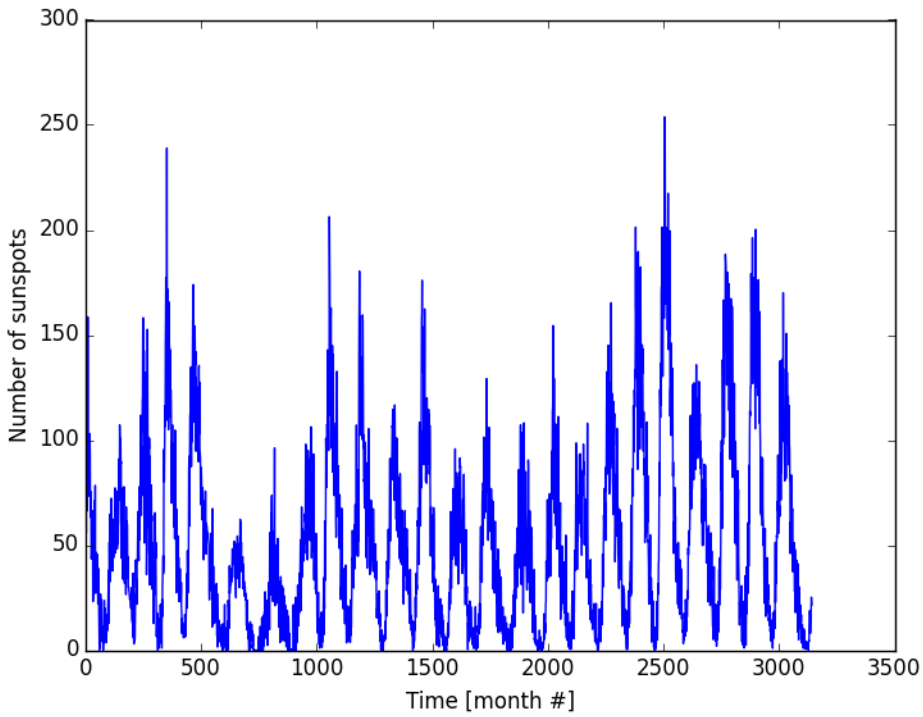
Computational Physics HW2

Luke Bouma

July 27, 2015

1 Plotting experimental data

1.1 Plotting `sunspots.txt` in Python:

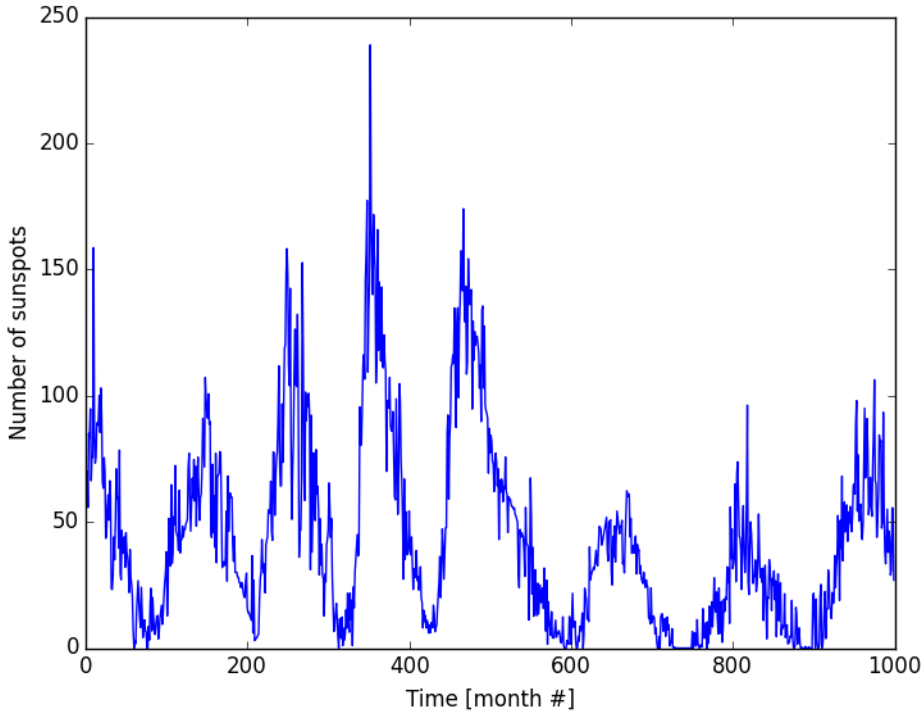


The figure above is the output from

```
from numpy import loadtxt
import matplotlib.pyplot as plt

data = loadtxt("sunspots.txt",float)
time = data[:,0]
sunSpotN = data[:,1]

plt.xlabel("Time [month #]")
plt.ylabel("Number of sunspots")
```

```
plt.plot(time, sunSpotN)
plt.show()
```

1.2 Modifying the program to display the first 1000 data points:



Where the only output lines that were changed are:

```
time = data[0:1000,0]
sunSpotN = data[0:1000,1]
```

