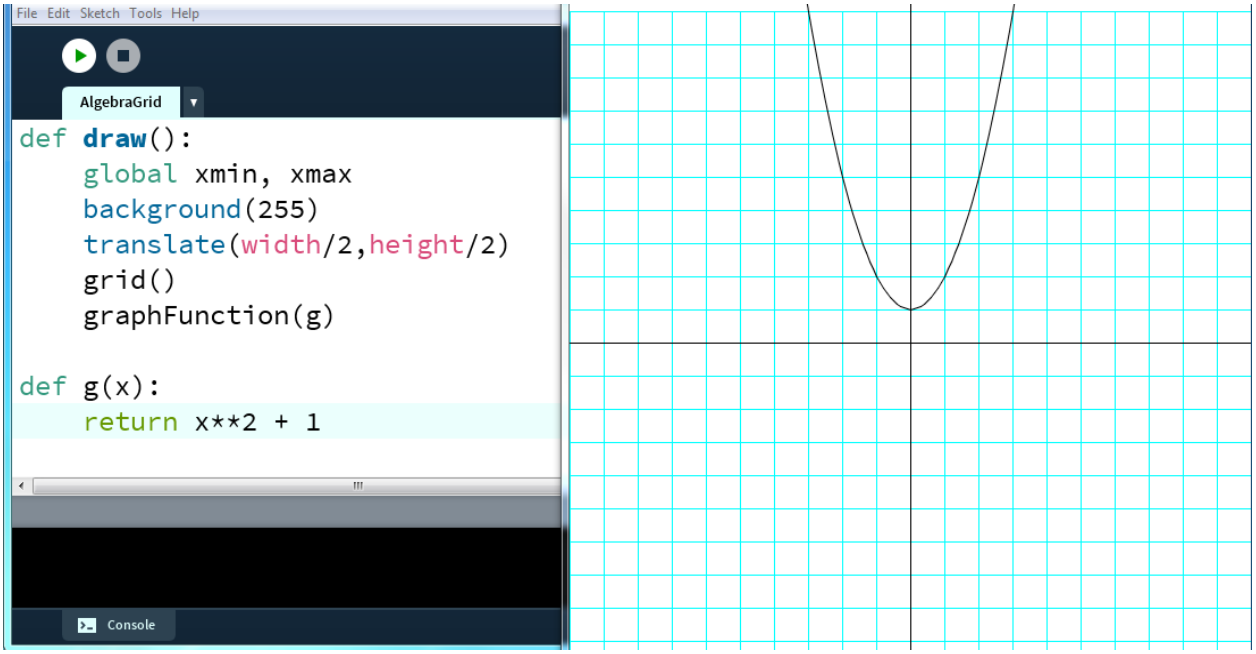
*Imaginary numbers are a fine and wonderful refuge of the divine spirit almost an amphibian between being and non-being. - Gottfried Leibniz*

## Complex Numbers

Numbers containing the square root of -1 have gotten a bad name in math classes. Calling something "imaginary" makes it seem like there's no real purpose for them. But they have a lot of real-world applications in electromagnetism, for instance. But in this chapter I hope to give some flavor of the beautiful art that can be made using "complex numbers," meaning numbers with a real part and an imaginary part. Using Python, manipulating these numbers becomes easier and we can use them for some very magical purposes.

There's a lot of confusion over why we ever needed to invent a number such as i, the square root of -1. Even textbooks say it was needed to solve equations like $x^2 + 1 = 0$. But there's no real number that makes that equation true, and when we graph the function $f(x) = x^2 + 1$




there's obviously no point where $f(x) = 0$. The curve never crosses the x-axis, so that's the end of it. There's certainly no reason to create a whole new kind of number to solve quadratics like this.

