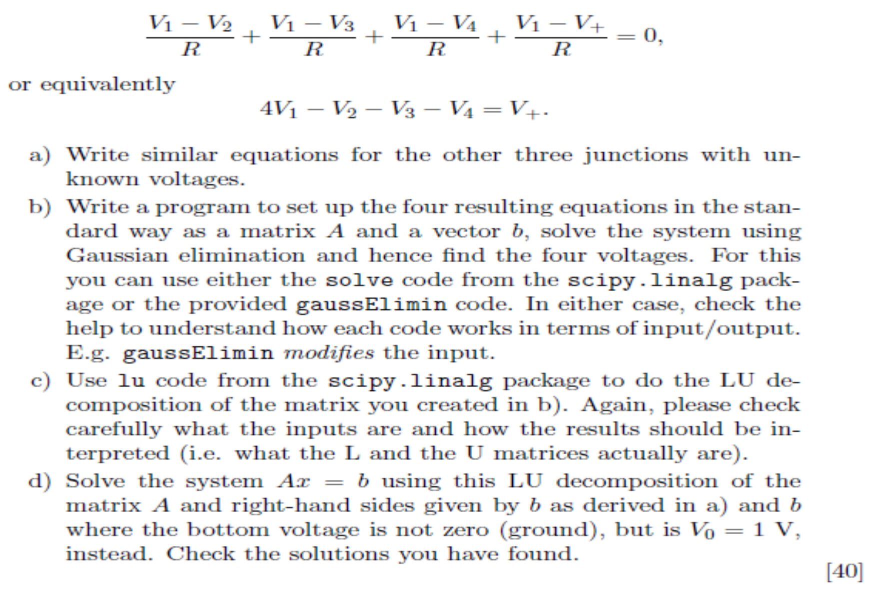
Anjelah Balachandran

University of Sussex – Scientific Computing
Assignment 1

Problem Sheet 1



$$\frac{V_1 - V_2}{R} + \frac{V_1 - V_3}{R} + \frac{V_1 - V_4}{R} + \frac{V_1 - V_+}{R} = 0,$$

or equivalently

$$4V_1 - V_2 - V_3 - V_4 = V_+.$$

a) Write similar equations for the other three junctions with unknown voltages.

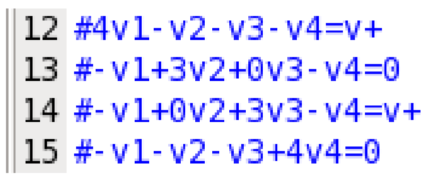
b) Write a program to set up the four resulting equations in the standard way as a matrix A and a vector b, solve the system using Gaussian elimination and hence find the four voltages. For this you can use either the `solve` code from the `scipy.linalg` package or the provided `gaussElimin` code. In either case, check the help to understand how each code works in terms of input/output. E.g. `gaussElimin` *modifies* the input.

c) Use `lu` code from the `scipy.linalg` package to do the LU decomposition of the matrix you created in b). Again, please check carefully what the inputs are and how the results should be interpreted (i.e. what the L and the U matrices actually are).

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 bottom voltage is not zero (ground), but is $V_0 = 1$ V, instead. Check the solutions you have found.

[40]

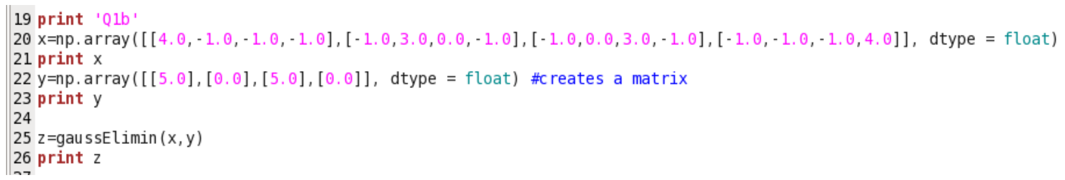
Solutions-



```
#4v1-v2-v3-v4=v+
#-v1+3v2+0v3-v4=0
#-v1+0v2+3v3-v4=v+
#-v1-v2-v3+4v4=0
```

1a) Ohms law and Kirchhoff's current law were used to calculate the voltages at the other three junctions.

