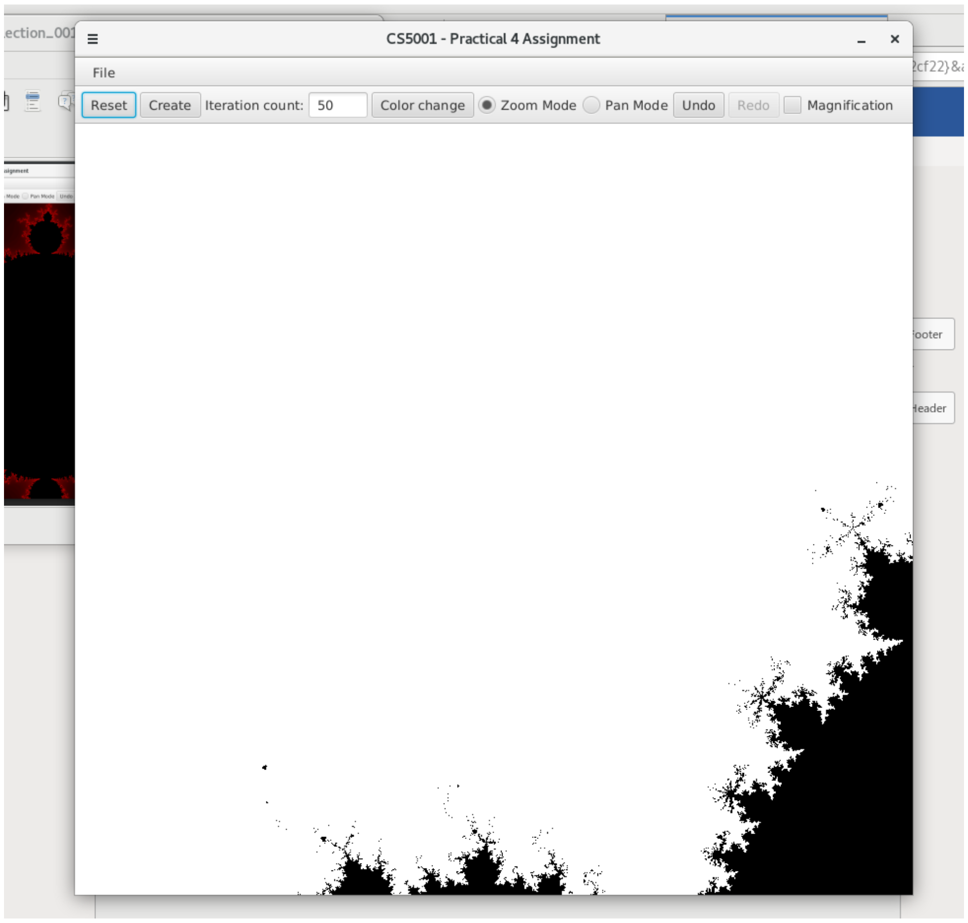


Figure 4.) This is the result of the zooming step of the previous figure.

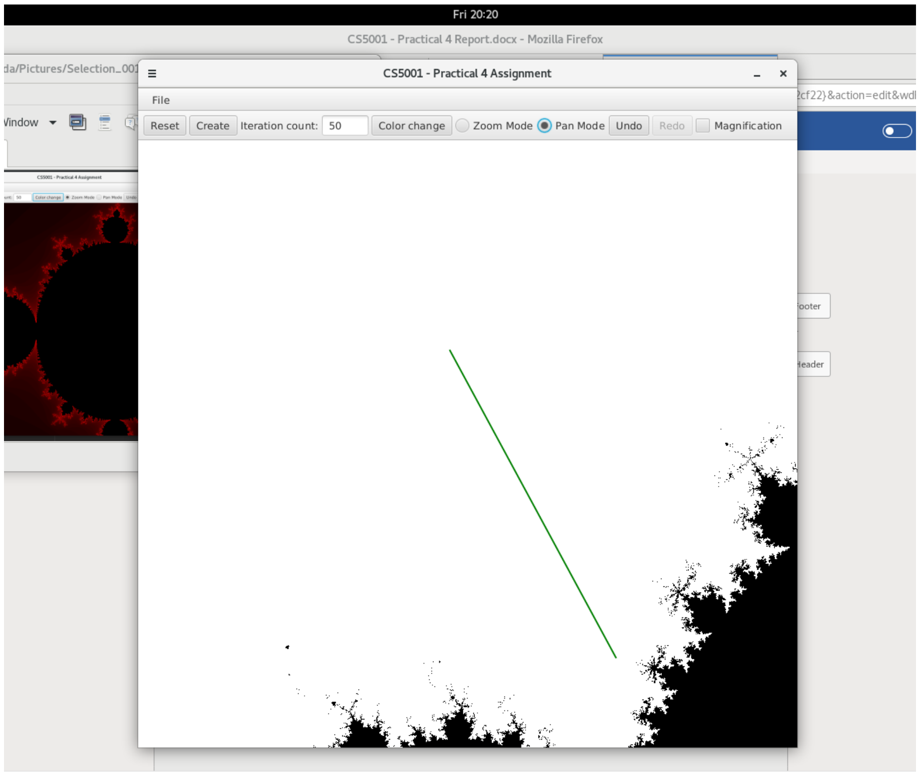


Figure 5.) The figure above displays the panning capabilities of the GUI. A green line anchored to the first mouse click displays the direction and the amount of the panning action.