1.3 Calculating running average of the data.

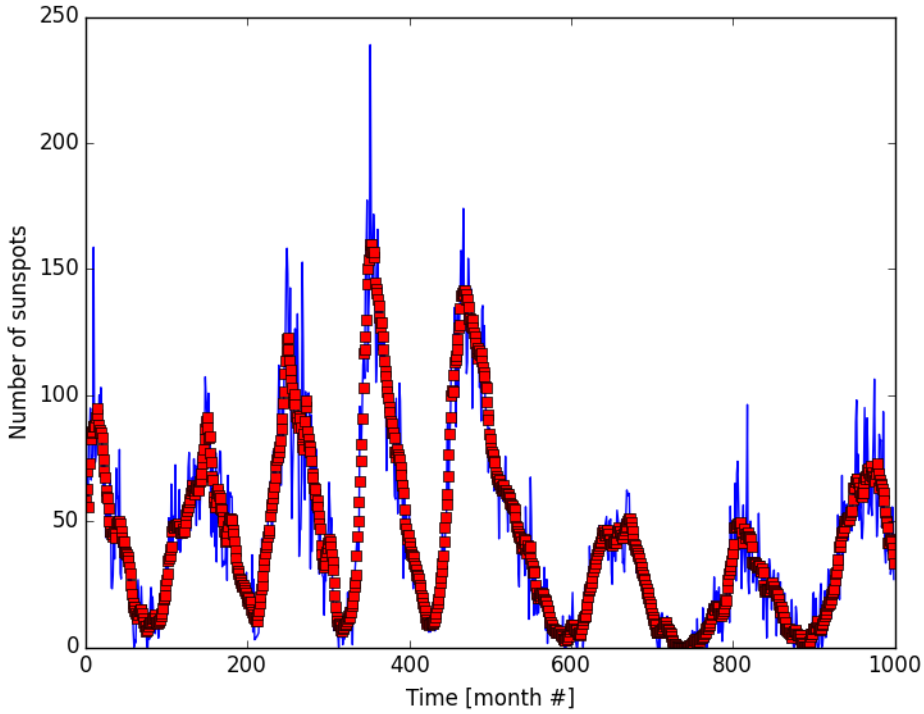
Define the running average to be

$$Y_k = \frac{1}{2r} \sum_{m=-r}^{r} y_{k+m},$$

where $r = 5$ and y_k are the sunspot numbers. This wound up being a bit messier than I wanted, but it still makes good sense:

```
#Import
from numpy import loadtxt, sum, zeros, size
import matplotlib.pyplot as plt

data = loadtxt("sunspots.txt",float)
time = data[:,0]
sunSpotN = data[:,1]
```

```
#Smoothing
r = 5
Y = zeros(size(sunSpotN))
for k in range(size(sunSpotN)):
    if k < 5 or k > 3100:
        Y[k] = sunSpotN[k]
    else:
        for m in range(k-5,k+5,1):
            Y[k] += 1/(2*r) * sunSpotN[m]

#Plotting
monthN = 1000
plt.xlabel("Time [month #]")
plt.ylabel("Number of sunspots")
plt.plot(time[0:monthN], sunSpotN[0:monthN], 'b-',
         time[0:monthN], Y[0:monthN], 'rs')
plt.show()
```



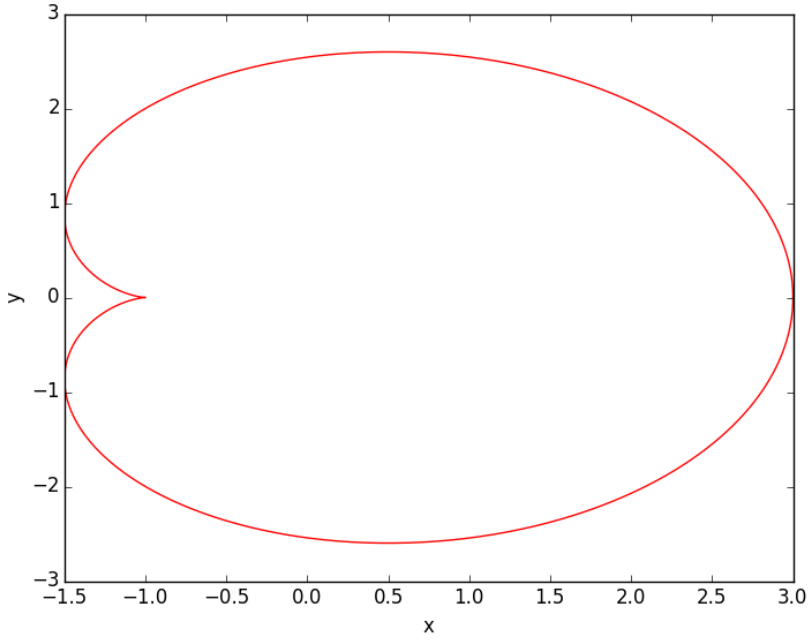


Figure 1: Daw, I love you too!

