

Lab 6 - Fractal Dimension

Thursday 05 October, 2006 - Due: Thursday 12 October

We will *take a closer look* (pun intended) at the self-similar Sierpinski Triangle. Then, you'll modify the parameters of the dynamical model to obtain a different fractal. Finally, you'll calculate the fractal dimension of these fractals. Along the way, we'll introduce some aspects of "event-driven programming" using graphical user interface (GUI) controls available in VPython. *As always, please include in your writeup any sections of Python code you write.*

1 A closer look at the Sierpinski Triangle

Open `lab6_sierpinski.py` in IDLE and run it.

This is a variation of the code presented in lecture. Visually, it has been enhanced by the temporary highlighting of new points.* In addition, the graphics are now drawn on a 3D `display` rather than a 2D `gdisplay`, which has been useful for plotting. This was done in order to get more control of visualization.

In terms of the Python code itself, the specification of the parameters of the dynamical model has been simplified using Python arrays.

In order to get a closer look at this self-similar structure, you are going to add some code that will zoom in or out on a specific point using the mouse. This is an example of **event-driven programming**. (This basic idea behind this feature has already been used in, for example, lab 5's `logistic_bifurcation.py`.)

Here's the plan:

- left-click on a point: zoom in and re-center
- right-click on a point: zoom out and re-center
- middle-click (or simultaneously-left-and-right-click) on a point: re-center

VPython provides some functions (<http://www.vpython.org/webdoc/visual/mouse.html>) that will help us easily do this.

Associated with each VPython `display` (by default, named `scene`), is a `mouse` object, which is accessed as `scene.mouse`. When a mouse button is pressed, a "mouse event" is generated and is added to list of mouse events to be processed. By studying the mouse event, we can change the flow of execution of our program.

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

*The highlighting scheme seems to work in Python 2.4 if the `rate` argument is not too high.

Within the while block, add these lines:

```
if scene.mouse.events:
    MOUSE_EVENT=scene.mouse.getevent()
    print MOUSE_EVENT.press, MOUSE_EVENT.pos
```

This code reads “if there are any mouse events to be processed, retrieve the first unprocessed event and call it `MOUSE_EVENT`, then print out the state of the depressed button and the scene location for that event”.

- ()★ **Run the modified program and watch your shell window as you press various buttons on the mouse.** Observe that releasing a mouse-button also generates a mouse event. To see this, hold a mouse button down, then release. Don’t just click the button. Note that dragging the mouse also generates a mouse event.
- (Q1)★ **Include a copy-and-paste of your shell-window contents indicating these mouse events.**

We wish to continue by processing the mouse presses and drive the program accordingly.

Open `lab6_sierpinski_part2.py`. (1) Copy-paste the `zoom_scale_step` statement just after the `rad` statement, near the top (as suggested). (2) Copy-paste the rest into the `if scene.mouse.events:` block, just after the `print` statement.

Let me explain the last half of the pasted code. (You’ll explain the first half.)

```
target=MOUSE_EVENT.pos
step=(target-scene.center)/20.
for i in range(0,20):
    rate(50)
    scene.center += step
    scene.scale *= scale_step

rad /= scale_factor
for o in scene.objects:
    o.radius = rad
```

- The VPython vector `step` is the displacement vector from the current `scene.center` to `target`, the location that was mouse-clicked,... divided by 20. Then, we take 20 steps in incrementing `scene.center` and scaling `scene.scaling` by a multiplicative factor determined earlier. After 20 steps, the view should be centered at `target` and zoomed-in by some integer power of 2. During this animated change of view, the fractal computation is not being done.
- Since we are using finite-sized spheres in “VPython space” to model points, we have to resize all of our spheres.

(Q2)★ **In your own words, explain the logical tests and subsequent statements in this first half of this block of pasted code.**

(Q3)★ **Include at least three (3) window captures which suggest the self-similarity of the Sierpinski Triangle. Why don’t your window captures show perfect self-similarity?**

♣ Of course, check alignment of the blocks.

♣ `x += step` is a compact shorthand for `x = x + step`. Similar comments apply to `*=` and `/=`.

2 A variation on the Sierpinski theme

As was discussed in lecture, the Sierpinski Triangle could be modeled as a nonlinear dynamical system of the form

$$\begin{aligned}x_{new} &= a_i x + b_i y + e_i \\y_{new} &= c_i x + d_i y + f_i\end{aligned}$$


where the index i is chosen randomly from the set $\{0, 1, 2\}$ and

$a_0 = 0.5$	$a_1 = 0.5$	$a_2 = 0.5$
$b_0 = 0.0$	$b_1 = 0.0$	$b_2 = 0.0$
$c_0 = 0.0$	$c_1 = 0.0$	$c_2 = 0.0$
$d_0 = 0.5$	$d_1 = 0.5$	$d_2 = 0.5$
$e_0 = 0.0$	$e_1 = 0.5$	$e_2 = 0.25$
$f_0 = 0.0$	$f_1 = 0.0$	$f_2 = 0.5$