1b)

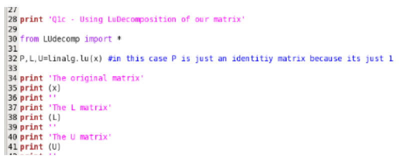


```
19 print 'Q1b'
20 x=np.array([[4.0,-1.0,-1.0,-1.0],[-1.0,3.0,0.0,-1.0],[-1.0,0.0,3.0,-1.0],[-1.0,-1.0,-1.0,4.0]], dtype = float)
21 print x
22 y=np.array([[5.0],[0.0],[5.0],[0.0]], dtype = float) #creates a matrix
23 print y
24
25 z=gaussElimin(x,y)
26 print z
```

In line 22 and 24, I have formed a matrix by using an array. I then used Gaussian elimation to solve for the unknown voltages. Gaussian elimation uses the first equation to remove the x1 term from all other equations by subtracting λ=aj1/a11 times the first equation from all the jth equation (j>1).

Anjelah Balachandran

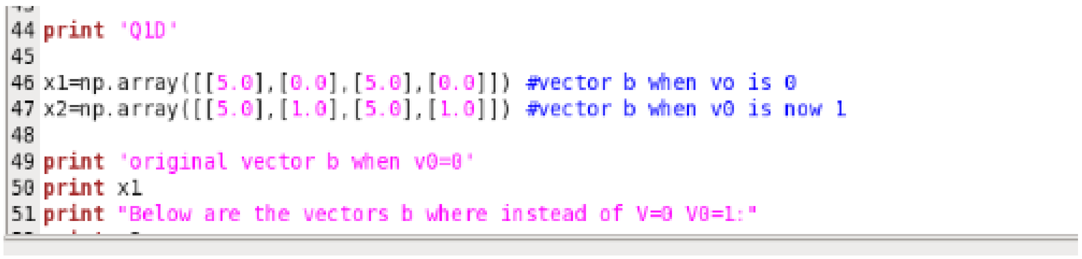
1c)



```
27
28 print 'Q1c - Using LuDecomposition of our matrix'
29
30 from LUdecomp import *
31
32 P,L,U=linalg.lu(x) #in this case P is just an identitiy matrix because its just 1
33
34 print 'The original matrix'
35 print (x)
36 print ''
37 print 'The L matrix'
38 print (L)
39 print ''
40 print 'The U matrix'
41 print (U)
```

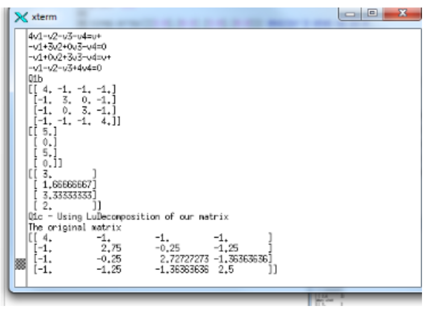
The LU decomposition is applied where L is a lower triangular matrix, U is an upper triangular matrix and P is a permutation matrix. P is needed to resolve certain singularity issues and in this case is the identity matrix. The LUdecomp function factorises any square matrix A such that A=LU. In lines 34 I have used this to split up my matrix x.

I have then proceeded to print the L and U matrix in lines 40 and 43.



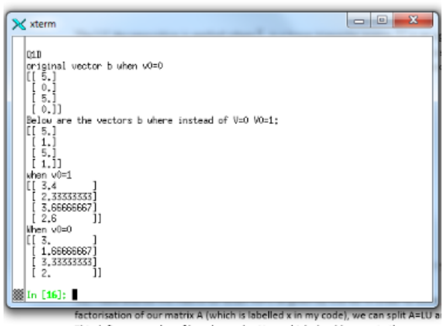
```
44 print 'Q1D'
45
46 x1=np.array([[5.0],[0.0],[5.0],[0.0]]) #vector b when vo is 0
47 x2=np.array([[5.0],[1.0],[5.0],[1.0]]) #vector b when v0 is now 1
48
49 print 'original vector b when v0=0'
50 print x1
51 print "Below are the vectors b where instead of V=0 V0=1:"
```

