Cover Page

I, \_*Xiaoxi Zheng*\_\_\_ affirm that the work submitted is my own and that the Honor  
Code was neither bent nor broken.

The easiest part of this HW is the setup of this project, since the structure of code  
was quite straight forward implemented on top of HW9. The more difficult parts of the HW is buried within implementation of the recursion of calling the Mandelbrot and Julia methods when zooming. I spend some times to draft out the algebra behind zooming, and making sure the Cartesian coordinates and the complex coordinates line up after zooming. I also spend times debugging for Array out of bound exceptions when trying to convert coordinates back and forth between the Cartesian plane and the complex plane.

I believe the objective of this assignment was for us to understand the Timer object and implement Animation frame by frame using JPanel and JFrames. How these classes and objects call on each other, and making sure we understand the structural relationship between them.

Code

import javax.swing.SwingUtilities;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JOptionPane;

import javax.swing.JFileChooser;

import javax.swing.JMenuBar;

import javax.swing.JMenu;

import javax.swing.JMenuItem;

import javax.swing.JLabel;

import javax.swing.Timer;

import javax.imageio.ImageIO;

import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.Color;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Point;

import java.awt.image.BufferedImage;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.MouseEvent;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseMotionAdapter;

import java.awt.event.KeyEvent;

import javax.swing.KeyStroke;

import java.io.File;

import java.io.IOException;

public class hw10{

private static final int WIDTH = 600;

private static final int HEIGHT = 450;

public static void main( String[] args){

new hw10();

}

public hw10() {

new ImageFrame(WIDTH,HEIGHT);

}

//####################################################################

private class ImageFrame extends JFrame {

//private static final int INFINITE = 1000;

private fractalDisplayPanel panel;

private JLabel label;

private Timer timer;

private int width = 600;

private int height = 450;

private double x = 0;

private double y = 0;

//threashold for divergence test

private int tmax = 100;

public boolean mandelbrot;

private BufferedImage image = null;

private int [] colorSchema = new int [101];

private double r0 = -2;

private double r1 = 2;

private double deltaR = r1-r0;

private double i0 = -1.5;

private double i1 = 1.5;

private double deltaI = i1-i0;

//Julia constant μ saved

private double[] constant = new double[2];

//=========================

public ImageFrame(int width, int height){

this.setTitle("CAP 3027 2015 - HW09 -XiaoxiZheng");

this.setSize( width, height );

this.setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE);

addMenu();////add a menu to the frame

image = simulatedImage(width,height);

panel = new fractalDisplayPanel( image);

this.getContentPane().add( panel, BorderLayout.CENTER );

label = new JLabel("Click and hold to zoom (LMB to zoom in/RMB to zoom out)");

this.getContentPane().add( label, BorderLayout.SOUTH );

this.pack();

this.setVisible( true );

}

private void addMenu(){

JMenu fileMenu = new JMenu("File Menu");

//load IFS description

JMenuItem mandelbrot = new JMenuItem("Mandelbrot");

mandelbrot.addActionListener( new ActionListener(){

public void actionPerformed( ActionEvent event){

//initial μ value @(-2 + 1.5i) ---> Top Left

r0 = -2;

r1 = 2;

deltaR = r1-r0;

i0 = -1.5;

i1 = 1.5;

deltaI = i1-i0;

produceDefaultMandelbrot(r0,i0,r1,i1);

}

} );

fileMenu.add(mandelbrot);

JMenuItem julia = new JMenuItem("Julia Set");

julia.addActionListener( new ActionListener(){

public void actionPerformed(ActionEvent event){

//initial Z value @(-2 + 1.5i) ---> Top Left

constant = promptForMiu();

r0 = -2;

r1 = 2;

deltaR = r1-r0;

i0 = -1.5;

i1 = 1.5;

deltaI = i1-i0;

produceDefaultJulia(r0,i0,r1,i1, constant[0], constant[1]);

}

} );

fileMenu.add(julia);

