Linn Boldt-Christmas

Scientific Computing
Assignment 1

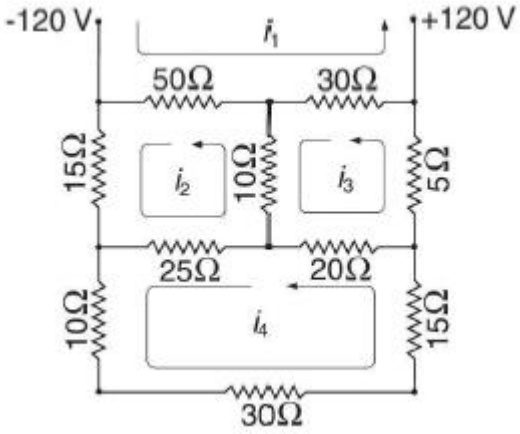
Candidate no. 117418
Due in November 2nd, 2015

University of Sussex

Physics & Astronomy

QUESTION 1

Consider the following circuit:



Determine the loop currents i_1 to i_4 in the electrical network shown, as follows:

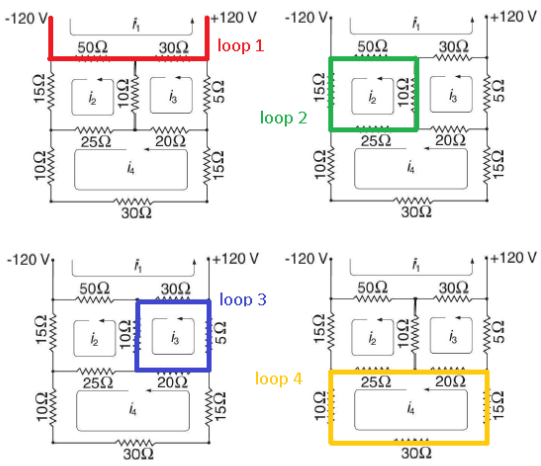
(a) Use Ohm's law and the Kirchhoff current law, which says that the total net current flow out of (or into) any junction in a circuit must be zero to write the 4 equations to be solved.

Kirchhoff's current law comes from the conservation of charge. Charge entering a circuit cannot simply leave a circuit; it has nowhere to go. The law tells us that the sum of currents at any junction or node in a circuit should be equal to zero.

Ohm's law tells us that the product of the resistance and the current should equal the voltage across that circuit or component. In an equation, when voltage = V, current = I, resistance = R:

$$V = IR$$

We can use this knowledge to create and execute a number of simultaneous equations. Using the diagram, let's highlight the segments that we will be analysing in order to create equations describing them, ie all of the current loops.



We want to find the voltage in all of these loops by multiplying their resistance (given) by their respective currents (i_n). In some of the resistors, there are multiple current loops interacting with it. For these, we take the direction of the loop that we are computing to be positive and the countering current to be negative.

For highlighted (red) loop 1:

$$50(i_1 - i_2) + 30(i_1 - i_3) = 50i_1 - 50i_2 + 30i_1 - 30i_3$$
$$= 80i_1 - 50i_2 - 30i_3 = 240V$$

For highlighted (green) loop 2:

$$50(i_2 - i_1) + 15i_2 + 25(i_2 - i_4) + 10(i_2 - i_3)$$
$$= 50i_2 - 50i_1 + 15i_2 + 25i_2 - 25i_4 + 10i_2 - 10i_3$$
$$= -50i_1 - 100i_2 - 10i_3 - 25i_4 = 0V$$

For highlighted (blue) loop 3:

$$30(i_3 - i_1) + 10(i_3 - i_2) + 20(i_3 - i_4) + 5i_3$$
$$= 30i_3 - 30i_1 + 10i_3 - 10i_2 + 20i_3 - 20i_4 + 5i_3$$
$$= -30i_1 - 10i_2 + 65i_3 - 20i_4 = 0V$$

