

**MAT 128B Project 1:**  
**Using Iteration Methods to Understand Fractal Geometry**

Melissa Fiacco & Carlos Palomo

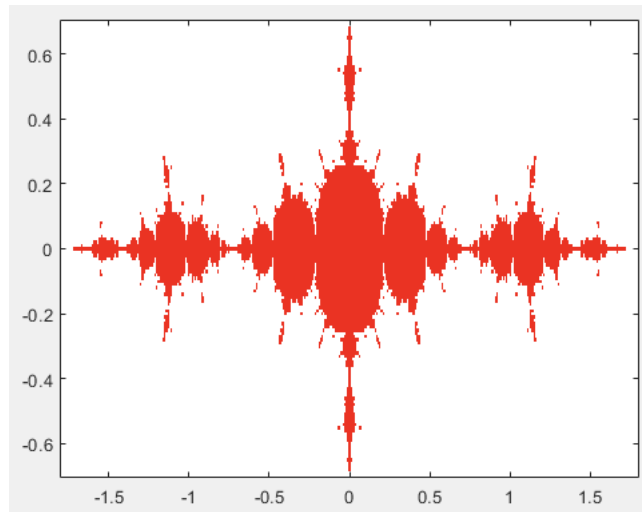
Date: 19 February 2018

## Introduction

In this project we are going to implement a series of computer programs that use iteration methods to generate different figures on the plane. We will be focusing on the Filled Julia set, Julia set, and the Mandelbrot Set: The Julia set is a set of complex numbers that do not converge to any limit when a given mapping is repeatedly applied to them, and the Mandelbrot Set is a particular set of complex numbers that has a highly convoluted fractal boundary when plotted representing convergence of the Julia set. Both sets produce interesting fractal geometry in the complex plane, which we will be able to generate using our iteration methods.

### Part I: An introduction to Fractals

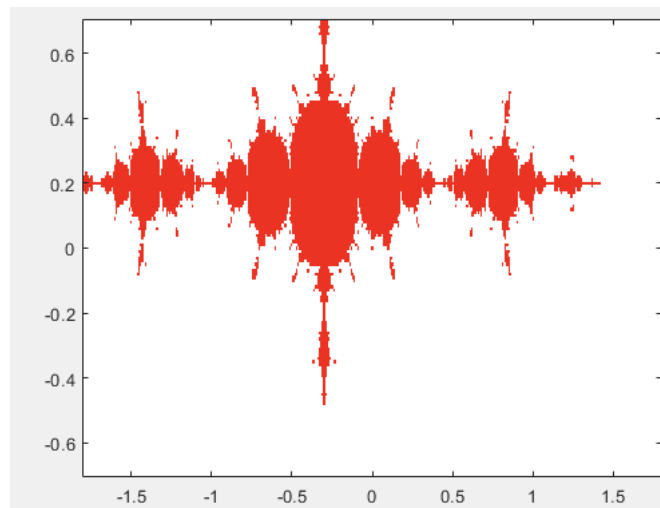
$\phi(z) = z^2$  can be transformed into a fixed point problem to predict its behavior based on the value  $z$ . If  $|z_0| \leq 1$ , the orbit remains bounded and is referred to as the Filled Julia set. We will not find a singular fixed point, but the Julia set will find a sequence of points mapped closely to themselves (never repeating) infinitely in an orbit contained within unit circle. Note  $z_0 \leq e^i = \cos(1) + i\sin(1)$  which describes the Filled unit circle in the complex plane. This process is what creates the fractal- a pattern occurs as the orbit is created. Implementing the program on page 100, `Julia.m`, we produce the following figure, replicating figure 4.13 in the book:



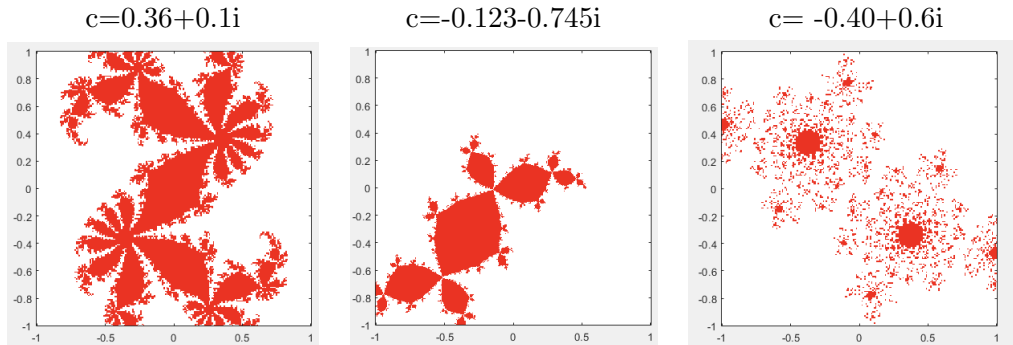
**Part II: Generate (and plot) other examples changing the value of  $c$  in the function**

We can generate different fractal patterns when we plot  $\phi(z) = z^2 + c$  using various  $c$  values (see below for various examples). However, we must be careful to restrict  $|z_0| \leq 2$ ; If we do not make this restriction, the orbit is unbounded and  $z_0$  is not contained in the Julia set. This is due to  $z_0$  existing outside of the unit circle and  $\phi(z) = z^2 + c$  growing outside of the unit circle, spiraling out; The computer program will reject our value, and end the program.