//Save image

JMenuItem saveImage = new JMenuItem("Save Image");

saveImage.addActionListener( new ActionListener(){

public void actionPerformed( ActionEvent event){

saveImage();

}

} );

fileMenu.add(saveImage);

//Exit

JMenuItem exitItem = new JMenuItem("Exit");

exitItem.addActionListener( new ActionListener(){

public void actionPerformed(ActionEvent event){

System.exit( 0 );

}

} );

fileMenu.add( exitItem);

//attach menu to a menu bar

JMenuBar menuBar = new JMenuBar();

menuBar.add( fileMenu);

this.setJMenuBar( menuBar);

}

public void mandelbrot(double ru, double iu,double ru1,double iu1){

//interpolate colors and store them in ColorSchema [] array

interpolateColor();

deltaR = ru1- ru;

deltaI = iu1 - iu;

double realIncr = (ru1- ru)/(width - 1);

double imagIncr = (iu1 - iu)/(height -1);

//initial μ value @(-2 + 1.5i) ---> Top Left

double real = ru;

//mandelbrot algorithm

//looping thru a 600\*450 bounded region

for(int x=0; x< width-1; x++){

double img = iu1;

for(int y=0; y<height-1; y++){

double [] complexZ = new double [2];

// temp variable to store real and img part of z when computing;

complexZ[0] = 0;

complexZ[1] = 0;

int t = 0;

while(t!=tmax){

//z = (z\*z) + u

//update realZ, imgZ

complexZ = zSquarePlusMiu(complexZ[0],complexZ[1],real,img);

if(sumOfSquare(complexZ[0],complexZ[1]) > 4.0) {

break;//diverging

}

else{

t++;

}

}

if(t == tmax){

//Plot black

double [] bitmapCoord = new double [2];

//temp variable to store the converted bitmap interpretation of complex number

bitmapCoord = toBitmapCoord(real,r0,r1,img,i0,i1);

if((int)bitmapCoord[0]< 0|| (int)bitmapCoord[1] < 0 ||(int)bitmapCoord[0] > width-1 || (int)bitmapCoord[1] > height-1 ){

//do nothing..out of bound

}

else{

//System.out.println("(bitmapCoord(x,y): " + (int)Math.round(bitmapCoord[0])+ ", " + (int)Math.round(bitmapCoord[1]));

image.setRGB((int)(bitmapCoord[0]),(int)(bitmapCoord[1]),colorSchema[t]);

}

}

else{

//diverged and μ is not in Mandelbrot set

//plot μ using colorSchema[t].

double [] bitmapCoord = new double [2]; //temp variable to store the converted bitmap interpretation of complex number

bitmapCoord = toBitmapCoord(real,r0,r1,img,i0,i1);

if((int)bitmapCoord[0]< 0|| (int)bitmapCoord[1] < 0 ||(int)bitmapCoord[0]> width-1|| (int)bitmapCoord[1] > height-1 ){

//do nothing..out of bound

}

else{

//System.out.println("(bitmapCoord(x,y): " + (int)Math.round(bitmapCoord[0])+ ", " + (int)Math.round(bitmapCoord[1]));

image.setRGB((int)(bitmapCoord[0]),(int)(bitmapCoord[1]),colorSchema[t]);

}

//System.out.println("(bitmapCoord(x,y): " + (int)Math.round(bitmapCoord[0])+ ", " + (int)Math.round(bitmapCoord[1]));

}

img -= imagIncr;

}

real += realIncr;

}

}

private void produceDefaultMandelbrot(double r0\_,double i0\_, double r1\_ ,double i1\_){

//display default image

new Thread(new Runnable() {

public void run(){

mandelbrot(r0\_,i0\_,r1\_,i1\_);

SwingUtilities.invokeLater(new Runnable() {

public void run() {

mandelbrot = true;

panel.setImage(image);

}

});

}

}).start();

}

public void julia(double rz, double iz, double rz1, double iz1, double rMiu\_, double iMiu\_){

double [] complexU = new double [2];

complexU[0] = rMiu\_;

complexU[1] = iMiu\_;

