Lab 1 - Chaos, Fractals, and Computer Graphics

Luciano Giannini, Ryan Pepin, Mike Emmett
LC 207: Mathematical Experiments in Computer Science
(Dated 09/25/2019)

Giannini, Pepin, Emmett

Contents

Introduction 2

Program 1: Drawing Sierpinski Gasket 4

Program 2: Following Valentines Remarks 5

Program 3: Drawing Sierpinski's Carpet 7

Program 4: Drawing Our Own Fractal 9

Program 5 & 6: The Julia Set 11

Problem 7: The Mandelbrot Set 13

Problem 8: The Chaos Theory 14

Problem 9: Representing the Chaos Theory in a Graph 15

Problem 10: The Butterfly Effect 16

Problem 11: Themes of Arcadia 17

What We Liked About This Lab 18

What We Did Not Like About This Lab 19

What We Learned From This Lab 20

Introduction

In the following lab, we will be looking at the mathematics discussed throughout the play *Arcadia*. The play of *Arcadia* is a particularly interesting one because it takes place during two different time periods, at the same time. The first part of the story describes a young girl, Thomasina, and her tutor Septimus discussing and developing their own way of working through an iterated algorithm. The second part of the play talks about how Hannah and Valentine are exploring and understanding the type of math that Thomasina was demonstrating back in 1809, and how it relates to Valentine's work today. Throughout this play, there were similarities relating both storylines such that the setting is in the same country house in Sidley Park, Septimus's copy of Mr. Chater's novel with the notes he slipped inside of it still being present in the storyline, and the constant turtle being present on both tables.

Arcadia is significant to this lab because it discusses Thomasina's idea that everything in existence has a formula that maps itself. As Valentine was explaining to Hannah later on in the story Thomasina was working through equations, by using an x value to figure out a y value and then plugging that found y value back into the equation as the next x value. Valentine discusses how what Thomasina was doing back in the 1800s was very similar to the work he is doing today involving feedback and fractal geometry. Using The feedback method Thomasina was able to demonstrate anticipation of fractal geometry in her hope of finding an equation to plot everything. Fractal Geometry uses complex numbers to model complex objects and gives us a deeper understanding of how the world is mapped through the eyes of mathematicians. As Valentine described a fractal object can best be described as a photograph that is blow up and then the detail is blow up again and so on and so on an infinite amount of times. In this lab, the first example of a fractal object we encounter is a Sierpinski Triangle. The Sierpinski Triangle is

an equilateral triangle that is divided into an infinite amount of smaller triangles recursively for an infinite amount of times. The next fractal objects we explore are the Julia set, and Mandelbrot set. Both of these are using Thomasina's idea of feeding information gathered from an equation back into the previous one. This concept was also used to derive population patterns, and to predict future generations. This idea was brought about when Valentine and Hanna started exploring the hunting journal from 1809. All of these methods allow for these two sets to create vivid colorful images, and complex plots of data regarding population, that we explore later on in this lab. As we begin this lab we are asked to use the Sierpinski Triangle to further deepen our understanding of fractal geometry, and how it works.

Problem 1: Drawing Sierpinski Gasket

Objective: Write a program that will draw the Sierpinski Gasket. Given the recursive method:

a. void Sierpinski(int x1, int y1, int x2, int y2, int x3, int y3, int depth, Graphics g)
b. where xi, yi for i = 1 to 3, are the vertices of a triangle and depth is the maximum level of recursion.

```
public void sierpienski(int x1, int y1, int x2, int y2, int x3, int y3, int depth, Graphics g)
{
   g.setColor(Color.red);
   setBackground(Color.white);

   if(depth != 0)
   {
      depth--;

   g.drawRect(x1,y1,1,1);
   g.drawRect(x2,y2,1,1);
   g.drawRect(x3,y3,1,1);

   g.drawLine(x1,y1,x2,y2);
   g.drawLine(x2,y2,x3,y3);
   g.drawLine(x3,y3,x1,y1);

   sierpienski(x1,y1,(x1+x2)/2,(y1+y2)/2,(x3+x1)/2,(y3+y1)/2,depth,g);
   sierpienski((x1+x2)/2,(y1+y2)/2,x2,y2,(x2+x3)/2,(y2+y3)/2,depth,g);
   sierpienski((x3+x1)/2,(y3+y1)/2,(x2+x3)/2,(y2+y3)/2,x3,y3,depth,g);
   }
}
```

FIG 1.1: Sierpinski Draw Method

The program begins by asking the user for the depth of the Gasket. Then we are able to start the drawing process of the figure.

We first plot three endpoints(A, B, C) of an equilateral triangle and draw a line to each vertex. Next, we need to find the midpoints of AB, BC, and CD. Using recursion we are able to call the Sierpinski method again using the midpoints that we found. We continue these steps until the depth is equal to zero.