If we change our  $z_0$  values, we shift where the fractal is centered. It is possible to choose an initial value that will cause the program to fail, so we must be careful. As an example, we took our code from Part I, and changed  $z_0$  from  $-0.8$  and  $-1.8$  to  $-0.9$  and  $-1.5$ , respectively, and the resulting fractal is now slightly shifted up and to the left, which reflects chosen  $z_0$ :

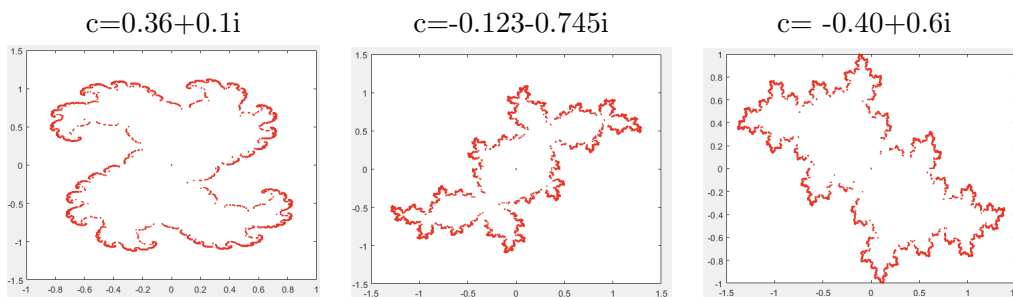


Examples of various  $c$  values and their graphs:



### Part III: Constructing the Julia Set

In Part II we constructed the Filled Julia set. In this part, we have constructed the Julia set, the boundary of the Filled Julia set for  $\Psi = \pm\sqrt{w - c}$ . The Inverse Iterative method utilizes multiple properties of complex numbers by splitting  $w$  and  $c$  into real and imaginary parts such that  $z = w - c$ , and then finds  $\sqrt{z}$ .



### Part IV: Computing the Fractal Dimension

The fractal dimension  $D$  is the complexity of a fractal: it measures the roughness of an object. The higher the fractal dimension, the more complex, and more rapidly the object changes as it is scaled. For example, if we split a square into 4 squares, there will be  $N=4$  squares similar to  $A$ , and each side is  $\frac{1}{2}$  its original length, so  $D = \frac{\log 4}{\log 2} = 2$ . With fractals, we cannot calculate this as easily, thus we must use algorithms to estimate the dimension. We created two algorithms based off the paper [On calculation of fractal dimension of images](#) by Ajay Kumar Bisoi and Jibitesh Mishra. The first method is the Reticular Cell Counting method and the second is Differential Box-Counting method. Using these two algorithms we attempted to find the fractal dimension for plot found in Part I. The Reticular Cell

Counting method found a fractal dimension of 2 and the Differential Box-Counting method found a fractal dimension of 1. Applying the algorithms to the filled unit circle we found a fractal dimension of 1