//interpolate colors and store them in ColorSchema [] array

interpolateColor();

deltaR = rz1- rz;

deltaI = iz1 - iz;

double realIncr = (rz1- rz)/(width - 1);

double imagIncr = (iz1 - iz)/(height -1);

//initial μ value @(-2 + 1.5i) ---> Top Left

double real = rz;

//imgU = iu;

//Julia algorithm

//looping thru a 600\*450 bounded region

for(int x=0; x< width-1; x++){

double img = iz1;

//realU = realU + 4/width;

for(int y=0; y<height-1; y++){

double [] complexZ = new double [2]; // temp variable to store real and img part of z when computing;

complexZ[0] = real;

complexZ[1] = img;

int t = 0;

while(t!=tmax){

//z = (z\*z) + u

complexZ = zSquarePlusMiu(complexZ[0],complexZ[1],complexU[0],complexU[1]);

if(sumOfSquare(complexZ[0],complexZ[1]) > 4.0) {

break;//diverging

}

else{

++t;

}

}

if(t ==tmax){

//Plot black

double [] bitmapCoord = new double [2]; //temp variable to store the converted bitmap interpretation of complex number

bitmapCoord = toBitmapCoord(real,r0,r1,img,i0,i1);

if((int)bitmapCoord[0]< 0|| (int)bitmapCoord[1] < 0 ||(int)bitmapCoord[0]> width-1|| (int)bitmapCoord[1] > height-1 ){

//do nothing..out of bound

}

else{

//System.out.println("(bitmapCoord(x,y): " + (int)Math.round(bitmapCoord[0])+ ", " + (int)Math.round(bitmapCoord[1]));

image.setRGB((int)(bitmapCoord[0]),(int)(bitmapCoord[1]),colorSchema[t]);

}

}

else{

//diverged and μ is not in Mandelbrot set

//plot μ using colorSchema[t].

double [] bitmapCoord = new double [2]; //temp variable to store the converted bitmap interpretation of complex number

bitmapCoord = toBitmapCoord(real,r0,r1,img,i0,i1);

if((int)bitmapCoord[0]< 0|| (int)bitmapCoord[1] < 0 ||(int)bitmapCoord[0]> width-1|| (int)bitmapCoord[1] > height-1 ){

//do nothing..out of bound

}

else{

//System.out.println("(bitmapCoord(x,y): " + (int)Math.round(bitmapCoord[0])+ ", " + (int)Math.round(bitmapCoord[1]));

image.setRGB((int)(bitmapCoord[0]),(int)(bitmapCoord[1]),colorSchema[t]);

}

}

img -= imagIncr;

}

real += realIncr;

}

}

public void produceDefaultJulia(double r0,double i0,double r1,double i1, double constant0, double constant1){

new Thread(new Runnable() {

public void run(){

julia(r0,i0,r1,i1, constant0, constant1);

SwingUtilities.invokeLater(new Runnable() {

public void run() {

mandelbrot = false;

panel.setImage(image);

}

});

}

}).start();

}

private double [] zSquarePlusMiu(double rZ,double iZ,double rU,double iU){

double [] answer = new double [2];

//compute Z^2 + μ in complex form

answer[0] = (rZ\*rZ) - (iZ\*iZ)+ rU;

answer[1] = ((rZ\*iZ) + (iZ\*rZ)) + iU;

//answer[1] = (2\*rZ\*iZ) + iU;

return answer;

}

private double sumOfSquare(double realZ\_,double imgZ\_){

double answer = 0.0;

answer = realZ\_\*realZ\_ + imgZ\_\*imgZ\_;

return answer;

}

private double [] toBitmapCoord(double realU\_,double r0\_,double r1\_,double imgU\_,double i0\_,double i1\_){

double answer [] = new double [2];

double deltaR\_ = r1\_-r0\_;

double deltaI\_ = i1\_-i0\_;

answer[0] = ((realU\_ - r0\_ ) / deltaR\_\* (width-1));

answer[1] = (imgU\_ - i0\_ ) / deltaI\_ \* (height-1);

return answer;