2 Curve plotting

2.1 Deltoid curve plot

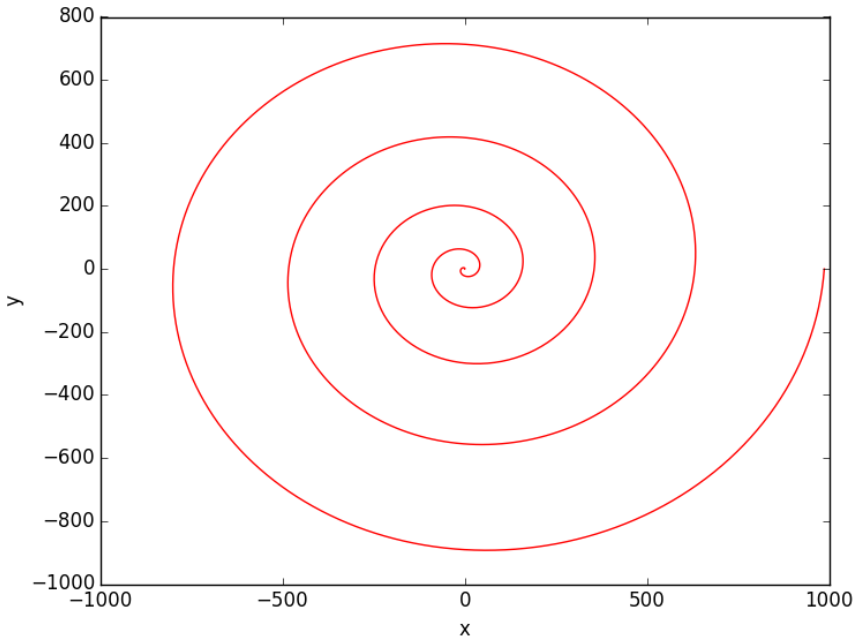
Plot the parametric equations

$$x = 2\cos\theta + \cos 2\theta, \quad y = 2\sin\theta - sin2\theta$$

where $0 \leq \theta < 2\pi$. Take a set of values of θ between zero and 2π, and calculate x and y for each from the equations above, then plot y as a function of x. We get Fig. 1 from the script

```
from math import sin, cos, pi
import matplotlib.pyplot as plt
from numpy import linspace, size, zeros

len = 100
x, y = zeros(len), zeros(len)
theta = linspace(0,2*pi,len)
for j in range(0,size(theta)):
    x[j] = 2*cos(theta[j]) + cos(2*theta[j])
    y[j] = 2*sin(theta[j]) + sin(2*theta[j])

plt.xlabel("x")
plt.ylabel("y")
plt.plot(x, y, 'bs')
plt.show()
```

2.2 Plotting the Galilean spiral, $r = \theta^2$ for $0 \leq \theta \leq 24\pi$.



is generated from

```
from math import sin, cos, pi
import matplotlib.pyplot as plt
from numpy import linspace, size, zeros

len = 1000
x, y, r = zeros(len), zeros(len), zeros(len)
theta = linspace(0,10*pi,len)
for j in range(0,size(theta)):
    r[j] = theta[j]**2
    x[j] = r[j]*cos(theta[j])
    y[j] = r[j]*sin(theta[j])

plt.xlabel("x")
plt.ylabel("y")
plt.plot(x, y, 'r-')
plt.show()
```

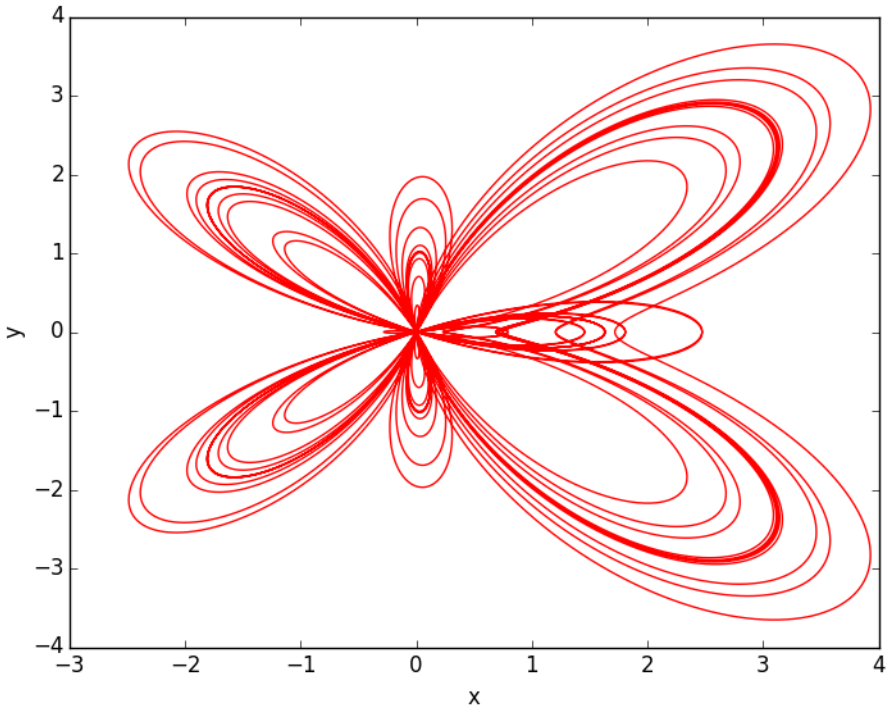
2.3 Fey's function polar plot

Using the script

```
from math import sin, cos, exp, pi
import matplotlib.pyplot as plt
from numpy import linspace, size, zeros
```

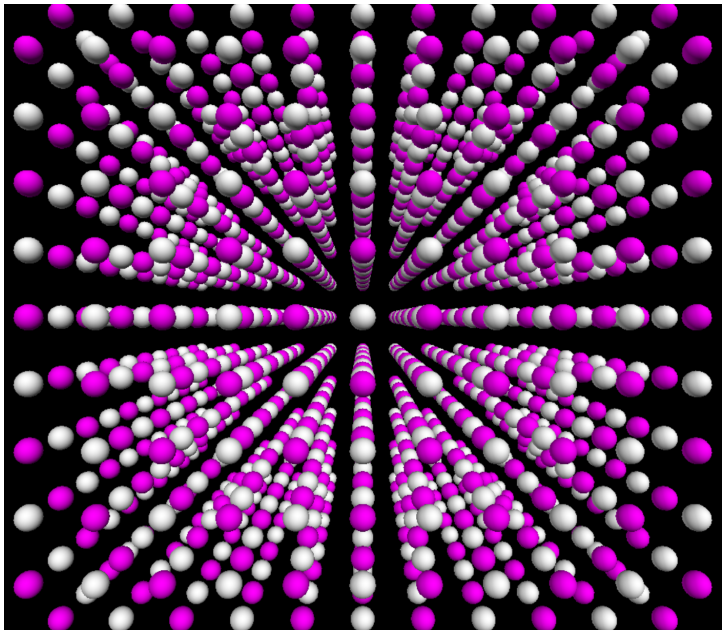
```
len = 100000
x, y, r = zeros(len), zeros(len), zeros(len)
theta = linspace(0,24*pi,len)
for j in range(0,size(theta)):
    r[j] = exp(cos(theta[j])) - 2*cos(4*theta[j]) + \
           sin(theta[j]/12)**5
    x[j] = r[j]*cos(theta[j])
    y[j] = r[j]*sin(theta[j])

plt.xlabel("x")
plt.ylabel("y")
plt.plot(x, y, 'r-')
plt.show()
```

we generate



3 Plotting atomic lattice (VPython)

This is the one that I'm skipping/avoiding because I already did it in the notes, to generate



which is an approximate NaCl lattice. Doing a FCC (face-centered cubic) lattice would be a question of adding another atom $\sqrt{2}a/2$ offset on the diagonal from the central atom.

