PHYS CS15A: Homework 1

Frank Lee

October 10, 2017

1 example_plot.py

Consider the following three figures produced from example_plot.py. The algorithms and code used will be explained afterwards.

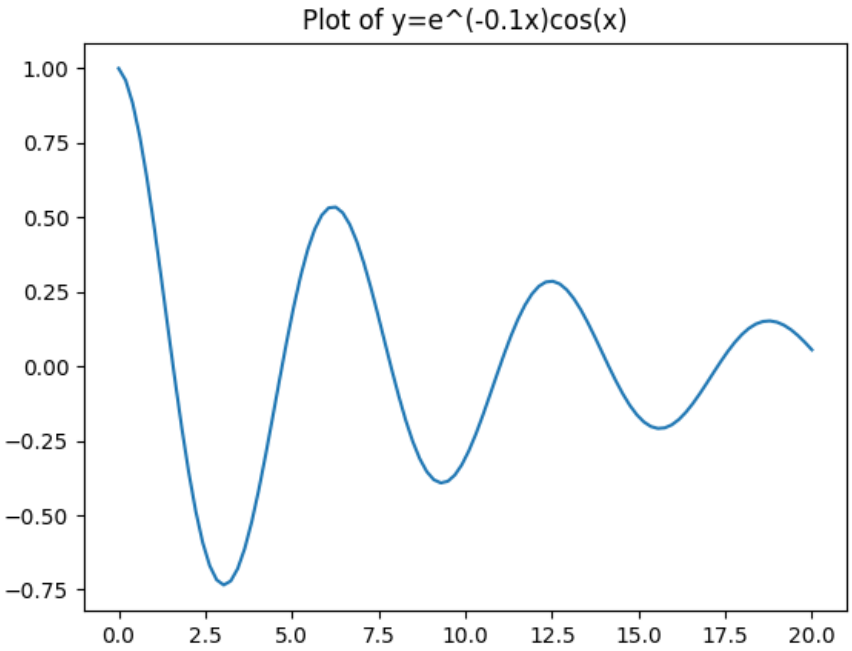

Figure 1: Figure 1 produced by example_plot.py

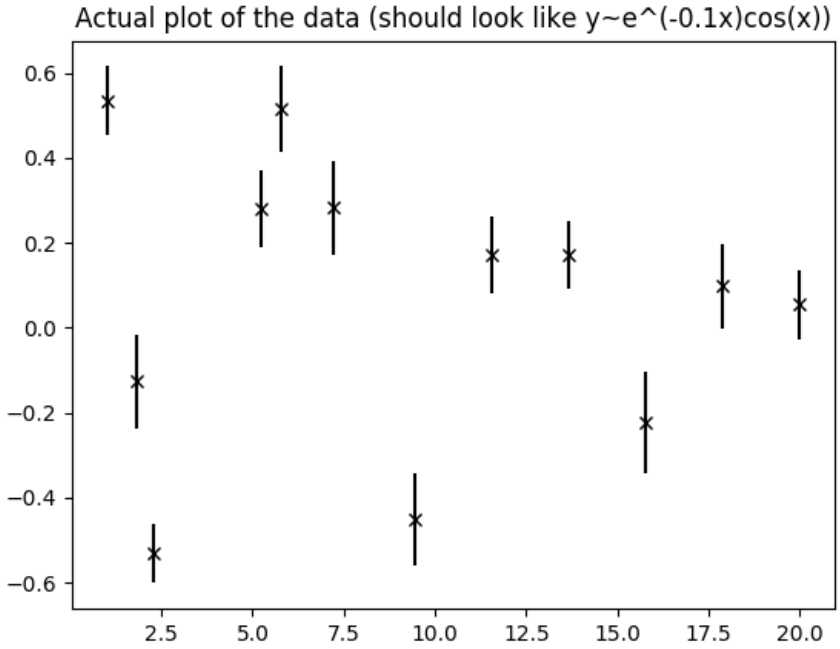

Figure 2: Figure 2 produced by example_plot.py

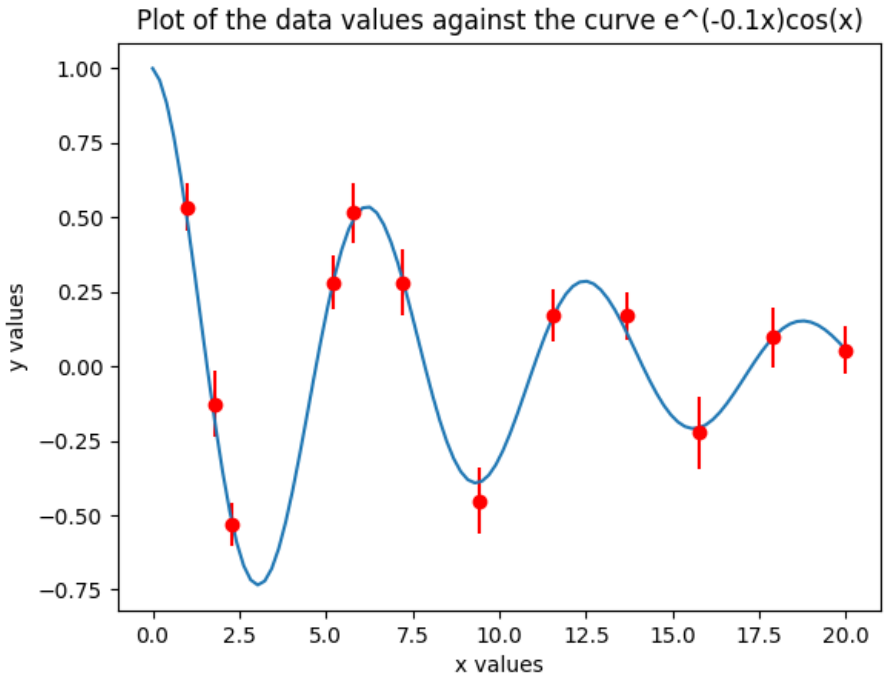

Figure 3: Figure 3 produced by example_plot.py

Figure 1 is simply a plot of the function $e^{-0.1x}\cos x$ where $x \in [0, 20]$. The function was plotted by standard means from Python's `matplotlib` library:

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

x_values = np.linspace(0,20,100)
y_values = (np.exp(-0.1*x_values))*(np.cos(x_values))
plt.figure(1)
plt.title("Plot of y=e^(-0.1x)cos(x)")
plt.plot(x_values, y_values)
```

Listing 1: Plotting figure 1

Figure 2 is a plot of just the data values[1], provided by the external file data.txt. We therefore must pull the data from our external file via the lines:

```
ExperimentInput = np.loadtxt("data.txt", delimiter=",")
Experiment = np.transpose(ExperimentInput)

x_data = Experiment[0]
y_data = Experiment[1]
y_error = Experiment[2]

plt.figure(2)