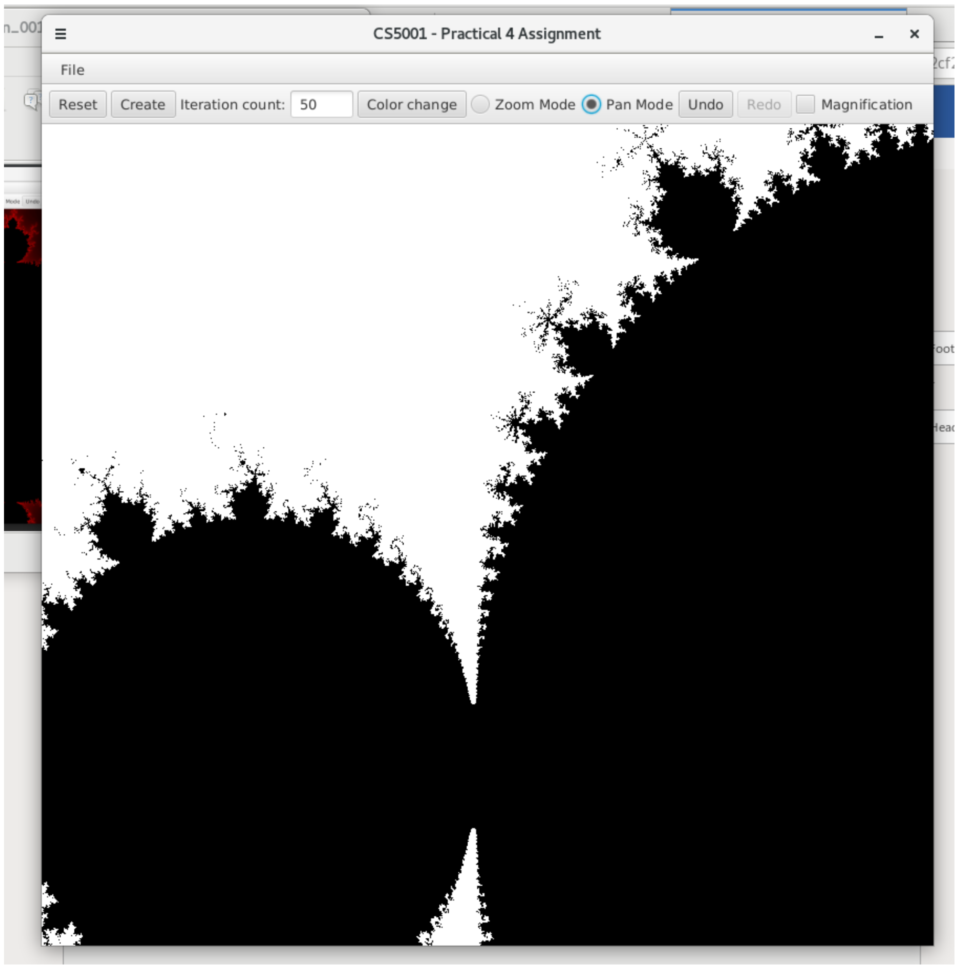


Figure 6.) The figure above displays the result of the panning action from the previous image.

Evaluation

The program developed fulfills the basic requirements and most the enhancements. However, even though the functionality is achieved, the program is not very modular and requires restructuring (creating new classes, refactoring methods) that would improve the flow. Better organization improves readability and currently some of the functions are too extensive. Standalone event handlers should be added and moved to a different class (I.e. components regarding mouse events and data storage).

The process of storing parameters should potentially use an XML format instead of writing the parameters into a text file. This would make storage more flexible and use a more standard/proven approach of storing data.

The *FileChooser* dialog when the load button is pressed should only allow for text files to be displayed. An extension filter should be implemented to reduce the probability of the user selecting a file with an unsupported format. The program currently does not check if all parameters have been loaded so if there are duplicate "valid" parameters, the program will load a mixture of the default parameters and the stored values. More testing should be performed to check if the program can handle "edge cases" especially in the *DataStorage* class.

Having two superimposed canvas 1 for the Mandelbrot set and 1 for drawing the magnification scale and the zooming in rectangle could also be an improvement. Currently, whenever the user wants the magnification scale to be displayed, the whole canvas is redrawn. However, since the slowest process is the Mandelbrot set computation, the benefits achieved from implementing might be small.

Some Javadoc comments were missing from the project and overall the documentation/comments should be better. Feedback should be provided to the user in case an invalid entry (I.e. not a number) is inserted into the max iteration input field.

An about page should also be added to explain to the user how to use the GUI and contain relevant information regarding the software (I.e. software version). Another aspect that would be interesting to explore would be the possibility of the *MandelModel* to inherit the *MandelbrotCalculator* and check if this could improve the overall structure of the program. The program is also designed to only work with screen/canvas with a 1:1 ratio (width: height) which limits the ability to zoom. This is the reason why resizing of the entire window was disabled. Further exploration on how to preserve the ratio when changing the window would be required.

The use of stacks to implement the redo and undo methods is simple however it means this functionality is limited by the maximum stack size. A circular array (or similar structure) should be implemented which would allow for data to be overwritten since it could allow for the most recent undo/redo to be stored at the cost of losing the old data. Alternatively, if the stack gets full, the older elements should be popped.

Another issue was that whenever the user changes the *maxIterations* using the *inputField* followed by a request to change color provides the wrong image. This occurs because the scaling of the brightness when redrawing the image would apply the new *maxIterations* value to the Mandelbrot data set generated using the old value. This should be address by only refreshing the model's parameters before recalculating the data.