1d) I have altered the matrix for when V0=1 as given by line 49. In order to solve Ax=b I used LU factorisation of our matrix A (which is labelled x in my code), we can split A=LU and solve for Lz=b. This defines our value of b and we solve Ux=z which should generate the same answer from part b. I used Gaussian elimination again with our new matrix x2 (line 59) as Vo is now 1 with the upper triangular matrix and this generated a new solution since V0=1. As shown below in the code after its run.



```
4v1-v2-v3-v4=v+
-v1+3v2+0v3-v4=0
-v1+0v2+3v3-v4=v+
-v1-v2-v3+4v4=0
Q1b
[[ 4. -1. -1. -1.]
 [-1.  3.  0. -1.]
 [-1.  0.  3. -1.]
 [-1. -1. -1.  4.]]
[[ 5.]
 [ 0.]
 [ 5.]
 [ 0.]]
[[ 3.        ]
 [ 1.66666667]
 [ 3.33333333]
 [ 2.        ]]
Q1c - Using LuDecomposition of our matrix
The original matrix
[[ 4.         -1.         -1.         -1.        ]
 [-1.          2.75       -0.25       -1.25       ]
 [-1.         -0.25        2.72727273 -1.36363636]
 [-1.         -1.25       -1.36363636  2.5       ]]
```

$\longrightarrow$



```
Q1D
original vector b when v0=0
[[ 5.]
 [ 0.]
 [ 5.]
 [ 0.]]
Below are the vectors b where instead of V=0 V0=1:
[[ 5.]
 [ 1.]
 [ 5.]
 [ 1.]]
when v0=1
[[ 3.4       ]
 [ 2.33333333]
 [ 3.66666667]
 [ 2.6       ]]
When v0=0
[[ 3.        ]
 [ 1.66666667]
 [ 3.33333333]
 [ 2.        ]]

In [16]: 
```

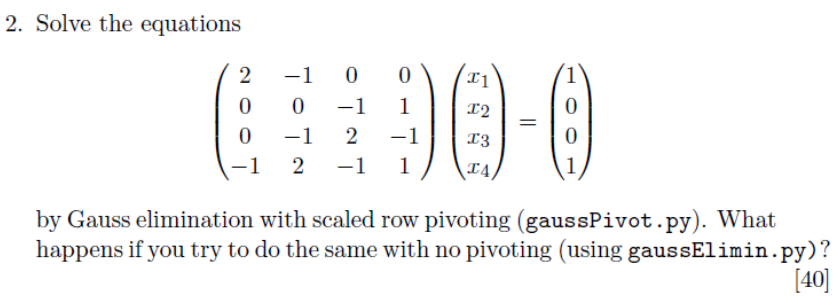
factorisation of our matrix A (which is labelled x in my code), we can split A=LU a

1b

1d

Anjelah Balachandran

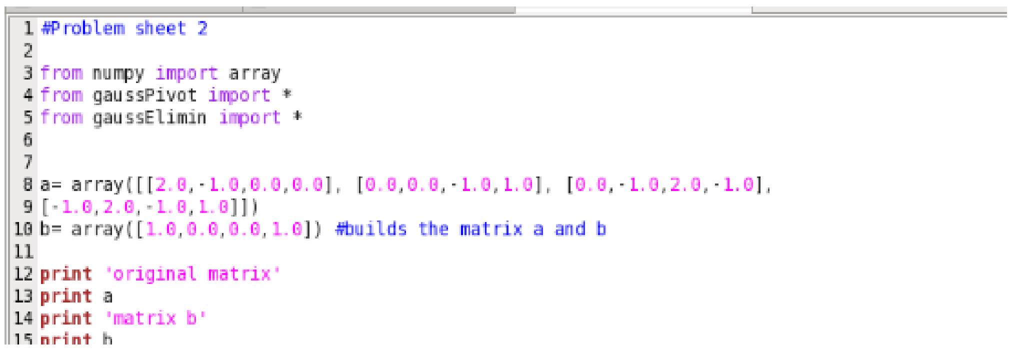
Problem Sheet 2



2. Solve the equations

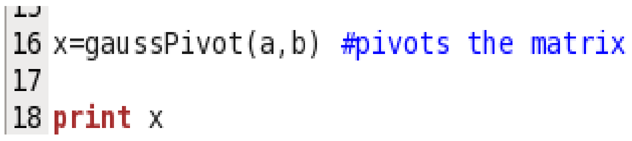
$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & -1 & 2 & -1 \\ -1 & 2 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

by Gauss elimination with scaled row pivoting (gaussPivot.py). What happens if you try to do the same with no pivoting (using gaussElimin.py)? [40]