}

private void interpolateColor(){

int ARGBNewGeneral = 0;

double[] colorInfoLeft;

double[] colorInfoRight;

//ignores delta alpha bc in this hw has no changes in alpha

double deltaRGen;

double deltaGGen;

double deltaBGen;

double redGeneral; //start paiting @ left

double greenGeneral; //start paiting @ left

double blueGeneral; //starting paiting @ left

//color[0] = white, color[40] = red, color[100] = Blue

//interpolate from 0---50

//follows the rule of thumb of right hand side of canvas - left hand side of canvas

colorInfoLeft = extraction(16711680);//extract white

colorInfoRight = extraction(16747520);//extract orange

//ignores delta alpha bc in this hw has no changes in alpha

deltaRGen = (colorInfoRight[1] - colorInfoLeft[1])/(49); //[1]--channel for red

deltaGGen = (colorInfoRight[2] - colorInfoLeft[2])/(49); //[2]--channel for green

deltaBGen = (colorInfoRight[3] - colorInfoLeft[3])/(49); //[3]--channel for blue

redGeneral = colorInfoLeft[1]; //start paiting @ left

greenGeneral = colorInfoLeft[2]; //start paiting @ left

blueGeneral = colorInfoLeft[3]; //starting paiting @ left

for(int x = 0; x<49;x++){

redGeneral = redGeneral + deltaRGen; greenGeneral = greenGeneral + deltaGGen;

blueGeneral = blueGeneral + deltaBGen;

//clamping

if(redGeneral>255){

redGeneral = 255;

}

if(redGeneral<0){

redGeneral = 0;

}

if(greenGeneral>255){

greenGeneral = 255;

}

if(greenGeneral<0){

greenGeneral = 0;

}

if(blueGeneral>255){

blueGeneral = 255;

}

if(blueGeneral<0){

blueGeneral = 0;

}

ARGBNewGeneral = toIntARGB(255,redGeneral,greenGeneral,blueGeneral);

colorSchema[x] =ARGBNewGeneral;//record the color information in the colorArray for future use }

}

//50---100

//follows the rule of thumb of right hand side of canvas - left hand side of canvas

colorInfoLeft = extraction(16747520);//extract orange

colorInfoRight = extraction(16711680);//extract red

//ignores delta alpha bc in this hw has no changes in alpha

deltaRGen = (colorInfoRight[1] - colorInfoLeft[1])/(49); //[1]--channel for red

deltaGGen = (colorInfoRight[2] - colorInfoLeft[2])/(49); //[2]--channel for green

deltaBGen = (colorInfoRight[3] - colorInfoLeft[3])/(49); //[3]--channel for blue

redGeneral = colorInfoLeft[1]; //start paiting @ left

greenGeneral = colorInfoLeft[2]; //start paiting @ left

blueGeneral = colorInfoLeft[3]; //starting paiting @ left

for(int x = 49; x<100;x++){

redGeneral = redGeneral + deltaRGen; //bc red starts from the left

greenGeneral = greenGeneral + deltaGGen;

blueGeneral = blueGeneral + deltaBGen;

//clamping

if(redGeneral>255){

redGeneral = 255;

}

if(redGeneral<0){

redGeneral = 0;

}

if(greenGeneral>255){

greenGeneral = 255;

}

if(greenGeneral<0){

greenGeneral = 0;

}

if(blueGeneral>255){

blueGeneral = 255;

}

if(blueGeneral<0){

blueGeneral = 0;

}

ARGBNewGeneral = toIntARGB(255,redGeneral,greenGeneral,blueGeneral);

colorSchema[x] =ARGBNewGeneral;//record the color information in the colorArray for future use }

}

}

private double[] extraction(int ARGB\_){

double[] extractionArray;

extractionArray = new double[4];

//extractionArray -- extraction[0] = alpha values;