For highlighted (yellow) loop 4:

$$25(i_4 - i_2) + 10i_4 + 30i_4 + 15i_4 + 20(i_4 - i_3)$$
$$= 25i_4 - 25i_2 + 10i_4 + 30i_4 + 15i_4 + 20i_4\_20i_3$$
$$= -25i_2 - 20i_3 - 100i_4 = 0V$$

These equations were commented in within the code so that the code becomes more self-explanatory:

```
# Simultaneous equations for the loops:
# 80*i1-50*i2-30*i3 = 240

# 50*i1-100*i2-10*i3-25*i4 = 0

# -30*i1-10*i2+65*i3-20*i4 = 0

# -25*i2-20*i3-100*i4 = 0
```

(b) Write a program to set up the four resulting equations in the standard way as a matrix A *and a vector* b, *solve the system using Gaussian elimination and hence find the four currents.*

Matrix *A* is one 4x4 matrix consisting of the four segment equations in each of the rows, and each column representing one of the i_n terms. Our other matrix *b* is consisting of the four results in each row, so, a single column matrix (ie a vector).

```
import scipy as sp
import gaussElimin as gE

A = sp.array([[80.0,-50.0,-30.0,0.0],[-50.0,100.0,-10.0,-
25.0],[-30.0,-10.0,65.0,-20.0],[0.0,-25.0,-20.0,100.0]])
# array of coefficients in simultaneous equations
b = sp.array([240.0,0.0,0.0,0.0]) # results for all the
simultaneous equations


print "For the first array, plug in all of the
coefficients of the simultaneous equations derived from
the circuit diagrams. This array, A:"

print A

print "For the second array, plug in all the results of
the simultaneous equations derived from the circuit
diagram. This array, b:"


print b
```

Next, we want to actually use these matrices in our Gaussian elimination. This is where we use the gaussElimin module (gE) that we imported earlier.

```
x = gE.gaussElimin(A,b)

print "Using the Gaussian elimination module, we can
solve:"
print x
```

When the code runs for part (b):

```
For the first array, plug in all of the coefficients of the
simultaneous equations derived from the circuit diagrams. This
array, A:
[[  80.  -50.  -30.    0.]
 [ -50.  100.  -10.  -25.]
 [ -30.  -10.   65.  -20.]
 [   0.  -25.  -20.  100.]]
```

For the second array, plug in all the results of the simultaneous equations derived from the circuit diagram. This array, b:

```
[ 240.    0.    0.    0.]
```

Using the Gaussian elimination module, we can solve:

```
[ 8.36478985  5.32910389  5.42426646  2.41712926]
```

(c) Use lu *code from the* scipy.linalg *package to do the LU decomposition of the matrix you created in (b).*

The point of the LU decomposition is to split a matrix into a lower triangular matrix and an upper triangular matrix. To do this, we have to import linalg from scipy as the question tells us.

```
from scipy import linalg

P, L, U = linalg.lu(A) # computes LU factorisation explicitly
where P is the permutation matrix, L is the lower triangular
matrix, and U is the upper triangular matrix

print "We can use an LU (Lower-Upper) decomposition to
decompose our resultant matrix into an upper part and a lower
part. The lower part:"
print L

print "And the upper part:"
print U
```

When we run our code for part (c):

```
We can use an LU (Lower-Upper) decomposition to decompose our
resultant matrix into an upper part and a lower part. The
lower part:
[[ 1.          0.          0.          0.        ]
 [-0.375       1.          0.          0.        ]
 [ 0.          0.52631579  1.          0.        ]
 [-0.625      -0.78947368  0.50411629  1.        ]]
And the upper part:
[[ 80.         -50.         -30.           0.        ]
 [  0.         -47.5         30.47727273 -30.45454545]
 [  0.           0.         -46.49521531  84.71062535]
```

```
[   0.            0.            0.          -91.74706795]]
```