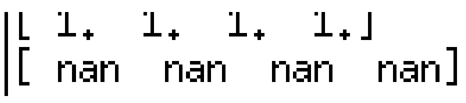
```
 1 #Problem sheet 2
 2
 3 from numpy import array
 4 from gaussPivot import *
 5 from gaussElimin import *
 6
 7
 8 a= array([[2.0,-1.0,0.0,0.0], [0.0,0.0,-1.0,1.0], [0.0,-1.0,2.0,-1.0],
 9 [-1.0,2.0,-1.0,1.0]])
10 b= array([1.0,0.0,0.0,1.0]) #builds the matrix a and b
11
12 print 'original matrix'
13 print a
14 print 'matrix b'
15 print b
```

2a) I formed a 2d arrays for the matrices (lines 8 and 9). I then used Gaussian pivot to pivot the matrix which rearranges the rows.



```
16 x=gaussPivot(a,b) #pivots the matrix
17
18 print x
```

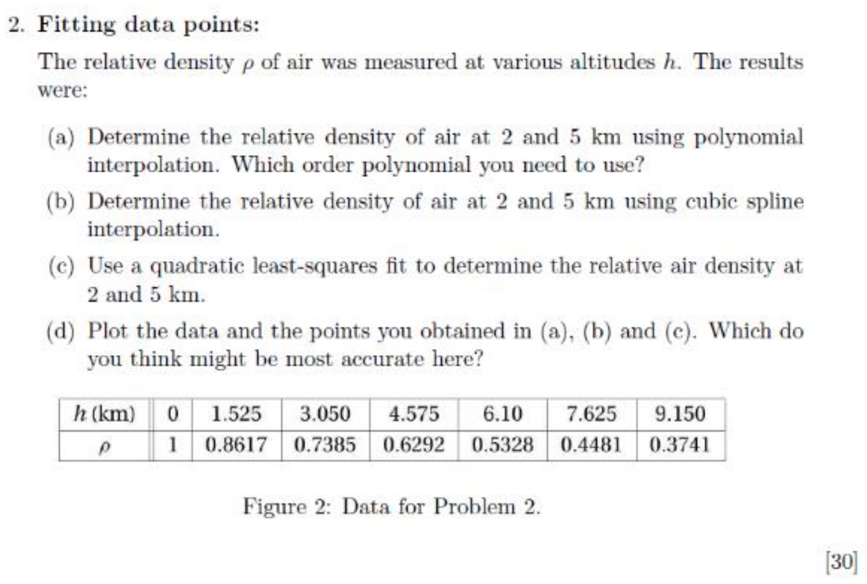
Upon trying to do Gaussian elimination as shown below, a nan error appeared. This is because integers were being divided through by zero, this is why Gaussian pivoting can be used to avoid this.



```
[ 1.   1.   1.   1.]
[ nan  nan  nan  nan]
```

Anjelah Balachandran

Problem Sheet 3



2. **Fitting data points:**

The relative density ρ of air was measured at various altitudes h. The results were:

(a) Determine the relative density of air at 2 and 5 km using polynomial interpolation. Which order polynomial you need to use?

(b) Determine the relative density of air at 2 and 5 km using cubic spline interpolation.

(c) Use a quadratic least-squares fit to determine the relative air density at 2 and 5 km.