## Part V: Connectivity of the Julia Set

We created a program which computes  $\text{orb}(0)$  to determine connectivity of the Julia set. We say divergence occurs if  $|z| > 100$ , i.e.  $\text{orb}(0)$  is unbounded, thus the Julia set is not connected. After 1500 iterations, if  $|z| < 100$ , but our function does not converge to fixed point, we assume  $\text{orb}(0)$  is bounded, and Filled Julia set is connected. We tested our function with various  $c$  values we used in part II. Based on the figures created in part II, we expected  $c = 0.36 + 0.1i$  and  $c = -0.123 - 0.745i$  to be connected and  $c = -0.4 + 0.6i$  not connected, which is what our program gave us.

```
>> OrbZero
Computing orb(0) to determine connnectivity of Filled Julia Set with

c =

    0.3600 + 0.1000i

Orb(0) is still bounded after 1500 iterations, assume connnectivity.
>> OrbZero
Computing orb(0) to determine connnectivity of Filled Julia Set with

c =

   -0.1230 - 0.7450i

Orb(0) is still bounded after 1500 iterations, assume connnectivity.
>> OrbZero
Computing orb(0) to determine connnectivity of Filled Julia Set with

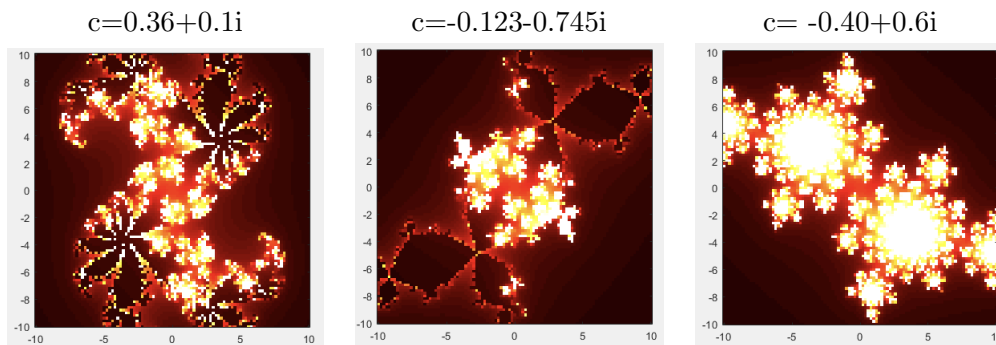
c =

   -0.4000 + 0.6000i

Orb(0) is unbounded, Filled Julia set is disconnected.
```

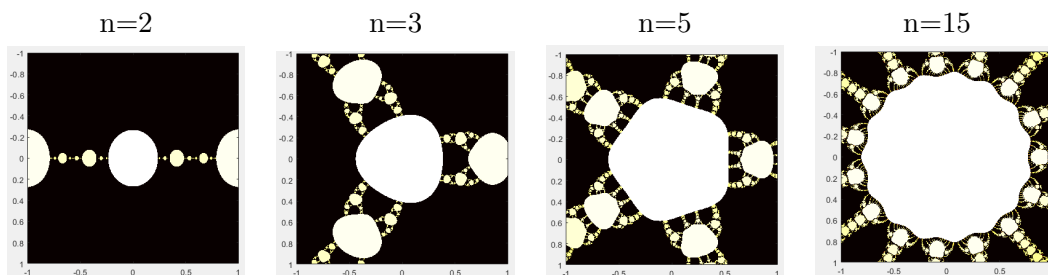
## Part VI: Coloring Divergent Sets

We can extend part V to create a color coded figure that tells us how long it takes for an orbit to diverge given a  $c$  value. Our color map is darker the higher number of iterations it takes to diverge for a range of  $z$  values, given  $c$ . We tested our  $c$  values from part II, and received the following images. Note, since we diverge when  $|z| > 100$ , we extended the domain and range to include all values within  $|z| < 100$ .



## Part VII: Newtons Method in the Complex Plane

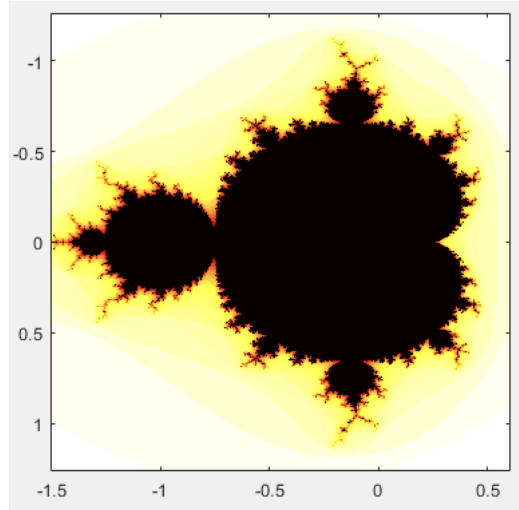
If we apply similar methods from the coloring orbits program, we can find the fixed points of a complex function using Newtons Method. We used the same colormap, with black meaning divergence after no iterations or small amount of iterations, and white implying convergence. The fixed points exist on the boundaries of the two colors. We tested our code for the function  $z^n + 1$  for various  $n$ .



## Part VIII: The Mandelbrot Set

For  $\phi(z) = z^2 + c$ , the Mandelbrot set shows where the Julia set is connected for various  $c$  values (the black region), and how long it takes for  $\phi(z) = z^2 + c$  with  $z_0 = 0$  to diverge. The lighter the color (away from black), the less iterations required to determine

divergence. As we get closer to the black region, the darker the color is, and the longer it takes to determine divergence. The black region is where the color map is equivalently 0, meaning no divergence occurs



## Reference



















Greenbaum, Anne; Chartier, Timothy P.. Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms (p. 98). Princeton University Press. Kindle Edition.

