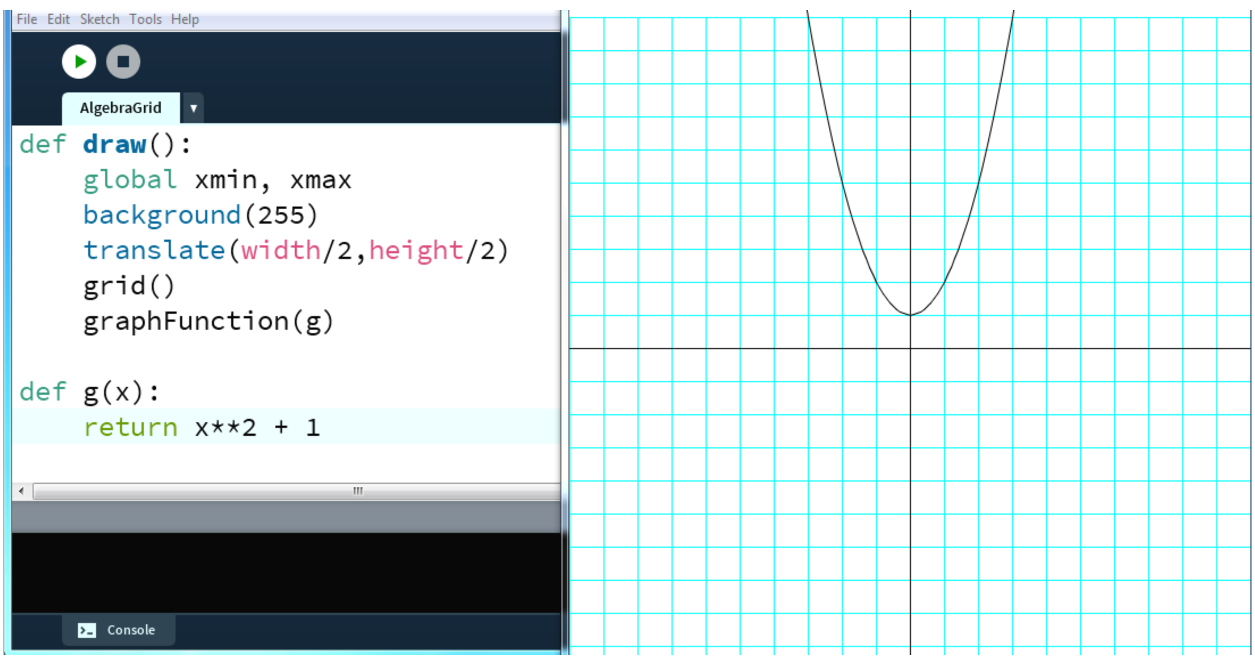
6

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!

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

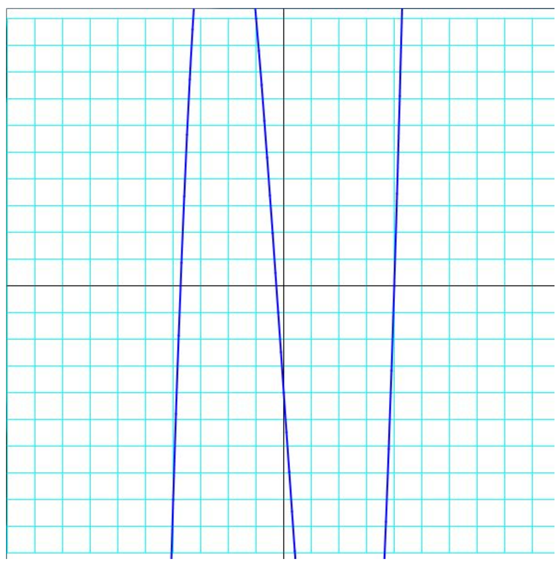
Except if, like the Italian mathematicians Cardano and Bombelli in the 1500s, you're trying to solve a depressed cubic like

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

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

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

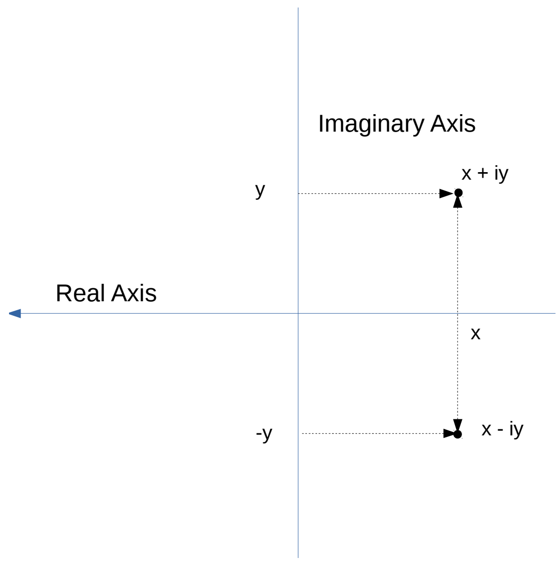
As a medieval mathematician you'd be tempted to throw out such a result, because you'd have no idea what to do with the square root of -121. Let's graph it to make sure there are no real solutions. We'll replace the function in our grapher above with $g(x) = x^3 - 15x - 4$ and look for where it crosses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

