Homework 1

Varun Nair

January 24, 2019

In [229]:
```
%precision %g
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

In [230]:
```
from math import sqrt, pi
from cmath import sqrt as csqrt
import numpy as np
from scipy import constants as C
import matplotlib.pyplot as pyplot
```

0.1 CP 2.5 Quantum Potential Step

This problem is to find the transmission (T) and reflection (R) probabilities for a given particle (with mass and energy) and potential step based on the wavevector forms of the probabilities:

$$T = \frac{4k_1k_2}{(k_1+k_2)^2} \text{ and } R = \left(\frac{k_1-k_2}{k_1+k_2}\right)^2.$$

The wavevectors k_1 and k_2 are defined in their respective regions as

$$k_1 = \frac{\sqrt{2mE}}{\hbar} \text{ and } k_2 = \frac{\sqrt{2m(E-V)}}{\hbar}$$

In [231]:
```
#This cell defines constants

h = 6.626e-34 #in kg m^2 s^-1
hbar = h / (2*pi)
```

In [232]:
```
#defines wavevector functions

def k1(m, E): #m is mass and E is energy
    return (sqrt(2*m*E)) / hbar

def k2(m, E, V): #V is height of potential step
    return (sqrt(2*m*(E-V))) / hbar
```

```
        #probability functions defined below

        def T(m, E, V): #transmission probability
            num = 4 * k1(m, E) * k2(m, E, V)
            den = (k1(m, E) + k2(m, E, V))**2
            return num / den

        def R(m, E, V): #reflection probability
            num = k1(m, E) - k2(m, E, V)
            den = k1(m, E) + k2(m, E, V)
            return (num / den)**2
```

```
In [233]: print("For a particle of mass 9.11e-31 kg and energy 10 eV, it will have \
          transmission probability T = {:4.3f} and \
          reflection probability R = {:4.3f} \
          if it encounters a potential step of height 9 eV.".format(T(9.11e-31, 10, 9), R(9.11e-
```

```
For a particle of mass 9.11e-31 kg and energy 10 eV, it will have transmission probability T = 0
```

0.2 CP 2.6 Planetary Orbits

Knowing the distance that a planet's perihelion and aphelion are from the sun and its linear velocity at one of those points is powerful, allowing one to calculate many facets about the planet's orbit.

Among other things, you can calculate the following

$$\begin{aligned} \text{Semi-major axis:} \quad & a = \frac{1}{2}(\ell_1 + \ell_2), \\ \text{Semi-minor axis:} \quad & b = \sqrt{\ell_1 \ell_2}\,, \\ \text{Orbital period:} \quad & T = \frac{2\pi ab}{\ell_1 v_1}\,, \\ \text{Orbital eccentricity:} \quad & e = \frac{\ell_2 - \ell_1}{\ell_2 + \ell_1}. \end{aligned}$$

```
In [234]: #this defines constants

          M = 1.9891e30 #mass of the sun in kg
          G = 6.6738e-11 # gravitational constant in m^3 kg^-1 s^-2
```

```
In [235]: #defining function to calculate period and eccentricity

          def findv2(l1, v1):
              '''This function calculates the linear velocity of an object at its aphelion given
                 by solving the quadratic equation ax**2 + bx + c = 0'''
              a = 1
              b = (-2*G*M) / (l1 * v1)
```

```
    c = -(v1**2 - ((2*G*M) / l1))

    #the discriminant in the quadratic equation
    disc = b**2 - (4*a*c)

    #only returns 1 solutions because we are only interested in the smaller solution
    return (-b - sqrt(disc)) / (2*a)

def findl2(l1, v1, v2):
    '''Finds distance from Sun to aphelion. Set equal to l2'''
    return (l1*v1) / v2

def semimajor(l1, l2):
    '''Finds semimajor axis of the orbit. Set equal to a.'''
    return 0.5 * (l1+l2)

def semiminor(l1, l2):
    '''Finds semiminor axis of the orbit. Set equal to b.'''
    return sqrt(l1*l2)

def period(l1, v1, a, b):
    '''Find the period of an object orbitting the Sun. Set equal to T.'''
    return (2*pi*a*b) / (l1*v1)

def eccentricity(l1, l2):
    """Finds the orbit's eccentricity. Set equal to ecc."""
    return (l2 - l1) / (l1 + l2)
```

In [236]:
```
def process(l1, v1):
    '''Puts together individual functions to find period and eccentricity at once'''
    v2 = findv2(l1, v1) #units: m/s
    l2 = findl2(l1, v1, v2) #units: m
    a = semimajor(l1, l2) #units: m
    b = semiminor(l1, l2) #units: m