## Commit History











 [crpalomo](#) / [MAT128](#)

Branch: **Project\_1** ▾

Commits on Feb 20, 2018

<b>Part 4 Method 2- doesn't work</b> ...  mfiacco committed an hour ago	Verified		<a href="#">45777c5</a>	<a href="#">↔</a>
<b>Merge branch 'Project_1' of https://github.com/crpalomo/MAT128 into P...</b> ...  crpalomo committed 2 hours ago			<a href="#">3685b1e</a>	<a href="#">↔</a>
<b>Part4 one method</b> ...  crpalomo committed 2 hours ago			<a href="#">8db3389</a>	<a href="#">↔</a>
<b>Part 7-Newton's</b> ...  mfiacco committed 7 hours ago	Verified		<a href="#">3524920</a>	<a href="#">↔</a>
<b>Merge branch 'Project_1' of https://github.com/crpalomo/MAT128 into P...</b> ...  crpalomo committed 7 hours ago			<a href="#">8a02632</a>	<a href="#">↔</a>
<b>Update Latex and Upload 7</b> ...  crpalomo committed 7 hours ago			<a href="#">00bd35e</a>	<a href="#">↔</a>
<b>Part 3- Melissa's Attempt</b> ...  mfiacco committed 9 hours ago	Verified		<a href="#">0b45cb8</a>	<a href="#">↔</a>
<b>Latex File and Need File</b> ...  crpalomo committed 14 hours ago			<a href="#">4afb62a</a>	<a href="#">↔</a>
<b>Uploading Part 3</b> ...  crpalomo committed 22 hours ago			<a href="#">29da182</a>	<a href="#">↔</a>

Commits on Feb 18, 2018

<b>Part 6- Not sure if correct</b> ...  mfiacco committed 2 days ago	Verified		<a href="#">9921258</a>	<a href="#">↔</a>
<b>Debugged Orb</b>  mfiacco committed 2 days ago	Verified		<a href="#">3d47bb2</a>	<a href="#">↔</a>
<b>Fixed End of Code</b> ...  mfiacco committed 2 days ago	Verified		<a href="#">cf986be</a>	<a href="#">↔</a>
<b>Calculating Orb(0)- PART V</b> ...  mfiacco committed 2 days ago	Verified		<a href="#">ddf89ec</a>	<a href="#">↔</a>
<b>Pushing from Fork</b> ...  mfiacco committed 2 days ago	Verified		<a href="#">06b24e5</a>	<a href="#">↔</a>

Commits on Feb 14, 2018

<b>Fixed Graphing problem</b> ...  crpalomo committed 6 days ago			<a href="#">2f49769</a>	<a href="#">↔</a>
--	--	---	-------------------------	-------------------

Commits on Feb 12, 2018

<b>Update 1</b> ...  crpalomo committed 8 days ago			<a href="#">56a1b75</a>	<a href="#">↔</a>
--	--	---	-------------------------	-------------------



## MatLab Codes

### Part I Code

```

1 - phi = inline('z^2');
2 - fixpt1 = (1 + sqrt(6))/2;
3 - fixpt2 = (1 - sqrt(6))/2;
4 - colormap([1 0 0; 1 1 1]);
5 - M = 2*ones(141,361);
6 - for j=1:141
7 -     y = -0.7 + (j-1)*0.01;
8 -     for i=1:361
9 -         x = -1.8 + (i-1)*0.01;
10 -        z = x + 1i*y;
11 -        zk = z;
12 -        iflag1 = 0;
13 -        iflag2 = 0;
14 -        kcount = 0;
15 -        while kcount < 100 && abs(zk) < 2 && iflag1 < 5 && iflag2 < 5
16 -            kcount = kcount + 1;
17 -            zk = phi(zk);
18 -
19 -            err1 = abs(zk-fixpt1);
20 -            if err1 < 1.e-6
21 -                iflag1 = iflag1 + 1;
22 -            else
23 -                iflag1 = 0;
24 -            end
25 -
26 -            err2 = abs(zk - fixpt2);
27 -            if err2 < 1.e-6
28 -                iflag2 = iflag2 + 1;
29 -            else
30 -                iflag2 = 0;
31 -            end
32 -        end
33 -        if iflag1 >= 5 || iflag2 >= 5 || kcount >= 100
34 -            M(j,i) = 1;
35 -        end
36 -    end
37 - end
38 - image([-1 1],[-1 1],M),
39 - axis xy