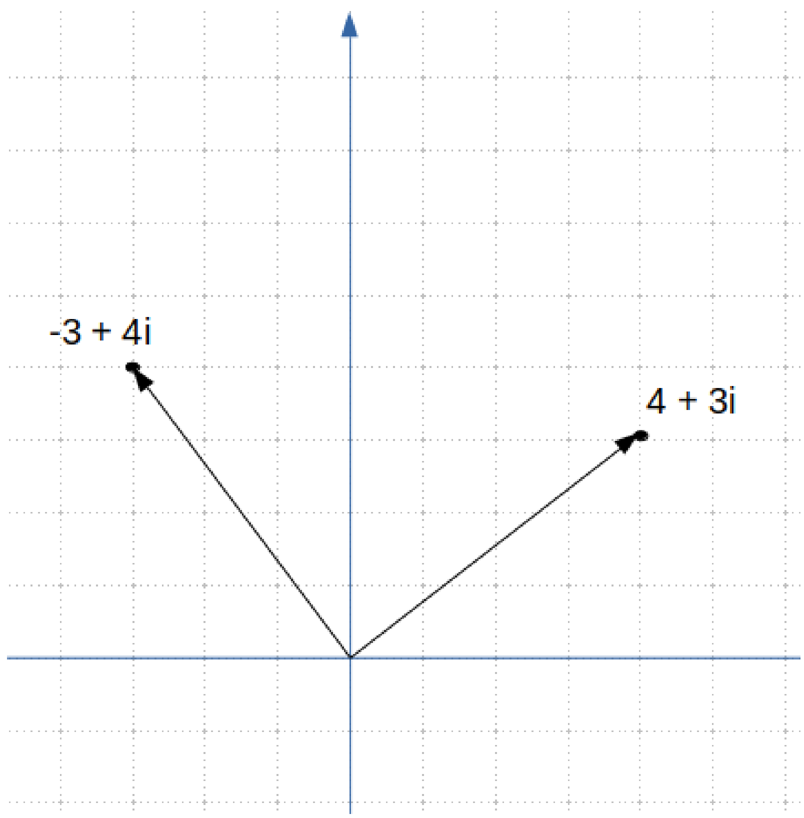
Geometric help

It helps to change our coordinate system a little. Now the real numbers are on the horizontal axis and the imaginary numbers are on the vertical axis:



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.

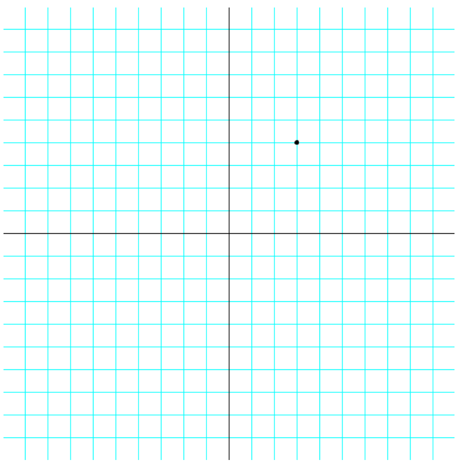


Multiply by i again and you rotate 90 degrees more and it’s like multiplying by -1.

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)
```

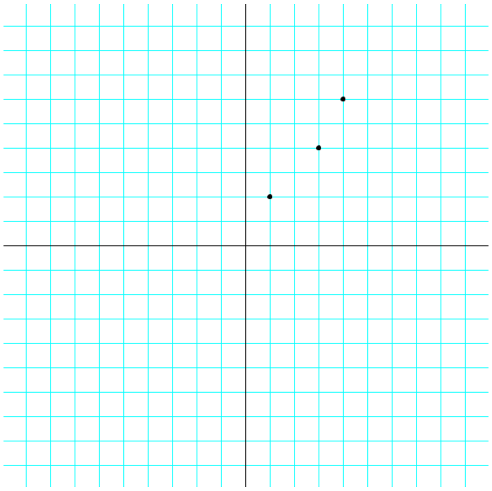
Here's what it looks like:



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:

$$(a + bi)(c + di)$$
$$= ac + adi + bci + bdi^2$$
$$= ac + (ad + bc)i + bd(-1)$$
$$= ac - bd + (ad + bc)i$$
$$= [ac-bd, ad+ bc]$$