It was in the 1500s when Italian mathematicians used to hold public competitions to see who was smarter, that cubic equations started to get some real attention. Like the quadratic formula, there's a cubic formula which solves a specific type of cubic called a "depressed cubic" of this form:

$$x^3 - px = q$$

The formula is even uglier than the quadratic but it works!

$$t=\sqrt[3]{-\frac{q}{2}+\sqrt{\frac{q^2}{4}+\frac{p^3}{27}}}+\sqrt[3]{-\frac{q}{2}-\sqrt{\frac{q^2}{4}+\frac{p^3}{27}}}$$

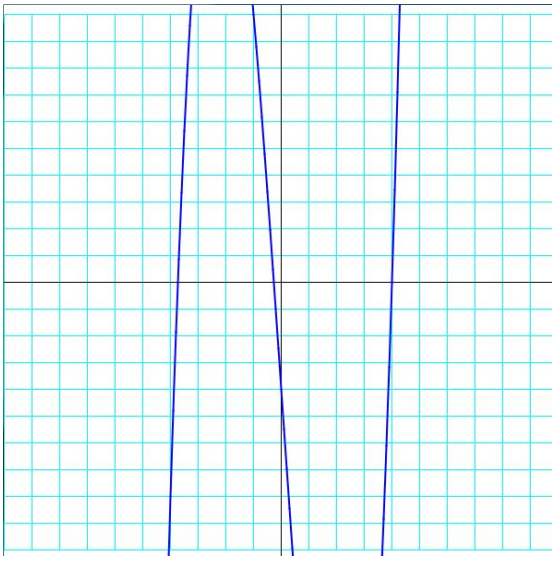
Except if, like the Italian mathematicians Cardano and Bombelli in the 1500s, you're trying to solve a depressed cubic like

$$x^3 - 15x = 4$$

Plug -15 and 4 into the cubic formula and it reduces to this:

$$x=\sqrt[3]{2+\sqrt{-121}}+\sqrt[3]{2-\sqrt{-121}}$$

As a medieval mathematician you'd be tempted to throw out such a result, because you had no idea what to do with the square root of -121. Let's graph it to make sure there are no real solutions. We'll replace the function in our grapher above with $g(x) = x^3 - 15x - 4$ and look for where it crosses the x-axis.



The curve crosses the x-axis three times. Meaning there are 3 real solutions to this equation! This is why the Italian mathematicians had to treat the result above seriously and deal with seemingly impossible numbers. Bombelli used trial and error to find out the cube root of 2 + 11i (from the above example) but we'll write some functions to help.

## *Geometric help*

What does multiplying by -1 do? We could look at it as rotating 180 degrees over the origin, like this:

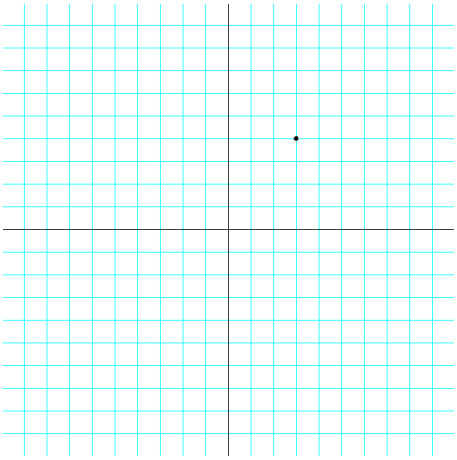
So what would the square root of -1 represent? A 90 degree rotation. Multiply by i again and you rotate 90 degrees more and it's like multiplying by -1. We can use our trigonometric rotations using sine and cosine to refer to any complex number.

R(cos

Here's how to graph a complex number. The first number is the horizontal number and the second number is the vertical number.

```
def graphComp(z):
    '''graphs a complex number
    z = x + iy'''
    fill(0)#black point
    ellipse(z[0]*xscl,z[1]*yscl,5,5)
```

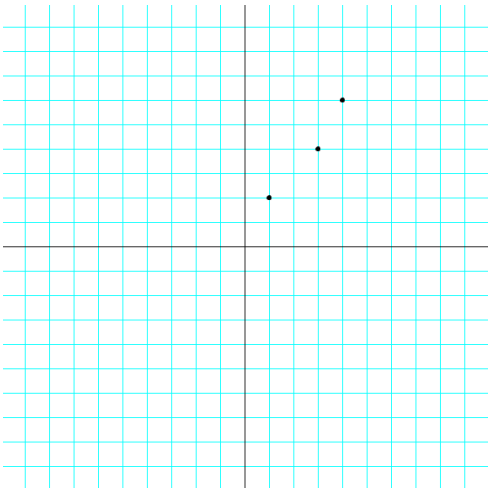
Here's what it looks like:



Adding two complex numbers together is just adding their x-values and adding their y-values. Write a function addComp(a,b) to do this.

```
def addComp(a,b):
    '''adds two complex numbers'''
    return [a[0]+b[0],a[1]+b[1]]
```

We defined the function called addComp, gave it two complex numbers (in list form [x,y]) and it returns another list. The first term of the list is the sum of the first terms of the complex numbers we gave it. The second term is the sum of the second terms (index 1) of the two complex numbers. If we graph the situation, this is what it looks like:



Adding complex numbers is just like taking steps in the x-direction and in the y-direction. It's not the most interesting part of studying complex numbers. But when you multiply them together (think rotation), things start getting interesting.

You can multiply two complex numbers together using FOIL:

(a + bi)(c + di) = ac + adi + bci + bdi2 = ac + (ad + bc)I + bd(-1) = ac – bd + (ad + bc)i = [ac-bd, ad+ bc]

```
def cMult(u,v):
    '''Returns the product of two complex numbers'''
    return [u[0]*v[0]-u[1]*v[1],u[1]*v[0]+u[0]*v[1]]
```

So to multiply u = 1 + 2i and v = 3 + 4i, you'd get

>>> u = [2,1]

>>> v = [3,4]

>>> cMult(u,v)

[2, 11]

The product is 2 + 11i.

## *What's Happening with Complex Numbers?*

The pattern is hard to see, until you find the **angle of rotation** a complex number represents. In u = 1 + 2i the 2 is the y-value and the 1 is the x-value. You can find the angle of this rotation by using the inverse of the tangent function, or atan:

```
from math import atan
def theta(x,y):
    return atan(y/x)
>>> theta(2,1)
```

```
0.4636476090008061
```

Remember that's in radians. Change the return line to

```
    return degrees(atan(y/x))
```

and theta(2,1) will return

```
26.56505117707799
```

in degrees. That means the complex number 2 + i represents a rotation of 26.6 degrees. How about 3 + 4i?

```
>>> theta(3,4)
53.13010235415598
```

53.13 degrees. When you multiply 2 + i by 3 + 4i you get 2 + 11i:

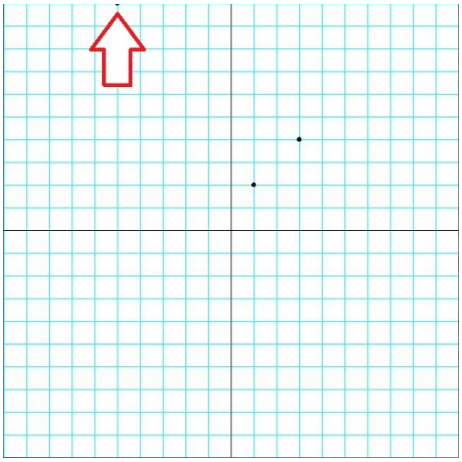
```
>>> cMult([2,1],[3,4])
[2, 11]
```

What rotation does 2 + 11i represent?

>>> theta(2,11)

79.69515353123397

Which is the sum of the rotations of 2 + i and 3 + 4i. When you multiply two complex numbers together, you add their angles of rotations, their thetas. What about their magnitudes? Magnitude is the distance the point is away from the origin. You find its length using the Pythagorean Theorem:



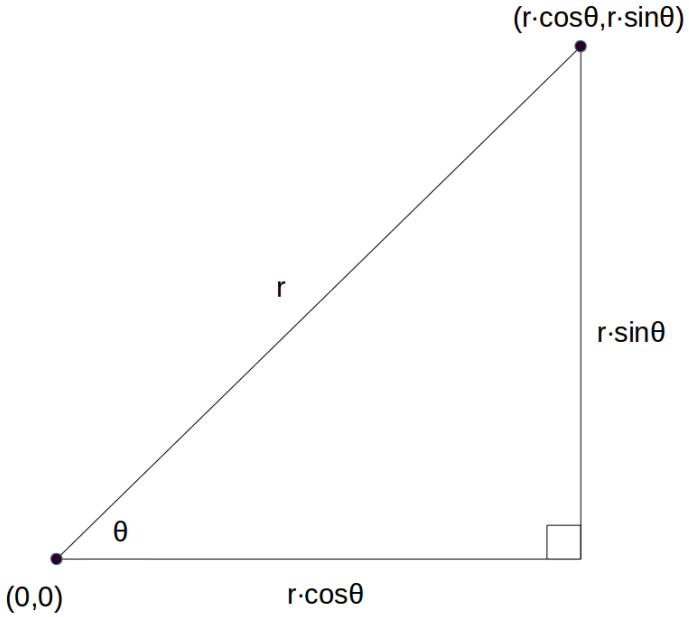
To multiply complex numbers using this notation, you simply multiply their magnitudes and add their angles.

## *Polar Form*

The same complex number can be expressed in a + bi form and in polar form:

$$z = r\cos(\theta) + i * \sin(\theta)$$

Now the situation looks like this:




This means

## *Mandelbrot Set Exploration*

There's a way to make complex numbers create surprisingly complicated and beautiful art: color every pixel on the grid according to how many iterations it'll take it to diverge. The formula we'll plug it into has another step than our squaring function. We'll square it and then add the original complex number to the square and repeat that process until it diverges. If it never diverges, we'll leave it black. For example, if z = 0.25 + 1.5i:

```
>>> z = [0.25,1.5]
```

We'll square z by multiplying it by itself and saving the square to a "z2" variable:

```
>>> z2 = cMult(z,z)
>>> z2
[-2.1875, 0.75]
```

Then we'll add z2 and z:

```
>>> addComp(z2,z)
[-1.9375, 2.25]
```

We need to test if this is more than two units away from the origin, using the Pythagorean theorem. Let's create a "magnitude" function:

```
def magnitude(z):
    #returns the distance from the origin
    return (z[0]*z[0] + z[1]*z[1])**0.5
```

We'll check if the magnitude is greater than 2:

```
>>> magnitude([-1.9375, 2.25])
2.969243380054926
```

So the complex number z = 0.25 + 1.5i diverges after only 1 iteration! How about z = 0.25 + 0.75i?

```
>>> z = [0.25,0.75]
>>> z2 = cMult(z,z)
>>> z3 = addComp(z2,z)
>>> magnitude(z3)
1.1524430571616109
```

It's still within 2 units of the origin, so let's replace z with this new value and put it back through the process again. First we'll create a new variable, z1, which we can use to square the original z:

```
>>> z1 = z
```

Repeat the process and find the magnitude:

```
>>> z2 = cMult(z3,z3)
>>> z3 = addComp(z2,z1)
>>> magnitude(z3)
0.971392565148097
```

It doesn't look like it's going to diverge, but we've only repeated the process twice. Let's automate the steps. What functions are we going to need for this task? We already have squaring, adding and finding the magnitude. Let's call a function mandelbrot, after the French mathematician Benoit Mandelbrot who first explored this process using computers in the 1970s. We'll repeat the squaring and adding process a maximum number of times, or until the number diverges:

```
def mandelbrot(z,num):
      '''runs the process num times
```

```
      and returns the diverge count'''
      count=0
      #define z1 as z
      z1=z
      #iterate num times
      while count <= num:
            #check for divergence
            if magnitude(z1) > 2.0:
                  #return the step it diverged on
                  return count
            #iterate z
            z1=addComp(cMult(z1,z1),z)
            count+=1
      #if z hasn't diverged by the end
      return num
```

In Processing, let's modify our "grid.pyde" sketch, make sure you have all your complex number functions ("addComp","cMult" and "magnitude", not to mention "arange") and use "println" to print a value to the Processing console:

```
def setup():
    size(600,600)

def draw():
    z = [0.25,0.75]
    println(mandelbrot(z,10))

def mandelbrot(z, num):
    '''runs the process num times
    and returns the diverge count'''
```