(d) Solve the system Ax = b *using this LU decomposition of the matrix* A *and right hand-sides given by* b *as derived in (a) and* b *where the left voltage is not -120 V, but is 0 V, instead. Check the solutions that you have found.*

For this section, we want to confirm that we have sensible answers by going backwards and seeing if we end up with the *b* vector that we started with.

```
print "Next, use the LU decomposition to solve Ax = b"

y1 = gE.gaussElimin(L,b) # lower triangular matrix and b

x1 = gE.gaussElimin(U,y1)# upper triangular matrix and the
gE of lower and b

print x1

print "Now we want to use our LU decomposition to prove that
we can go the other direction and get an answer from it
too."
print "Using a new array where the voltage across is 120 V
instead of 240 V (as the range is 0 V to +120 V instead of -
120 V to +120 V:"

c = sp.array([240.0,0.0,0.0,0.0]) # new b array where the -
120 V is = 0 V

y2 = gE.gaussElimin(L,c) # same process as with y1 and x1

x2 = gE.gaussElimin(U,y2)


print x2
```

When the code runs:

```
Next, use the LU decomposition to solve Ax = b

[-0.17583304 -0.27122297 -0.29567614 -0.1508547 ]

Now we want to use our LU decomposition to prove that we can
go the other direction and get an answer from it too.

Using a new array where the voltage across is 120 V instead of
240 V (as the range is 0 V to +120 V instead of -120 V to +120
V:

[-0.09829751 -2.65021772 -3.84509716 -2.66964198]
```

QUESTION 2

Using the most efficient *methods available to you (both the provided tridiagonal matrix solver from the book and the internal* scipy.linalg.solve_banded) *solve the symmetric, tridiagonal system*

$$\begin{aligned} 4x_1 - x_2 &= 9 \\ -x_{i-1} + 4x_i - x_{i+1} &= 5, \qquad i = 2 \ldots 9 \\ -x_{n-1} + 4x_n &= 5 \end{aligned}$$

with n = 10.

A tridiagonal matrix is a matrix where the only nonzero elements occur along the three main diagonals (ie the main diagonal, the one above that, and the one below).

This is system that will require a 10x10 matrix to be solved. That matrix will be a tridiagonal matrix because, as described, it will be composed entirely of zeroes except for in the main diagonal (i.e. where horizontal n = vertical n) where it will be equal to 4; along the diagonal above (i.e. where horizontal n = vertical n-1) where it will be equal to -1; and finally, along the diagonal above (i.e. where horizontal n = vertical n+1)

```
import numpy as np
import scipy as sp
from scipy import linalg

print "Following the equations given to us in the
question, the boundaries they impose call for the
following matrix in order for them to be represented:"
M = np.zeros([10,10]) # creates an array of zeroes

for i in range (0,9,1): # for within the array, up until
the 9th line out of 10, these rules apply:
     M[i,i-1]=-1 # the integer at (n, n-1) = -1
     M[i,i]=4 # the integer at (n,n) = 4
     M[i,i+1]=-1 # the integer at (n, n+1) = -1

for i in range (9,10,1): # continuing the rules for the
10th and last line because there was an error on the last
line under the previous constrictions as we fell outside
of our range
     M[i,i-1]=-1 # the integer at (n, n-1) = -1
     M[i,i]=4 # the integer at (n,n) = 4

print M
```

This matrix *M* is the tridiagonal matrix described in the first paragraph. When the code runs, the matrix looks like this:

Following the equations given to us in the question, the boundaries they impose call for the following matrix in order for them to be represented:

```
[[ 4. -1.  0.  0.  0.  0.  0.  0.  0. -1.]
 [-1.  4. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  4. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  4. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  4. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  4. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  4. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  4. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  4. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  4.]]
```