```

### Part II Graph 1 Code

```

1 - phi = inline('z^2 - 1.25'); % Define the function whose fixed points we seek.
2 - fixpt1 = (1 + sqrt(6))/2; % These are the fixed points.
3 - fixpt2 = (1 - sqrt(6))/2;
4
5 - colormap([1 0 0; 1 1 1]); % Points numbered 1 (inside) will be colored red;
6 - % those numbered 2 (outside) will be colored white.
7 - M = 2*ones(141,361); % Initialize array of point colors to 2 (white).
8
9 - for j=1:141
10 -    y = -.9 + (j-1)*.01; % Try initial values with imaginary parts between
11 -    % -0.7 and 0.7
12 -    for i=1:361
13 -        x = -1.5 + (i-1)*.01; % and with real parts between
14 -        % -1.8 and 1.8.
15 -        z = x + 1i*y; % 1i is the MATLAB symbol for sqrt(-1).
16 -        zk = z;
17 -        iflag1 = 0; % iflag1 and iflag2 count the number of iterations
18 -        iflag2 = 0; % when a root is within 1.e-6 of a fixed point;
19 -        kount = 0; % kount is the total number of iterations.
20 -
21 -        while kount < 100 && abs(zk) < 2 && iflag1 < 5 && iflag2 < 5
22 -            kount = kount+1;
23 -            zk = phi(zk); % This is the fixed point iteration.
24 -            err1 = abs(zk-fixpt1); % Test for convergence to fixpt1.
25 -            if err1 < 1.e-6
26 -                iflag1 = iflag1 + 1;
27 -            else
28 -                iflag1 = 0;
29 -            end
30 -            err2 = abs(zk-fixpt2); % Test for convergence to fixpt2.
31 -            if err2 < 1.e-6
32 -                iflag2 = iflag2 + 1;
33 -            else
34 -                iflag2 = 0;
35 -            end
36 -        end
37 -        if iflag1 >= 5 || iflag2 >= 5 || kount >= 100 % If orbit is bounded, set this
38 -            M(j,i) = 1; % point color to 1 (red).
39 -        end
40 -    end
41 - end
42 - image([-1.8 1.8],[-.7 .7],M), % This plots the results.
43 - axis xy % If you don't do this, vertical axis is inverted.

```

## Part II Graph 2 Code

```

1  %%Part 2; C1
2  phi= inline('z^2+0.36 + 0.1i'); %Define the function whose fixed points we seek.
3  fixpt1 = (1+sqrt(1-(144-40i)/100))/2; %These are the fixed points.
4  fixpt2 = (1-sqrt(1-(144-40i)/100))/2;
5
6  colormap([1 0 0; 1 1 1]); %Points numbered 1 (inside) will be colored red;
7  %Those numbered 2 (outside) will be white.
8  M = 2*ones(201,201); %Initialize array of point colors to 2 (white).
9
10 for j=1:201 %Try initial values with imaginary parts between
11     y = -1 + (j-1)*.01; % -.7 and .7
12     for i=1:201 %and with real parts between
13         x = -1 + (i-1)*.01; % -1.8 and 1.8.
14         z = x + 1i*y; % 1i is the Matlab symbol for sqrt(-1).
15         zk = z;
16         iflag1 = 0; %iflag1 and iflag2 count the number of iterations
17         iflag2 = 0; % when a root is within 1.e-6 of a fixed point;
18         kount = 0; % kount i < 5 && iflag2 < 5
19
20         while kount < 100 && abs(zk) < 2 && iflag1 < 5 && iflag2 < 5
21             kount = kount+1;
22             zk = phi(zk); % This is the fixed point iteration.
23
24             err1 = abs(zk-fixpt1); %Test for convergence to fixpt1.
25             if err1 < 1.e-6, iflag1 = iflag1 +1; else, iflag1 = 0; end;
26
27             err2 = abs(zk-fixpt2); %Test for convergence to fixpt2.
28             if err2 < 1.e-6, iflag2 =iflag2+1; else, iflag2 = 0; end;
29
30         end;
31         if iflag1 >= 5 || iflag2 >= 5 || kount >= 100, %If orbit is bounded,
32             M(j,i) = 1; % point color to 1 (red)
33         end;
34     end;
35 end;
36
37 image([-1 1],[-1 1],M), %This plots the results.
38 pbaspect ([1 1 1]);
39 axis xy %otherwise the vertical axis is inverted.
40