4 Deterministic chaos and the Feigenbaum plot

Consider the *logistic map*, defined by

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

For a given constant r take a value of x, say $x = 1/2$, and feed it into the RHS of this equation to get x'. Then take the value you get, feed it back in, and so on. I.e., you *iterate*. One of three things will happen:

1. The value settles down to a fixed number, and stays there. For instance, $x = 0$ is one such fixed point of the logistic map.

2. You never settle to a single value, but instead settle into a periodic pattern, rotating around a set number of values, repeating them in sequence forever. This is a *limit cycle*.

3. It goes crazy. It generates a seemingly random sequence of numbers that appear to have no rhyme or reason to them at all. This is *deterministic chaos*, in the sense that it really does look chaotic, but deterministic because we know that the values have a rule behind them. The behavior is determined, it just isn't obvious that this is the case.

We write a program to calculate and display the behavior of the logistic map:

```
import matplotlib.pyplot as plt
from numpy import linspace, size

#Returns result of n operations of logistic map on x, given r
def logisticMap(r, x, n):
    i, xPrime = 0, 0.
```

```
    while i < n:
        xPrime = r*x*(1-x)
        x = xPrime
        i += 1
    return xPrime

#Returns list of latter 1000 mappings to plot
def getXlist(r, x):
    i = 0
    while i < 250:
        x.append(logisticMap(r, x[size(x)-1], 1))
        i += 1
    return x

xMin, xMax = 3.84, 3.86
for r in linspace(xMin, xMax, 501):
    print(r)
    x = [logisticMap(r, 1/2, 1000)]
    x = getXlist(r,x)
    for i in range(size(x)):
        plt.scatter(r, x[i], 1)
plt.xlabel("r")
plt.ylabel("x (Orbit)")
plt.title("Logistic Map (Feigenbaum Plot)")
plt.axis([xMin,xMax,0,1])
plt.show()
```

which, once plotted in Figs. 2 and 3 gives pretty nice results. Answering the questions:

a) Fixed points look like single orbits (values of x) corresponding to single values of r. Limit cycles are when two or more values of x correspond to a single value of r. Chaos is when a seemingly random large number of x points are given by a single value of r.

b) The switch from orderly (fixed points, limit cycles) to chaotic happens roughly at $r = 3.56995$ (obviously, wiki).

Aside: This deterministic chaos is seen in more complex physical systems too, most notably fluid dynamics and the weather. We get an idea through these plots of what chaotic behavior means: change initial conditions by just a little, and you'll get a rapid onset of highly diverse, hard-to-predict results. The classic example is Edward Lorenz's butterfly effect (although scifi writer Ray Bradbury may have suggested the idea in a story 25 years earlier).

We can definitely go further than these two figures though. Generating them with roughly 10^6 points took upwards of ten minutes on my laptop, and we should be able to do much better. Noting the hint at the end of the problem to vectorize the code, we rewrite the whole program and find nicer results:

```
import matplotlib.pyplot as plt
from numpy import linspace, array, vstack, full, copy

#Returns result of n operations of logistic map on x, given r
```

