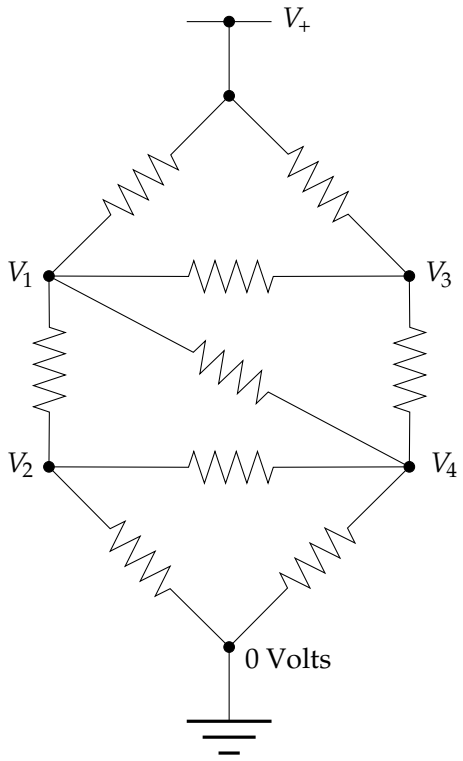
UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. Ilian Iliev, Office: Pev III 4C5

Assignment 1
Deadline: 12pm on Thursday, October 30th, 2014.
Penalties will be imposed for submissions beyond this date.
Final submission date: Friday, October 31st, 2014.
No submissions will be accepted beyond this date.

1. Consider the following circuit of resistors:



All the resistors have the same resistance R. The power rail at the top is at voltage $V_+ = 5\,\mathrm{V}$. What are the other four voltages, V_1 to V_4?

To answer this question we 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. Thus for the junction at voltage V_1, for instance, we have

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

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]

Solution:

a) The analogous equations based on the Ohm's and Kirchoff's laws for the other 3 junctions are:

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

or equivalently

$$3V_2 - V_1 - V_4 = 0.$$

for V_2,

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

or equivalently

$$3V_3 - V_1 - V_4 = V_+.$$

for V_3, and

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

or equivalently

$$4V_4 - V_1 - V_2 - V_3 = 0.$$

for V_4. Therefore, the coefficient matrix A is

$$A = \begin{pmatrix} 4 & -1 & -1 & -1 \\ -1 & 3 & 0 & -1 \\ -1 & 0 & 3 & -1 \\ -1 & -1 & -1 & 4 \end{pmatrix}$$

and the right hand side vector is:

$$b = \begin{pmatrix} V_+ \\ 0 \\ V_+ \\ 0 \end{pmatrix}$$

[10]

b) A sample Python script which sets up this problem is as follows:

```
from numpy import *
import scipy as Sci
import scipy.linalg
from gaussElimin import *
from LUdecomp import *
from numpy import linalg

Vplus=5 #5V
a=array([[4.0, -1.0, -1., -1.],[-1.0, 3.0, 0.0, -1.0],\
        [-1.0, 0.0, 3.0, -1.],[-1.0, -1.0, -1.0, 4.0]])
b=array([Vplus, 0.0, Vplus, 0.0])
pylab.show()
```

The solution is obtained by doing:

```
x1=linalg.solve(a,b) #using the in-build Scipy code
print x1
```

or

```
gaussElimin(a,b) # using the provided Gauss elimination code
print b          # Note that after the call the vector b
                 # contains the solution
```

This yields:

```
[ 3.          1.66666667  3.33333333  2.        ]
[ 3.          1.66666667  3.33333333  2.        ]
```

i.e. the results from the two codes are identical, as expected. The V_2 and V_4 are lower than V_1 and V_3, which makes sense physically, being closer to the high V_+ source and further form the ground. [10]

c) Then, the solution using the internal routine is given simply by:

```
a=array([[4.0, -1.0, -1., -1.],[-1.0, 3.0, 0.0, -1.0],\
          [-1.0, 0.0, 3.0, -1.],[-1.0, -1.0, -1.0, 4.0]])
ac=a.copy()
P,L,U=Sci.linalg.lu(ac)
print('L=',dot(L,P))
print('U=',U)
print('Check',dot(dot(L,P),U))
```

If we used `gaussElimin(a,b)` above we need to re-define the coefficient matrix again, as that code modifies the original one. Alternatively, we could make a copy beforehand, as shown above, and then use that, without modifying the original matrix.

Note that the `lu` code provides a split of A into a product of a lower triangular matrix L, diagonal one P and an upper triangular matrix U, i.e. $A = LPU$. If we just want a split into L and U we can simply multiply L and P above (which again gives a lower triangular matrix, but this new one is with non-empty main diagonal).