```

## Part II Graph 3 Code

```

1  phi= inline('z^2-0.123-0.745i'); %Define the function whose fixed points we seek.
2  fixpt1 = (1+sqrt(.492+2.98i))/2; %These are the fixed points.
3  fixpt2 = (1-sqrt(.492+2.980i))/2;
4
5  colormap([1 0 0; 1 1 1]); %Points numbered 1 (inside) will be colored red;
6  %Those numbered 2 (outside) will be white.
7  M = 2*ones(141,361); % Initialize array of point colors to 2 (white).
8
9  for j=1:141 % Try initial values with imaginary parts between
10     y = -.9 + (j-1)*.01; % -.7 and 0.7
11     for i=1:361 % and with real parts between
12         x = -1.5 + (i-1)*.01; % 1i is the Matlab symbol for sqrt(-1).
13         z = x + 1i*y;
14         zk = z;
15         iflag1 = 0; %iflag1 and iflag2 count the number of iterations
16         iflag2 = 0; % when a root is within 1.e-6 of a fixed point;
17         kount = 0; % kount i < 5 && iflag2 < 5
18
19         while kount < 100 && abs(zk) < 2 && iflag1 < 5 && iflag2 < 5
20             kount = kount+1;
21             zk = phi(zk); % This is the fixed point iteration.
22
23             err1 = abs(zk-fixpt1); %Test for convergence to fixpt1.
24             if err1 < 1.e-6, iflag1 = iflag1 +1; else, iflag1 = 0; end;
25
26             err2 = abs(zk-fixpt2); %Test for convergence to fixpt2.
27             if err2 < 1.e-6, iflag2 =iflag2+1; else, iflag2 = 0; end;
28
29         end;
30         if iflag1 >= 5 || iflag2 >= 5 || kount >= 100, %If orbit is bounded,
31             M(j,i) = 1; % point color to 1 (red)
32         end;
33     end;
34 end;
35
36 image([-1 1],[-1 1],M), %This plots the results.
37 pbaspect ([1 1 1]);
38 axis xy %otherwise the vertical axis is inverted.
39

```

## Part II Graph 4 Code

```

1 - phi= inline('z^2-4+.6i'); %Define the function whose fixed points we seek.
2 - fixpt1 = (1+sqrt(1-(-1.6+2.4i)))/2; %These are the fixed points.
3 - fixpt2 = (1-sqrt(1-(-1.6+2.4i)))/2;
4
5 - colormap([1 0 0; 1 1 1]); %Points numbered 1 (inside) will be colored red;
6 - %Those numbered 2 (outside) will be white.
7 - M = 2*ones(201,201); %Initilize array of point colors to 2 (white).
8
9 - for j=1:201; %Try initial values with imaginary parts between
10 - y = -1 + (j-1)*.01; % -.7 and .7
11 - for i=1:201; %and with real parts between
12 - x = -1 + (i-1)*.01; % -1.8 and 1.8.
13 - z = x + 1i*y; % 1i is the Matlab symbol for sqrt(-1).
14 - zk = z;
15 - iflag1 = 0; %iflag1 and iflag2 count the number of iterations
16 - iflag2 = 0; % when a root is within 1.e-6 of a fixed point;
17 - kount = 0; % kount i < 5 && iflag2 < 5
18
19 - while kount < 100 && abs(zk) < 2 && iflag1 < 5 && iflag2 < 5
20 - kount = kount+1;
21 - zk = phi(zk); % This is the fixed point iteration.
22
23 - err1 = abs(zk-fixpt1); %Test for convergence to fixpt1.
24 - if err1 < 1.e-6, iflag1 = iflag1 +1; else, iflag1 = 0; end;
25
26 - err2 = abs(zk-fixpt2); %Test for convergence to fixpt2.
27 - if err2 < 1.e-6, iflag2 =iflag2+1; else, iflag2 = 0; end;
28
29 - end;
30 - if iflag1 >= 5 || iflag2 >= 5 || kount >= 100; %If orbit is bounded,
31 - M(j,i) = 1; % point color to 1 (red)
32 - end;
33 - end;
34 - end;
35
36
37 - image([-1 1],[-1 1],M); %This plots the results.
38 - pbaspect ([1 1 1]);
39 - axis xy %Otherwise the vertical axis is inverted.

```

## Part III Code

```

1 - c = .36 + .1i;
2
3 - zkr = zeros(361*141, 1);
4 - zki = zeros(361*141, 1);
5
6 - for j = 1:141
7 - y = -.7 + (j-1)*.01;
8 - for i = 1:361
9 - x = -1.3 + (i-1)*.01;
10 - z = x + 1i*y;
11 - zk = z;
12 - kount = 0;
13 - while kount < 100
14 - rzk = real(zk); %Spliting up into Real & Imaginary Parts
15 - rc = real(c); %to calculate r & theta
16 - imzk = imag(zk);
17 - imc = imag(c);
18 - kount = kount + 1;
19 - r = sqrt((rzk-rc)^2 +(imzk-imc)^2); %Finding r using real & im parts
20 - if (rzk-rc) > 0
21 - theta = atan((imzk-imc)/(rzk-rc)); %finding theta using real & im parts
22 - elseif (rzk-rc) < 0
23 - theta = atan((imzk-imc)/(rzk-rc))+pi; %Since we have to keep
24 - % within range of arctan
25 - end
26 - ran = randi(2); %We have to randomly pick if we have pos
27 - if ran == 2 %or negative sqrt value
28 - ran = -1;
29 - end
30 - zk = ran*sqrt(r)*(cos(theta/2) + 1i*sin(theta/2));
31 - end
32 - zkr(i + (j-1)*261) = real(zk); %Stores our real & imag values
33 - zki(i + (j-1)*261) = imag(zk);
34 - end
35 - end
36
37 - plot(zkr, zki,'r.') %Plots red boundary of Julia set, real as x, im as y

