UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. Ilian Iliev, Office: Pev 3 4A5

Problem Sheet 1 (Problem 1 will be assessed)
Deadline: 12pm on Wednesday, October 30th, 2013.
Penalties will be imposed for submissions beyond this date.
Final submission date: Thursday, October 31st, 2013
No submissions will be accepted beyond this date.

1. Use the built-in NumPy function `random.random` to create the $n \times n$ random matrix A, in which all elements have values between 0 and 10.

   (a) Use this to create an $n \times n$ matrix A for n=50, 100, 200, 300, and 400 using a `for` loop.

   (b) For this matrix A construct a vector b so that the solution to $Ax = b$ is $x = [1, 2, \dots n]$.

   (c) Test your construction by applying Gauss elimination to compute x. What is the largest element-by-element error for each n? Plot the decimal log of the max element-by-element error vs. n. Comment on the results. [30]

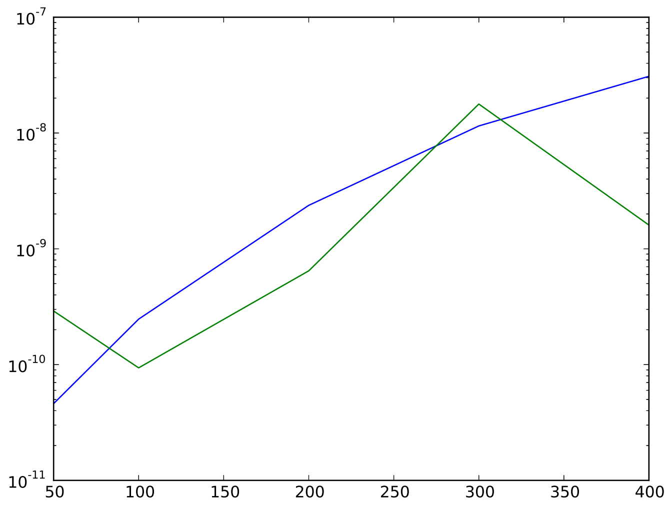
   Solution: A sample Python script which solves this problem is as follows:

```
from numpy import *
from gaussElimin import *
from numpy import linalg
import pylab

xx=zeros(5) # we will store here the errors for plotting
y=zeros(5)
ii=

for i in (50,100,200,300,400):
      #fill i x i matrix with random values between 0 and 10
      a=10.*random.random((i,i))

      x0=zeros(i)  #initialize solution vector
      for j in range(i):
            x0[j]=j+1

        b=dot(a,x0) #right hand side that should give solution (1,1, ..., 1)

        #linalg.solve(a,b)

        gaussElimin(a,b) # call provided Gauss elimination code
        print max(b-x0)

        print(i,ii)
        xx[ii]=i        #store i's and corresponding errors for plotting
        y[ii]=max(b-x0)
        ii=ii+1

pylab.semilogy(xx,y)   #plot errors
pylab.show()
```

This yields:

```
In [33]: run prob1_random.py
2.90263812985e-10
(50, 0)
9.37205868468e-11
(100, 1)
6.4517280407e-10
(200, 2)
1.7782156192e-08
(300, 3)
1.60294177931e-09
(400, 4)
```

Errors are random, but generally increase with matrix size, due to compounding of the round-up errors (you can run the code several times to check that). The plot for my values (two sets of errors) is given below:



2. (a) Find the upper- and lower-triangular matrices **L** and **U** so that

$$A = LU = \begin{pmatrix} 4 & -1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 4 & -1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 4 & -1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & -1 & 4 & -1 \\ 0 & 0 & \cdots & 0 & 0 & -1 & 4 \end{pmatrix}$$

(4's along the main diagonal and -1's along the next 2 diagonals), where A is a 5×5 matrix. You can use either the provided LU decomposition code or the build-in `scipy.linalg.lu` function. In either case, 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). If you have time, try both methods. Do the results match?

(b) Solve the system $Ax = b$ using the LU decomposition of the matrix A from (a) above and right-hand sides given by $b = (1.0, 2.0, 4.0, 6.0, 3.0)$ and $b = (-3.0, 5.0, -4.0, 0.0, 4.0)$. Check the solutions you have found (note that the provided codes for Gaussian elimination and LU decompositions *modify* the matrix A, so make a copy first).