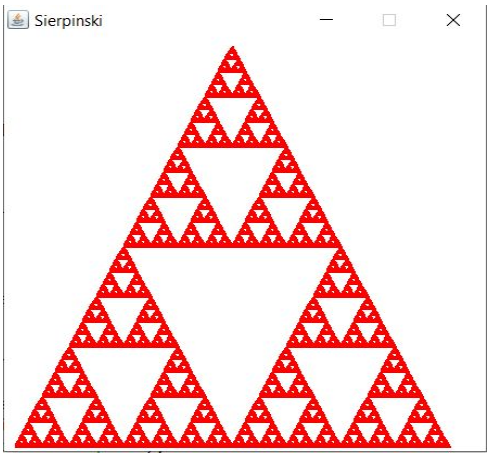


FIG 1.2: Drawing of the Sierpinski Gasket

Program 2: Following Valentines Remarks

In the play of Arcadia Valentine states that "You never know where to expect the next dot. But gradually you start to see this shape.....The unpredictable and the predetermined unfold together to make everything the way it is."

In this program, our objective is to use the idea from Valentine to show an unpredictable outcome. We have to implement the following iterative algorithm:

1. Hardwire into your program three points of an equilateral triangle (x_1,y_1), (x_2,y_2) (x_3,y_3). These can be screen coordinates. Remember (0,0) is at the top left corner.
2. Pick one of the three vertices at random, call it (x,y)
3. Then we have to repeat the following until 10,000 points have been drawn:
    a. Pick a vertex, (x_1,y_1), (x_2,y_2) or (x_3,y_3) at random. Call it v.
    b. Place (draw a pixel) a point p exactly halfway between (x,y) and point v.
    c. Set (x, y) equal to point p.

In FIG 2.1 shows us creating the three points of an equilateral triangle.

```
public Valentine_Shape()
{
   a = new Point(200,0);
   b = new Point(0,400);
   c = new Point(400,400);
}
```

FIG 2.1

We then have to pick one of the vertices at random which is demonstrated by the switch statement. The reasoning for this step is to include all random possibilities. From the choosing of one of the random vertices, we are able to then continue with the program.

```
int  a = rand.nextInt(3);
switch(a)
{
case 0:
{ xy = new Point(x1,y1);  break;}
case 1:
{ xy = new Point(x2,y2);  break;}
case 2:
{ xy = new Point(x3,y3);  break;}
}
```

FIG 2.2: Picking a random point for xy

```
for(int i = 0; i <= 1000000; i++)
{
int  randomnum = rand.nextInt(3);
switch(randomnum)
{
case 0:
{v = new Point(x1,y1); break;}
case 1:
{v = new Point(x2,y2); break;}
case 2:
{v = new Point(x3,y3); break;}
}
 Point p = Point.getMidpoint(xy,v);

g.drawLine(p.x(),p.y(),p.x(),p.y());
xy = p;
}
```

FIG 2.3

As of now, there is a randomly chosen point xy and we have to find a random vertex to start off the program. Once we choose a random vertex we need to find the midpoint between points xy and v(which is one of the vertices). Likewise from problem 1 using the midpoint formula we then draw a point where the midpoint is and then save the midpoint to xy. Now as the program runs xy will always be changing because we are using the answer from the midpoint of xy and v as the next xy. The vertex is randomly chosen from the switch statement.

From Valentine's idea we see that the outcome of this idea is unpredictable in theory, but once we plot each point that is calculated we see the bigger picture.

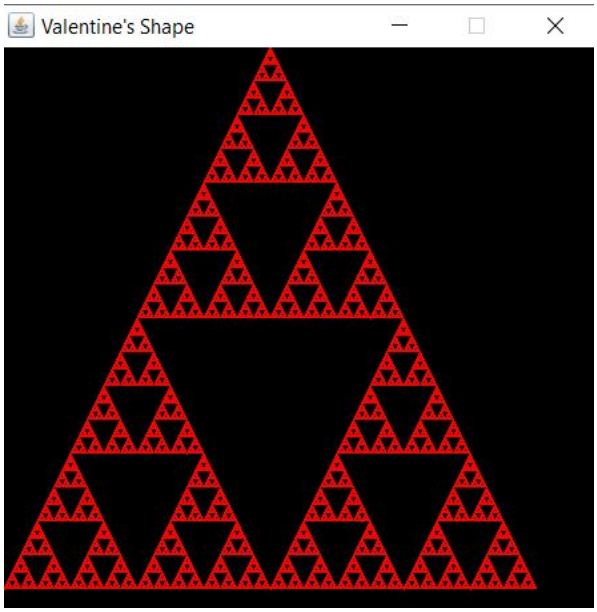


FIG 2.4: The Drawing from Valentines Theory

Program 3: Drawing Sierpinski's Carpet

The Sierpinski's Carpet is a plane fractal that uses a similar technique of Sierpinski's Gasket to program. There are two conditions that change, one being the carpet begins with a square with four different vertices and two, the midpoint that we find is ⅓ the distance from point xy.

