

Lab 9 - Complex Dynamics: Julia Sets

Thursday 25 October, 2006 - Due: Thursday 3 November

there is
no Lab 8

In this lab, you will *take a closer look* at a particular Complex Dynamical System, called a Julia Set. Since we will need complex numbers, you'll learn about working with complex numbers in Python. As in the case of the Sierpinski Triangle, the Julia sets display some interesting structures at all length scales. To see this, we will first highlight the structure with colors, then we will add a zooming feature. With these features, you'll now be free to roam about the various Julia Sets.

As always, please include in your writeup any sections of Python code you write.

♣ Read thoroughly. Every sentence has been very carefully constructed to guide you through this lab.

1 Complex numbers in VPython

The complex numbers were introduced in order to solve equations like

$$x^2 + 1 = 0,$$

which has no solutions among the real numbers (i.e., there is no real-valued x that solves that equation). Doing some algebra (e.g., the quadratic equation), we need

$$x = \pm\sqrt{-1}.$$

Mathematicians define the **imaginary unit** $i \equiv \sqrt{-1}$ (so, that $i^2 = -1$). Then, the complex numbers have the form $z = x + yi$, where x and y are real-valued.

! Some use "j" instead of "i". VPython uses "j".

()★ **Write (copy-paste) a short VPython program using the complex number type.**

```
from visual import *

x=sqrt(-1)
z=sqrt(-1+0j)

print z, z*z

z2=3+4j
print z2, z2.real, z2.imag, z2.conjugate()
print z2*z2, z2*z2.conjugate(), abs(z2)**2.
print atan2(z2.imag,z2.real), atan2(z2.imag,z2.real)*180.0/pi

z3=complex(3,4)
print z3, z3.real, z3.imag

print cos(pi), exp( complex(0,pi) )
```

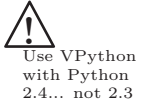
Observe that `x=sqrt(-1)` causes an error (“math domain error”) since the argument “-1” is assumed to be real-valued. To tell Python that we want to use complex-numbers. write -1 as `-1+0j`. Note that there is no multiplication symbol next to the `j`. Comment out that incorrect line, and run the program again.

(Q1)★ **In your write-up, include the CORRECTED copy of this program, together with its shell output.**

2 A Julia Set, in living color

Open `lab9_julia-faces.py` in IDLE and run it.

You should notice that it is a little faster than the version presented in class. Rather than using VPython’s `box()` for each picture-element (“pixel”), we are using VPython’s `faces()`, which is a low-level graphical object that lets us draw triangular faces.



From this segment of code, observe that

```
while ( (abs(z) < 2.0) and (iter<MAXITS) ):
    z = z*z + a #JULIA
    iter += 1

if(iter==MAXITS):
    bins[i,j]=1
```

marks the initial point in the complex plane black if it survives `MAXITS` iterations of not escaping the circle of radius 2 in the complex plane. **It might be interesting to see which points escape the circle of radius 2, color-coded by the number of iterations needed to escape.**

We are going to map an integer to a color... in a fancy way to take advantage of the ability to select Red, Green, and Blue components in a Python tuple: (R,G,B) , where $-1 \leq R, G, B \leq 1$.

The basic idea is can be seen in `lab9_color-demo.py`, which you should open in IDLE, then run. In this demo, with a selection of 64 integers, we form tuples as:

```
0.0    (0, 0, 0)
1.0    (0, 0, 1)
2.0    (0, 0, 2)
3.0    (0, 0, 3)
4.0    (0, 1, 0)
5.0    (0, 1, 1)
6.0    (0, 1, 2)
59.0   (3, 2, 3)
60.0   (3, 3, 0)
61.0   (3, 3, 1)
62.0   (3, 3, 2)
63.0   (3, 3, 3)
```

which are essentially the integers converted to “base 4” ($4^3 = 64$).

To get color-tuples, we divide each component by the largest “base 4” digit, 3. For example: `(0.0, 0.0, 0.0)` is black, `(1.0, 0.0, 0.0)` is red, `(0.3, 0.3, 0.3)` is a dark-grey, `(1.0, 1.0, 0.0)` is yellow, `(1.0, 1.0, 1.0)` is white.

Now, we're just about ready to include a color-code for the "number of iterations to escape".

♣ Code snippets are available at [lab9_julia-faces-snippets.py](#).

- First, let's make better use of `bins[]`.
Just after the `while(abs(z) < 2.0)...` loop, replace these two lines

```
if (iter==MAXITS):  
    bins[i,j]=1
```

with this one line

```
bins[i,j]=iter
```

♣ watch the
indentation

which records the number of iterations needed to escape the `while` loop.