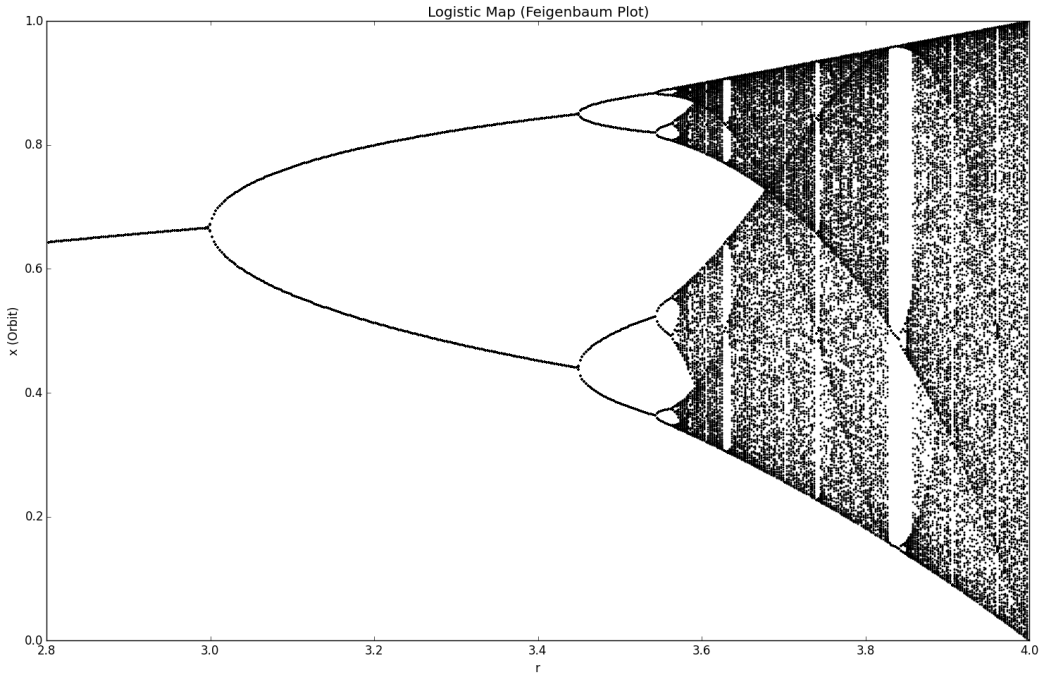


Figure 2: The logistic map, $x_{n+1} = rx_n(1 - x_n)$, for $2.8 \leq r \leq 4.0$.

```
def logisticMap(r, x, n):
    i, xPrime = 0, 0.
    while i < n:
        xPrime = r*x*(1-x)
        x = xPrime
        i += 1
    return xPrime

#Returns full matrix of x values with each map iteration for numX
def getMatrix(r, x, numX):
    i = 0
    while i < numX:
        if i == 0:                      #obnoxious, but seemed necessary
            x = vstack((x, logisticMap(r, x, 1)))
        else:
            x = vstack((x[0:i,:], logisticMap(r, x[i-1,:], 1)))
        i += 1
    return x

numR = 1000       #number of r values
numX = 1000       #number of x values for each r value
rMin, rMax = 2.2, 3      #r between min and max
alp = 0.02        #from 0 to 1, changes contrast of plotted pts
```

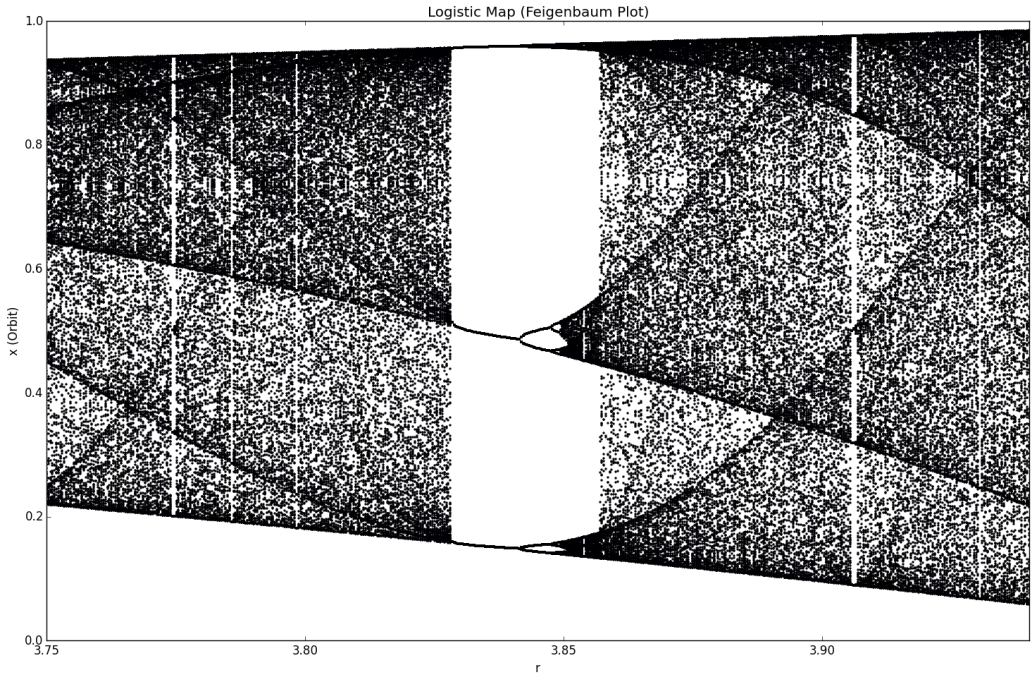


Figure 3: The logistic map in the first large island of stability, which begins at $1 + \sqrt{8}$.

```
r = linspace(rMin, rMax, numR)
x = array([1/2] * numR)

x = logisticMap(r, x, 1000) #stabilize starting values over 1000 iterations
x = getMatrix(r, x, numX) #get the data of x orbits for each r (vectorized)

print("Got matrix, now plotting.")

rArr = full((numX,numR), 1.) #used to vectorize scatter plot
for i in range(numX):
    rArr[i,:] = copy(r)

plt.xlabel("r")
plt.ylabel("x (Orbit)")
plt.title("Logistic Map (Feigenbaum Plot)")
plt.axis([rMin,rMax,0,1])
plt.scatter(rArr, x, s=4, alpha=alp)
plt.show()
```

This program runs like, at least a factor of 100 faster than the other. So obviously we push it further, and make more plots:

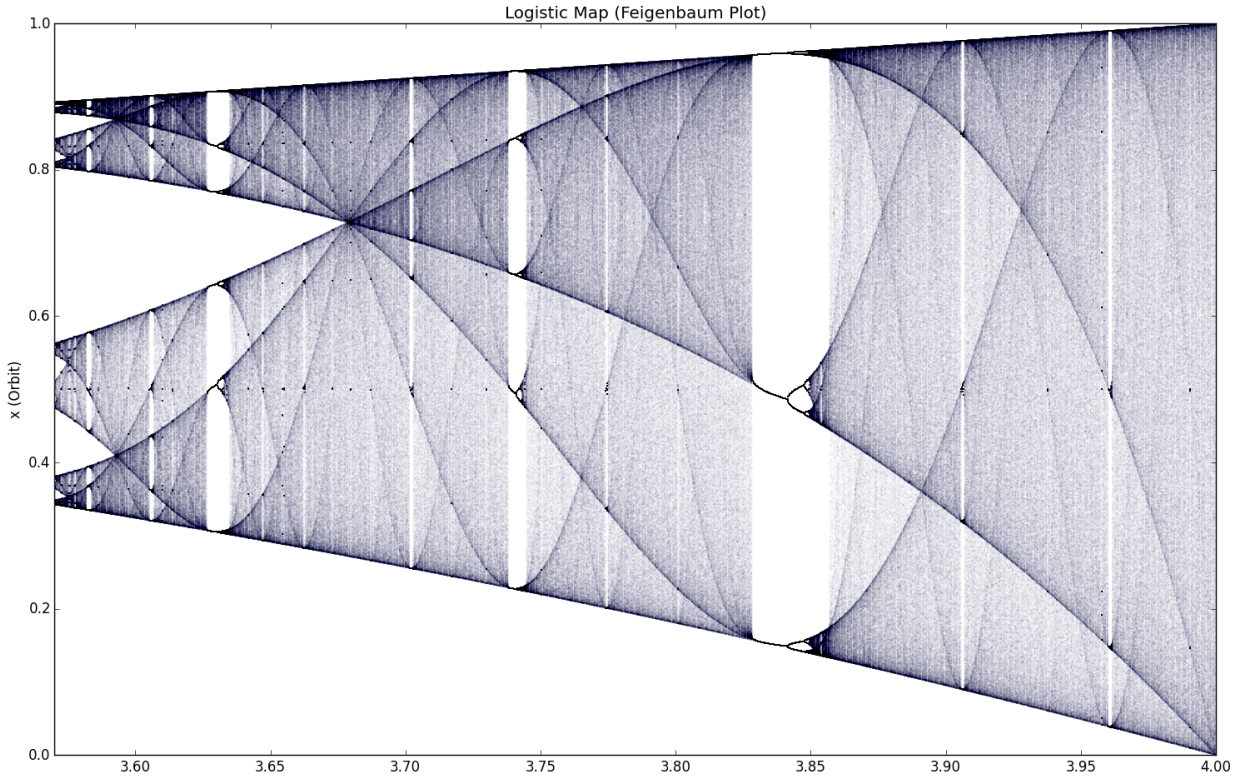


Figure 4: More points.

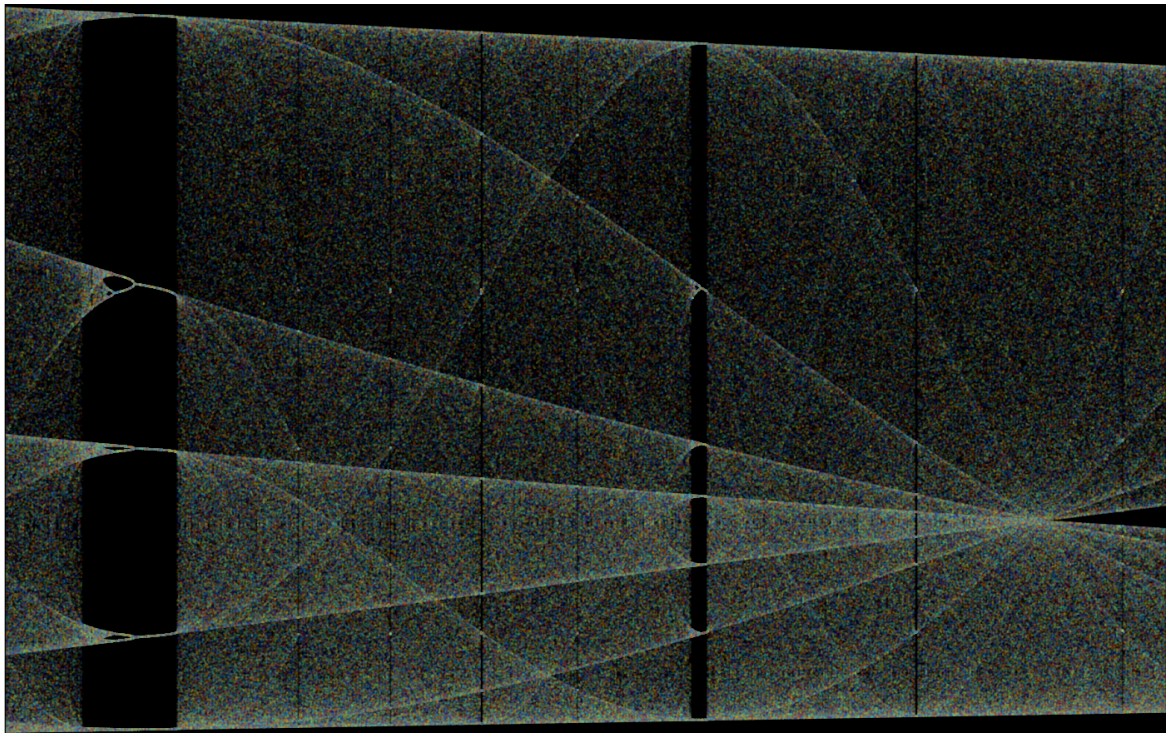


Figure 5: Getting artistic.