Objective:

1. Hardwire the coordinates of the four corners of a square into a graphics program.
2. Find the midpoints of the four sides of the square
3. Use a random number generator to select one of these eight points, call it (x,y)
4. Use a random number to select one of the four vertices or one of the four midpoints, call it v.
5. Draw a point p between (x,y) and v such that the distance from p to v is 1/3 the distance from (x,y) to v.
6. Call the new point (x,y)
7. repeat 4-6 indefinitely

```
for(long i = 0; i <= 10000000; i++)
{
int  randomnum = rand.nextInt(8);
switch(randomnum)
{
case 0:
{v = new Point(x1,y1); break;}
case 1:
{v = new Point(x2,y2); break;}
case 2:
{v = new Point(x3,y3); break;}
case 3:
{v = new Point(x4,y4); break;}
case 4:
{ v = new Point(mp1.x,mp1.y);  break;}
case 5:
{ v = new Point(mp2.x,mp2.y);  break;}
case 6:
{ v = new Point(mp3.x,mp3.y);  break;}
case 7:
{ v = new Point(mp4.x,mp4.y);  break;}
}
 Point p = Point.getTripoint(xy,v);

g.drawLine(p.x(),p.y(),p.x(),p.y());
xy = p;
```

FIG 3.1: The method we use to draw the Sierpinski Carpet

First, the program creates four points that represent the four corners of a square(A, B, C, D). Then we find the midpoints between each set i.e. The midpoint from A to B, B to C, C to D, and finally D to A. Likewise in Program 2, the program picks a starting location for point xy from one of the vertices. From there the program starts a loop to run many times. Inside the loop, we need to create a random number and have a switch statement that will save point v as one of the eight total points, as seen in FIG 3.1.

After getting the values for points xy and v finding the midpoint that is ⅓ the distance from xy is where some mathematics shows up.

```
public static Point getTripoint(Point a, Point b)
	{
	int ax = a.x();
	int ay = a.y();
	int bx = b.x();
	int by = b.y();
	double  px = ((bx-ax)/3);
	double py = ((by-ay)/3);
	int px1 = (int)px;
	int py1 = (int)py;
	int px2 = bx - px1;
	int py2 = by - py1;

	Point x = new Point(px2, py2);
		return x;
	 }
```

FIG 3.2: The Midpoint method for Sierpinski's Carpet

With this method, we are able to get a midpoint that is ⅓ away from the distance from point xy. The way this method works is by passing in two different points which in our case is xy and v. Then using the Point class the program is able to break each point into its x and y components. Subtracting each of the x components from the two different points then dividing by 3 and vice versa with the y components. The point needs to be ⅓ the distance from xy to v. So, we are able to subtract the value found for each x and y by the x and y components of v. See FIG 3.2. Then using those values we create a new point and return it.

Once we return it the program draws a point and save point p to xy. So for each pass through the loop xy is the answer from the previous loop. Each time the program loops through it picks a new vertex and uses the xy from the previous until the loop is done. Eventually, the output will look like FIG 3.3.

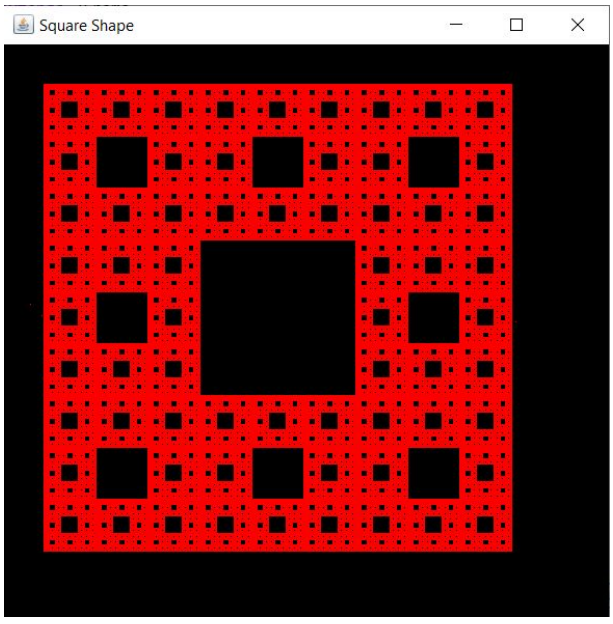


FIG 3.3: Sierpinski Carpet

Program 4: Drawing Our Own Fractal

Objective: Design and implement an algorithm, similar to the algorithms of either program 1,2, or 3. The output should be some self-similar geometric figure.