    T = period(l1, v1, a, b) #units: s
    years = T / 31536000
    ecc = eccentricity(l1, l2)

    output = [l2, v2, years, ecc]
    return output
```

In [237]:
```
print("For the Earth...\n the distance to the aphelion is {:5.3f} m \n velocity is {:5
orbital period is {:5.5f} years \n orbital eccentricity is {:5.5f}"\
      .format(process(1.4710e11, 3.0287e4)[0],\
              process(1.4710e11, 3.0287e4)[1],\
              process(1.4710e11, 3.0287e4)[2],\
              process(1.4710e11, 3.0287e4)[3])) #Earth's numbers
```

```
For the Earth...
 the distance to the aphelion is 152027197208.660 m
 velocity is 29305.399 m/s
 orbital period is 1.00022 years
 orbital eccentricity is 0.01647
```

```
In [238]: print("For Halley's comet...\n the distance to the aphelion is {:5.3f} m \n velocity i
          orbital period is {:5.5f} years \n orbital eccentricity is {:5.5f}"\
                .format(process(8.7830e10, 5.4529e4)[0],\
                        process(8.7830e10, 5.4529e4)[1],\
                        process(8.7830e10, 5.4529e4)[2],\
                        process(8.7830e10, 5.4529e4)[3])) #Halley's numbers
```

```
For Halley's comet...
 the distance to the aphelion is 5282214660876.441 m
 velocity is 906.681 m/s
 orbital period is 76.08170 years
 orbital eccentricity is 0.96729
```

0.3 CP 2.9 The Madelung constant

The Madelung constant gives the electric potential in a solid based on the surrounding atoms. This is found by summing the potential due to all atoms as the number of atoms in all directions approaches infinity. The equation relating the Madelung constant to the total potential is

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L \\ \text{not } i=j=k=0}}^{L} V(i,j,k) = \frac{e}{4\pi\epsilon_0 a} M.$$

Thus, we can write (omitting the conditions of the sum for brevity)

$$M = \sum V(i,j,k) \frac{4\pi\epsilon_0 a}{e} = \sum \pm \frac{e}{4\pi\epsilon_0 a\sqrt{i^2+j^2+k^2}} \frac{4\pi\epsilon_0 a}{e} = \pm \frac{1}{\sqrt{i^2+j^2+k^2}}.$$

This allows us to see that the Madelung constant M (for the case where each atom is of unit charge) does not depend on anything other than the relative positions of the atoms to each other, i.e., the spacing between adjacent atoms does not affect its value.

```
In [239]: def Madelung(L):
              '''The entire function for finding the Madelung constant for sodium chloride.
                 L is the number of atoms extending from origin'''
              sum = 0

              for i in range(-L, L):
                  for j in range(-L, L):
                      for k in range(-L, L):
                          if i == 0 and j == 0 and k == 0:
                              pass
```

```
                        else:
                            if (i + j + k) % 2 == 0: #adds all the potentials from sodium atom
                                sum += -1 / (sqrt(i**2 + j**2 + k**2))
                            elif (i + j + k) % 2 == 1: #adds all the potentials from chlorine
                                sum += +1 / (sqrt(i**2 + j**2 + k**2))

            return sum
```

```
In [240]: %%time

          Ls = [10, 100] #values of L to calculate the Madelung constant for

          [print("The Madelung constant for L ={:3.0f} is {:1.3f}. \n".format(x, Madelung(x))) f
```

```
The Madelung constant for L = 10 is 1.748.

The Madelung constant for L =100 is 1.748.

CPU times: user 13.1 s, sys: 181 ms, total: 13.2 s
Wall time: 15.3 s
```

0.4 CP 3.6 Deterministic chaos and the Feigenbaum plot

For certain values of r in the logistic equation

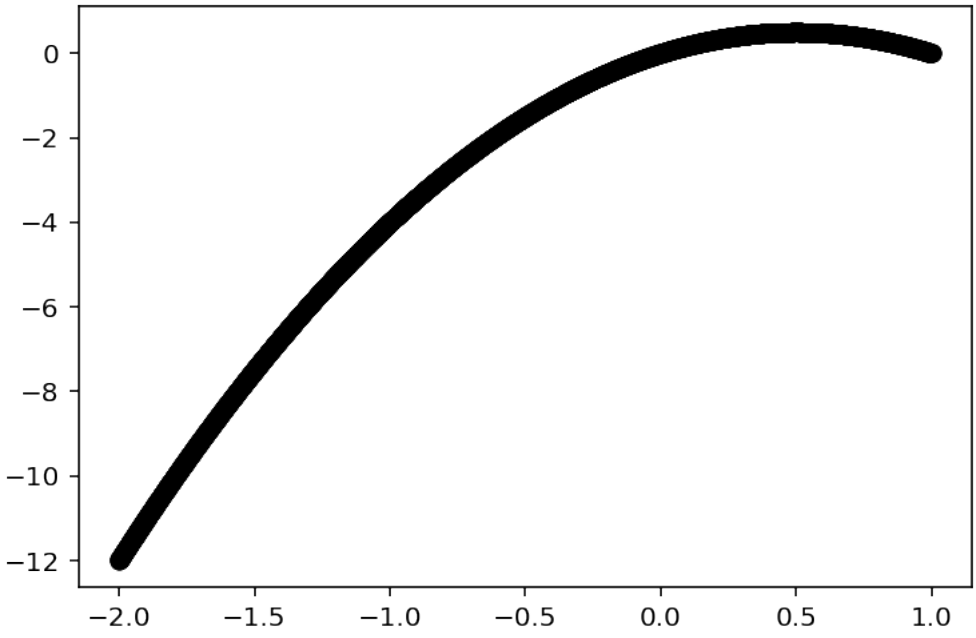
$$x' = rx(1 - x),$$

the iterative map of the results will appear to be random. This is deterministic chaos.

