LINUX VOICE TUTORIAL

PYTHON: MAKE YOUR OWN JULIA FRACTALS

TARIQ RASHID

With a smattering of Python and some clever maths you can create even more beautiful chaos inside your Linux box.

Julia sets are closely related to the Mandelbrot Set, which we explored last issue with some basic Python. All of these are produced by the code we'll develop here from the same simple iterated function z2+c as the Mandelbrot fractals.

You can see that they are different to the Mandelbrot set, and yet there's something about them that is similar. You can also see that some Julia sets are a single object like the Mandelbrot, but some are many pieces, with some even becoming so fragmented they are almost like dust.

It might not be obvious, but while there is only one Mandelbrot, there are infinitely many Julia sets. The recipe, or algorithm, for creating Julia sets is the same as that for the Mandelbrot set except for one key difference.

Let's remind ourselves of the recipe for the Mandelbrot set. We choose a rectangle on the complex plane and test many evenly spaced points to see whether they're part of the Mandelbrot set. We do this by seeing if repeated iteration of a function z2+c, where c is the point being tested, results in the orbit escaping and diverging, or not. And if it does, we colour the point according to how fast the point diverges.

This recipe is essentially the same for making Julia sets. The difference is that the roles of the z and c values are reversed (see diagram below).

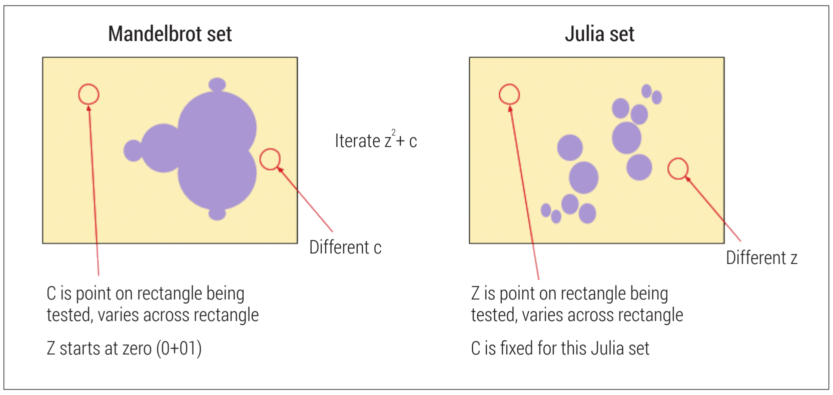
The following shows the calculating function julia(z, c, maxiter), based on the mandel(c, maxiter) function we developed in the last issue:

LV PRO TIP

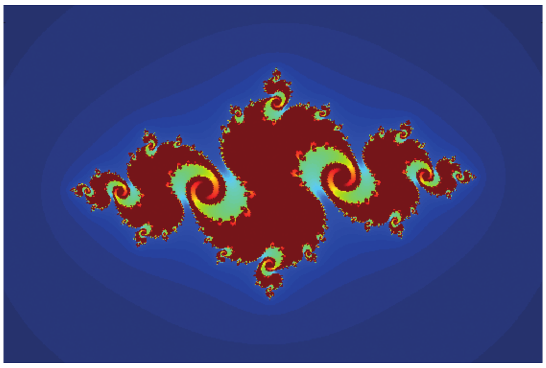
Explore the Mandelbrot and Julia fractals using the interactive XaoS open source software at **http://bit.ly/1vLdz52** or through a web browser **http://bit.ly/1lbXYL1**

Instead of **c** representing the point on the complex being tested (as in the Mandlebrot set), it is kept constant for all the points being tested. Similarly instead of **z** starting from zero, it starts as the complex number representing the point being tested.

```
def julia(z, c, maxiter):
  for iteration in xrange(maxiter):
    z = (z*z) + c
    if abs(z) > 4:
      break
```



Mandelbrot set
Julia set
Iterate z^2+ c
Different c
Different z
C is point on rectangle being tested, varies across rectangle
Z starts at zero (0+01)
Z is point on rectangle being tested, varies across rectangle
C is fixed for this Julia set


Just one look tells you that the Julia set is very closely related, but subtly different to the Mandelbrot set.

```
      pass
    pass
  return iteration
```

You can see that z doesn't start at (0+0i) anymore but is a parameter passed to the function. The parameter c is also passed but is a fixed complex number and not a number representing the point being tested, because z does that now.