Our group has already demonstrated an algorithm for a triangle and a square. We wanted to see if a pentagon would work similarly.

```
public UnpredictableImaginationPanel()
{
   a = new Point(210,10);
   b = new Point(20,160);
   c = new Point(90,380);
   d = new Point(340,380);
   e = new Point(400,160);

}
```

FIG4.1: Assigning the vertices of the pentagon

These are the points that we chose for the pentagon. We use the algorithm from Sierpinski's carpet, but alternated point v, which is one of the vertices of the pentagon.

```
for(int i = 0; i < 1000000; i++)
{
   num1 = r.nextInt(5)+1;
   switch(num1)
   {
      case 1:
            v = a;
            break;
      case 2:
            v = b;
            break;
      case 3:
            v = c;
            break;
      case 4:
            v = d;
            break;
      case 5:
            v = e;
            break;
   }
   Point p = Point.getTripoint(xy,v);
   xy = p;
   g.drawRect(p.x,p.y,1,1);
}
```

FIG 4.2: Algorithm For Drawing the Pentagon

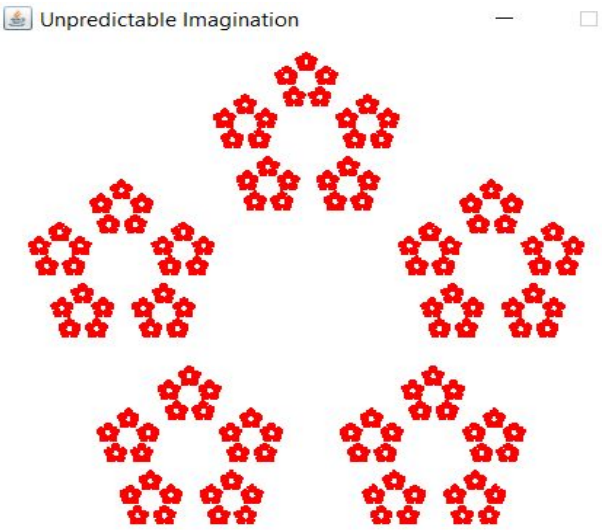


FIG 4.3: Sierpinski's Pentagon

In our program, we tried using some different algorithms for creating a self-similar geometric figure. First, we tried using the midpoint formula to create the new point p, but it was a scattered mess(FIG 4.4). So we then tried to use the Tripoint method from the Sierpinski's Carpet algorithm. Which gave us something that looked relatively close to a Sierpinski figure.

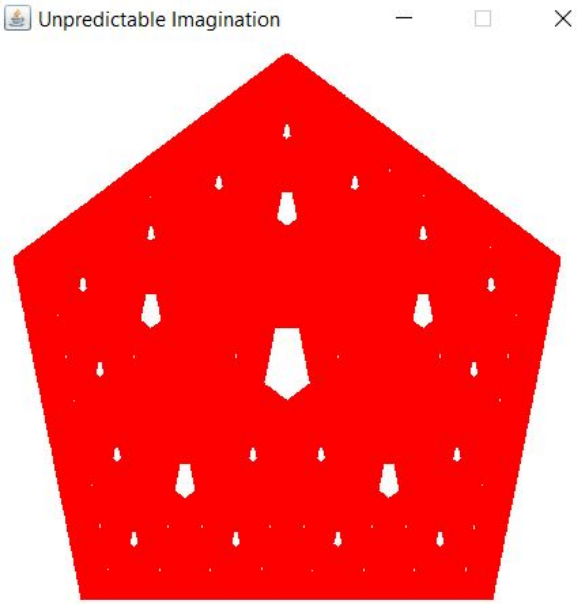


FIG4.4: Our First Trial With the Midpoint Formula

Problems 5-6: The Julia Set

Objective: Create fractals by running complex numbers through a f(x) = x^2 + c algorithm and setting x to the previous value for f(x). c is constant.

```
public Complex f( Complex z)
{
   Complex a = new Complex(.2,.6);
   return (z.mul(z)).add(a);
}
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Complex c;
  int xPixel = 0;
  int yPixel = 0;
  for (double x = lower_x; x < upper_x; x += .005)
   {
    xPixel++;
    yPixel=0;
    for (double y = lower_y; y < upper_y; y += .005)
    {
      yPixel++;
       int i;
       c = new Complex(x,y);
       for (i = 0;i<max;i++)
       {
         c = f(c);
         if (c.abs()>2)
          break;
        }
       int clr = max - i;
       float  red = (clr * 23  %   256)/256.0f;  // the f makes 256.0 a float, rather than a double

       float green = (clr * 6   %   256)/256.0f;

       float blue = (clr * 13   %   256)/256.0f;

       Color color = new Color(red, green, blue); // parameters are float

        g.setColor(color);
       g.drawRect(xPixel,yPixel,1,1);
    }
```

FIG 5.1: Algorithm for Julia Set

The fractal images created from this program are very intricate since each pixel found on the screen has been run through the algorithm to determine its color. Wherever there is a color change that means that the pixel has escaped the prison set and becomes a part of the escape set at a different rate than the pixels around it. By changing around the parameters for c, we find that the images are in different shapes but the same colors.