```
In [241]: #the logistic map function

          def logistic(r, x):
              return r * x * (1-x)
```

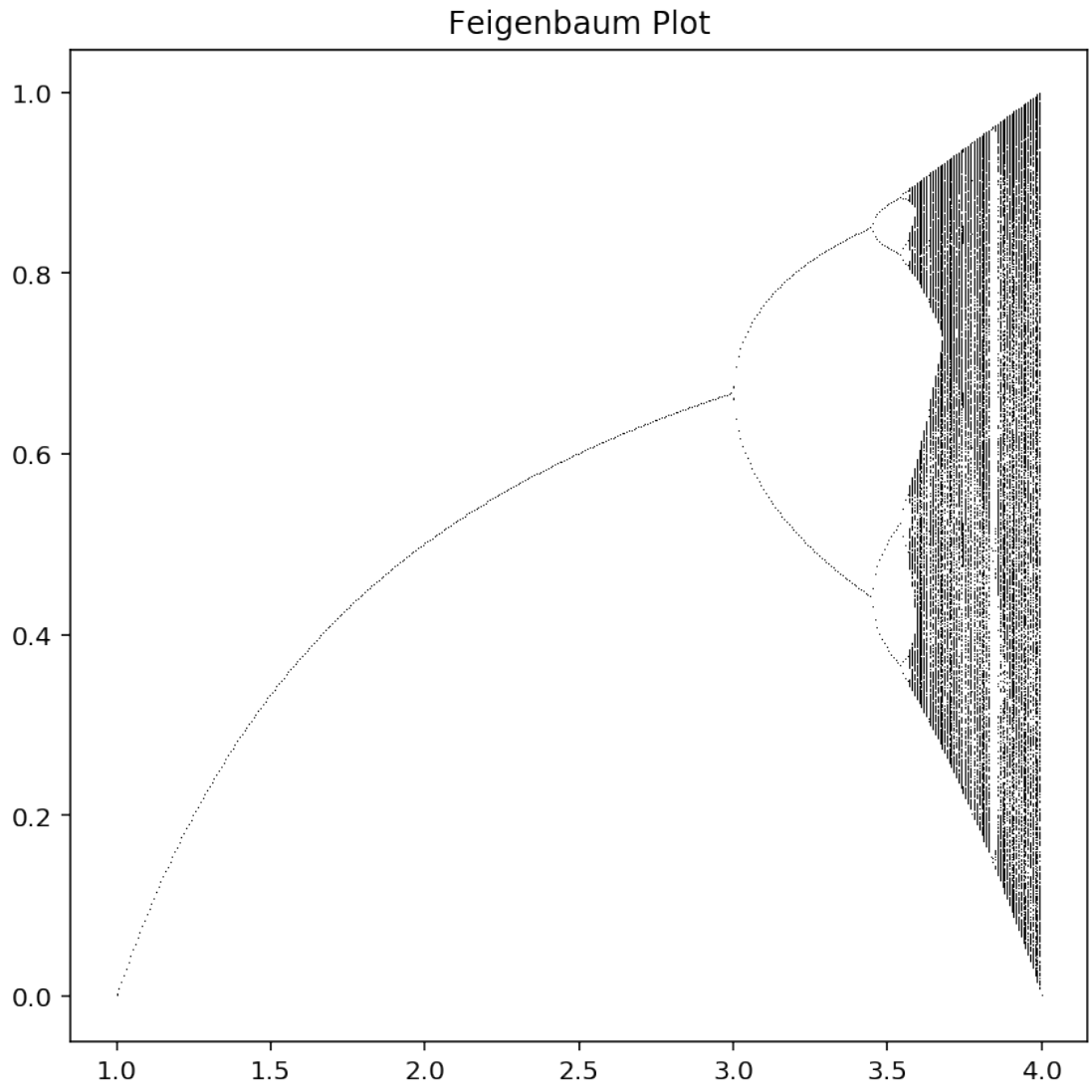
```
In [242]: #x = np.linspace(0, 1, 100)
          fig, axis = pyplot.subplots(1, 1)
          axis.plot(x, logistic(2, x), 'ko')
```