Solution: (a) We fist have to setup the matrix A, e.g. as follows:

```
from numpy import dot,zeros
```

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

n=5

A=zeros((n,n))

A[0,0]=4.
for i in range(1,n):
    A[i,i]=4.
    A[i-1,i]=-1.
    A[i,i-1]=-1.
```

Note the imported modules above: `scipy` and `scipy.linalg` are needed for the in-build LU decomposition routine, while `LUdecomp` is the routine from our textbook, provided on Study Direct.

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

```
Ac=A.copy()
P,L,U=Sci.linalg.lu(Ac)
print('L=',dot(L,P))
print('U=',U)
```

Note that this 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 one is with non-empty main diagonal).

The result from running the above commands is:

```
('L=', array([[ 1.        ,  0.        ,  0.        ,  0.        ,  0.
       [-0.25      ,  1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        , -0.26666667,  1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -0.26785714,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , -0.26794258,  1.        ]]))
('U=', array([[ 4.        , -1.        ,  0.        ,  0.        ,  0.
       [ 0.        ,  3.75      , -1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  3.73333333, -1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  3.73214286, -1.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  3.73205742]]))
```

Similarly, using the provided LU decomposition code from the textbook, we do:

```
Ac=A.copy()
a=LUdecomp(Ac)

print('L\U=',a)
```

obtaining as a result:

```
('L\\U=', array([[ 4.        , -1.        ,  0.        ,  0.        ,  0.
       [-0.25      ,  3.75      , -1.        ,  0.        ,  0.        ],
       [ 0.        , -0.26666667,  3.73333333, -1.        ,  0.        ],
       [ 0.        ,  0.        , -0.26785714,  3.73214286, -1.        ],
       [ 0.        ,  0.        ,  0.        , -0.26794258,  3.73205742]]))
```

Here the L and U matrices are included into one matrix (in order to save space we do not need to store the zeros), where everything on and below the main diagonal belongs to L and the elements above the main diagonal belong to U.

The results from the two LU decompositions agree. This means that the same LU decomposition method (Doolittle reduction) is used in both codes. Remember that we mentioned in class that the LU decomposition is not unique and there exist other ways to do it.

We can check our results by multiplying LPU:

```
print('Check',dot(dot(L,P),U))
```

which gives:

```
('Check', array([[ 4., -1.,  0.,  0.,  0.],
       [-1.,  4., -1.,  0.,  0.],
       [ 0., -1.,  4., -1.,  0.],
       [ 0.,  0., -1.,  4., -1.],
       [ 0.,  0.,  0., -1.,  4.]]))
```

i.e. we recover the original matrix A.

(b) We first set up the two right-hand sides:

```
b1=array([1.,2.,4.,6.,3.])
b2=array([-3.,5.,-4.,0.,4.])
```

Then, we make independent copies of the L and U (remember that the Gauss elimination code changes them, so we do not want to overwrite the originals) and call the Gauss elimination code twice (method is as shown in the lecture notes):

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

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

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

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

Note here that because of the LU decomposition, we actually do not need the full Gauss elimination, but can only do the back-substitution phase (challenge the better ones in the class to do that!).

We then have the solution, and can also check it by multiplying it with the original matrix A:

```
print 'solution 1:',b1
print 'check 1',A,dot(A,b1)

print 'solution 2:', b2
print'check 2',dot(A,b2)
```

resulting in:

```
solution 1: [ 0.52307692  1.09230769  1.84615385  2.29230769  1.32307692]
```

```
check 1 [[ 4. -1.  0.  0.  0.]
 [-1.  4. -1.  0.  0.]
 [ 0. -1.  4. -1.  0.]
 [ 0.  0. -1.  4. -1.]
 [ 0.  0.  0. -1.  4.]] [ 1.  2.  4.  6.  3.]
solution 2: [-0.51666667  0.93333333 -0.75        0.06666667  1.01666667]
check 2 [ -3.00000000e+00   5.00000000e+00  -4.00000000e+00  -5.55111512e-17
   4.00000000e+00]
```

which shows that both solutions are indeed correct.

Note that in the second solution we get -5.55111512e-17 instead of the original 0, but withing the computer precision they are the same. This occurs due to round-off errors.

As a general note, we of course can also include all above commands into a script e.g. `prob1.py` and run that.