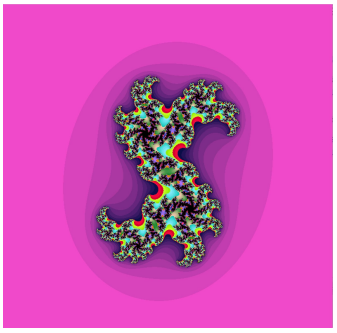
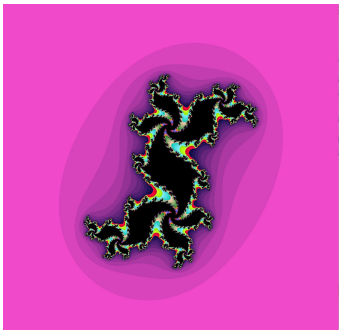


FIG 5.2

FIG 5.3



In order to accomplish this, we use embedded for loops to shift from pixel to pixel and run each pixel through the method f(Complex z). This method then takes the value given for z and returns z^2 + c (c being a constant Complex number). The resulting f(z) value is set to z and fed back into the equation for 50 iterations (max) or until the absolute value of z is greater than 2 (in the escape group). We count how many iterations the point takes to escape, which determines the color. When the pixel escapes so fast from the prison set that it only takes one iteration, we see the color set to black.

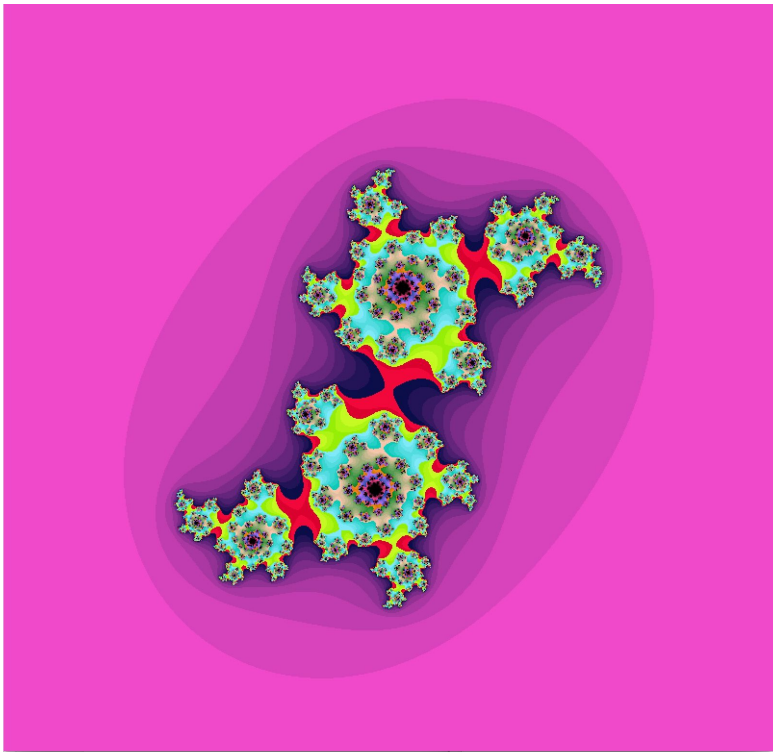


FIG 5.4

Problem 7: The Mandelbrot Set

Objective: Create fractals like the Julia Set (Mandelbrot Set is technically a specific type of Julia Set), but instead of keeping c as a constant Complex number, c will vary for each pixel.

```
public Complex f( Complex z, Complex a)
{
  return (z.mul(z)).add(a);
}
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Complex a;
  int xPixel = 0;
  int yPixel = 0;
  for (double x = lower_x; x < upper_x; x += .005)
   {
    xPixel++;
    yPixel=0;
    for (double y = lower_y; y < upper_y; y += .005)
    {
      yPixel++;
       int i;
       Complex c = new Complex(0,0);
       a = new Complex(x,y);

       for (i = 0;i<max;i++)
       {
         c = f(c,a);
         if (c.abs()>2)
          break;
        }
       int clr = max - i;
       float  red = (clr * 23  %   256)/256.0f;  // the f makes 256.0 a float, rather than a double

       float green = (clr * 6   %   256)/256.0f;

       float blue = (clr * 13   %   256)/256.0f;

       Color color = new Color(red, green, blue); // parameters are float

        g.setColor(color);
       g.drawRect(xPixel,yPixel,1,1);
```

FIG 7.1

Very similar code to the Julia set, but now the c value varies from pixel to pixel rather than always being .2 + .6i or some similar value. Likewise, the z value which we set to (x,y) in the Julia Set is now always beginning at (0,0) and then being fed back into itself to grow.