```
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:

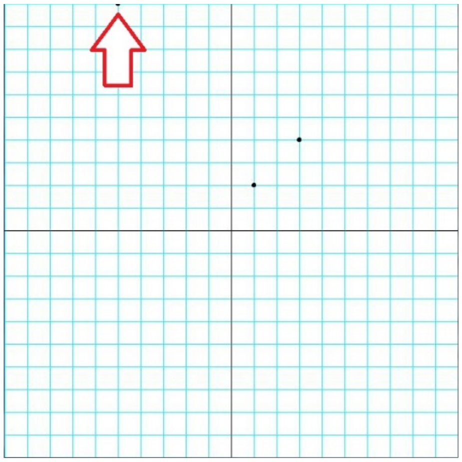
```
>>> cMult([2,1],[3,4])
[2, 11]
```

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**.

Mulitply their magnitudes? Won't they get really large?

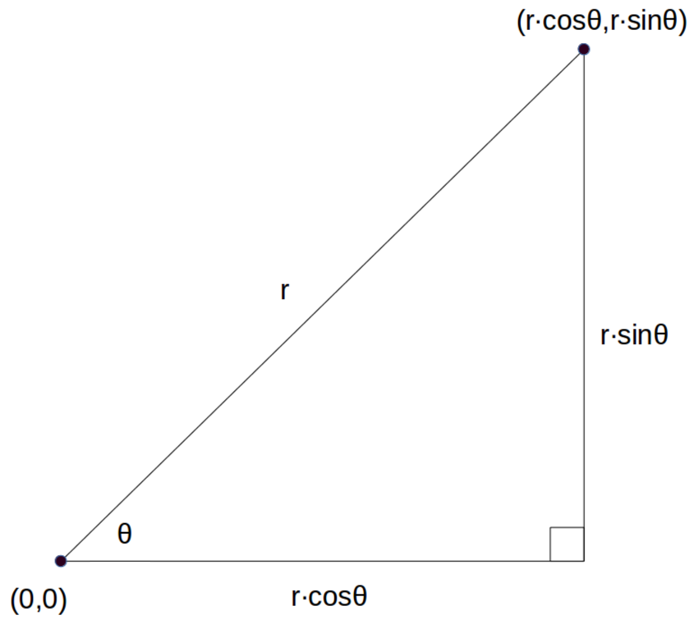
Yes, mulitplying numbers larger than 1 makes them larger. Multiplying 1 by itself stays the same, and multiplying numbers smaller than 1 makes them smaller. On the complex plane, it looks like this:

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 get too big and fly off the grid. Getting too big in math terms is called "diverging." 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!

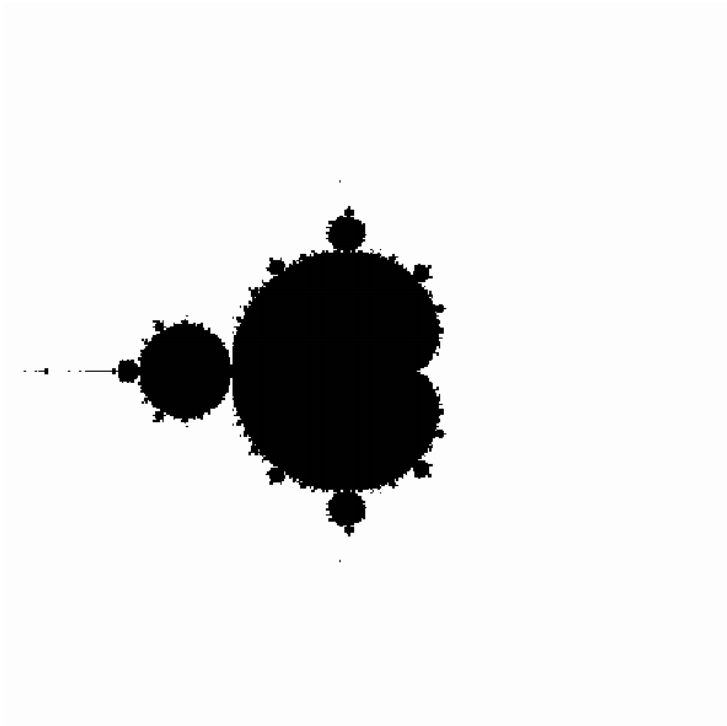


Figure: The famous Mandelbrot Set.