plt.errorbar(x_data, y_data, yerr=y_error, fmt='kx')  #plotting the
   data points
plt.title("Actual plot of the data (should look like y~e^(-0.1x)cos(
   x))")
```

Listing 2: Plotting figure 2

We then superimpose our data with the actual plot of $e^{-0.1x}\cos x$. We simply just reuse the lines we had for figures 1 and 2 such that we have:

```
plt.figure(3)
plt.plot(x_values, y_values)
plt.errorbar(x_data, y_data, yerr=y_error, fmt='ro')
```

Listing 3: Plotting figure 2

with of course our title and axis labels.

[1]That I fabricated so that it looks like the data values 'fit' $e^{-0.1x}\cos x$

2 RateT.py

Arrhenius's Equation relates the rate constant k of a particular chemical reaction to its absolute temperature, T by the equation

$$k = Ae^{-E_a/RT} \tag{1}$$

where A is a prefactor constant, E_a is the activation energy and R is the gas constant[2]. Taking the natural log of equation (1) gives:

$$\ln k = -\frac{E_a}{R}\frac{1}{T} + \ln A \tag{2}$$

Equation (2) implies that plotting $1/T$ against $\ln k$ gives a straight line. We will use this method to determine the activation energy and the prefactor constant.

2.1 Part (a)

We need the absolute temperature instead of temperature in degrees celsius. This conversion can be done by the following:

```
RawInputValues=np.loadtxt("RateT.txt",delimiter=",")
InputValues=np.transpose(RawInputValues)
TempKelvin=InputValues[0]+273.15              #converting from
   Celsius to Kelvin
RateConstant=InputValues[1]
```

Listing 4: Establishing our data points from RateT.txt

2.2 Parts (b), (c) and (d)

As indicated by equation (2), we must plot $1/T$ against $\ln k$ in order to get a straight line. Therefore, we plot:

```
plt.figure(1)
plt.errorbar(1/TempKelvin, np.log(RateConstant), fmt="ro")
```

Listing 5: Plotting RateT.txt in an Arrhenius plot

We can also include a linear best fit using `numpy.polyfit()`. Define m and b to be the slope and y-intercept of our best fit respectively. We can then have

```
m,b=np.polyfit(1/TempKelvin, np.log(RateConstant),1)     #Assuming a
   linear best fit
plt.plot(1/TempKelvin,m*(1/TempKelvin)+b,'-')
```

Listing 6: Determining and plotting the line of best fit (linear regression)

With this, we can plot our data points along with our line of best fit such that

[2] $R \approx 8.314$ $\text{JK}^{-1}\text{mol}^{-1}$

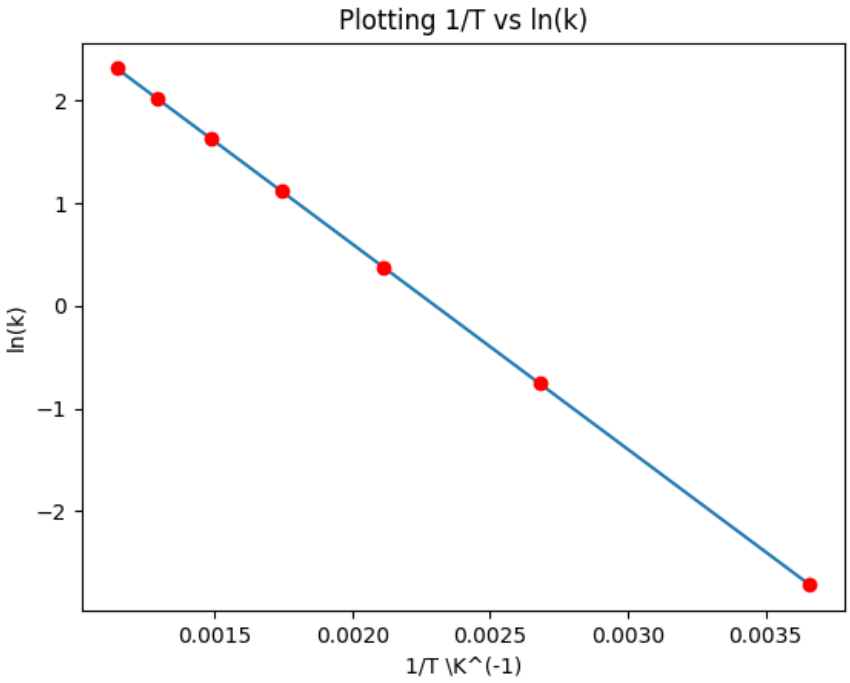

Figure 4: Figure 1 produced by example_plot.py

We can just retrieve our m and b value by simply executing `print(m,b)`, which shows us $m = -2001.529...$ and $b = 4.606....$ Therefore, $E_a = -R \times -2001.529 = 16.6$ kJ and $A = \exp 4.606 = 100.2$. Therefore:

$$\boxed{E_a = 16.6 \text{ kJ}} \quad \boxed{b = 100.2} \tag{3}$$

3 HeatCapacity.py

3.1 Part (a)

We are given that the specific heat capacity, C is related to the temperature by the following relationship

$$C = aT + bT^3 \tag{4}$$

We can just rewrite this as

$$C = a(T + bT^3) \tag{5}$$

where $a, b \in \mathbb{R}$ are different constants than that from (4). We want to rewrite (4) so that it looks like (5) since given the 'correct' or best fit value of b, we know that:

$$\frac{C}{(T + bT^3)} = a \tag{6}$$

In other words, this ratio must be constant given that we have found the 'right' value of b. This is the cornerstone of the method we will use. We will first guess our value of b by iterating b through a range of say $[0, 5]$ in arbitrary small increments. For each guess we have for b, we then compute the ratio from (6) using the data values we are given from HeatCapacity.txt and place them in an array (or list in Python).

After completing our iteration, we will then have a whole array of constants a for each guess value of b. Provided that we have actually iterated through the correct b, we should in practise arrive at an array of constants with zero variance. We can place the variances in another array and then implement a simple algorithm that determines the smallest variance. We now have a value for b, and so calculating a is relatively straightforward. Below is the code used to find the 'k' value, which is same as our b value:

```
def findkvalue():
  vararray=[]
  k=0
  i=0
  while (k<=0.1):
    length=len(InputValues[0])
    constantarray=[]
    for i in range(0,length):
      approx=TempValues[i]+k*(TempValues[i])**3
      actual=HeatCap[i]
      constant=actual/approx
      constantarray.append(constant)
    summ=0
    for data in constantarray:
      summ=summ+data
    mean=summ/length
    summm=0
    for moredata in constantarray:
      summm=summm+(moredata-mean)**2
    var=summm/length
    vararray.append(var)
```

```
22      k=k+0.001
23    print(vararray)
24    findmin(vararray,len(vararray),0.001)
```

Listing 7: Determining the k-value in $a(T + kT^3)$

Listing 7 involves the calling of another function `findmin(array_name,array_length,index_step)` and is defined as

```
1  def findmin(arry,len,indexstep):
2    min=arry[0]
3    for i in range(1,len):
4      if arry[i]<min:
5        min=arry[i]
6        minind=i
7    print("The min value is: ",min," at index: ",minind,".\n")
8    print("Therefore, the k-value is: ",minind*indexstep," with
      variance: ",min,".")
```

Listing 8: Definition of findmin(...)

Here is a brief walk-through of listings 7 and 8. The while loop starting at line 5 assumes that our k-value is somewhere in the range $[0, 0.1]$. This guess was deduced by trial and error, and also the fact that our data values have low orders of magnitude, so it makes sense for k to be small. Lines 8 to 12 iterate through our given data values and computes the ratio given by equation (6). We then store these constants in the array `constantarray[]`. Now in lines 14 to 21, we compute the mean, which will be used to calculate the variance of these constants for each k and places this variance in another array `vararray[]`. Line 22 indicates we are incrementing k in values of 0.001 and line 23 is simply placed so that we can simply see the array we have created for checking purposes. Finally, we have line 24, which not only calculates the minimum variance in each `vararray[]` for each k but it also calculates the k-value, based on the index position in the array.

3.2 Part (b)

Using our method from part (a), we can calculate the value of k to be exactly $k = 0.005$. This is because in our `vararray[]`, it was found that a k value of 0.005 had *zero* variance. Now that we have k for $a(T + kT^3)$, we must determine a, which we denote as `newconst`. We do this by implementing the following function:

```
1  def findnewconst():
2    length=len(InputValues[0])
3    conss=[]
4    for i in range(0,length):
5      right=TempValues[i]+0.005*(TempValues[i])**3
6      left=HeatCap[i]
7      cc=left/right
8      conss.append(cc)
9    print(conss)
10   av=0
11   for data in conss:
12     av=av+data
```