5 Least squares and the photoelectric effect

This is the method of least squares. If we have N data points with coordinates (x_i, y_i), then the sum of the squares of the distances between some guess line $y = mx + c$ and the points is

$$\chi^2 = \sum_{i=1}^{N} (mx_i + c - y_i)^2 = \sum_{i=1}^{N} m^2 x_i^2 + c^2 + y_i^2 + 2mx_i c - 2mx_i y_i - 2cy_i.$$

The least squares fit is the line that minimizes χ^2. Find this minimum by differentiating with respect to both m and c, and setting the derivatives to zero:

$$\begin{aligned} m\sum_{i=1}^{N} x_i^2 + c\sum_{i=1}^{N} x_i - \sum_{i=1}^{N} x_i y_i &= 0 \\ m\sum_{i=1}^{N} x_i + cN - \sum_{i=1}^{N} y_i &= 0. \end{aligned}$$

For convenience, define

$$E_x = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad E_y = \frac{1}{N}\sum_{i=1}^{N} y_i, \quad E_{xx} = \frac{1}{N}\sum_{i=1}^{N} x_i^2, \quad E_{xy} = \frac{1}{N}\sum_{i=1}^{N} x_i y_i,$$

so that we can rewrite our equations

$$\begin{aligned} mE_{xx} + cE_x &= E_{xy} \\ mE_x + c &= E_y. \end{aligned}$$

Solving simultaneously for m and c gives

$$m = \frac{E_{xy} - E_x E_y}{E_{xx} - E_x^2}, \quad c = \frac{E_{xx}E_y - E_x E_{xy}}{E_{xx} - E_x^2}.$$

These are the equations for the least-squares fit of a straight line to N data points. They give the values of m and c for the last line that best fits the given data.

5.1 Millikan: read data points, make graph

```
from numpy import loadtxt
import matplotlib.pyplot as plt

data = loadtxt("millikan.txt", float)
x = data[:, 0]
y = data[:, 1]

plt.plot(x, y, 'k.')
plt.show()
```

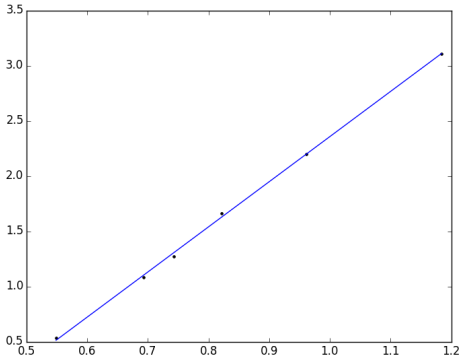
5.2 Find $E_y, E_x x, E_x y$ and print m,c of best-fit line

```
N = size(x)
E_x = 1/N * sum(x)
E_y = 1/N * sum(y)
E_xx = 1/N * sum(x**2)
E_xy = 1/N * sum(x*y)

m = (E_xy - E_x*E_y) / (E_xx * E_x**2)
c = (E_xx*E_y - E_x*E_xy) / (E_xx - E_x**2)

print("m=%.3f" % m, "c=%.3f" % c)
```

5.3 Draw straight line

Draw the straight line by evaluating $mx_i + c$ using the values of m, c from the previous part. Store these values to a list or array, then graph this array as a solid line. This gives a program:

```
from numpy import loadtxt, size
import matplotlib.pyplot as plt

data = loadtxt("millikan.txt", float)
x = data[:, 0]/1e15
y = data[:, 1]

N = size(x)
E_x = sum(x) / N
E_y = sum(y) / N
E_xx = sum(x**2) / N
E_xy = sum(x*y) / N

m = (E_xy - E_x*E_y) / (E_xx - E_x*E_x)
c = (E_xx*E_y - E_x*E_xy) / (E_xx - E_x**2)
bestFit = m*x + c
print("m=%.3f" % m, "c=%.3f" % c)
plt.plot(x, y, 'k.')
plt.plot(x, bestFit)
plt.show()
```



5.4 Calculating Planck's constant

Given that this data is actually of a photoelectric experiment, where the voltage required to stop ejected electrons, V, is measured as a function of ν, the frequency of incoming light, so our equation $y = mx + c$ actually is $V = \frac{h}{e}\nu - \phi$, so that $y = V, m = h/e, c = -\phi$.

We are given that $e = 1.602 \times 10^{-19}$C, and since we have $m = 4.088 \times 10^{-15}\text{VHz}^{-1}$, we can calculate

$$\begin{aligned} h = m * e &= 4.088 \times 10^{-15} \times 1.602 \times 10^{-19} \\ &= 6.549 \times 10^{-34}\text{VCs} \\ &= 6.549 \times 10^{-34}\text{Js} \end{aligned}$$

which is indeed within a few percent of the literature value.

6 Trapezoidal rule and Romberg rule for integrals

Evaluate

$$I = \int_0^1 \sin^2 \sqrt{100x}\ \mathrm{d}x$$

6.1 Use the trapezoidal rule to calculate it to accuracy $\varepsilon = 10^{-6}$

The approximation used in the trapezoidal rule integrating f over $[a, b]$ is

$$\begin{aligned} I &\approx h\left(\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{k=1}^{N} f(a + kh)\right), \quad \text{for} \\ \varepsilon &= \frac{1}{12}h^2(f'(a) - f'(b)). \end{aligned}$$

For us, $f(x) = \sin^2\sqrt{100x}$, so $f'(x) = \frac{d}{dx}(\sin\sqrt{100x} \cdot \sin\sqrt{100x}) = 10x^{-1/2}\cos\sqrt{100x}\sin\sqrt{100x}$, which simplified is $5\sin(20\sqrt{x})/\sqrt{x}$ and to avoid dividing by zero,

$$\begin{aligned} 5\sin(20\sqrt{x})/\sqrt{x} &= \frac{5}{\sqrt{x}}\left(20\sqrt{x} - \frac{(20\sqrt{x})^3}{3!} + \frac{(20\sqrt{x})^5}{5!} - \frac{(20\sqrt{x})^7}{7!} + \frac{(20\sqrt{x})^9}{9!} + O(x^{11/2})\right) \\ &= 5\left(20 - \frac{20^3 x}{3!} + \frac{20^5 x^2}{5!} - \frac{20^7 x^3}{7!} + \frac{20^9 x^4}{9!} - \frac{20^{11} x^5}{11!} + O(x^5)\right) \end{aligned}$$