//extraction[1] = red values; & etc with ARGB

extractionArray[0] = ARGB\_>>>24;

extractionArray[1] = (ARGB\_<<8) >>> 24;

extractionArray[2] = (ARGB\_<<16)>>>24;

extractionArray[3] = (ARGB\_<<24)>>>24;

return (extractionArray);

}

private int toIntARGB(double alpha\_, double red\_, double green\_, double blue\_){

//System.out.println((alpha\_<<24)|(red\_<<16)|(green\_<<8)|(blue\_));

return ((((int)alpha\_)<<24)|(((int)red\_)<<16)|((int)(green\_)<<8)|((int)blue\_));

}

private void saveImage(){

try

{

if(mandelbrot){

File outputfile = new File("mandelbrot.png");

javax.imageio.ImageIO.write(image, "png", outputfile );

}

else{

File outputfile = new File("JuliaSet.png");

javax.imageio.ImageIO.write(image, "png", outputfile );

}

}

catch ( IOException e )

{

JOptionPane.showMessageDialog( ImageFrame.this,

"Error saving file",

"oops!",

JOptionPane.ERROR\_MESSAGE );

}

}

private double [] promptForMiu(){

double [] temp = new double[2];

double [] error = new double[2];

error[0] = -100000;

error[1] = 100000;

String input1 = JOptionPane.showInputDialog("Please enter the real part of Mu ");

String input2 = JOptionPane.showInputDialog("Please enter the imaginary part of Mu");

if(valideInput(input1) && valideInput(input2)){

temp[0] = Double.parseDouble(input1);

temp[1] = Double.parseDouble(input2);

return temp;

}

else if (input1 == null || input2 == null){ //User clicked "Cancel"

System.exit(0);

return error;

}

else{

return promptForMiu();

}

}

private boolean valideInput(String input\_){

try{

double num = Double.parseDouble(input\_);

if(num<-1000 || num > 1000){

JOptionPane.showMessageDialog(null, "Invalid Input", "alert", JOptionPane.ERROR\_MESSAGE);

return false;

}

return true;

}

catch(NumberFormatException e){

JOptionPane.showMessageDialog(null, "Invalid Input", "alert", JOptionPane.ERROR\_MESSAGE);

return false;

}

}

protected BufferedImage simulatedImage(int width\_,int height\_){

while (true) {

if (width\_ < 0 || height\_ < 0)

return null;

try {

BufferedImage img = new BufferedImage(width\_,height\_,BufferedImage.TYPE\_INT\_RGB);

return img;

} catch (OutOfMemoryError err) {

JOptionPane.showMessageDialog(this, "Ran out of memory! Try using a smaller image size.");

}

}

}

//nested FractalDisplayPanel class

private class fractalDisplayPanel extends JPanel{

// panel size

private final int WIDTH, MAX\_X;

private final int HEIGHT, MAX\_Y;

// image displayed on panel

private BufferedImage image;

private Graphics2D g2d;

private Timer timer;

//[T]---zooming-In; [F]---zooming-OUT

private boolean zoom;

//the point in which the mouse clicked

private Point mouseLoc;

private final int MILLESECONDS\_BETWEEN\_FRAMES = 17;

private final double ZOOM\_PERCENTAGE = 0.05;

private static final double MAXIMUM\_ZOOM = 1E-11;

private final double ZOOM\_RATE = ZOOM\_PERCENTAGE / 2;

private final double LEFT\_ZOOM\_IN\_BOUND = ZOOM\_RATE;

private final double RIGHT\_ZOOM\_IN\_BOUND = 1.0 - ZOOM\_RATE;

private final double LEFT\_ZOOM\_OUT\_BOUND = -ZOOM\_RATE;

private final double RIGHT\_ZOOM\_OUT\_BOUND = 1.0 + ZOOM\_RATE;

//------------------------------------------------------------------------

// constructor

public fractalDisplayPanel( BufferedImage image ){

this.image = image;

g2d = image.createGraphics();

// define panel characteristics

WIDTH = image.getWidth();

HEIGHT = image.getHeight();