What we now want is a more economic matrix that stores all of the non-zero terms, called a CM matrix. This is used because whilst our matrix isn't that large, many larger tridiagonal matrices will have thousands upon thousands of "empty" cells in the matrix where the terms are just zero, and this is a tremendous waste of space (and processing power).It basically takes the diagonal terms, shift them 45 degrees and store them in much smaller rows or columns.

```
print "Now to convert it into a cm matrix, i.e. a matrix
storing (economically) the non-zero matrix elements."

CM = np.array([[0,-1,-1,-1,-1,-1,-1,-1,-1,-
1],[4,4,4,4,4,4,4,4,4,4],[-1,-1,-1,-1,-1,-1,-1,-1,-1,0]],
dtype=float)# the dtype=float is to avoid rounding errors

print CM
```

When the code runs, the matrix looks like this:

Now to convert it into a cm matrix, i.e. a matrix storing (economically) the non-zero matrix elements.

```
[[ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1.  0.]]
```

Our final puzzle piece is the RHS matrix – that is, the matrix that includes the right hand side of the system of equations. In our case, these are the things that the equations given in the question are equal to, so, 9 for the first term and then nine 5s for the remaining nine terms.

```
print "In order to complete the tridiagonal calculation,
we also need the RHS matrix (vector), which is composed
of the answers to each of the equations given to us in
the question ie holding the right hand side of the
system."

RHS = np.array([[9],[5],[5],[5],[5],[5],[5],[5],[5],[5]],
dtype=float) # again, avoiding rounding errors with
floats

print RHS
```

RHS printed by running the code:

```
In order to complete the tridiagonal calculation, we also need
the RHS matrix (vector), which is composed of the answers to
each of the equations given to us in the question ie holding
the right hand side of the system.
[[ 9.]
 [ 5.]
 [ 5.]
 [ 5.]
 [ 5.]
 [ 5.]
 [ 5.]
 [ 5.]
 [ 5.]
 [ 5.]]
```

Finally, we can now combine all our information to get a solution using the recommended linalg.solve_banded module:

```
print "Finally, we obtain our tridiagonal matrix's
solution:"

TM = sp.linalg.solve_banded((1,1),CM,RHS) # the (1,1)
means that we want to include the 1 diagonal above and 1
diagonal below the main diagonal

print TM
```

Running code:

```
Finally, we obtain our tridiagonal matrix's solution:
[[ 2.90191936]
 [ 2.60767745]
 [ 2.52879042]
 [ 2.50748425]
 [ 2.50114659]
 [ 2.4971021 ]
 [ 2.48726181]
 [ 2.45194513]
 [ 2.3205187 ]
 [ 1.83012968]]
```

QUESTION 3

The table shows the drag coefficient c_D *of a sphere as a function of Reynolds number* Re.

Re	0.2	2	20	200	2000	20 000
c_D	103	13.9	2.72	0.800	0.401	0.433

(a) Use the natural cubic spline to find c_D *at* Re = *5, 50, 500, and 5000.*

First, we need to create an array with these values of Re and c_D so that Python know what to plot, and also import all of the necessary modules. We then plot it on a log-log scale.

```
import numpy as np

from scipy import interpolate

import matplotlib.pyplot as plt

from polyFit import *

from swap import *

from error import *


Re = np.array([0.2,2,20,200,2000,20000]) # the Re values
given to us, on x axis

cD = np.array([103,13.9,2.72,0.800,0.401,0.433]) # the cD
values given to us, on y axis

plt.figure(1)

plt.loglog(Re,cD) # plotting Re and cD on a log-log scale
```

Then, we use the natural cubic spline module from scipy.interpolate to create a cubic spline interpolator. In order to make a plot of it, we also have to create an array of the new Re values at which we want the interpolator to act.

```
f = interpolate.interp1d(Re,cD,kind='cubic') # using
cubic spline to interpolate Re vs. cD plot


newRe = np.array([5,50,500,5000]) # an array of the
values at which we want Re

newcD = f(newRe) # making the y values of the cubic
function using the new Re values we are using
```

```
plt.loglog(Re,cD,'o',newRe,newcD,'-') # first plot is
circles, new plot is dashes

plt.title('Re vs cD on loglog scale')

plt.xlabel('logRe')

plt.ylabel('logcD')
```