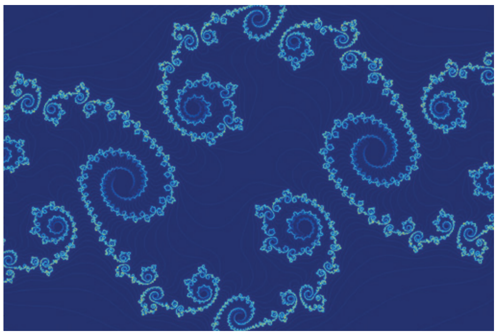
The Python program to plot Julia sets is presented here for easy reference:

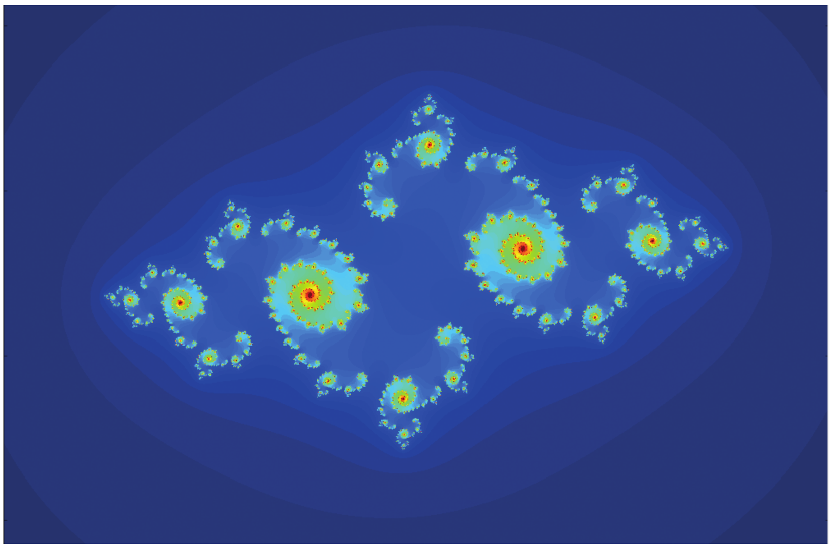
```
# loads the numerical and plotting extensions to Python
# needed if you're using a locally installed IPython, not for some online IPython providers
%pylab inline

# set the location and size of the atlas rectangle
xvalues = linspace(-2, 2, 1000)
yvalues = linspace(-2, 2, 1000)

# size of these lists of x and y values
xlen = len(xvalues)
ylen = len(yvalues)

# value of c (unique for each Julia set)
c = complex(-0.35, 0.65)

# julia function, takes the fixed parameters z and c and the maximum number of iterations maxiter, as inputs
def julia(z, c, maxiter):
    # start iterating and stop when it's done maxiter times
  for iteration in xrange(maxiter):

    # the main function which generates the output value of z from the input values using the formula (z^2) + c
    z = (z*z) + c

    # check if the (pythagorean) magnitude of the output complex number z is bigger than 4, and if so stop iterating as we've diverged already
    if abs(z) > 4:
      break
      pass
    pass

  # return the number of iterations we actually did, not the final value of z, as this tells us how quickly the values diverged past the magnitude threshold of 4
  return iteration

# create an array of the right size to represent the atlas, we use the number of items in xvalues and yvalues
atlas = empty((xlen,ylen))

# go through each point in this atlas array and test to see how many iterations are needed to diverge (or reach the maximum iterations when not diverging)
for ix in xrange(xlen):
  for iy in xrange(ylen):

    # at this point in the array, work out what the actual real and imaginary parts of x are by looking it up in the xvalue and yvalue lists
    zx = xvalues[ix]
    zy = yvalues[iy]
    z = complex(zx, zy)

    # now we know what c is for this place in the atlas, apply the mandel() function to return the number of iterations it took to diverge
    # we use 80 maximum iterations to stop and accept the function didn't diverge
    atlas[ix,iy] = julia(z,c,80)
```

Pretty as a picture

Here's an image produced by one of the filters provided by the same extension that we got the Gaussian filter from. It's beautiful – and is the cover of the author's ebook (**http://www.amazon.co.uk/Make-Your-Mandelbrot-Tariq-Rashid-ebook/dp/B00JFIEC2A**). See if you can recreate it yourself.

The image is the result of a filter that highlights edges (a Sobel filter) applied to the same Julia images we created previously. The code is at **http://bit.ly/1qpnlsE**.





