Fractal Friday!

Adapted from the series on @phy.sics, now on @theorama_org

May 8, 2020

A fractal is a pattern that never ends and is self-similar on every scale. They are generated by something called a positive feedback loop, which is essentially when a function that's reiterated yeilds the same result. Fractal patterns could be found everywhere in nature, from seashells to trees to galactic structures. Although the nature of fractals are complicated, the algorithms and processes that are used to produce them are fairly simple. And that's what *Fractal Friday* is all about, using basic algorithms to produce interesting fractal patterns. By default, all algorithms will be written in MATLAB and Python. Enjoy!

1 Week 1: The Sierpinski Triangle!

The first pattern we're going discussing is what is perhaps the most iconic Fractal Pattern, the Sierpinski triangle. The algorithm that is used to produce this pattern is known as a Chaos Game. The game goes as following [1]:

1. Pick 3 points which could be connected to a triangle (but don't connect them). For clarity's sake, we'll label them Points **A, B, and C**.
2. Pick an arbitraty point that exists within the region enclosed by the "imaginary triangle".
3. Get a dice and roll it. If the dice rolls to 1 or 2, mark a new point where its coordinates is half the distance between the initial point and **A**. If the dice rolls 3 or 4, do the same but for point **B**, and if it rolls 5 or 6, do the same for point **C**.
4. Using the new point as the new starting point, repeat the process a large number of time (our script reiterated 15,000 times).

[1] Note: You can actually try this out on pen and paper, however it will take a while for you to notice a clear pattern

The output of this Chaos game should be something that resembles the following:

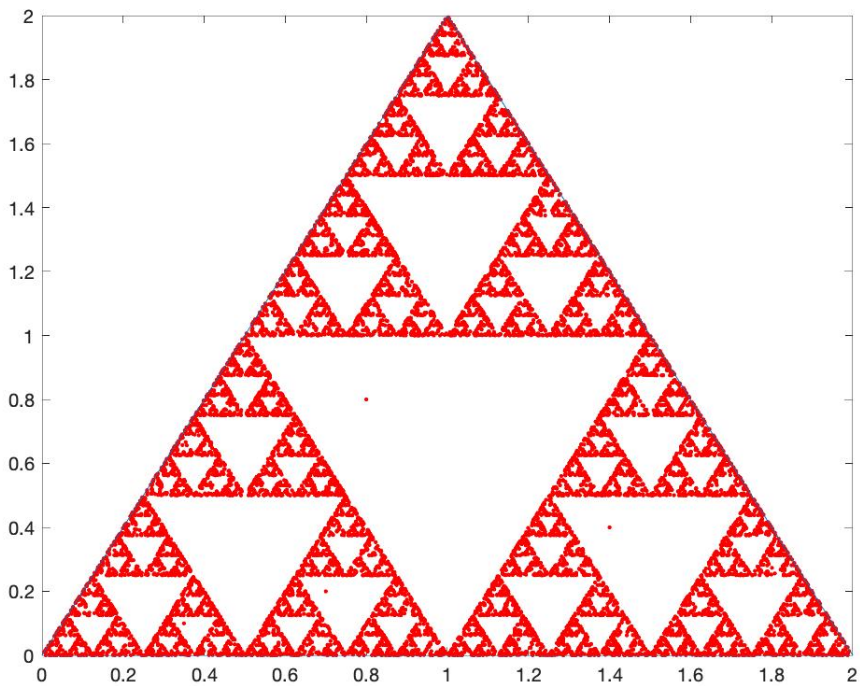

Fig. 1: A Sierpinski Triangle Formed by the MATLAB Script on Page 3.

Matlab Code

```
%% triangle co-ordinates
m=[0 1 2];
n=[0 2 0];

%starting point
ax=zeros(1,15001);
ay=zeros(1,15001);
ax(1)=0.8;
ay(1)=0.8;

%% dice
r=randi(6,1,15000); %15000 random integers frpm 1 to 6
for i=1:15000
    if r(1,i)==1
        x=0;
        y=0;
        elseif r(1,i)==2
        x=0;
        y=0;
        elseif r(1,i)==3
        x=1;
        y=2;
    elseif r(1,i)==4
        x=1;
        y=2;
    else
        x=2;
        y=0;
    end
    ax(1,i+1)=(ax(1,i)+x)/2;
    ay(1,i+1)=(ay(1,i)+y)/2;
    plot(ax(1,i),ay(1,i),'r.');
    hold on;
end

line([m(1),m(2)],[n(1),n(2)]);
line([m(2),m(3)],[n(2),n(3)]);
line([m(1),m(3)],[n(1),n(3)]);
```

Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
import time

ax=np.zeros((1,15001))
ay=np.zeros((1,15001))
ax[0][0]=0.7
ay[0][0]=0.7

a=np.random.randint(low=1,high=6,size=15001)
for i in range(0,15001):
    if a[i]==1:
        x=0
        y=0
    elif a[i]==2:
        x=0
        y=0
    elif a[i]==3:
        x=2
        y=0
    elif a[i]==4:
        x=2
        y=0
    elif a[i]==5:
        x=1
        y=2
    else:
        x=1
        y=2
    ax[0][i]=(ax[0][i-1]+x)/2
    ay[0][i]=(ay[0][i-1]+y)/2

plt.figure(1)
for i in range(15000):
    plt.plot(ax[0][i],ay[0][i],'b.')
plt.show()
```

2 Week 2: Barnsley Fern!

The next pattern is a pattern that seems to shock people as soon as they see it. Not because it is too abstract to be fathomed by the average mind, but for the exact opposite reason. It is shocking because it is something that is natural and something we see in our everyday lives. It is shocking because such a simple mathematical process can generate a familliar structure.

There are 4 2-dimensional transformations necessary to produce Barnsley's Fern, each one being an iterated function. That is what we call an ISF, or Iterated Function System. The system goes as following:

- $f_1 = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
- $f_2 = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$
- $f_3 = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.26 \end{bmatrix}$
- $f_4 = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.44 \end{bmatrix}$

The output of these four functions are:

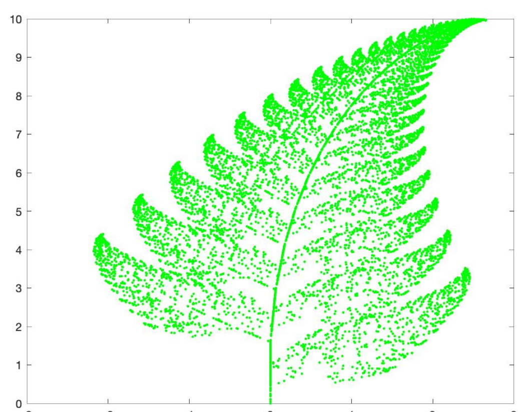

Fig. 2: A Barnsley Fern Formed by the MATLAB Script on Page 6.

f_1 produces the stem of the Fern, f_2 produces the smaller leaflets, f_3 produces the largest left-hand side leaflet, and f_4 the largest right-hand side leaflet.

Fig. 2 was produced with 10,000 iterations!

Matlab Code

```
r=rand(1,10000);
x(1,10000)=zeros;
y(1,10000)=zeros;
x(1,1)=0;
y(1,1)=0;
for i=1:9999
    if r(i)<0.01
        x(i+1)=0;
        y(i+1)=0.16*y(i);
    elseif r(i)<0.85
        x(i+1)=0.85*x(i)+0.04*y(i);
        y(i+1)=-0.04*x(i)+0.85*y(i)+1.6;
    elseif r(i)<0.93
        x(i+1)=0.2*x(i)-0.26*y(i);
        y(i+1)=0.23*x(i)+0.22*y(i)+1.6;
    else x(i+1)=-0.15*x(i)+0.28*y(i);
        y(i+1)=0.26*x(i)+0.24*y(i)+0.44;
    end
    plot(x(i),y(i),'g.');
    hold on;
end
```

Python Code

```
import numpy as np
import matplotlib.pyplot as plt
import time

x=np.zeros((1,10001))
y=np.zeros((1,10001))
x[0][0]=0.01
y[0][0]=0.01

r=np.random.randint(low=0,high=10000,size=10000)

for i in range(0,10000):
    if r[i]<100:
        x[0][i+1]=0
        y[0][i+1]=y[0][i]*0.16
    elif r[i]<8500:
        x[0][i+1]=0.85*x[0][i]+0.04*y[0][i]
        y[0][i+1]=-0.04*x[0][i]+0.85*y[0][i]+1.6
    elif r[i]<9300:
        x[0][i+1]=0.2*x[0][i]-0.26*y[0][i]
        y[0][i+1]=0.23*x[0][i]+0.22*y[0][i]+1.6
    else:
        x[0][i+1]=-0.15*x[0][i]+0.28*y[0][i]
        y[0][i+1]=0.26*x[0][i]+0.24*y[0][i]+0.44