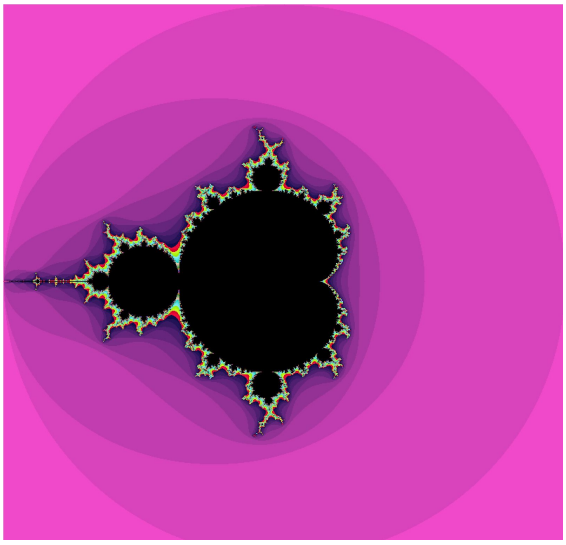


FIG 7.2:

Mandelbrot Set

Problem 8: Growth Rate Population

Objective: Use the Logistic Equation (y = rx(1-x)) to show how small changes to the x value can result in a huge change in whether the population becomes stable, periodic, or chaotic.

```
public class GrowthRate
{
  public static void growth(double r)
  {
    System.out.println("Growth Rate " + r);
    double x = .5;
    //double z = .6;
    for (int i=0;i<20;i++)
    {
      double y = (r*x) - (r*(x*x));
      //double q = (r*z) - (r*(z*z));
      System.out.println((i+1)+" "+y);
      x = y;
      //z = q;
    }
  }
  public static void main(String[]args)
  {
    for (double r=.5;r<=5;r+=.01)
    {
      growth(r);
    }
  }
}
```

FIG 8.1: Growth Rate Method

Stable

```
Growth Rate 1.5000000000
1 0.37500000000000006
2 0.3515625000000001
3 0.3419494628906251
4 0.3375300415791572
5 0.33540526891609446
6 0.33436286174912533
7 0.333846507648091
8 0.33358952546889625
9 0.33346133094949937
10 0.3333973075663318
11 0.33336531431077887
12 0.33334932228788183
13 0.3333413274271377
14 0.3333373302843773
15 0.33333533178489194
16 0.33333433255312206
17 0.33333383294173
18 0.33333358313715733
19 0.3333334582351518
20 0.33333339578421917
```

FIG 8.2

Periodic

```
Growth Rate 3.2000000000
1 0.8000000000000004
2 0.5119999999999996
3 0.7995392000000004
4 0.5128840565227515
5 0.7994688034800597
6 0.5130189943751087
7 0.7994576185134753
8 0.5130404310855616
9 0.799455830902729
10 0.51304385708274
11 0.7994555449356965
12 0.5130444051432486
13 0.7994554991822685
14 0.513044492830395
15 0.7994554918617534
16 0.5130445068602736
17 0.7994554906904716
18 0.5130445091050531
19 0.7994554905030667
20 0.5130445094642169
```

FIG 8.3

Chaotic

```
Growth Rate 3.9000000000
1 0.9750000000000005
2 0.09506249999999827
3 0.33549992226561975
4 0.8694649252589937
5 0.4426331091131286
6 0.9621652553368983
7 0.1419727793615806
8 0.4750843861995216
9 0.9725789275368876
10 0.10400971326753217
11 0.363447601972798
12 0.9022784261127808
13 0.34387106474847373
14 0.8799326467511769
15 0.41203961733731687
16 0.9448255872191544
17 0.2033077681250246
18 0.6316975062256791
19 0.9073574907303557
20 0.32783351150888596
```

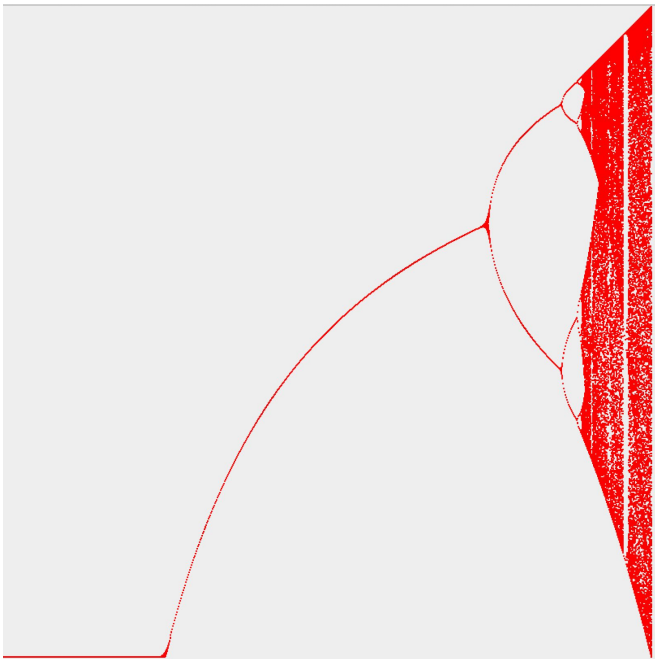
FIG 8.4

To represent the growth of the population, the program takes in an r-value for the growth rate and runs it through the Logistic Equation. The x-value always starts out as .5 and is reset to f(x) each run-through. For each growth rate, we run the program 20 times to see if the population becomes stable, periodic, or chaotic. We then increment the growth rate by .01 and try run the next growth rate through the Logistic Equation 20 times to see how changing the growth rate a tiny amount changes the way the population behaves.