Some Julia fractals are more broken up, while others are more contiguous.

```

    pass
  pass

# set the figure size
figsize(18,18)

# plot the array atlas as an image, with its values represented as colours, peculiarity of python that we have to transpose the array

imshow(atlas.T, interpolation="nearest")
```

Try plotting different Julia sets by changing the **c** parameter. Remember it is the single parameter **c** which uniquely defines the Julia set.

3D fractal landscapes

This section will require you to use a locally installed version of *IPython*, because the online *IPython* services can't access your OpenGL graphics subsystem to render the three-dimensional plots.

Let's try to turn the two-dimensional flat images of the Mandelbrot and Julia sets into three dimensions. One way to do this is to use the colour information in each image array to represent altitude; the height of a landscape above sea level as it were. We should then see mountainous landscapes shaped by the values of arrays filled by the Mandelbrot or Julia calculations.

Surface plot

Let's try it on a very simple array first. The following code prepares a 3x2 array:

```
a = zeros( [3,2] )
a[0,0] = 1
a[0,1] = 2
a[1,0] = 4
a[2,1] = 3
print a
```

Plotting a simple flat image based on these values, using **imshow(a, interpolation="nearest")** results in the simple 2D plot as expected.

LV PRO TIP

The code for the full programs to calculate and plot the Mandelbrot and Julia fractals is online at **http://bit.ly/1qpnlsE**.

LV PRO TIP
Don't forget that in Python, like many programming langauges, counting elements of arrays and lists starts at 0 not 1.

LV PRO TIP
You can explore these 3D views by rotating them and dragging them with your pointer, and use the menus to select different ligthting, colour palettes and many other options.

Now let's plot this same array in three dimensions, with the third dimension given by the value of each cell. To do this we need to use a new extension to Python called **mayavi**.

Telling *IPython* that we want to use **mayavi** uses the **import** instruction, just as before. The special **paylab** instruction is to ensure that the common set of numerical extensions are automatically loaded in the local *IPython*:

```
%pylab inline
import mayavi.mlab
```

The instructions to plot a surface are simple:

```
mayavi.mlab.surf(a, warp_scale="auto")
mayavi.mlab.show()
```

The first instruction prepares the surface plot and allows *IPython* to automatically scale the heights. The second instruction is needed to actually show the plot. This time the result will appear in a separate window outside your browser. You can do quite a lot to this 3D plot, including rotating it around and changing the lighting that illuminates it. The images at the bottom of the page show the above small array plotted in this way, and again rotated using the mouse. Try it yourself.

For easy reference, the entire code for creating a simple 3x2 array and plotting a three-dimensional plot of it as as follows:

```
%pylab inline
import mayavi.mlab

a = zeros( [3,2] )
a[0,0] = 1
a[0,1] = 2
a[1,0] = 4
a[2,1] = 3

mayavi.mlab.surf(a, warp_scale="auto")
mayavi.mlab.show()
```

We now apply this same idea to the larger arrays created by the Mandelbrot code we wrote earlier. We simply add the new instructions to create the three-dimensional plots, remembering to import the **mayavi** extension too. It's worth keeping the previous **imshow()** instruction to see the flat view too. The instructions are as follows. The **warp_scale** was adjusted to 1.0 to give a reasonable height, the default setting created hills that were a little too steep:

```
mayavi.mlab.surf(atlas.T, warp_scale=1.0)
mayavi.mlab.show()
```

These three-dimensional height fields as some people call them, give an interesting perspective of the Mandelbrot set.

Let's try that again but with the original rectangle zooming into details of the Mandelbrot set, and also change the colour palette by using the interactive menus. Some of these landscapes are quite haunting. Let's try the same idea with Julia sets.