And if we want to see the individual results printed:

```
print "\n\n Cubic spline interpolation results:" #
printing the numerical value of the function at the given
new Re values when using the cubic spline interpolation


print "Value at Re = 5: " + (str(f(5)))

print "Value at Re = 50: " + (str(f(50)))

print "Value at Re = 500: " + (str(f(500)))

print "Value at Re = 5000: " + (str(f(5000)))
```

When that code runs:

```
Cubic spline interpolation results:

Value at Re = 5: -95.8099340778

Value at Re = 50: 2550.62121393

Value at Re = 500: -455562.249652

Value at Re = 5000: 97675156.105
```

(b) Do the same as in (a), but using a polynomial interpolant

This time, we have to use the polyFit module (the individual one, not the polyfit included in numpy). We also need to log our arrays of Re and cD in order for them to comply with the loglog scale (since we're plotting everything on the same graph as required by question c).

```
lRe=np.log(Re)

lcD=np.log(cD)
```

```
p = polyFit(lRe,lcD,3) # this gives the coefficients of
the polynomial on the logged values of Re and cD

print "The polynomial interpolation results are: "

print p

x = np.arange(np.log(Re[0]),np.log(Re[5]),0.1) # making a
range of the relevant Re log values to use with our
polyfit

y = p[0]+x*(p[1])+(x**2)*(p[2])+(x**3)*(p[3]) # the
polyfit


plt.loglog(np.e**x, np.e**y, '+') # putting them back
into exponentials

plt.show(1)
```

To see the individual results printed for the polynomial interpolation:

```
print "\n\n Polynomial interpolation results:" # printing
the numerical value of the function at the given new Re
values when using the polynomial/barycentric
interpolation

print "Value at Re = 5: " +
(str(interpolate.barycentric_interpolate(Re,cD,5)))

print "Value at Re = 50: " +
(str(interpolate.barycentric_interpolate(Re,cD,50)))

print "Value at Re = 500: " +
(str(interpolate.barycentric_interpolate(Re,cD,500)))

print "Value at Re = 5000: " +
(str(interpolate.barycentric_interpolate(Re,cD,5000)))

p = polyFit(lRe,lcD,3) # this gives the coefficients of
the polynomial on the logged values of Re and cD

print "The polynomial interpolation points are: "

print p
```