Nothing will appear on the screen yet, but in the console, you'll see the number 4 printed out. Come to find the complex number z = 0.25 + 0.75i diverges after 4 iterations. I printed out each step:

```
0.7905694150420949
1.1524430571616109
0.971392565148097
1.1899160852817983
2.122862368187107
```

So now we'll go through every pixel on the screen and put their location into the Mandelbrot process. They'll return a number, and if the pixel never diverges, we'll color it black. Going over all the pixels requires a nested loop for x and y in the draw function:

```
def draw():
    #origin in center:
    translate(width/2,height/2)
    #go over all x's and y's on the grid
    for x in arange(xmin,xmax,.01):
        for y in arange(xmin,xmax,.01):
```

Then we'll declare a complex number z to be x + iy and run that through the mandelbrot function:

```
            z=[x,y]
            #put it into the mandelbrot function
            col=mandelbrot(z,100)
```

The mandelbrot function will square and add the complex number 100 times and return the number of iterations it took for the number to diverge. This number will be saved to a variable called "col" since "color" is a keyword in Processing. That number will determine what color we make that pixel. For now, let's just get a Mandelbrot Set on the screen by making every pixel that never diverges (col = num) black. Otherwise the rectangle is white:

```
            #if mandelbrot returns 0
            if col == 100:
                  fill(0) #make the rectangle black
            else:
                  fill(255) #make the rectangle white
            #draw a tiny rectangle
            rect(x*xscl,y*yscl,1,1)
```

Run this and you should see the famous Mandelbrot Set!

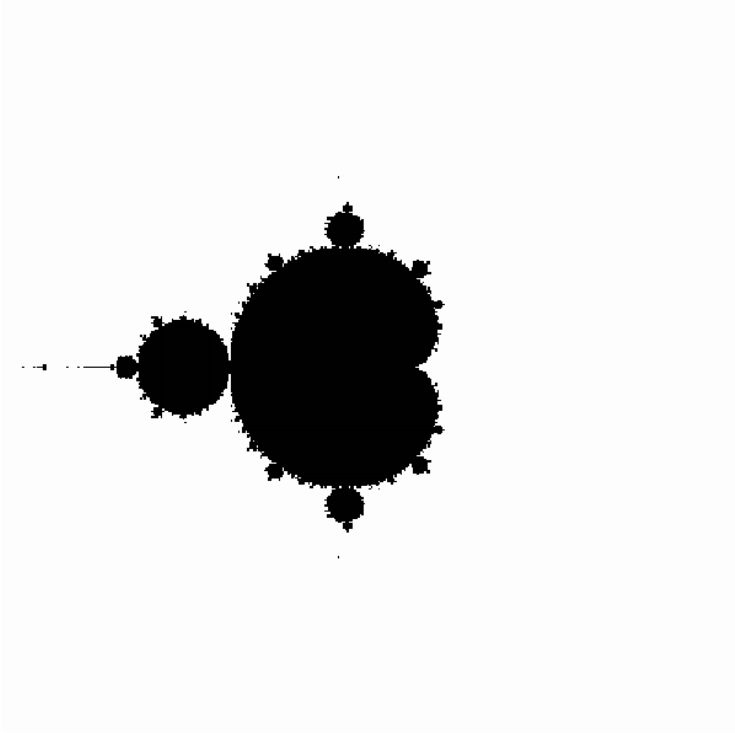


*Figure: The famous Mandelbrot Set.*

Isn’t it amazing? If not amazing, at least it’s a bit unexpected. I highly recommend searching out the YouTube videos people have posted of zooming in to spots on the Mandelbrot Set and it just keeps getting more and more complicated. Let’s give it some color. Let Processing know you’re using the HSB scale, not the RGB:

```
def setup():
      size(600,600)
      colorMode(HSB)
      noStroke()
```