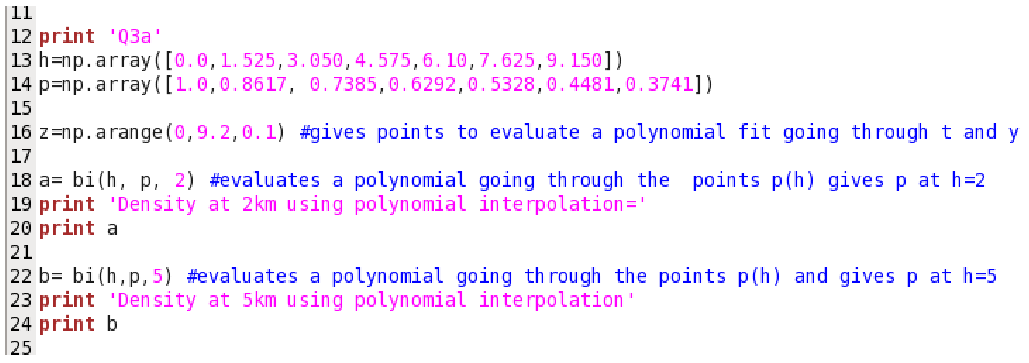
(d) Plot the data and the points you obtained in (a), (b) and (c). Which do you think might be most accurate here?

h (km)	0	1.525	3.050	4.575	6.10	7.625	9.150
ρ	1	0.8617	0.7385	0.6292	0.5328	0.4481	0.3741

Figure 2: Data for Problem 2.

[30]

3a) I arranged my table into array much like the previous questions as shown below by lines 13 and 14.

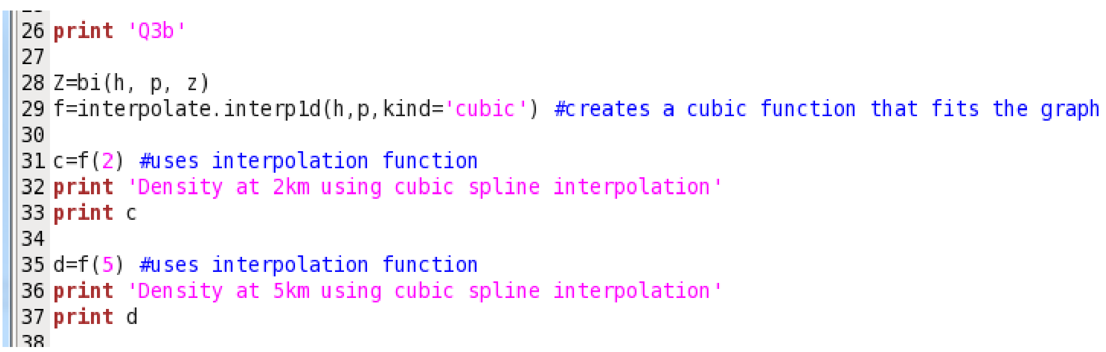


```
11
12 print 'Q3a'
13 h=np.array([0.0,1.525,3.050,4.575,6.10,7.625,9.150])
14 p=np.array([1.0,0.8617, 0.7385,0.6292,0.5328,0.4481,0.3741])
15
16 z=np.arange(0,9.2,0.1) #gives points to evaluate a polynomial fit going through t and y
17
18 a= bi(h, p, 2) #evaluates a polynomial going through the  points p(h) gives p at h=2
19 print 'Density at 2km using polynomial interpolation='
20 print a
21
22 b= bi(h,p,5) #evaluates a polynomial going through the points p(h) and gives p at h=5
23 print 'Density at 5km using polynomial interpolation'
24 print b
25
```

In line 16, I used the np.arange function to generate points to evaluate a polynomial fit going through t and y. I then used the barycentric function which I imported at the start of the code in order to fit a polynomial function to my data athat interpolates points within our data set.

This produces a graph of n+1 polynomial order, where n+1 is the number of data points, which in this case is 6. **Therefore our order of polynomial is 5.**

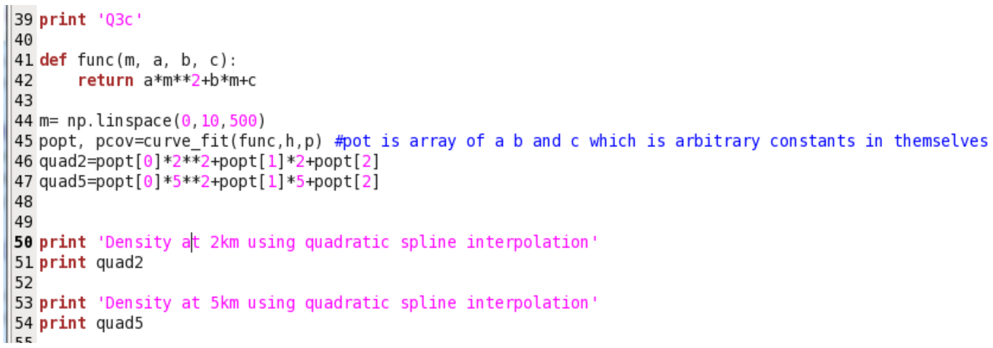
3b)



```
print 'Q3b'

Z=bi(h, p, z)
f=interpolate.interp1d(h,p,kind='cubic') #creates a cubic function that fits the graph

c=f(2) #uses interpolation function
print 'Density at 2km using cubic spline interpolation'
print c

d=f(5) #uses interpolation function
print 'Density at 5km using cubic spline interpolation'
print d
```