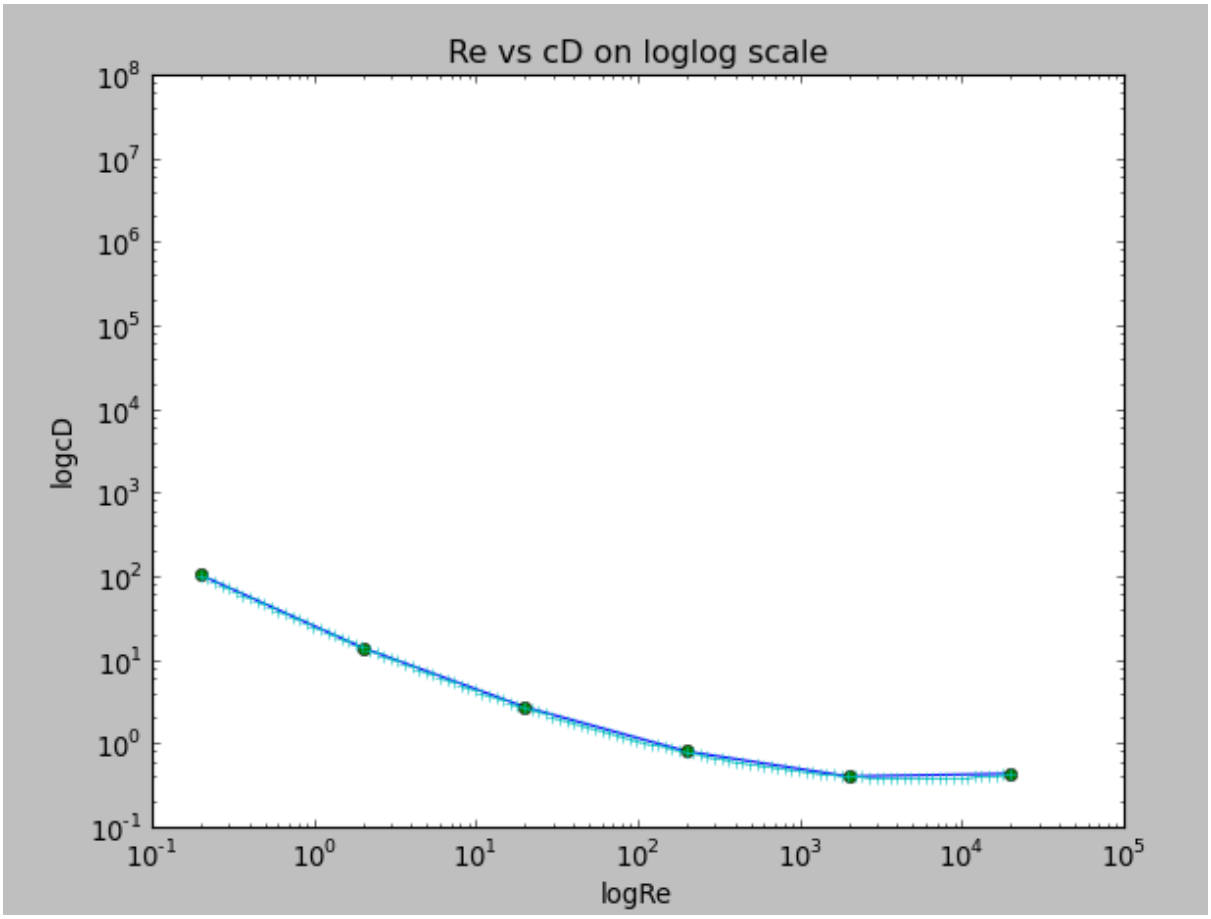
When the code is run, we get the polynomial interpolation results printed as such:

Polynomial interpolation results:

Value at Re = 5: -96.1004620719

```
Value at Re = 50: 2581.14411797

Value at Re = 500: -646765.83191

Value at Re = 5000: 1658455173.23

The polynomial interpolation points are:

[  3.22071242e+00  -8.37872943e-01   2.54609617e-02
1.78702950e-03]
```

We also print a figure showing (i) the original Re v cD values in logged form as the plain darker blue line, (ii) the cubic spline interpolation as the dark green circles, and (iii) the polynomial interpolant as the lighter blue '+' signs.