plt.figure(1)
for i in range(10000):
    plt.plot(x[0][i],y[0][i],'b.')
plt.show()
```

3 Week 3: Mandelbrot Set!

This week, the pattern we're going to be discussing is what is known as the *Mandelbrot Set*. The Mandelbrot Set can be obtained by contour plotting the set of all complex numbers, z, where the following condition is applied: $f(z) = z^2 + c$ converges when z is reiterated from $z = 0$. (i.e $f(0), f(f(0)), f(f(f(0))), ...$)

Plotting $Im[c]$ vs. $Re[c]$, the following image is produced:

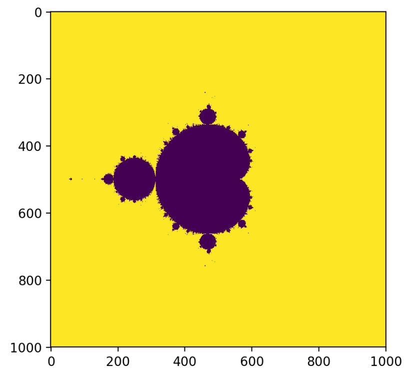

Fig. 3: The Mandelbrot Set Plotted by the Python Script below.

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

n=1000
Z=np.zeros([n,n],int)

xg=np.linspace(-2,2,n)
yg=np.linspace(-2,2,n)

for u,x in enumerate(xg):
    for v,y in enumerate(yg):
        z=0
        c=complex(x,y)
        for i in range(100):
            z=z*z+c
            if abs(z)>3:
                Z[v,u]=1
                break


plt.imshow(Z)
plt.show()
```

Matlab Code

```
%% power of Z
n=2;

%% square with side length 2 units and step size 0.01
[x,y]=meshgrid(-1:0.001:1,-1:0.001:1);

%% recurring function Z
z=abs((((((((((((((((((((((((((((((((((x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y).^n+x+1*i*y);

%% countour plot of absolute value of Z with (x,y) meshgrids
surf(x,y,z,'EdgeColor','none');

%% upper limit of Z can be changed for appropriate plot
zlim([0,1.5]);

caxis([0,1.5]);
view(2)
```

4 Week 4: Sin(x) Set

Let's say we have a complex number $z = x + iy$. A nice way to yield a fractal pattern from z would be be reiterating a function on z (See *Week 3 and 6*). That is exactly what we're going to be doing this week, with $sin(z)$. The code on page 12 performs a reiteration of $sin(z)$ 12 times, so the final output would be $sin(sin(sin(sin(sin(sin(sin(sin(sin(sin(sin(sin(x + iy))))))))))))$. The following pictures are cool patterns produced at specific x and y values:

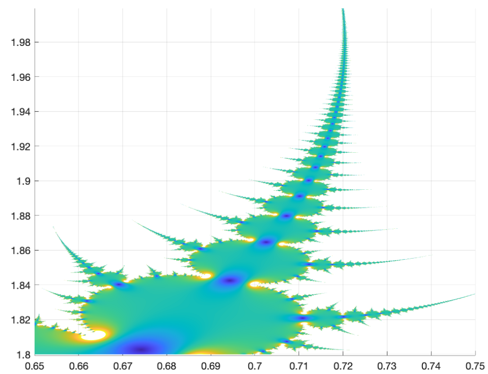

Fig. 4: A $sin(x + iy)$ reiteration, where $x = 0.65 \rightarrow 0.75$ and $y = 1.8 \rightarrow 2$.
"The Scorpion Tail"

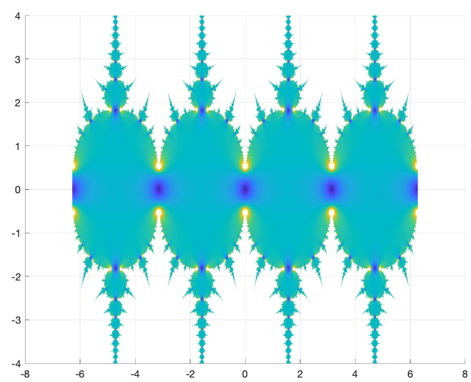

Fig. 5: A $sin(x + iy)$ reiteration, where $x = -2\pi \rightarrow 2\pi$ and $y = -4 \rightarrow 4$.

Python Code

```python
import numpy as np
import matplotlib.pyplot as plt

n=500
pi = np.pi

Z=np.zeros([n,n],float)

xl=np.linspace(-6,6,n)
yl=np.linspace(0,pi,n)

for u,x in enumerate(xl):
    for v,y in enumerate(yl):
        z=0
        c=complex(x,y)
        for i in range(100):
            z=np.sin(z)+c
            if abs(z)>100:
                Z[v,u]=1
                break

plt.imshow(Z,origin="lower")
plt.show()
```

Matlab Code

```matlab
%% background
[x,y]=meshgrid(-6:0.001:6,0:0.001:pi);
%% sine function initialization
z=sin(x+i*y);
z1=sin(z);
z2=sin(z1);
z3=sin(z2);
z4=sin(z3);
z5=sin(z4);
```

```
z6=sin(z5);
z7=sin(z6);
z8=sin(z7);
z9=sin(z8);
z10=sin(z9);
z11=abs(sin(z10));

surf(x,y,z11,'EdgeColor','none');

zlim([0,1]);
caxis([0,1]);
view(2)
```