- Second, near the top, just after the `MAXITS` assignment, include these lines

```
L=MAXITS**(1/3.)  
LL=L*L  
LLL=LL*L #this is MAXITS
```

Then, within the "`if B>0:` block", replace

```
colorcode=color.black
```

with

```
colorcode=( int(B/LL)/(L), int((B/L)%L)/(L), int(B%L)/(L) )
```

Admittedly, this scheme is flawed if `L` is not an integer (i.e., if `MAXITS` is not a perfect cube). By dividing by `L` instead of `L-1`, we avoid overflowing a color-component. While not perfect, it does do its job mapping an integer into a color-tuple.

(Q2)★ **In your write-up, include your REVISED copy of this program, together with a screen-capture of the display window.**

3 A Julia Set, close-up

In our visualization of a Julia Set, we only sample a region of space with a finite fixed level of resolution in space and iteration-time. To see more detail, we would like to zoom in on a selected region of interest, in order to *resample that region at a higher resolution*.

So, we have to: **use the existing code to create two functions:**

- `julia()` which resamples a small region centered at a given complex number `c`
- `draw_region()` which draws the region computed by `julia()`

Once that is done, we can insert some code to wait for mouse-events and zoom in accordingly.

```
julia() :
```

- ()★ **Highlight the “calculate the julia map” lines of code and hit [TAB] to indent that block. Then, just before this newly-indented block, insert this declaration:**

```
def julia(a,c,MAX,bins):
```

which defines the function named `julia` with parameters

- `a` (the complex number that defines the julia set),
- `c` (the complex number that defines the center of interest),
- `MAX` (the real-number that specifies the size of the region in the complex plane),
- `bins` (the array of “pixels” used to resample the region).

Next, our function needs to extract some information from these input parameters. The locally-defined variable `number` is defined from the shape of the `bins` array.

- ()★ **Insert this line, indented, at the top of this newly-created function `julia` :**

```
    number=bins.shape[0]-1
```

- ()★ **Finish up `julia` with these two changes:**

In order to focus on the region with center at the complex number `c` , you need to replace `z=complex(x,y)` with

```
        z=complex(x,y)+c
```

Finally, return `bins` to the calling function by inserting at the end of the `julia` function block:

```
return bins
```



Verify that your code looks like:

```
def julia(a,c,MAX,bins):
    number=bins.shape[0]-1

    ###calculate the julia map
    bins=zeros(((number+1),(number+1)))
    for i in arange (0,number+1):
        ...
        ...
        z=complex(x,y)+c
        ...
        ...
    return bins
```

```
draw_region() :
```

- ()★ Highlight the “draw the region” lines of code and hit [TAB] to indent that block. Then, just before this newly-indented block, insert this declaration and the `number` definition:

```
def draw_region(c,MAX,bins):  
    number=bins.shape[0]-1
```

- ()★ Finish up `draw_region` with these two changes:

In order to focus on the graphical region with center corresponding to the complex number `c`, you need to replace in *this newly-constructed* `draw_region`

```
x=-MAX+2*(MAX/number)*i  
y=-MAX+2*(MAX/number)*j
```

with

```
x=-MAX+2*(MAX/number)*i+c.real  
y=-MAX+2*(MAX/number)*j+c.imag
```



Verify that your code looks like:

```
def draw_region(c,MAX,bins):  
    number=bins.shape[0]-1  
  
    #####draw the region  
    nrm=vector(0,0,1)  
    side=2*float(MAX)/float(number)  
    ...  
    ...  
        x=-MAX+2*(MAX/number)*i+c.real  
        y=-MAX+2*(MAX/number)*j+c.imag  
    ...  
    ...
```

initializations :

- ()★ Near the start of the entire program, just after `a=complex(-0.75,0)`, insert the initializations of `c` and `bins` :

```
a=complex(-0.75,0)  
c=complex(0.,0.)  
bins=zeros(((number+1),(number+1)))
```

- ()★ Now that you’ve created those two functions, you can use them by:

```
bins=julia(a,c,MAX,bins)  
draw_region(c,MAX,bins)
```



()★ Now insert the entire block of code for the zooming functions from the end of [lab9_julia-faces-snippets.py](#) . You can choose the method of zooming by setting the value of `zoom_mode` appropriately.

(Q3)★ Explore the Julia set with `a=complex(-0.75,0)` at various locations and at various length scales. Include screen-captures in your write-up. (The shell-window prints the values of `a` and `c` and the zoom-range `MAX` .)

(Q4)★ Explore the Julia set with different values of `a` at various locations and at various length scales. Include screen-captures in your write-up. (Some interesting choices of `a` are in [lec9.pdf](#) .)