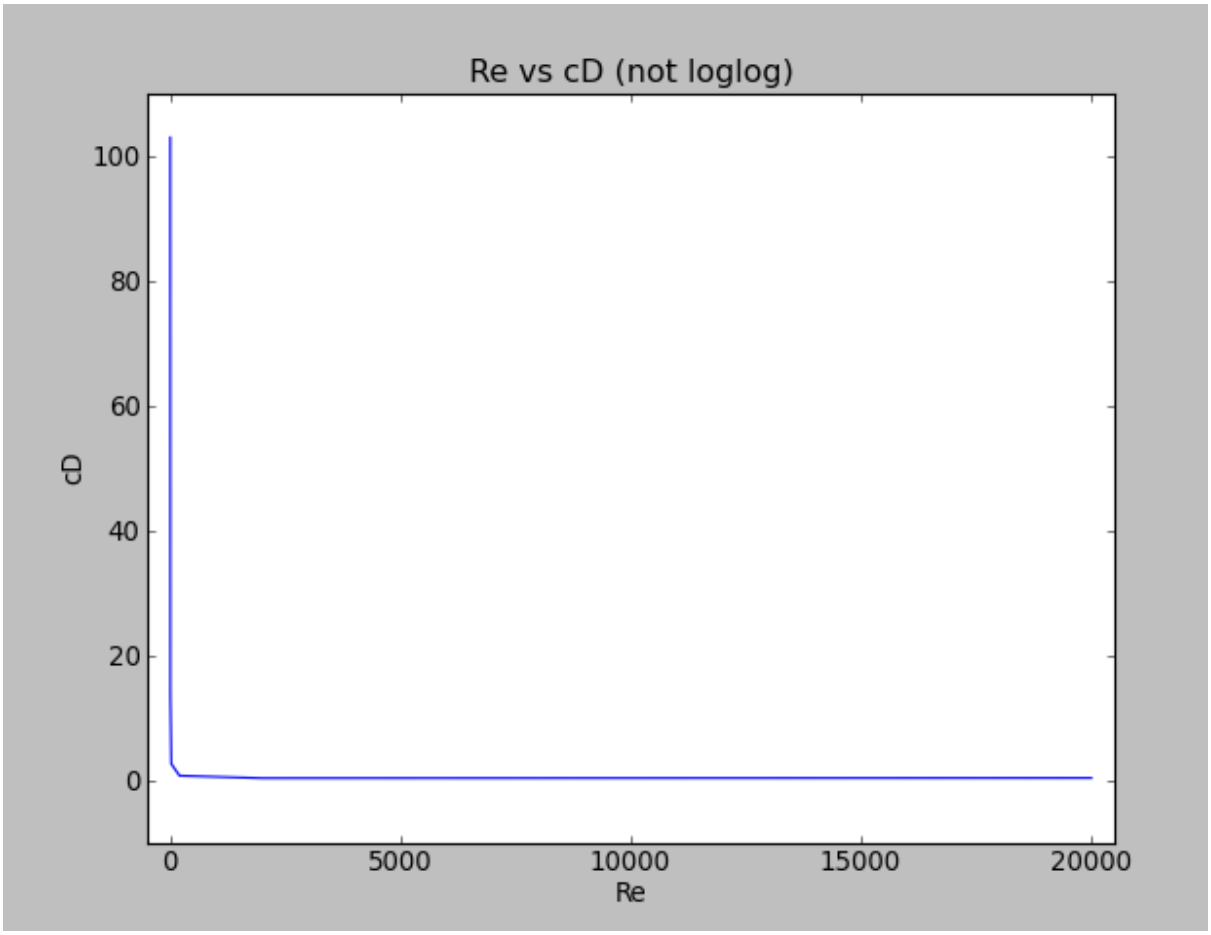


(c) What happens if you do not use a log scale? Plot all three on the same graph.

If you try to plot non-logged Re and cD on the same figure as the above log-log plots, Python tries to automatically log them when plotting them, giving you simply the same exact result as we already have. So, you have to put the non-logged Re and cD on a separate plot.

```
plt.figure(2)
plt.plot(Re,cD, '-')
plt.title('Re vs cD (not loglog)')
plt.xlabel('Re')
plt.ylabel('cD')
plt.axis([-500,20500,-10,110])
plt.show(2)
```

When this code runs:



This looks like the function has a very fast decay rate. As a low Reynolds number (Re) corresponds to a high cD drag coefficient on this graph, and since low Reynolds numbers correspond to laminar flow, this implies that laminar flow only happens when drag coefficients are very high, and that they can become turbulent very suddenly in response to a small change in Re.

This graph is not particularly helpful, especially comparatively to the much more informative log-log plot. The derivative of this plot would have a very large and negative magnitude (as the function will be some kind of e^{-nx} where n is very large and positive), meaning small changes in cD would go largely unnoticed.