I used barycentric interpolate here again much like in 3a. I used the interpolate function for 2km and 5km to generate the density at those values applied to the cubic function that fits the graph. We use cubic splines as they are a combination of polynomials.

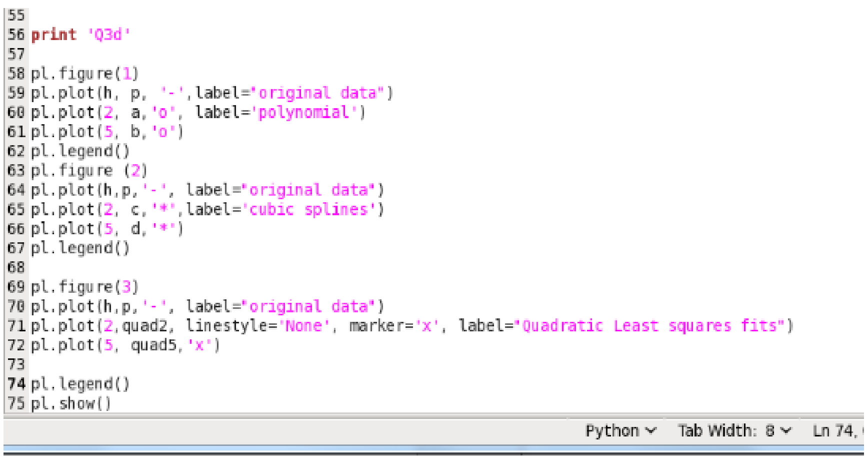
3c)



```
print 'Q3c'

def func(m, a, b, c):
    return a*m**2+b*m+c

m= np.linspace(0,10,500)
popt, pcov=curve_fit(func,h,p) #pot is array of a b and c which is arbitrary constants in themselves
quad2=popt[0]*2**2+popt[1]*2+popt[2]
quad5=popt[0]*5**2+popt[1]*5+popt[2]


print 'Density at 2km using quadratic spline interpolation'
print quad2

print 'Density at 5km using quadratic spline interpolation'
print quad5
```

In order to get non-linear least squares to fit the function I defined the function where m contains set an arbitrary set of values (for me 500), linspace here gives an evenly spaced values within that range. Popt is an array of a, b and c where we have ax^2 +bx+c – i.e the quadratic form. I have set quad2 and quad5 for the densities at 2 and 5 to find a fitted curved function for it.

3d)