```
13  av=av/length
14  print("The new constant is therefore:",av,".")
```

Listing 9: Finding the second constant

Notice in line 5, we use our solution $k = 0.005$ and we then compute the ratio as before. Implementing listing 9 gives us $a = 2$. We therefore have the following best fit to the data:

$$\boxed{C = 2(T + 0.005T^3) = 2T + 0.01T^3} \tag{7}$$

where we put our equation in the original form in (4) such that $\boxed{a = 2}$ and $\boxed{b = 0.01}$. Plotting equation (7) along side our data points:

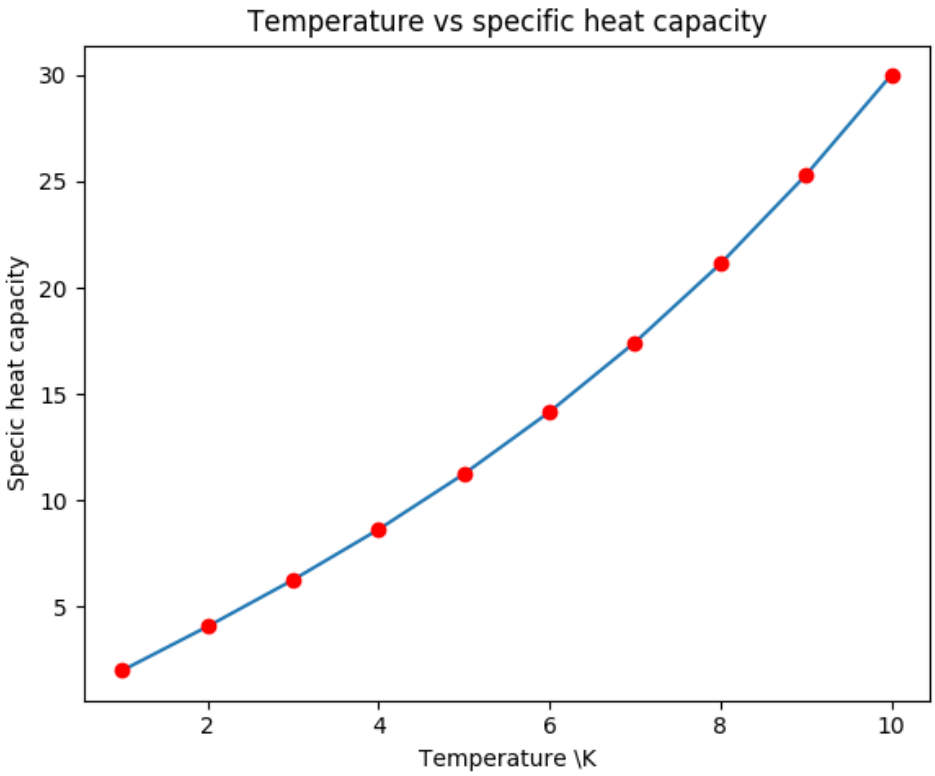

Figure 5: Plot of equation (7) against data points from HeatCapacity.txt

4 Flow.py

Below are plots using various combinations of log() scaling.

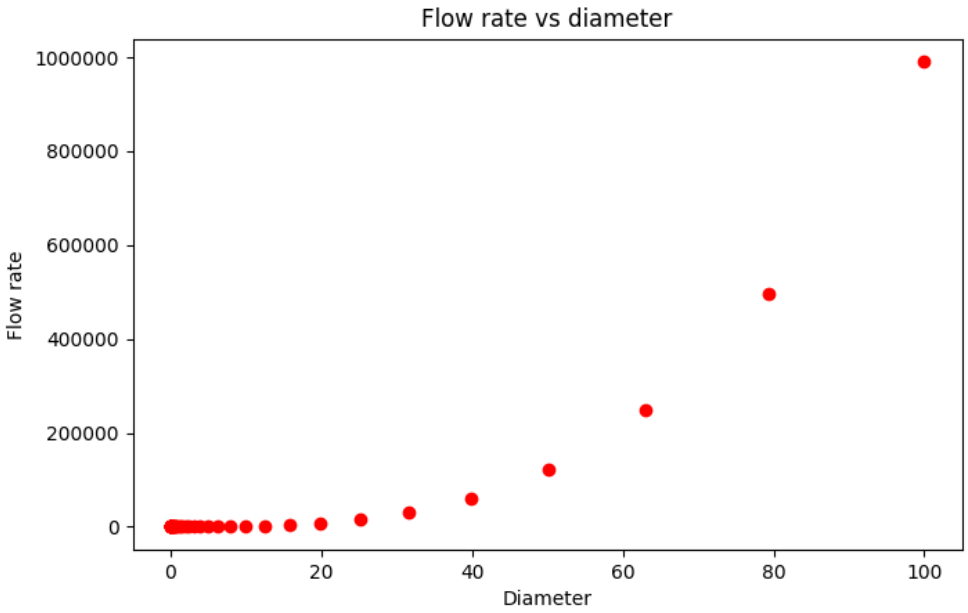

Figure 6: Flow rate vs diameter without any changes in scaling

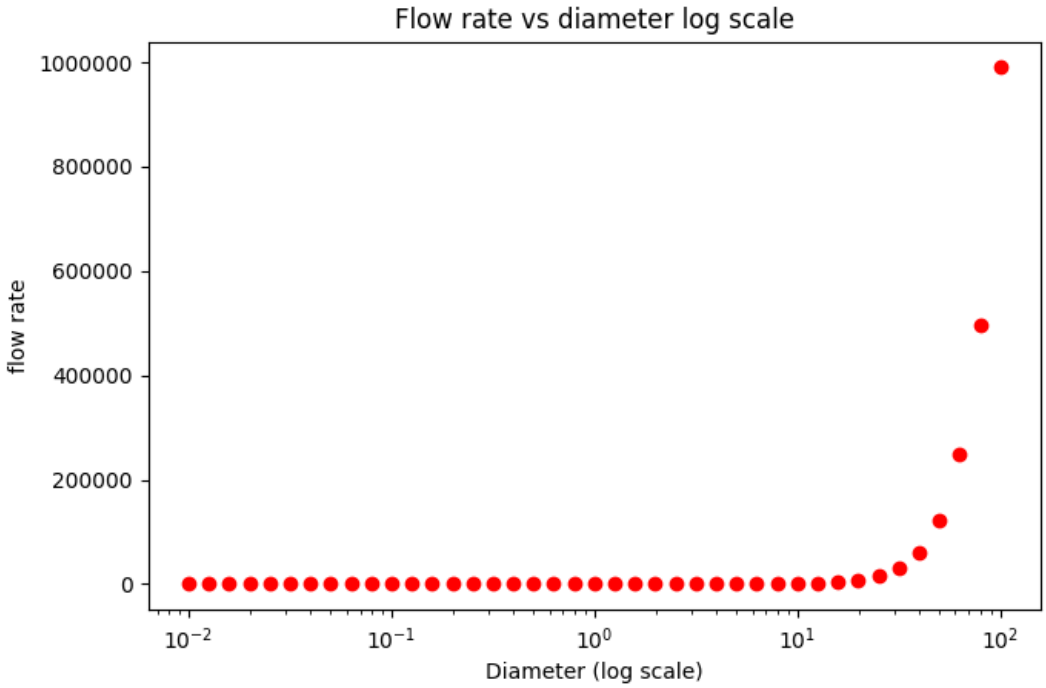

Figure 7: Flow rate vs diameter with a log scaling in the diameter

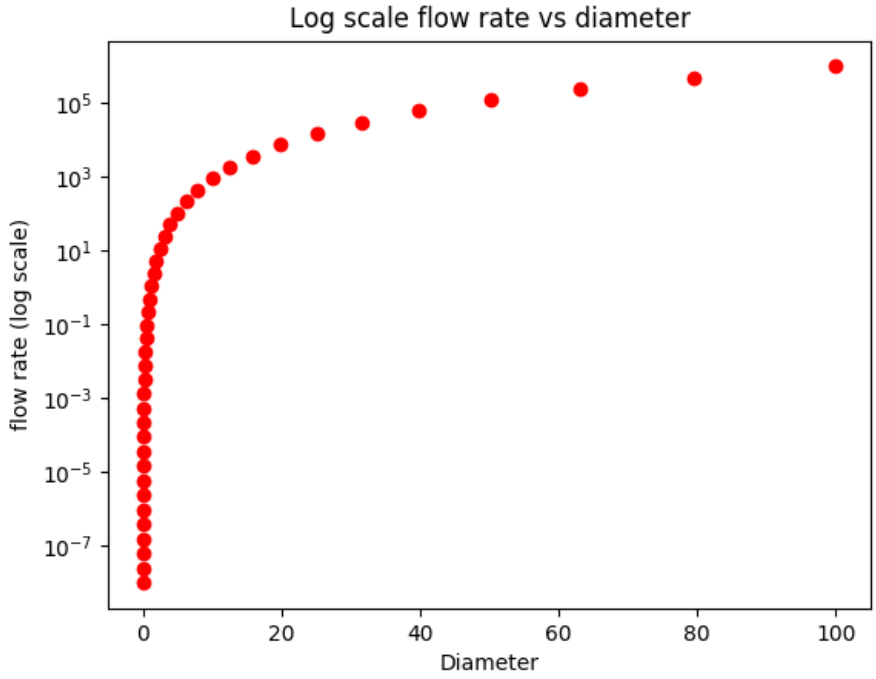