5 Week 5: Cos(z) Set

Last week we covered the reiteration of $sin(z)$ on the complex plane. We will do the exact same thing this week but on $cos(z)$. No more is needed to be said, so without further ado, look at those amazing patterns

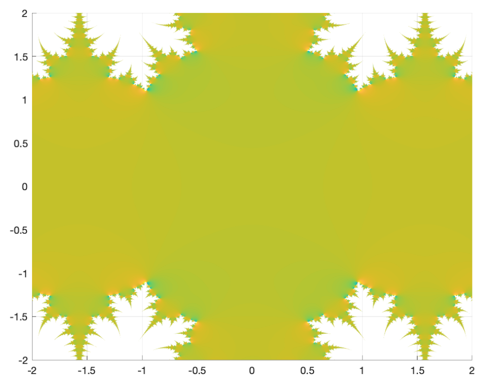

Fig. 6: A $cos(x+iy)$ reiteration, where $x = -2 \to 2$ and $y = -2 \to 2$.

As you can see, from -2 to 2 for both x and y, an amazing symmetrical shape is produced. Try out different intervals for yourslef!

Matlab Code

```
%% background
[x,y]=meshgrid(-2:0.001:2,-2:0.001:2);

%% cose function initialization
z=cos(x+i*y);
z1=cos(z);
z2=cos(z1);
z3=cos(z2);
z4=cos(z3);
z5=cos(z4);
z6=cos(z5);
z7=cos(z6);
z8=cos(z7);
z9=cos(z8);
z10=cos(z9);


z11=abs(cos(z10));


surf(x,y,z11,'EdgeColor','none');

zlim([0,1]);
caxis([0,1]);
view(2)
```

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

n=500
Z=np.zeros([n,n],float)

xl=np.linspace(-2*np.pi,2*np.pi,n)
yl=np.linspace(-3,3,n)

for u,x in enumerate(xl):
```

```
    for v,y in enumerate(yl):
        z=0
        c=complex(x,y)
        for i in range(100):
            z=np.cos(z)+c
            if abs(z)>300:
                Z[v,u]=1
                break

plt.imshow(Z,origin="lower")
plt.show()
```

6 Week 6: Fractal Tree!

Recursion is the process of a function referring to itself. If you have followed this series, you'd be very familiar with this process, because all fractals display recusion. In computer code, a recursion would happen when you have a function that has an operation which leads the function to call on itself. Suppose you have a tree that symmetrically branches out into 2 braches (symetrically meaning that the length and angle of the left branch is equal to the right brach), and then those 2 branches symmetrically branch out into 4, then 8, etc. That is a recursion process that leads to a fractal pattern. That fractal pattern is called a Fractal Tree.

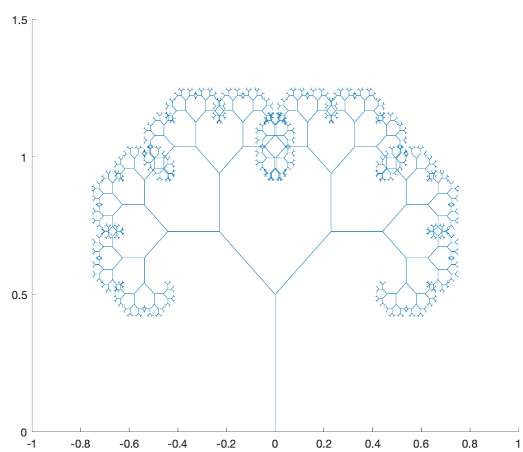

Fig. 7: A Fractal Tree Plotted by the MATLAB Script below.

Matlab Code

```
function def_xyz=tree(xold,yold,theta,length)

% xold,yold = value of x and y co-ordinate for every iteration
% theta= angle of rotation for every new branch
%length= length of branch for every new iteration


ratio=0.65; % multiplication factor for length of new branch that'll reduce the length

%new co-ordinates
xnew=xold+length*cos(theta);
ynew=yold+length*sin(theta);

if length > 0.005

    line([xold,xnew],[yold,ynew]);

    % brach on right side of every iteration
    tree(xnew,ynew,theta+pi/4,length*ratio);

    % brach on left side of every iteration
    tree(xnew,ynew,theta-pi/4,length*ratio);
    axis([-1 1 0 1.5]);
    pause(0.001);

end
end
```

Python Code

```
import matplotlib.pyplot as plt
import numpy as np
import random


def draw_tree(xold,yold,theta,length):
    ratio= 0.6
    xnew=xold+length*np.cos(theta)
    ynew=yold+length*np.sin(theta)
    if length >0.009:
        plt.plot([xold,xnew],[yold,ynew], '-r')
        draw_tree(xnew,ynew,theta+np.pi/5,length*ratio)
        draw_tree(xnew,ynew,theta-np.pi/5,length*ratio)


def main():
    draw_tree(1,1,np.pi/2,1)
    plt.show()
main()
```