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:

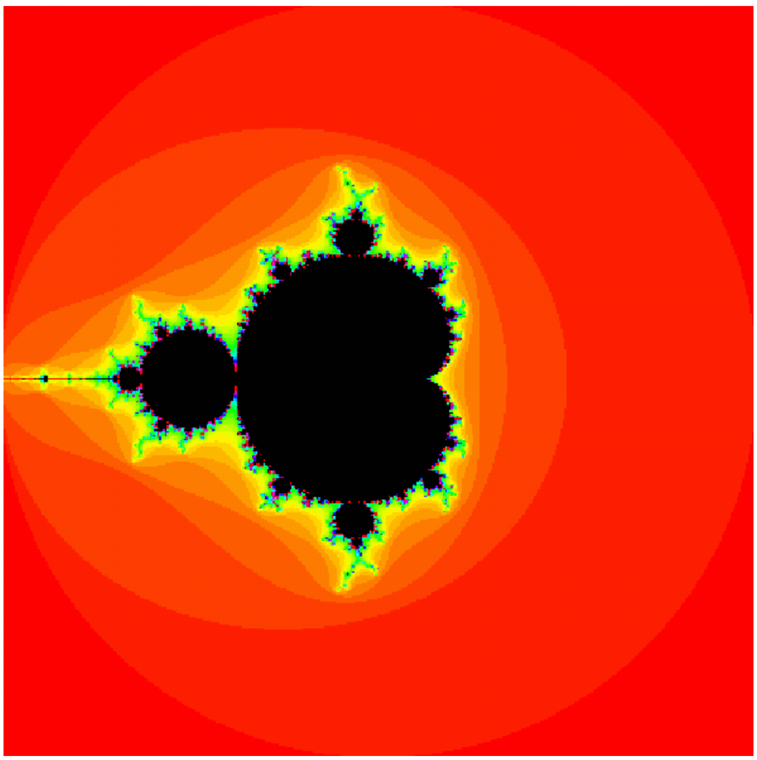


Figure: The Mandelbrot Set colored depending on divergence values

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:

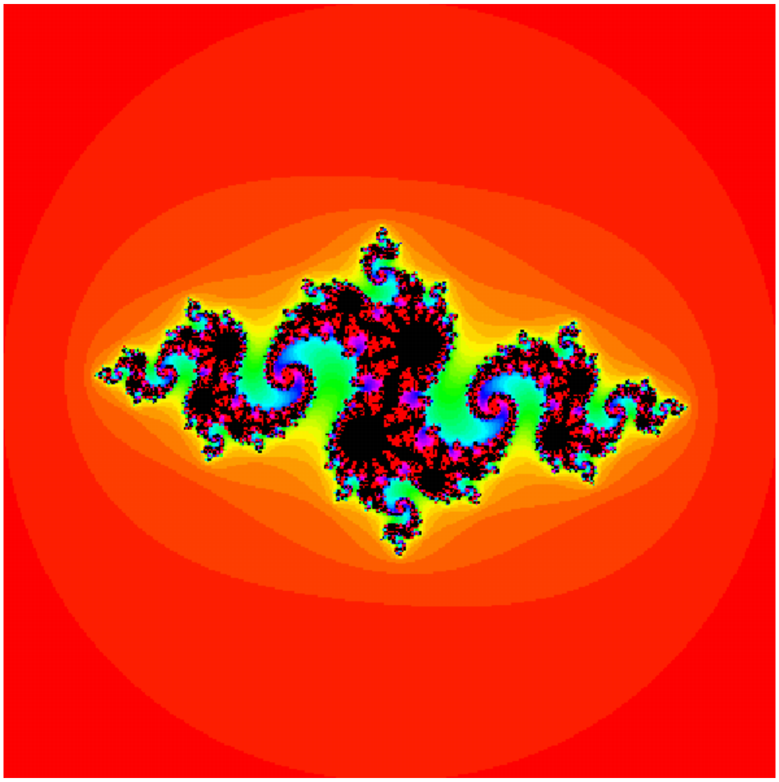


Figure: The Julia set for c = -0.8+0.156i

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:

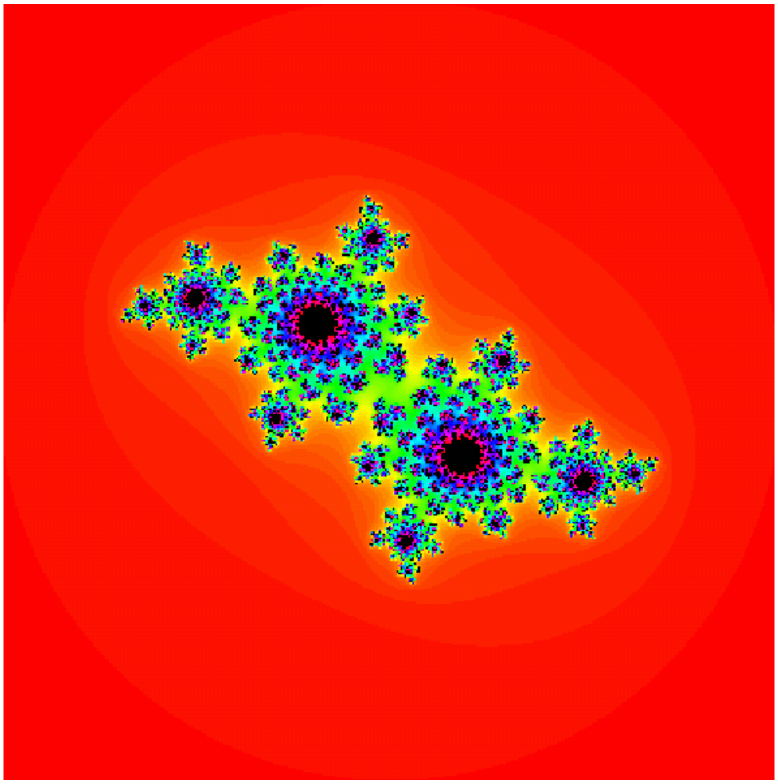


Figure: The Julia set for c = -0.4+0.6i

Transforming Pictures

What if we took all the pixels in a picture and transformed them according to a formula? If we took a section of the complex grid, in this case the numbers between 0 and 1, and squared all their locations, it would look like Figure 6-:

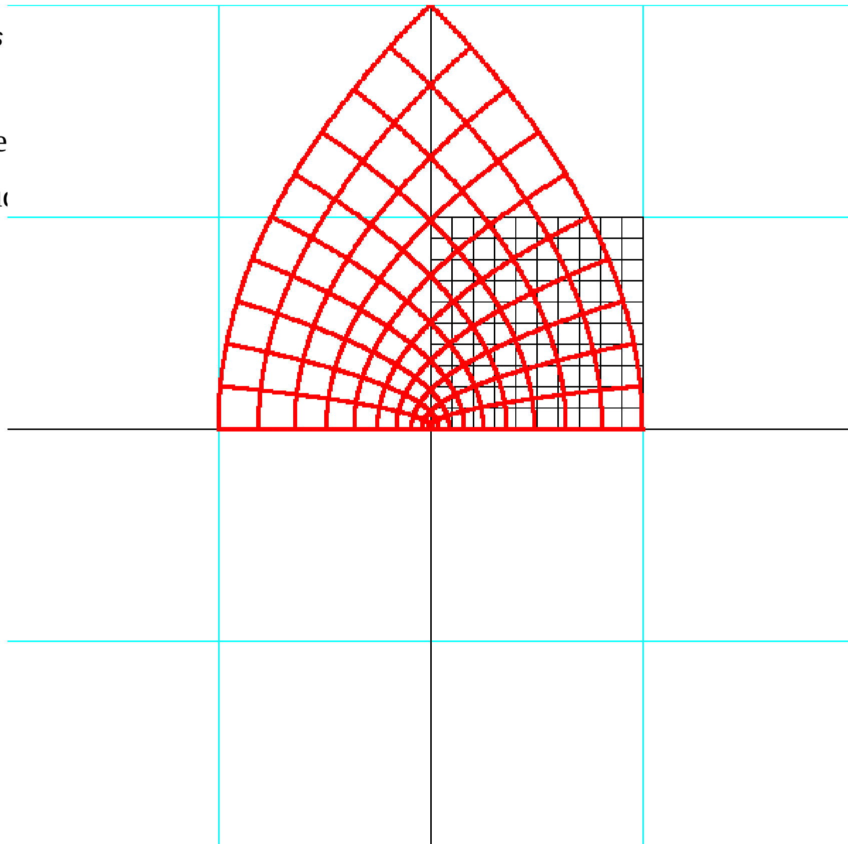
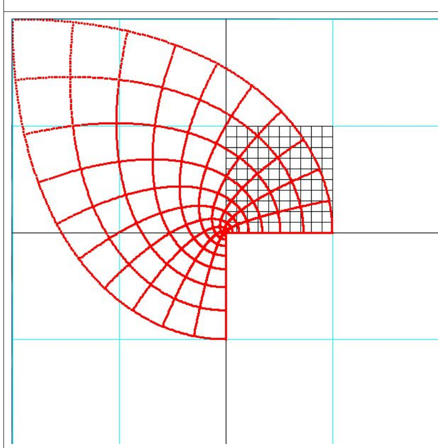
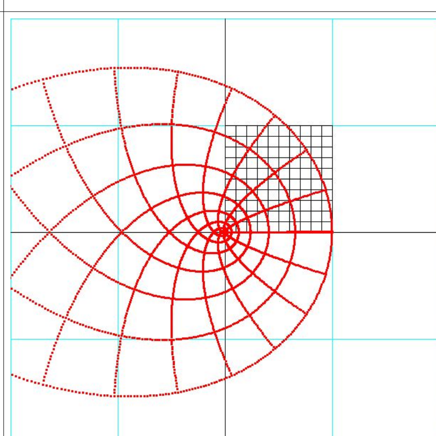
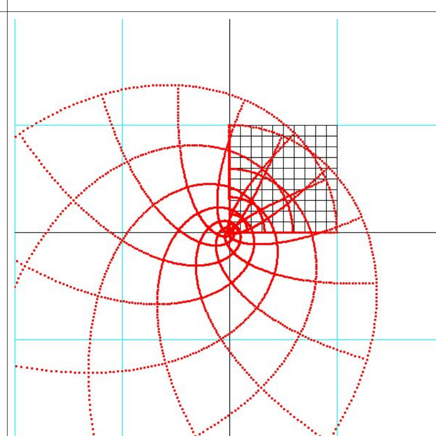