```
Out[242]: [<matplotlib.lines.Line2D at 0x11c23f438>]
```



In [243]:
```
#for a single value of r, starts with x and iterates a certain number of times
def iterate(r, N):
    '''applies logistic map with N iterations to starting value of xi for increasing r

    fig, figplot = pyplot.subplots(1, 1, figsize = (7, 7))
    figplot.set_title("Feigenbaum Plot")

    xi = 0.5 #starting value of x

    for i in range(N):
        step = logistic(r, xi) #utilizes logistic map function to find output
        if i - 1000 >= 0:
            figplot.plot(r, step, ',k')

        xi = step #with this, the output of logistic will become the next input
```

In [244]:
```
r = np.linspace(1, 4, 400) #array of possible r valus
iterate(r, 2000) #calls function
```



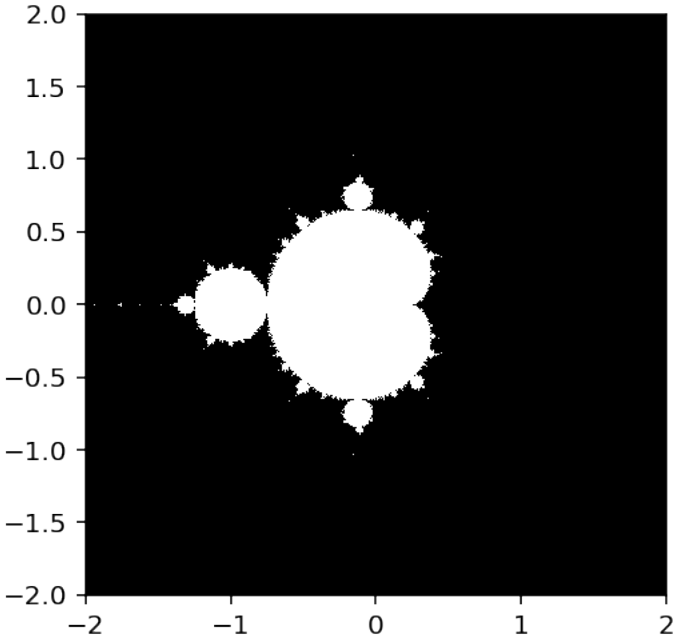
a) For a given r on the Feigenbaum Plot, you get a fixed point if there is only a single x value corresponding to it. Likewise you can read off a limit cycle if there are multiple (but countable) x values to read off after branches. Chaos corresponds to an infinite amount of possible x values, and can be seen as the darker, almost fully-shaded regions.

b) The edge of chaos appears to be at r = 3.5. The logistic map transitions from a fixed point to limit cycles at r = 3.0 and splits into double the number of branches a few times before r = 3.5. After this, you start to see full shading along a vertical line passing through a given r value.

0.5 CP 3.7 The Mandelbrot set

Like the logistic map, the Mandelbrot set is recursively applied. If after some large number of iterations, $|z'| < 2$ for every one of the iterations, then that point is in the set.

```
In [245]: def mandelbrot(z, c): #defines the mandelbrot function
              return pow(z, 2) + c

In [246]: def plot_mandelbrot(N, threshold, grid):
              '''This function will iterate the mandelbrot function <N> times,
                 compare the magnitude of the resulting complex number to <threshold>,
                 then plot values under the threshold on a specified size <grid>.'''

              #fig, ax = pyplot.subplots(1, 1, figsize = (10, 10))
              #ax.imshow(-----)
              x, y = np.ogrid[-2:2:1j*grid, -2:2:1j*grid] #creates the grid

              c = x + y*1j #defines c as a complex number based on location on grid
              z = 0 #starting value of z

              #iterates Mandelbrot function set number of times
              for i in range(N):
                  z = mandelbrot(z, c)
              #[z = mandelbrot(z, c) for i in range(N)]

              #the values that are in the Mandelbrot set
              inset = np.abs(z) < threshold

              pyplot.imshow(inset.T, extent = [-2, 2, -2, 2], aspect = 'equal')# ,cmap = 'jet')
              #pyplot.gray
              #ax = pyplot.figure(figsize = (10,10))

              color = np.ogrid[-2:2:1j*grid, -2:2:1j*grid] #initializing array for colors

              pyplot.show()

In [247]: plot_mandelbrot(100, 2, 4000)

/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:2: RuntimeWarning: overf

/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:2: RuntimeWarning: inval

/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:19: RuntimeWarning: inva
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2
−1
0
1
2