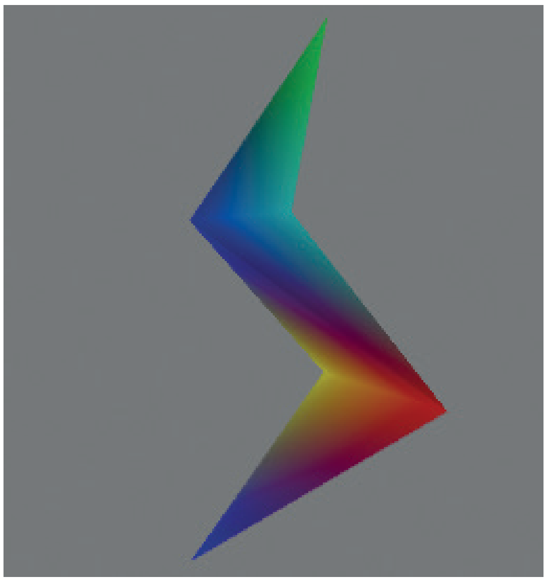
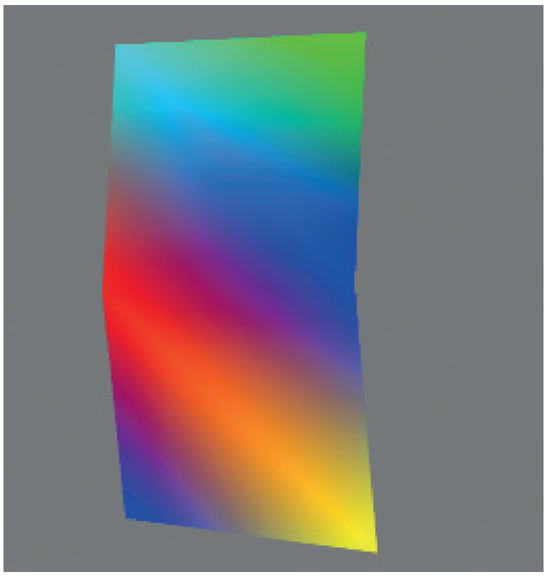
Gentler landscapes

Some of the landscapes we created were a little spiky and noisy. Let's see if we can calm them down, smooth off the sharp edges, and reduce the cragginess a little, all in the hope of creating a perhaps more gentle, more appealing, landscape.

One way to do this is to calculate a new image array, based on the original one, but with each new value somehow smoothed based on its neighbour's values. We want to calculate the average over a small group of neighbouring array elements, perhaps a 3x3 square or a bigger circle. Engineers and scientists who work with images or signals do this kind of thing fairly often to try to reduce noise.

Python provides a collection of such filters that can be applied to an array of values, which is often useful when they are in fact images. A common smoothing filter is a slightly more sophisticated version of the local average we just described, called a Gaussian blur filter. If you work with photo editing applications like *Photoshop* or *Gimp* then you'll be familiar with applying a Gaussian blur to smooth an image.

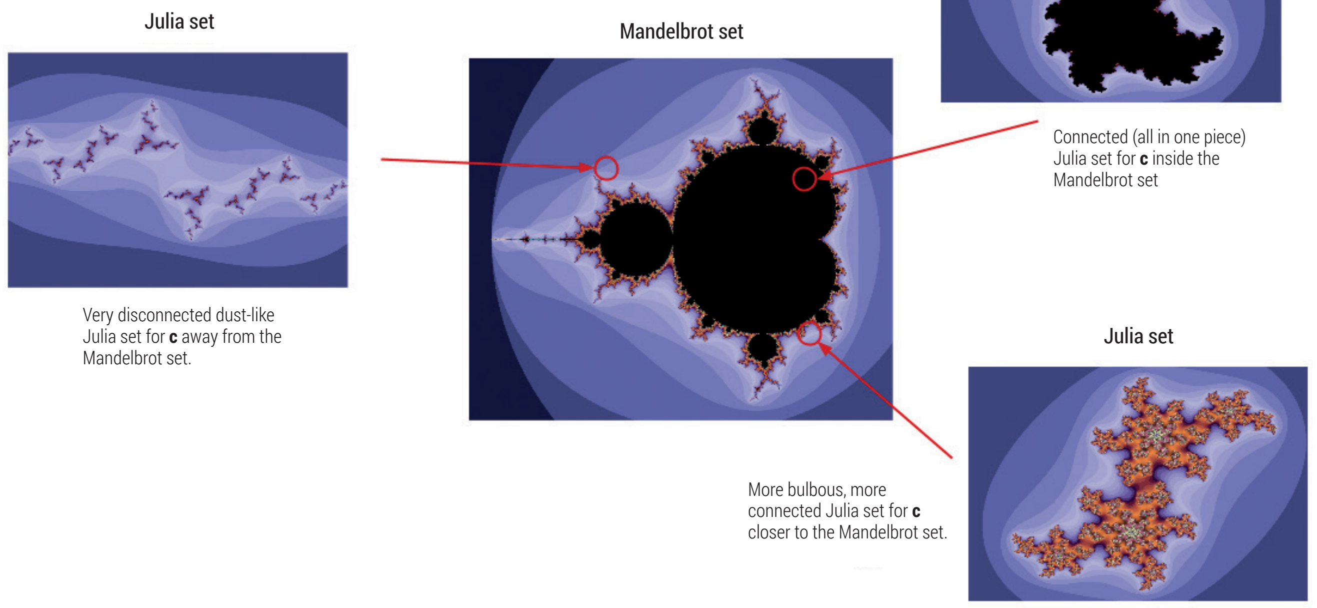
To get access to these filters, we'll need to import the Python extension at the top of our code as follows:

Small rotated and illuminated 3D plotted array.

The intimate relationship between Mandelbrot and Julia fractals

The Mandelbrot and Julia fractals are intimately connected. The relationship is illustrated in the following diagram. Each Julia fractal is uniquely defined by the chosen value of c in z2+c. For values of c that fall inside the Mandelbrot fractal, the Julia fractals are a single object. For values of c that fall outside the Mandelbrot, the Julia fractals are many separate pieces. The further c moves from the Mandelbrot set, the more fragmented and dust-like the Julia fractals become.


Julia set
Very disconnected dust-like Julia set for **c** away from the Mandelbrot set.
Mandelbrot set
Julia set
Connected (all in one piece) Julia set for **c** inside the Mandelbrot set
Julia set
More bulbous, more connected Julia set for **c** closer to the Mandelbrot set.

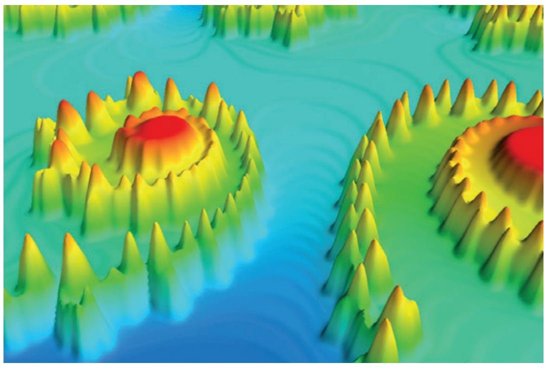
```
import scipy.ndimage
```

The following shows how the Gaussian blur filter can be used to create a smoothed image array from the original, which we called atlas in our previous code. The value 2 is the strength of the smoothing:

```
smoothed_atlas = scipy.ndimage.gaussian_filter(atlas.T, 2)
```

We can plot this as a flat image just as before using the **imshow** function:

```
# set the figure size
figsize(18,18)

# create a smoothed image of the original by applying a Gaussian blur filter
smoothed_atlas = scipy.ndimage.gaussian_filter(atlas.T, 2)
# plot the array atlas as an image, with its values represented as colours, peculiarity of python that we have to transpose the array
```


Add a 3D height field and a Gaussian blur to our Mandelbrot image and you get the Misty Mountains.

```
imshow(smoothed_atlas, interpolation="nearest")
```

Similarly we can plot the 3-dimensional landscape just as before using the **mayavi** extensions.

```
mayavi.mlab.surf(smoothed_atlas, warp_scale=0.9)
mayavi.mlab.show()
```

We'd suggest playing with the settings for the Gaussian smoothing filter to get an effect that works for you. For example, we've had great results applying the filter to the Julia set generated by c=(-0.768662 + 0.130477i), but there are many different variations and many kinds of smoothing effects you can produce.

Try It Yourself

We'd encourage you to explore other filters to see if they make interesting images. Perhaps you might explore different functions aside from z2+c which was the basis for both the Mandelbrot and Julia fractals.

And remember, you can't do any harm by experimenting and playing. We hope in this series of tutorials you've discovered the fun and convenient *IPython* environment for web-based Python programming, and become familiar with the basic building blocks of Python programming. We also hope you've experienced some of the surprise, excitement and beauty that even simple mathematics holds, particularly if your experience has been of boring trigonometry puzzles. Now go and play with some maths!

Tariq spends his time grappling with enterprise IT problems, informed by two decades of working with open technology.

LV PRO TIP
You can explore more filters, besides the Gaussian blur, to apply to your images at **http://bit.ly/1mNyMWE**