And color the rectangles according to the value that’s returned by the mandelbrot function:

```
            if col == 100:
                fill(0)
            else:
                #map the color from 0 to 100
                #to 0 to 255
                col1 = map(col,0,100,0,255)
                fill(col1,360,360)
            #draw a tiny rectangle
```

```
            rect(x*xscl,y*yscl,1,1)
```

Using the map function we can change the range of the col variable from between 0 and 100 to between 0 and 255. Then we make that the “H” or “hue” component of the HSB color mode. Run this and you should see a nicely colored Mandelbrot Set:

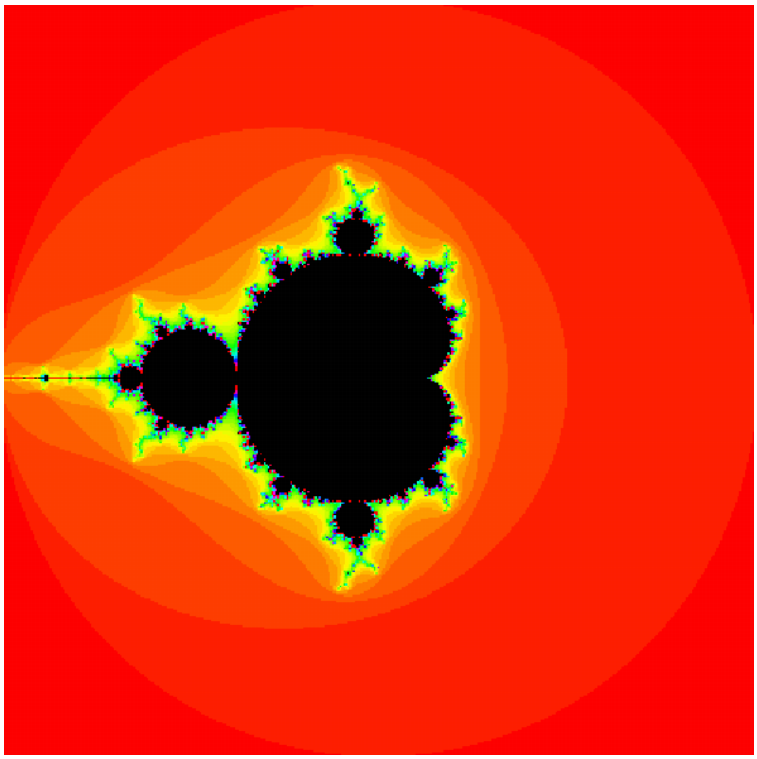


*Figure: The Mandelbrot Set colored depending on divergence values*

However, it just sits there. There’s a related set called the Julia set, which we can change dynamically using Processing.

## *The Julia Set*

The Julia Set is constructed just like the Mandelbrot Set, but after squaring the complex number, you don’t add the same complex number to it. You choose a complex number and keep adding that to the squared number. The Wikipedia page for the Julia Set gives a bunch of examples of beautiful Julia Sets and the complex numbers to use to create them. Let’s try to create the one using c = -0.8+0.156i. We can easily modify our mandelbrot function to be a julia function. Save your mandelbrot sketch as “julia.pyde” and change the mandelbrot function like this:

```
def julia(z,c,num):
    '''runs the process num times
```

```
    and returns the diverge count'''
    count = 0
    #define z1 as z
    z1 = z
    #iterate num times
    while count <= num:
        #check for divergence
        if magnitude(z1) > 2.0:
            #return the step it diverged on
            return count
        #iterate z
        z1 = addComp(cMult(z1,z1),c)
        count += 1
```

The complex number c will be different from z, so we'll have to pass that to the julia function:

```
        for y in arange(xmin,xmax,.01):
            #declare z
            z = [x,y]
            c = [-0.8,0.156]
            #put it into the julia program
            col = julia(z,c,100)
            #if julia returns 100
            if col == 100:
```

Run it and you'll get a much different design than the Mandelbrot Set:

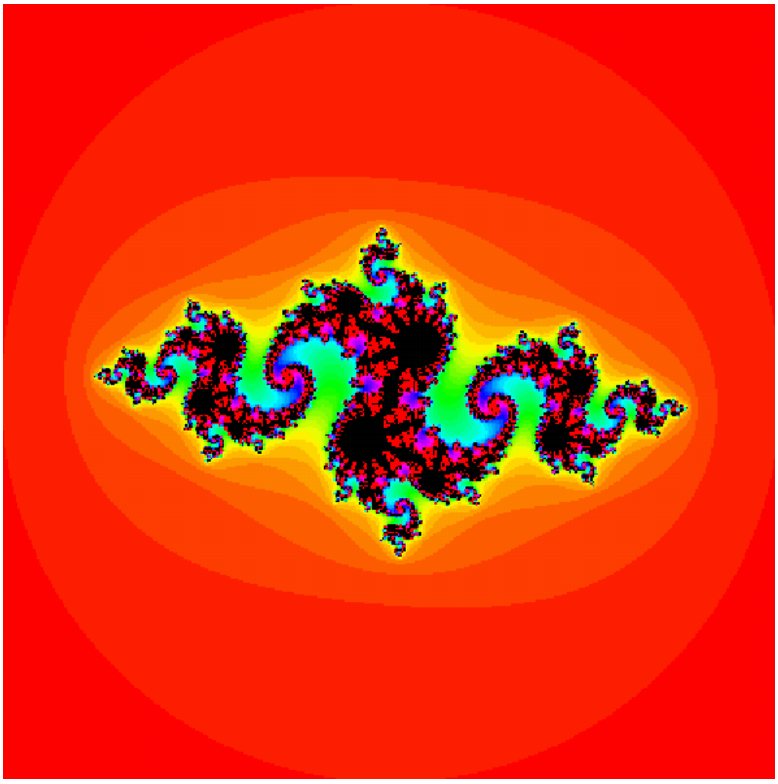


*Figure: The Julia set for c = -0.8+0.156i*

The great thing about the Julia Set is you can change c and have a different output. Change c to 0.4+0.6i and you should see this:

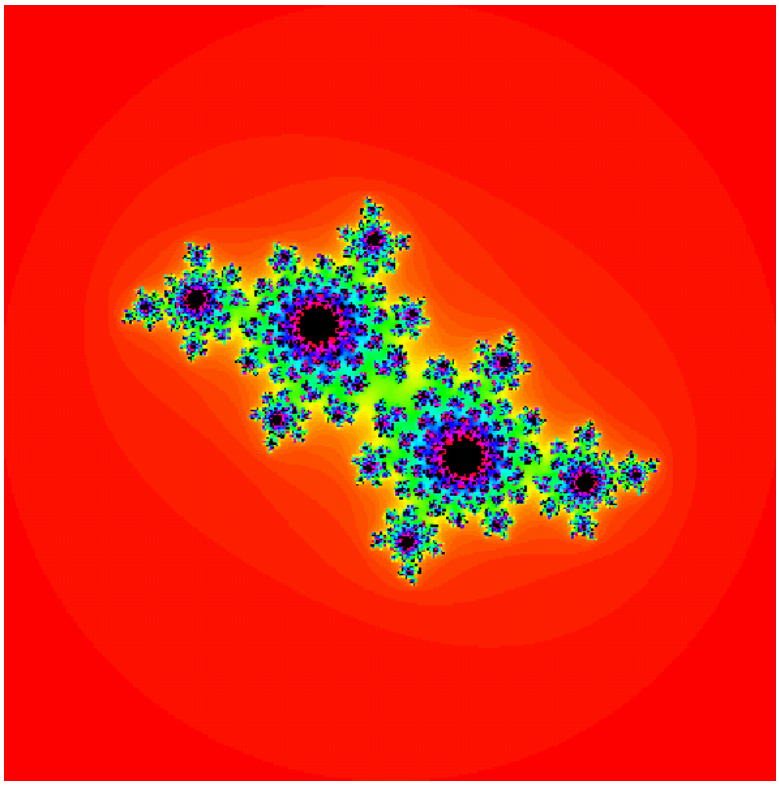


*Figure: The Julia set for  c =  -0.4+0.6i*