Figure 6-: the points … *d*

As you can see … on and (often) a change in magnitu… rs, it would look like this:

$Z = z^3$	$Z = z^4$	$Z = z^5$
		

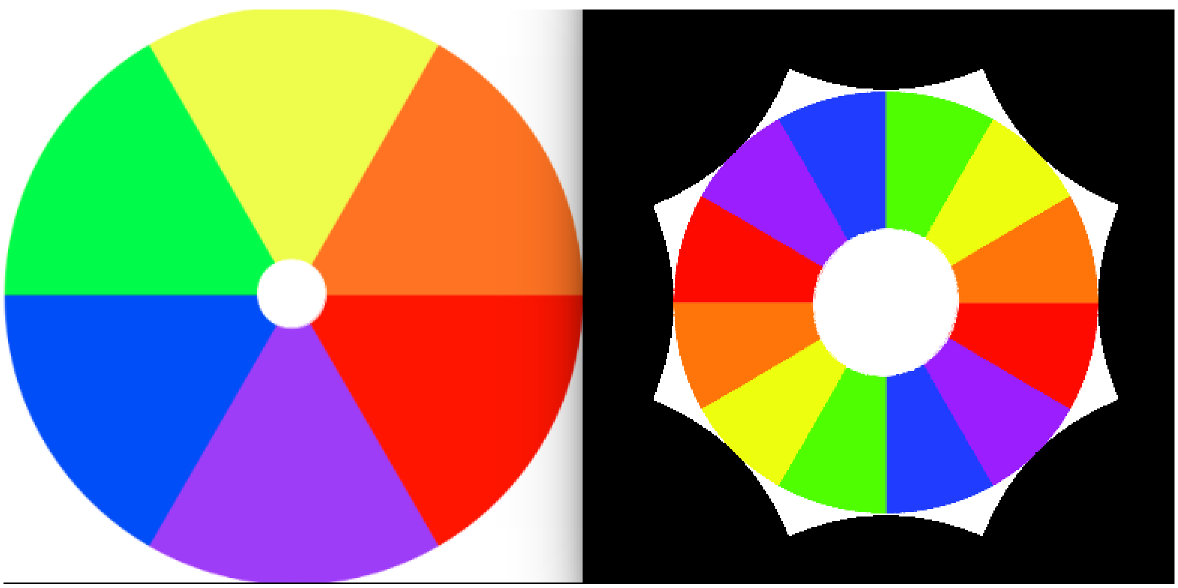
Complex Numbers are fun because they're meant to be transformed with functions, and the output can be displayed as Mandelbrot Sets or Julia Sets. But those two sets concern numbers that fly off, or "diverge," and what if you just want to show where a number ends up when you apply a certain function to it? The technical term is a number "maps" to another number. Let's start with a complex number like z = -0.4 + 0.5i. It's here on the grid

Let's square it:

```
>>> z = [-0.4,0.5]
>>> print(cMult(z,z))
[-0.08999999999999997, -0.4]
```

It "maps" to this point. How do you show that for a bunch of points? You can't just write out the output point, but you can use what's called a color wheel.