Problem 9: Graphing Growth Rate

Objective: Represent the growth rate using a graph. Use a constant growth rate for each iteration and run it through the Logistic Equation 500 times, only plotting the y-values from the 101st to the 500th run through.

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  for (double r=0;r<=4;r+=.005)
  {
    double x = .5;
    //double z = .6;
    for (int i=0;i<500;i++)
    {
      double y = ((r*x) - (r*(x*x)));
      //double w = ((r*z) - (r*(z*z)));
      int intr = (int)(r*200);
      //int intw = 800-(int)(w*800);
      int inty = 800-(int)(y*800);
      if (i>100)
      {
        //g.drawRect(intr,intw,1,1);
        //g.setColor(Color.BLUE);
        g.drawRect(intr,inty,1,1);
        g.setColor(Color.RED);
      }
      x = y;
      //z = w;
    }
  }
```



The resulting image shows how quickly the population turns chaotic. The x-axis represents the growth rate (r) and the y-axis represents f(x) as the population changes. It is easy to see that for a long time, the growth rate is relatively stable and for each r-value, there appears to be only one y-value. However, after the line splits, the populations quickly become chaotic with many y-values for each r-value. Since the program plots points from the 101st iteration to the 500th iteration, there are a possible 400 y-values per r-value. Although at first there only appears to be one point for each r-value (r,y), when the growth rate represents a chaotic population, there appears to be hundreds of y-values for each r-value.

Problem 10: The Butterfly Effect

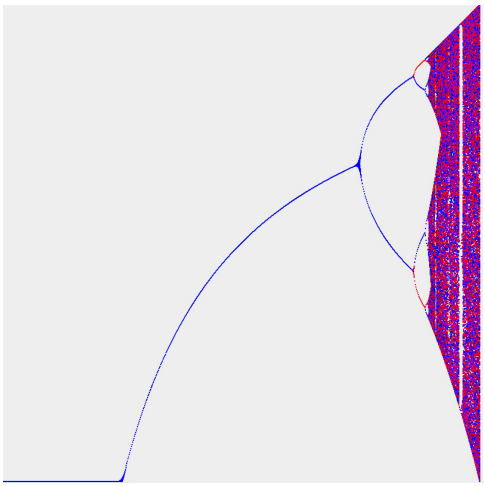
Objective: Represent the major changes that a tiny difference in growth rate can have on a population using the Logistic Equation.

Code **Result**

```
public class GrowthRate
{
  public static void growth(double r)
  {
    System.out.println("Growth Rate " + r);
    double x = .5;
    double z = .6;
    for (int i=0;i<20;i++)
    {
      double y = (r*x) - (r*(x*x));
      double q = (r*z) - (r*(z*z));
      System.out.println((i+1)+" "+y+" "+q);
      x = y;
      z = q;
    }
  }
  public static void main(String[]args)
  {
    for (double r=.5;r<=5;r+=.1)
    {
      growth(r);
    }
  }
}
```

```
Growth Rate 4.000000000000002
1 1.0000000000000004 0.9600000000000002
2 -1.7763568394002505E-15 0.15359999999999951
3 -7.105427357601018E-15 0.5200281599999989
4 -2.8421709430404285E-14 0.9983954912280582
5 -1.1368683772162042E-13 0.006407737294170435
6 -4.547473508865336E-13 0.025466712787757338
7 -1.8189894035469623E-12 0.0992726373101729
8 -7.275957614201088E-12 0.3576703231666232
9 -2.910383045701612E-11 0.9189690523700266
10 -1.1641532183145267E-10 0.2978597326246484
11 -4.65661287380021E-10 0.8365572492216857
12 -1.8626451503874465E-9 0.5469168719853283
13 -7.450580615427577E-9 0.9911952284924497
14 -2.980232268375493E-8 0.03490899002500081
15 -1.1920929428773352E-7 0.13476140976174086
16 -4.768372339943577E-7 0.46640308880307635
17 -1.9073498454724227E-6 0.9954849902321047
18 -7.629413933823427E-6 0.017978497818764705
19 -3.0517888567121615E-5 0.07062108573978154
20 -1.220752796345769E-4 0.26253499195486396
```

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  for (double r=0;r<=4;r+=.005)
  {
    double x = .5;
    double z = .6;
    for (int i=0;i<500;i++)
    {
      double y = ((r*x) - (r*(x*x)));
      double w = ((r*z) - (r*(z*z)));
      int intr = (int)(r*200);
      int intw = 800-(int)(w*800);
      int inty = 800-(int)(y*800);
      if (i>100)
      {
        g.drawRect(intr,intw,1,1);
        g.setColor(Color.BLUE);
        g.drawRect(intr,inty,1,1);
        g.setColor(Color.RED);
      }
      x = y;
      z = w;
    }
  }
```