```

## Part IV Reticular Cell Counting Code

```

1 - c = imread('FilledUnit.png');
2 - c = rgb2gray(c);
3 - [rows, columns, numberOfColorChannels] = size(image);
4 - if ndims(c)==3
5 -     if size(c,3)==3 && size(c,1)>=8 && size(c,2)>=8
6 -         c = sum(c,3);
7 -     end
8 - end
9 - if length(c)==numel(c)
10 -     dim=1;
11 -     if size(c,1)~=1
12 -         c = c';
13 -     end
14 - end
15 - width = max(size(c));
16 - p = log(width)/log(2);
17 - if p==round(p) || any(size(c)~=width)
18 -     p = ceil(p);
19 -     width = 2^p;
20 -     mz = zeros(width, width);
21 -     mz(1:size(c,1), 1:size(c,2)) = c;
22 -     c = mz;
23 - end
24 - n=zeros(1,round(p+1));
25 - n(round((p+1))) = sum(c(:));
26 - for g=(p-1):-1:0
27 -     siz = 2^(p-g);
28 -     siz2 = round(siz/2);
29 -     for i=1:siz:(width-siz+1)
30 -         for j=1:siz:(width-siz+1)
31 -             c(i,j) = ( c(i,j) || c(i+siz2,j) || c(i,j+siz2) || c(i+siz2,j+siz2) );
32 -         end
33 -     end
34 -     n(g+1) = sum(sum(c(1:siz:(width-siz+1),1:siz:(width-siz+1))));
35 - end
36 - n = n(end:-1:1);
37 - r = 2.^(0:p);
38 - loglog(r,n);
39 - xlabel('r, box size'); ylabel('n(r), number of boxes');
40 - coefficients = polyfit(x,y, 1);

```

## Part IV Differential Box-Counting Code

```

1
2 % Differential Box counting using image.
3
4 X= imread('FilledUnit.png');
5
6 X = rgb2gray(X);
7 axis square;
8
9 B=size(X,1);
10 temp=B;
11 h=1;
12 N(h)=0;
13 L(h)=0;
14
15 while temp>1 % temp=1 means each small grid is a pixel. There is no need to continue.
16     temp2=0;
17     for i=2:99 % select the value of L
18         if mod(temp,i)==0
19             L(h)=temp/i; temp=L(h);
20             break
21         end
22     end
23
24     for j=1:(B/L(h)) % locate the area of each part of the grid, j is by row, k is by column.
25         for k=1:(B/L(h))
26             area=X([L(h)*(j-1)+1:L(h)],L(h)*(k-1)+1:L(h)]);
27             mn=min(area(1:end)); %minimum gray level
28             mx=max(area(1:end)); %maximum gray level
29             nr=fix(mx/L(h))-fix(mn/L(h))+1; %nr(i,j)=l-k+1
30             temp2=temp2+nr;
31         end
32     end
33     N(h)=temp2; h=h+1;
34 end
35 r=B./L;
36 p=polyfit(log10(r),log10(N),1); %Best fit line
37 Dimension=p(1)

```

## Part V Connectivity of Julia Set Code

```

1 - fprintf('Computing orb(0) to determine connectivity of Filled Julia Set with\n')
2 - z = .1i;
3 - c = -.4+.6i;
4 - %Defining Function and Calculating our Fixed Points
5 - phi = @(z) z^2 + c;
6 - fxpt1 = (1 + sqrt(1-4*c))/2;
7 - fxpt2 = (1 - sqrt(1-4*c))/2;
8 - %Setting our initial values
9 - zk = z;
10 - iflag1 = 0;
11 - iflag2 = 0;
12 - kount = 0;
13 - orbit = ones(1500); %We will let it run for 1500 times. If it still does
14 - %diverge after this many times, we assume connected
15 - while kount < 1500 && abs(zk) <= 100 && iflag1 < 6 && iflag2 < 6
16 -     kount = kount + 1;
17 -     zk = phi(zk);
18 -     orbit(kount) = zk;
19 -     err1 = abs(zk - fxpt1);
20 -     if err1 < 1e-6
21 -         iflag1 = iflag1 + 1;
22 -     else
23 -         iflag1 = 0;
24 -     end
25 -     err2 = abs(zk - fxpt2);
26 -     if err2 < 1e-6
27 -         iflag2 = iflag2 + 1;
28 -     else
29 -         iflag2 = 0;
30 -     end
31 - end
32 -
33 - if kount < 1500
34 -     orbit = orbit(1:kount)';
35 - end
36 - if iflag1 >= 6 || iflag2 >= 6
37 -     fprintf('Filled Julia set is connected, converges to fixed point.\n');
38 - elseif abs(zk) > 100
39 -     fprintf('Orb(0) is unbounded, Filled Julia set is disconnected.\n');
40 - else
41 -     fprintf('Orb(0) is still bounded after 1500 iterations, assume connectivity.\n')
42 - end