Figure 8: Flow rate vs diameter with a log scaling in the flow rate

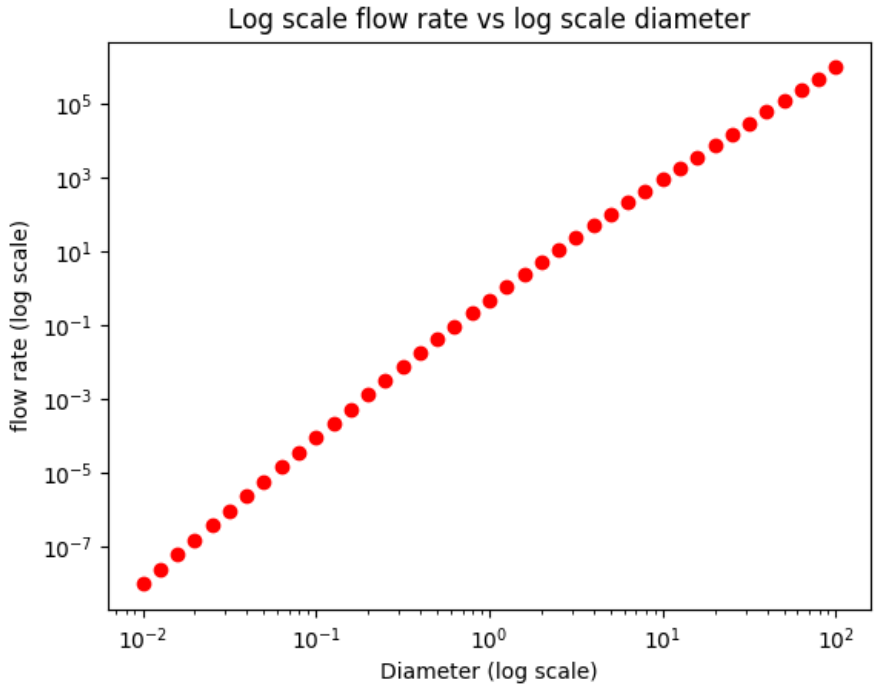

Figure 9: Flow rate vs diameter with a log scaling for both diameter and flow rate

4.1 Part (b)

Figure 9 with a "log-log" scale gives us the clearest and most descriptive presentation of the data. Figure 9 looks like a straight line, and a straight line plotted on a log-log scale indicates a *power* relationship. This power relationship makes it difficult to observe data on a 'normal' scale in that data is often clumped at one end and spread on another (apparent in figure 6).

We now know that the relation is of the form Ax^k where $A = 10^b$ and $k = m$ of the linear line with slope m and y-intercept b.

4.2 Part (c)

We will consider small values and large values of the diameter separately. Since our data value is given in ascending order of diameters, we can simply take the first couple of data points as our 'small' values and the last couple of data points as our 'large' values.

```
smalldatax=[]
smalldatay=[]
for i in range(0,6):
  smalldatax.append(np.log10(Diameter[i]))
  smalldatay.append(np.log10(FlowRate[i]))
```

Listing 10: Determining behaviour at small values of diameter

Here, we are taking the first 5 data points and taking their logarithms (and in base 10 since that is what our scaling did). We then attempt to fit a linear line of best fit for these small data values using our standard method:

```
m,b=np.polyfit(np.array(smalldatax), np.array(smalldatay),1)
print(m,b)
```

which gives us $m = 3.981...$ and $b = -0.040...$ and so $F \approx 10^{-0.04} d^{3.981} \approx d^4$. We utilise the same approach for large diameters, but instead we consider the last couple of data points. This means our for-loop iterates through the last couple of indices, which gives us: $m = 3.029$ and $b = -0.059$ and so $F \approx 10^{-0.059} d^{3.029} \approx d^3$. Therefore:

$$\boxed{F = d^4} \quad \text{for small } d \tag{8}$$

$$\boxed{F = d^3} \quad \text{for large } d \tag{9}$$