Every point on the color wheel also has a location which can be expressed in terms of complex numbers. So you put the input point into a function and color that point with the color of the output point on the color wheel. For the pixel corresponding to the number z = -0.4 + 0.5i, you'd color it with the color on the color wheel corresponding to the point z = -0.09, -0.4.



That color is …. Repeat this process for all points. Depending on the function you choose, the patterns you get from this process are really interesting, and maybe even beautiful.

Start a Processing sketch and name it “complex.pyde.” Save a picture with dimensions 400x400 in the same folder as your sketch. We’ll use Processing’s “Pixels” functions to create an empty 400x400 image and we’ll do the math to put the colors in it.

The loadImage function loads the image you’ve saved, and we’re assigning that to a variable called “myphoto.” What createImage does is create a “Pimage” datatype for storing images. It’s like a list of what’s in each pixel. And loadPixels makes sure the information about the pixel is saved in the right format, in our case, RGB colors. Let’s set it up:

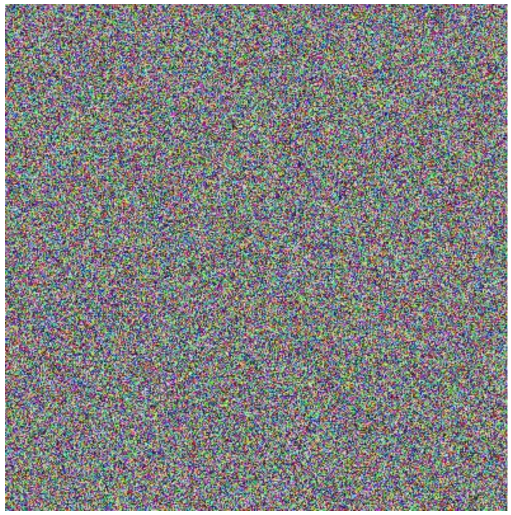
```
from __future__ import division

def setup():
    size(400,400);
    background(0)
    myphoto = loadImage("lavender.jpg");
    img = createImage(400,400,RGB); #create a new image
    img.loadPixels(); #create a list of pixels

    #to test:
    #set pixels to random colors
    for i,pixel in enumerate(img.pixels):
        img.pixels[i] = color(random(255),
                              random(255),
                              random(255))

    #draw image on screen
    image(img,0,0)
```

You can see this displays a collection of randomly colored pixels.



If you added `println(len(img.pixels))` to check the length of the pixels array, because it's 400 x 400 you'd expect the array to be 160,000 pixels long, right? Tell the program to print out the length of the img.pixels array and you'll find you're right.

```
#to test:
#set pixels to random colors
for i,pixel in enumerate(img.pixels):
    img.pixels[i] = color(random(255),
                          random(255),
                          random(255))
println(len(img.pixels))
#draw image on screen
image(img,0,0)
```

```
160000
```

What we have to do is square each pixel's complex form and then get the color of the resulting complex coordinate location in our lavender picture. So at the bottom of the sketch we'll put the "cMult" function we used in the Mandelbrot and Julia explorations. a and b will start at -2 and go up to positive 2.

We'll also create a "counter" variable which will count the number of pixels we're coloring. It'll start at 0 and we'll increment it every time we set a pixel's color.

```
#Create coefficients of complex number a + bi
#starting at xmin and ymin
b =-2.0;
```

```
a = -2.0;
counter = 0
while b < 2.0: #go up to 2
    a = -2.0; #reset after every loop!
    while a < 2.0: #go up to 2
        #Create complex number as p
        p = [a,b]
        q = cMult(p,p) #square p
```

So we started loops so a and b can keep going up (we'll increment them later) and we created a complex number p = a + bi, in our notation [a,b]. Then we squared p by multiplying it by itself using our cMult function and called that complex number q.

Next we'll check the color of the location of q in our color wheel or picture. But our scale is from -2 to 2 and we want it to be between 0 and the width and height of our image. We'll use Processing's map function. `map(q[0],-2,2,0,myphoto.width)` means take the first number in q and wherever it is between -2 and 2, scale it proportionally between 0 and the width of the photo. If it's a third of the way between -2 and 2, this will return the number a third of the way between 0 and (in our case) 400. We convert that number to an integer and assign it to a variable x. Do the same for y and you have a complex number x + iy which is the location on our image corresponding to the location of q = a + bi on our 4 x 4 grid.

```
        #convert range of values from -2 to 2 to 0 to 400
        x = int(map(q[0],-2,2,0,myphoto.width));
        iy = int(map(q[1],-2,2,0,myphoto.height));

        #get that color in "myphoto"
        c = myphoto.get(x,iy); ❶

        #put that color in the new image
        img.pixels[counter] = c;
```

At ❶ we used Processing's "get" function to get the color of the pixel in our photo at (x, iy) and saved the color to the variable c. In the next line we applied that color to the current pixel in the list of pixels in our output image.