which we implement to be accurate up to $O(x^6)$ to avoid the division by zero. The program we write is

```
from numpy import sin, sqrt

def factorial(x):
    if x == 1:
        return 1
    else:
        return x*factorial(x-1)
```

```
def f(x):
    return sin(sqrt(100*x))**2

def fPrime(x):
    return 5*(20 - 20**3*x/factorial(3)
              + 20**5*x**2/factorial(5)
              - 20**7*x**3/factorial(7)
              + 20**9*x**4/factorial(9)
              - 20**11*x**5/factorial(11))

def eps(h, a, b):
    return h**2 * (fPrime(a) - fPrime(b)) / 12

a, b, I = 0, 1, 0.

for i in range(50):
    N = 2**i
    h = (b-a)/N
    error = eps(h, a, b)
    print(i, h, error)
    if error < 1e-6:
        I = h*(f(a)/2 + f(b)/2)
        for k in range(1,N):
            I += h*f(a+k*h)
        break

print(I)
```

which results $I = 0.455832532307$, good to Mathematica's result to 10 decimals!

6.2 Romberg technique

Honestly I can't be bothered here. Looks pretty boring, and we can just do the higher-order Gaussian stuff in any case.

7 Heat capacity of a solid

Debye's theory of solids gives the heat capacity of a solid at temperature T to be

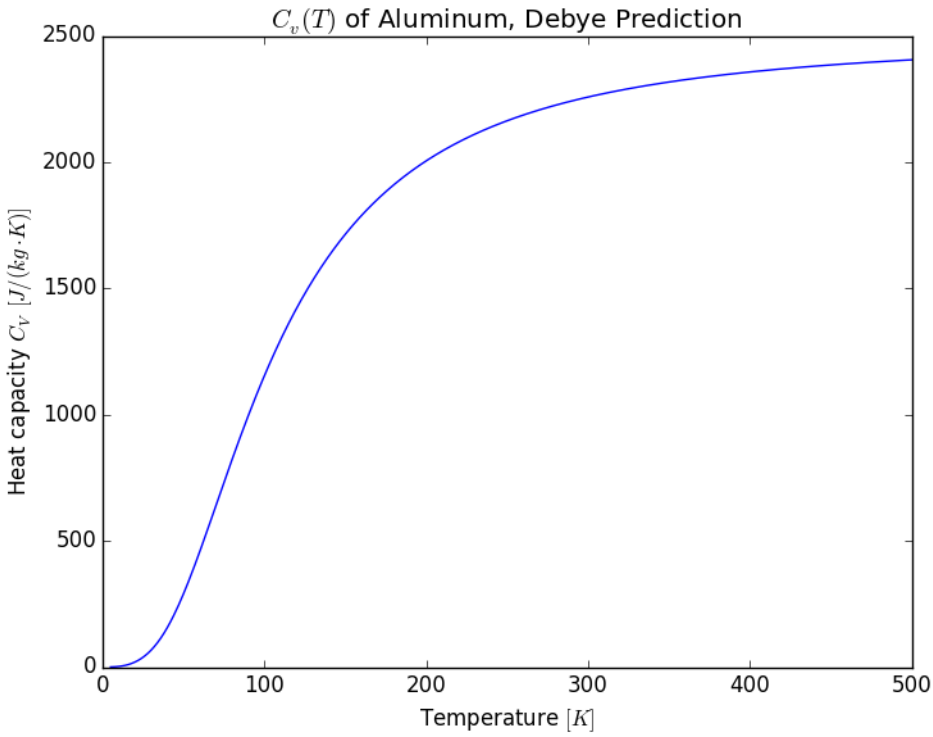
$$C_V = 9V\rho k_B \int_0^{\theta_D/T} \frac{x^4 e^x}{(e^x-1)^2}\,\mathrm{d}x,$$

for V the volume of the solid, ρ the number density of atoms, and θ_D the Debye temperature, a property of solids that depends on their density and the speed of sound.

a) Write a function `cv(T)` that calculates C_V for a given value of the temperature, for a sample of 1000 $\mathrm{cm}^3 = 10^{-3}$ m^3 of solid Al, which has number density $\rho = 6.022 \times 10^{28}\mathrm{m}^{-3}$, and $\theta_D = 428\mathrm{K}$. Use Gaussian quadrature to evaluate the integral, with $N = 50$ sample points.

```
from numpy import exp, linspace
from gaussxw import gaussxw, gaussxwab
import matplotlib.pyplot as plt

def f(x):
    return x**4 * exp(x) / (exp(x)-1)**2

def cv(T):
    V = 1e-3                #m**3
    rho = 6.022e28          #1/m**3
    kB = 1.38065e-23        #J/K
    thetaD = 428
    prefac = 9*V*rho*kB*(T/thetaD)**3

    #Get points and weights for integral
    N = 50
    a, b = 0., thetaD/T
    xp, wp = gaussxwab(N, a, b)
    #Perform integration
    s = 0.
    for k in range(N):
        s += wp[k]*f(xp[k])
    #print(s)
    return prefac * s

T = linspace(5,500,496)
cvList = []
for i in range(496):
    cvList.append(cv(T[i]))

plt.plot(T, cvList)
plt.xlabel('Temperature $[K]$')
plt.ylabel('Heat capacity $C_V$ $[J/(kg\cdot K)]$')
plt.title('$C_v(T)$ of Aluminum, Debye Prediction')
plt.show()
```

gives the figure


$C_v(T)$ of Aluminum, Debye Prediction
Heat capacity C_V $[J/(kg \cdot K)]$
Temperature $[K]$
0
500
1000
1500
2000
2500
0
100
200
300
400
500