The result from running the above commands is:

```
('L=', array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [-0.25      ,  1.        ,  0.        ,  0.        ],
       [-0.25      , -0.09090909,  1.        ,  0.        ],
       [-0.25      , -0.45454545, -0.5       ,  1.        ]]))
('U=', array([[ 4.        , -1.        , -1.        , -1.        ],
       [ 0.        ,  2.75      , -0.25      , -1.25      ],
       [ 0.        ,  0.        ,  2.72727273, -1.36363636],
       [ 0.        ,  0.        ,  0.        ,  2.5       ]]))
('Check', array([[  4.00000000e+00,  -1.00000000e+00,  -1.00000000e+00,
```

```
  -1.00000000e+00],
[ -1.00000000e+00,   3.00000000e+00,   0.00000000e+00,
  -1.00000000e+00],
[ -1.00000000e+00,  -6.93889390e-18,   3.00000000e+00,
  -1.00000000e+00],
[ -1.00000000e+00,  -1.00000000e+00,  -1.00000000e+00,
   4.00000000e+00]]))
```

We see that L and U are indeed lower and upper triangular matrix and their multiplication gives back the original matrix (except that some zeros became really small numbers instead, why do you think that happens?). [10]

d) We setup the two right-hand sides:

```
V0=0.0
b1=array([Vplus, V0, Vplus, V0])
V0=10.0
b2=array([Vplus, V0, Vplus, V0])
```

The only difference when the grounding is imperfect is replacing $V0$ above with non-zero value. We then solve the equations using the standard way. In fact, since the matrices are now triangular we really need to only do the back-substitution phase, making this much faster for different b's, since all the heavy lifting was done in c) above:

```
Lc=L.copy()
Uc=U.copy()

gaussElimin(Lc,b1)
gaussElimin(Uc,b1)

print 'check 1',A,dot(A,b1)

print 'solution 1:',b1

Lc=L.copy()
Uc=U.copy()

gaussElimin(Lc,b2)
gaussElimin(Uc,b2)
```

```
print 'solution 2:', b2
```

```
solution 1: [ 3.          1.66666667  3.33333333  2.        ]
solution 2: [ 3.4         2.33333333  3.66666667  2.6       ]
```

[10]

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`)?

[30]

Solution: We fist have to setup the matrix A and right hand side vector b, e.g. as follows:

```
from numpy import *
from numpy.linalg import *
from gaussPivot import *

A=array([[2.,-1.,0.,0.],[0.,0.,-1.,1.],[0.,-1.,2.,-1.],[-1.,2.,-1.,1.]])
b=array([1.,0.,0.,0.])
```

Important: make sure arrays are filled with floats, not integers!

Then, the solution is given simply by

```
x1=solve(Ac,bc)

print x1
```

Similarly, if we use the provided `gaussPivot` code we do:

```
Ac=A.copy()
bc=b.copy()

x2=gaussPivot(Ac,bc)

print x2
```

Note that here we first made copies of the arrays since the code modifies them. [10]

The result in both cases is as expected:

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

[5]

However, if we call the standard Gauss elimination without pivoting we run into problems:

```
x3=gaussElimin(A,b)

print x3
```

results in:

```
[ nan  nan  nan  nan]
```

i.e. 'Not a Number' because of division by zero accompanied by the appropriate error message. Clearly, even though the system of equations is quite simple and has a well-behaved matrix and an (obvious) solution, in such cases pivoting (i.e.swapping rows in order to avoid dividing by 0 or by a small number) is crucial to avoid such problems.

[15]

3. **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 1: Data for Problem 2.

[30]

Solution:

We first have to import the needed packages and set up the data points given in the problem and the points where we will want the interpolations evaluated, all as arrays:

```
import numpy as np
import scipy
from scipy import interpolate
from scipy import optimize
from scipy.optimize import curve_fit
import pylab as pl

h=np.array([0.0, 1.525, 3.050, 4.575, 6.10, 7.625, 9.150])
rho=np.array([1.0, 0.8617, 0.7385, 0.6292, 0.5328, 0.4481, 0.3741])

h0=np.array([2.0,5.0])
```

[2]

(a) Given N+1 points, a polynomial of order N is required for interpolation. In this case, there are 7 points, so we need a 6th order polynomial.