We represented The Butterfly Effect (the major changes that occur from a seemingly minuscule event) through both a graph and a table. The table shows that once the growth rate is up to the point of chaos, the results vary heavily when we change the value for x by even 0.1. This is also shown in the graph of the original function with the function with +.01 x-value, as you see the red dots and blue dots differ when the function becomes chaotic. The programs are still the exact same, simply run twice. Once with x=.5 and once with x=.6. The difference in the outputs is very tiny until the functions begin to hit chaos, at which point the difference in the outputs is very noticeable, as represented in the graph and table.

Problem 11: Themes of Arcadia

In *Arcadia* Thomasina and Valentine are both performing two similar types of mathematics, but they are both doing them in their own way and for different reasons. Thomasina is set on the idea that everything in existence holds a numerical value, and that objects value can be represented by its own formula. She uses this idea to perform a premature version of fractal geometry to carryout her iterative methods, and generate equations that represent complex objects. Valentine, on the other hand, preferred to use the same iterative method as Thomasina, but instead, he used it for the sole purpose of predicting future generations and seeing trends and patterns in populations. Throughout the play, Thomasina is questioning already known to be true theorems and algorithms, and the type of math she was performing tends to support this theme. Septimus tells her that it is impossible for her to derive equations for different objects in the world around her, but he also tells her that this is not a new idea, but no one has really carried it out. Being a curious person Thomasina aims to prove Septimus wrong and fills her journal with formulas and graphs that represent different objects she tries to find equations for. Now on the opposite side of the storyline, Valentine seems to be a more serious and factual person, that uses his mathematical skills not to spite someone, but to make discoveries. He finds the data that Thomasina uses to be extremely like his work, but he knows that the processes she was working through weren't used to its full potential. He is well aware of the fact that modern technology is the sole reason he can use the data, that she was working out by hand, to figure out equations for interesting concepts like population changes. This awareness supports the idea of how intertwined these two timelines are, and how similar they are in more aspects than just the settings and objects around the characters.

What We Liked About This Lab

Throughout this lab, we were able to see real applications of mathematics, in relation to computer science. From the Sierpinski figures, we see that an unpredictable outcome can clearly result in a visible diagram. These figures rely on specific formulas depending upon the polygon that we used. The Julia set and Mandelbrot set was really interesting, once we got to see the full picture. In these programs, we did some different trials with changing the constant, c. Which resulted in different looking graphics. The changing of the colors was very cool, but hard to understand at first. We saw we had to calculate how many iterations it took for the point to escape. Then with this information, we were able to change the color.

In addition, having Professor Bravaco and Professor Simonson in class while working through the lab question was really helpful. They explained each question in more detail than what was presented to us in the lab online. Also if we were stuck in the middle of writing code for one of the questions they would ask us where we were stuck at and had us explain how we were thinking about the problem.

What We Did Not Like About This Lab

As a group, we felt that the wording of some of the questions was rather difficult to understand. Although after some time we were able to figure out what was expected of us to do for each question. Having Ralph and Shai in class while working on the lab, was a great help to further understand what some of the questions were actually asking us to do. It may not be something that can be changed, but the work environment we perform the majority of the labs in was not ideal for our group work. There was a lot of noise going on with other groups discussing the same issues, and this caused us not to focus as well as we know we can.

What We Learned From This Lab

There were many new topics that were presented to us throughout this lab. It gave us first-hand experience into some simple algorithms that were written recursively and iteratively. Understanding how the Sierpinski's figures create themselves by recursion is really interesting and the images that were developed were not what we expected. We were able to keep everything simple with recursion which allowed us to recall the method as many times as we needed it to perform the designated task. We also learned how the Julia set and Mandelbrot set were created, and what the colors represented in each fractal image. This was the first time we used Java and our skills with graphics to perform a task like this, and it was rather enjoyable to see what we could use our skills to create. Each of these problems had concepts that related to the teachings in the play, and every one of them was new and interesting. The growth rate part of the lab was particularly interesting, because it allowed us to visualize changes in nature, that we originally were not directly able to predict beforehand. Working through the code in these assignments was challenging, but we learned from our mistakes and developed working codes in the end, and gained an understanding of how each of them works. When something didn't work we were able to decompress the method and understand how each little part affected the entire code. This was the key to making the programs do what we wanted to do, and it helped us develop skills in checking over each part of our programs.