Dimension size = new Dimension( WIDTH, HEIGHT );

setMinimumSize( size );

setMaximumSize( size );

setPreferredSize( size );

MAX\_X = WIDTH - 1;

MAX\_Y = HEIGHT - 1;

mouseLoc = null;

zoom = true;

//initialize Timer

timer = new Timer(MILLESECONDS\_BETWEEN\_FRAMES, new ImageZoomer());

this.addMouseListener( new MouseAdapter(){

public void mousePressed( MouseEvent event ){

if(event.getButton() == MouseEvent.BUTTON1){

// zooming in stuff

zoom = true;

mouseLoc = event.getPoint();

timer.start();

}

else if(event.getButton() == MouseEvent.BUTTON3){

// zooming out stuff

zoom = false;

mouseLoc = event.getPoint();

timer.start();

}

}

public void mouseReleased(MouseEvent event){

timer.stop();

}

});

this.addMouseMotionListener( new MouseMotionAdapter(){

public void mouseDragged(MouseEvent event){

mouseLoc = event.getPoint();

//claming mouse points within the border of display

int x = (int)mouseLoc.getX();

int y = (int)mouseLoc.getY();

if (x < 0)

x = 0;

else if (x > width-1)

x = width-1;

if (y < 0)

y = 0;

else if (y > height-1)

y = height-1;

mouseLoc.setLocation(x, y);

}

});

}

//------------------------------------------------------------------------

public void setImage( BufferedImage src ){

g2d.drawImage( src,

0, 0, MAX\_X, MAX\_Y,

0, 0, (src.getWidth() - 1), (src.getHeight() - 1),

null

);

repaint();

}

//------------------------------------------------------------------------

// behaviors

public void paintComponent( Graphics g ){

super.paintComponent( g );

g.drawImage( image, 0, 0, null );

}

private class ImageZoomer implements ActionListener{

public void actionPerformed(ActionEvent evt){

new Thread(new Runnable() {

public void run() {

// precompute variables that will be used multiple times

double newX = 2 \* (mouseLoc.getX() / MAX\_X - 0.5);

double newY = 2 \* (mouseLoc.getY() / MAX\_Y - 0.5);

double deltaR = r1 - r0;

double deltaI = i1-i0;

double xCoff = newX \* ZOOM\_RATE;

double yCoff = newY \* ZOOM\_RATE;

// Zooming in portion

if (zoom) {

double newr0 = r0 + (LEFT\_ZOOM\_IN\_BOUND + xCoff) \* deltaR;

double newr1 = r1 - (1.0 - (RIGHT\_ZOOM\_IN\_BOUND + xCoff)) \* deltaR;

double newi0 = i0 + (LEFT\_ZOOM\_IN\_BOUND + yCoff) \* deltaI;

double newi1 = i1 - (1.0 - (RIGHT\_ZOOM\_IN\_BOUND + yCoff)) \* deltaI;

if (newr1 - newr0 <= MAXIMUM\_ZOOM || newi1 - newi0 <= MAXIMUM\_ZOOM) {

SwingUtilities.invokeLater(new Runnable() {

public void run() {

JOptionPane.showMessageDialog(ImageFrame.this, "WARNING\_CANNOT\_ZOOM\_FURTHER", "ERROR!", JOptionPane.ERROR\_MESSAGE);

}

});

timer.stop(); // Make sure to stop the timer

return;

}

r0 = newr0;

r1 = newr1;

i0 = newi0;

i1 = newi1;

}

// Zooming out portion

else {

r0 = r0 + (LEFT\_ZOOM\_OUT\_BOUND - xCoff) \* deltaR;

r1 = r1 - (1.0 - (RIGHT\_ZOOM\_OUT\_BOUND - xCoff)) \* deltaR;

i0 = i0 + (LEFT\_ZOOM\_OUT\_BOUND - yCoff) \* deltaI;

i1 = i1 - (1.0 - (RIGHT\_ZOOM\_OUT\_BOUND - yCoff)) \* deltaI;