The polynomial interpolation itself is done by directly calling the appropriate Scipy routine:

```
y0=interpolate.barycentric_interpolate(h,rho,h0)
print 'y0=',y0
```

The result obtained is the polynomial evaluated at the points $h0$, or

```
y0= [ 0.82177677  0.60108587]
```

[7]

(b) Similarly, the cubic spline interpolation is done by directly calling the Scipy routine `interp1d` with the correct options and then evaluating the result at the required points:

```
ci=interpolate.interp1d(h,rho,kind='cubic')
y1=ci(h0)
print 'y1=',y1
```

The result is:

```
y1= [ 0.82176781  0.60108727]
```

[7]

(c) For the least square fitting we have to fist set up the function to be fitted, a quadratic polynomial here, and then call the Scipy routine `curve_fit`, followed by evaluating the result at the required points:

```
def f(x,a,b,c):
    return a*x**2+b*x+c

p1,c1=curve_fit(f,h,rho)
y2=p1[0]*h0**2+p1[1]*h0+p1[2]
print 'y2=',y2
```

The result is:

```
y2= [ 0.82305346  0.60073896]
```

[7]

(d) We plot the data points and the interpolated ones on the same plot, as follows

```
pl.clf()
pl.plot(h,rho)
pl.scatter(h0,y0)
pl.scatter(h0,y1)
pl.scatter(h0,y2)

pl.show()
```

This gives the plot below:

From both the values and the plot itself we see that all three approaches yield very similar values for the density at 2 and 5 km. The two interpolation results are practically identical, reflecting the fact that the data points provided suggest a very smooth, well-behaved function. The plot line itself is a connect-the-dots, i.e. a linear spline

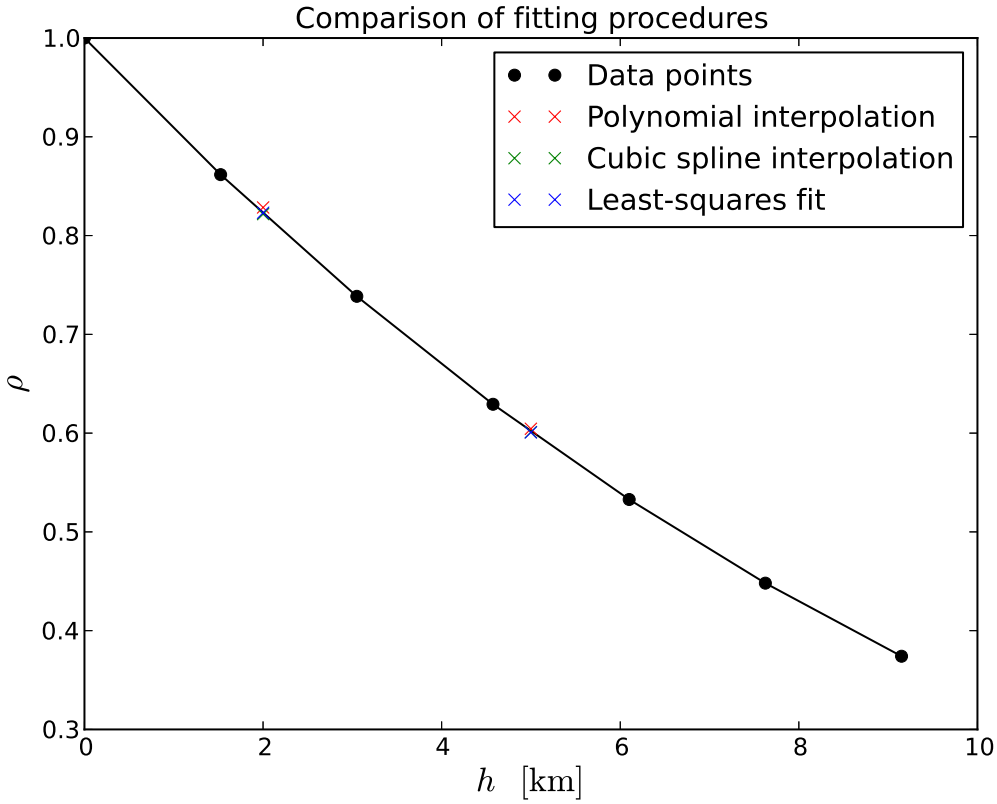


Figure 2: Plot for Problem 3.

interpolation, and the more sophisticated interpolation results can only be distinguished by zooming the plot. The least square fitting gives also similar results, but nevertheless slightly different ones, at sub-percent level. We can expect that approach to be least precise since it uses lowers number of parameters. This however is the best approach if the data itself has some (e.g. experimental) errors, since in that case ensuring that the curve goes precisely through all the data points (as interpolation does) makes little sense and can result in spurious oscillations. [7]