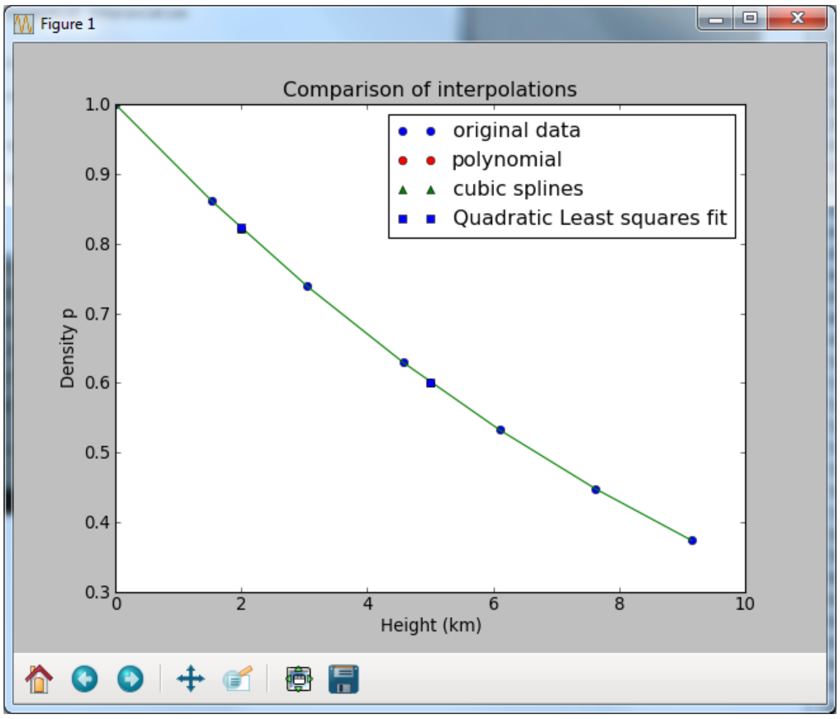
```

print 'Q3d'

pl.figure(1)
pl.plot(h, p, '-',label='original data")
pl.plot(2, a,'o', label='polynomial')
pl.plot(5, b,'o')
pl.legend()
pl.figure (2)
pl.plot(h,p,'-', label="original data")
pl.plot(2, c,'*',label='cubic splines')
pl.plot(5, d,'*')
pl.legend()

pl.figure(3)
pl.plot(h,p,'-', label="original data")
pl.plot(2,quad2, linestyle='None', marker='x', label="Quadratic Least squares fits")
pl.plot(5, quad5,'x')

pl.legend()
pl.show()
```

Python ⌄ Tab Width: 8 ⌄ Ln 74,

Anjelah Balachandran

I put the results all on one graph so it was easier to compare and analyse. I used polyfit and plot imported from scipy to produce the graph. All three data sets produced the same shape curve. As two of the data points (the polynomial interpolation and cubic spline interpolation were identical), the data points were on top of each other, which is why the polynomial orange point is not there to see.

Graphs produced-



Upon close inspection of the data by zooming in, the quadratic least squares fit appears to be the best approximation for the set of interpolations as the data points are closer to the curve produced whilst the cubic splines and polynomial fit are further away from the line.

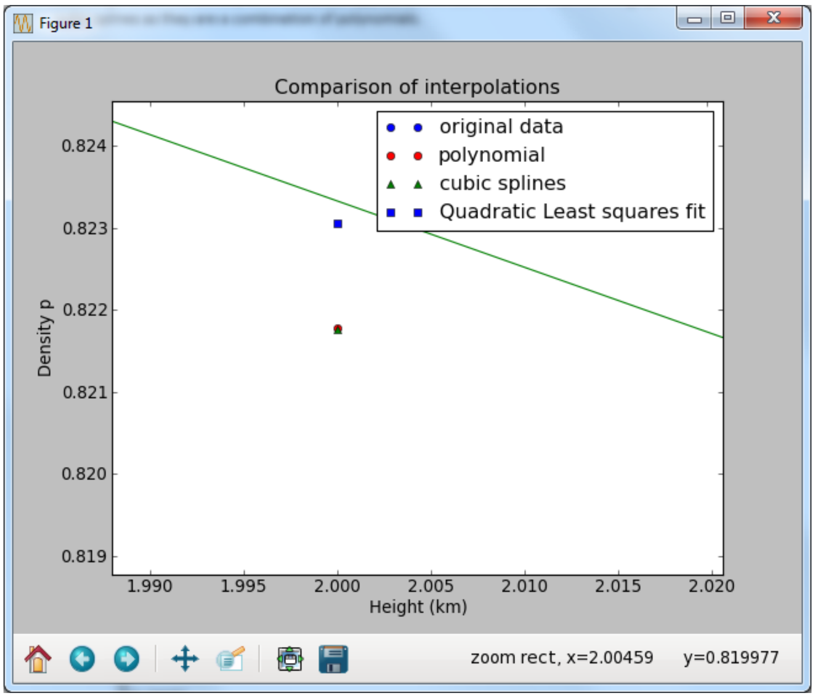


Figure 1
Comparison of interpolations
● ● original data
● ● polynomial
▲ ▲ cubic splines
■ ■ Quadratic Least squares fit
Density p
Height (km)
zoom rect, x=2.00459 y=0.819977