()★ **Modify your program to use the following nonlinear dynamical system:**

$a_0 = 1./3.$	$a_1 = 1./3.$	$a_2 = 1./3.$	$a_3 = 1./3.$	$a_4 = 1./3.$
$b_0 = 0.0$	$b_1 = 0.0$	$b_2 = 0.0$	$b_3 = 0.0$	$b_4 = 0.0$
$c_0 = 0.0$	$c_1 = 0.0$	$c_2 = 0.0$	$c_3 = 0.0$	$c_4 = 0.0$
$d_0 = 1./3.$	$d_1 = 1./3.$	$d_2 = 1./3.$	$d_3 = 1./3.$	$d_4 = 1./3.$
$e_0 = 0.0$	$e_1 = 2./3.$	$e_2 = 0.0$	$e_3 = 2./3.$	$e_4 = 1./3.$
$f_0 = 0.0$	$f_1 = 0.0$	$f_2 = 2./3.$	$f_3 = 2./3.$	$f_4 = 1./3.$

where the index i is now chosen randomly from the set $\{0, 1, 2, 3, 4\}$.

Note: the pattern should look like “a square that is recursively subdivided into (3×3) -squares, keeping the four corner-squares and the center square”.  Verify this, or else your answers to the rest of this lab may be completely wrong.

You may wish to change the `title` of the `display` window.

(Q4)★ **Include at least three (3) window captures which suggest the self-similarity of this “box-fractal” dynamical system.**

3 The fractional dimensionality of a fractal

As described in lecture, one can define the “dimension” of a figure, even one as strange as the fractals we’ve been discussing. In the simplest case, one defines the “box counting dimension” $d^{(0)}$ as

$$d^{(0)} = - \lim_{s \rightarrow 0} \frac{\ln N(s)}{\ln s},$$

where, after overlaying a grid of boxes of side s , $N(s)$ is the “number of non-empty boxes of side s ”. Instead of thinking in terms of “lengths of side s ”, one could think in terms of “divisions per unit length div ”: $div = 1/s$. Then, since $\ln(1/s) = \ln(s^{-1}) = (-1) \ln(s)$, we could write

$$d^{(0)} = + \lim_{div \rightarrow \infty} \frac{\ln N(div)}{\ln div},$$

Generally, this number is not an integer!

You’re going to try to determine approximations to $d^{(0)}$ for the box-fractal in the previous section.

- ()★ **Make a copy of your last program which generated the box-fractal.**

First, we have to limit the number of trials (to, say, 10,000) so that we can stop to evaluate our approximation to $d^{(0)}$.

- ()★ **Replace the statement enforcing an infinite loop, `while 1:`, with**

```
trials=int(1e4)
run=0
while(run<trials):
    run +=1
```

In order to speed up the computations, you may wish to completely delete or disable (from this copy) the block of code that zooms in on the fractal. In addition, you may delete or disable the `rate()` statement.

Next, we need to essentially make a 2D-histogram, in which we set up a grid of “bins” and keep a count of how many times a point lands in a particular bin.

- ()★ **Before the `trials` statement, insert the following lines, which declares the “number of divisions per unit length” and then creates a corresponding square matrix `bins` of integers, initialized to zero.**

```
divisions=64
bin=zeros( (divisions,divisions) )
```

- ()★ **Just after the iteration of the dynamical system (the `x=...` and `y=...` statements), insert the following lines**

```
bin[int( x*float(divisions) ),int( y*float(divisions) )] +=1
```

Thus, when a new point (x, y) is generated, we increment the count of the particular bin it lands in.



Note that the argument of `zeros()` is a pair, or more precisely, a **tuple** of integers.



Note that $0 \leq x < 1$ and $0 \leq y < 1$.

For “box counting”, we just need the number N of non-empty bins.

()★ **After the completion and exit of the `while` loop, add these lines:**

```
N=0
for k in range(0, divisions):
    for l in range(0, divisions):
        if bin[k,l]>0:
            N +=1

print divisions, divisions*divisions, trials, N
print math.log(divisions), math.log(N), math.log(N)/math.log(divisions)
```



(Q5)★ **Try different values for `divisions`.**

Include the completed program and the results from the shell window in your report. Include a plot of $\log(N)$ against $\log(\text{divisions})$ and use it to estimate a value for the fractal dimension

In order to visualize better what is happening, let me suggest the following two modifications:

()★ **Just after the creation of the `bin` matrix, i.e., `bin=zeros((divisions,divisions))`, add the following lines:**

```
for j in arange(0., 1.+1./divisions, 1./divisions):
    curve(pos=[ (0,j), (1,j)], color=color.red, thickness=rad)
    curve(pos=[ (j,0), (j,1)], color=color.red, thickness=rad)
```

which draws a grid for your bins. (The upper limit was nudged up to `1.+1./divisions` in order to draw the last grid lines.)

()★ **within the `if bin[k,l]>0:` block, just after the incrementing of `N`, i.e., `N += 1` add the following line:**

```
box(pos=( (k+.5)/divisions, (l+.5)/divisions, 2*rad ),
     axis=(0,0,1), length=1e-3, width=.8/divisions,
     height=.8/divisions, color=color.green )
```

which draws a green square if the bin is non-empty. (The `.5` s are needed to position the square appropriately in the grid.)

(Q6)★ **For a small value of `divisions`,**

include a window capture and a capture of the corresponding shell-window output.