}

// If the panel currently has a Mandelbrot set

if (mandelbrot) {

//call mandelbrot

mandelbrot(r0,i0,r1,i1);

}

else{

julia(r0,i0,r1,i1,constant[0],constant[1]);

}

// Update the image

SwingUtilities.invokeLater(new Runnable() {

public void run() {

panel.setImage(image);

}

});

}

}).start();

}//action performed

}//image zoomer

}//FractalPanel

}//Jframe

}//hw10

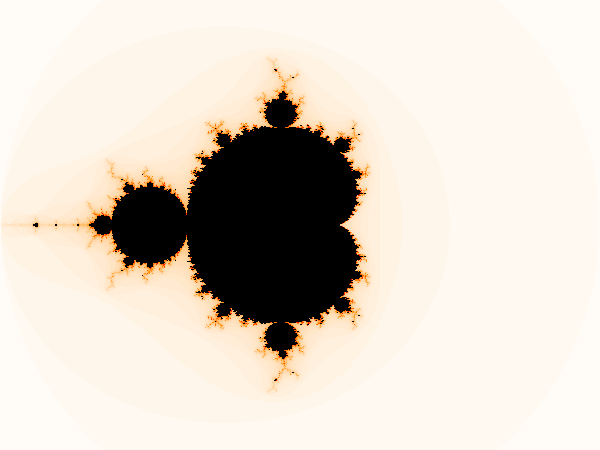
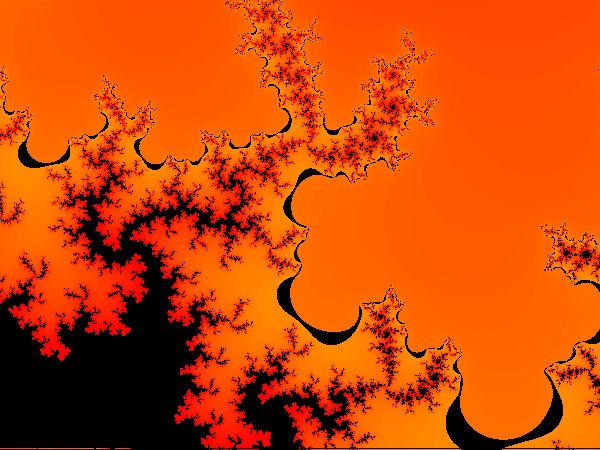
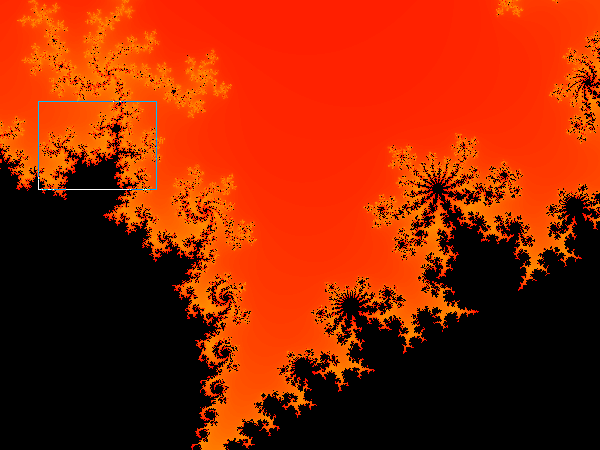
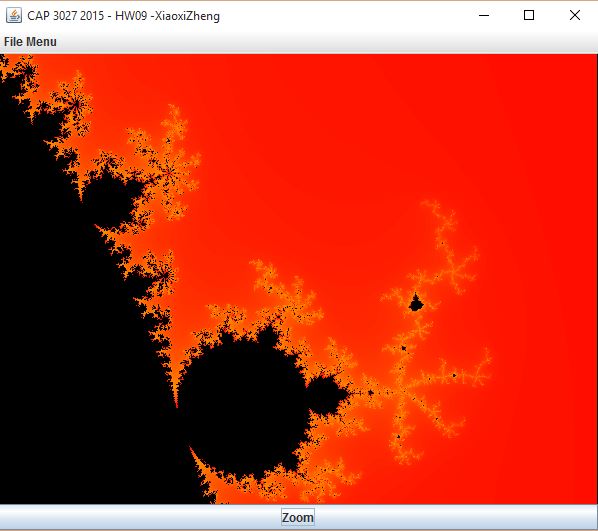
=====================================================================================