Now we just repeat that for every pixel in our 4x4 area. We have a width of 400 pixels to work with but if a goes between -2 and 2 we'll make the computer calculate the decimal for the next a by incrementing 4.0 / img.width. Just for the record, 4.0 / 400 is 0.01. So the next complex number the program will square is -1.99 – 2.0i.

We’ll also increment our counter variable because we’ve colored pixel 0 and next we’ll be coloring pixel 1.

```
            #increment a and counter
            a += 4.0/img.width
            counter += 1;

        #increment b
        b += 4.0/img.height;
```

Outside the loop, when all the pixel-coloring is done, we’ll (finally!) draw the image on the screen:

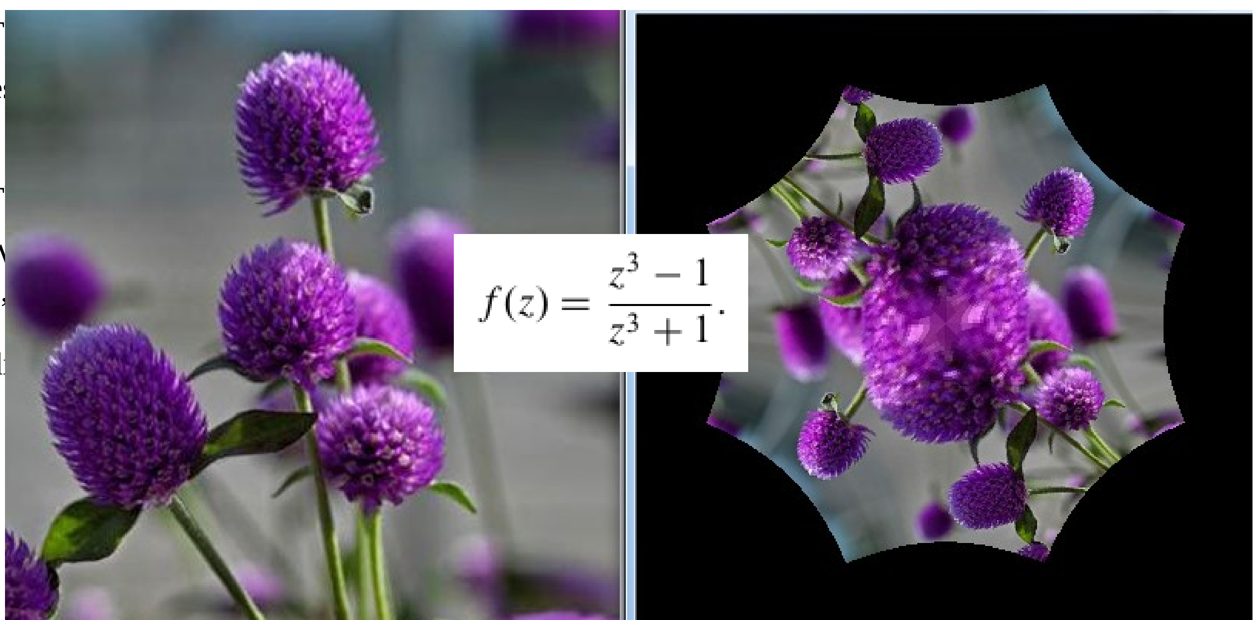
```
    image(img,0,0)
```

Run this and you’ll see the

Now, if instead of a standard color wheel you use a picture, of flowers for example, you can get even more beautiful output. I got this idea from the book Creating Symmetry by math professor Frank Farris.

T

corre



$$f(z) = \frac{z^3 - 1}{z^3 + 1}.$$

T

W

Here’

divid

$2^3 = 8$
$2^2 = 4$
$2^1 = 2$

$2^0 = 1$
$2^{-1} = ½$

So dividing by 2 is the same as taking 2 to the power -1. How do we take a complex number to a power? For the power 2, we multiply the magnitudes, which, since they're the same, it's squaring them. Then we add the angles of rotation together, which is the same as multiplying them by 2. Get the pattern yet? It's called DeMoivre's Theorem:

```
from math import atan,sqrt
...
def theta(x,y):
    return degrees(atan(y/x))

def magnitude(z):
    return sqrt(z[0]**2 + z[1]**2)

def power(z,n):
    r = magnitude(z)
    angle = theta(z[0],z[1])
    return [r*cos(angle),r*sin(angle)]
```