```

## Part VI Divergent Set Code

```

1 -
2 - z=.1i;
3 - c=.36+.1i;
4 - phi = @(z) z^2 + c;
5 - fixpt1 = (1 + sqrt(1-4*c))/2;
6 - fixpt2 = (1 - sqrt(1-4*c))/2;
7 - colormap hot;
8 -
9 - for j=1:101, %initial values with imaginary parts between
10 -     y = -1 + (j-1)*.02; % -1 and 1
11 -     for i=1:101,
12 -         x = -1 + (i-1)*.02;
13 -         zk = x + 1i*y;
14 -         iflag1 = 0;
15 -         iflag2 = 0;
16 -         kount = 0;
17 -
18 -         while kount < 1500 && abs(zk) <=100 && iflag1 < 6 && iflag2 < 6
19 -             kount = kount+1;
20 -             zk = phi(zk); % This is the fixed point iteration.
21 -
22 -             err1 = abs(zk-fixpt1); %Test for convergence to fixpt1.
23 -             if err1 < 1.e-6, iflag1 = iflag1 +1; else, iflag1 = 0; end
24 -
25 -             err2 = abs(zk-fixpt2); %Test for convergence to fixpt2.
26 -             if err2 < 1.e-6, iflag2 =iflag2+1; else, iflag2 = 0; end
27 -
28 -         end
29 -
30 -         if abs(zk)>100
31 -             M(j,i) =kount;
32 -         end
33 -     end
34 - end
35 -
36 - image([-10 10],[-10 10],M), %This plots the results.
37 - pbaspect ([1 1 1]); %sets the ratio of x axis to y axis equal to 1
38 - axis xy %otherwise the vertical axis is inverted.

```

## Part VII Newton Method Code

```

1  %PART 7: NEWTON'S METHOD
2
3  n=2;
4  phi = @(zz) zz.^n + 1;
5  phi_prime = @(w) n.*w.^(n-1);
6  nmax=800;
7  max=40;
8  xmin=-1; ymin=-1;
9  xmax=1; ymax=1;
10
11 %Creating linspace like in mandelbrot.m
12 [x,y] = meshgrid(linspace(xmin, xmax, nmax), linspace(ymin, ymax, nmax));
13 z=x+1i*y;
14 zk=zeros(size(z));
15 k=zeros(size(z));
16
17 for j = 1:max
18     zk = z - phi(z)./phi_prime(z); %Newton's Iteration
19     k(abs(zk) > 2 & k == 0) = max - j; %Checking divergence
20     z=zk;
21 end
22
23 x=[xmin,xmax];
24 y=[ymin,ymax]; %Setting x&y values for the image
25 figure,
26 imagesc(x,y,k),
27 colormap hot %Color scheme- Black means convergence, border is fixed points
28 axis square

```

## Part VIII Mandelbrot Set Code

```

1  %Mandelbrot Set
2  function mandelbrot(n, max)
3  n=800; %height&width of image
4  max=40; %Maximum number of iterations
5  xmin = -1.5;    xmax = .6; %x & y max and min
6  ymin = -1.26;   ymax = 1.26;
7
8  %We create a grid that our c will span
9  %It's a square around our x and y max/min
10 [x,y] = meshgrid(linspace(xmin, xmax, n), linspace(ymin, ymax, n));
11
12 %Creating an array of space to keep the iterative values, z
13 %and the number of iterations it took to diverge, k
14 c = x + 1i * y;
15 z = zeros(size(c));
16 k = zeros(size(c));
17
18 for j = 1:max
19     z = z.^2 + c;
20     k(abs(z) > 2 & k == 0) = max - j; %Checking divergence
21 end
22
23 x=[-1.5,.6];
24 y=[-1.26,1.26]; %Setting x&y values for the image
25 figure,
26 imagesc(x,y,k),
27 colormap hot %Color scheme of mandelbrot set
28 axis square

```