Question

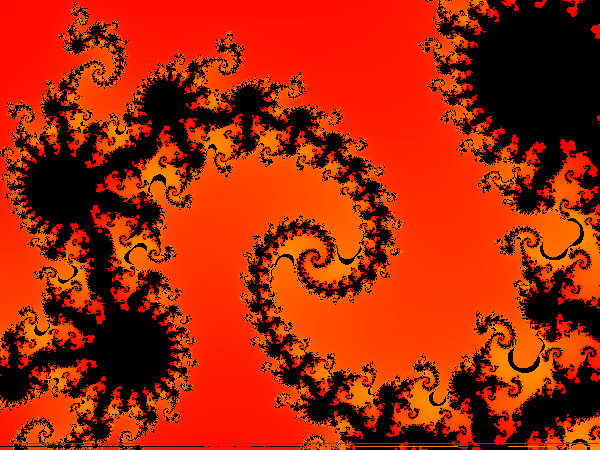
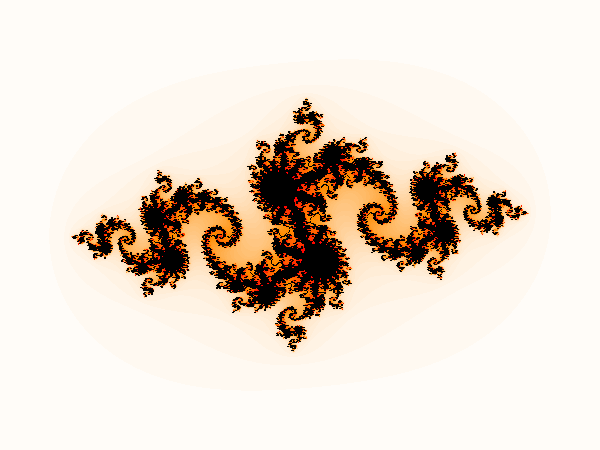
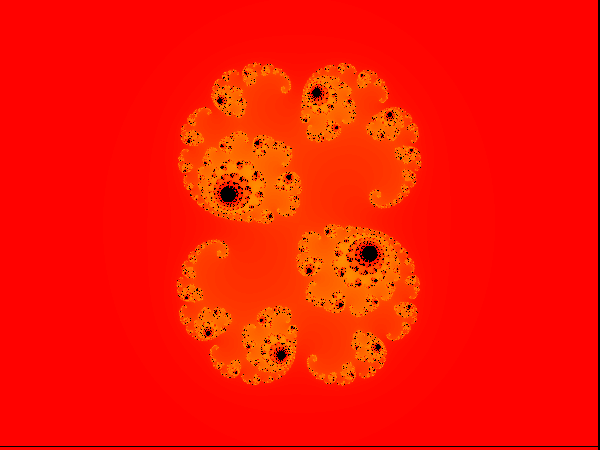
1. Does the program compile without errors?  
Yes.  
2. 2. Does the program compile without warnings?  
Yes  
3. 3. Does the program run without crashing?  
Yes  
4. 4. Describe how you tested the program.  
I ran several test cases with different inputs for Julia. And make sure the default image for Mandelbrot set is displaying and zooming correctly.   
5. Describe the ways in which the program does not meet assignment's specifications.  
In the GIFs I provided, the image appear to be tearing a bit, and slow to update for every frame. I based on the assumption that could be the nature of my Graphics Card and the amount of other programs I had running on my laptop at the time.   
6. Describe all known and suspected bugs.  
There are no known bugs.  
7. Does the program run correctly?  
Yes

Screenshots

## Mandelbrot set



## Julia Set



\*\*\*Please see folder for the animated GIFs\*\*\*