```
Instructor: Dr. Kenneth Duru
First Semester 2019
Mathematical Sciences Institute
Australian National University
```

- Math3511, Scientific Computing

Lab 1: Rounding Errors and Linear Systems of Equations

A. Rounding errors and error propagation

```
%matplotlib inline
import numpy as np
import scipy as sp
import math
```

A1. Implementing rounding in Python

<span, style="color:blue">1. (5 points) Discuss the function **frexp(x)** from the Python math module. In your own words describe what it does. Give some examples to demonstrate and show that the results are as expected (i.e. for your demonstrations work out the mantissa and exponent by hand).

<span, style="color:limegreen">**Answer**:

<span, style="color:blue">2. (5 points) Consider the function **phi** below which uses **frexp(x)**. We claim that this function rounds any real number to t binary digits. Give some examples to show this is true.

```
def phi(x,t):
    """
    rounding function to t binary digits
    """
    (xm, xe) = math.frexp(x)
    xr = round(xm*2.0**t)/2.0**t
    return math.ldexp(xr, xe)
```

<span, style="color:limegreen">**Answer**:

A2. Errors in function evaluation

<span, style="color:blue">3. (5 points) Consider the expression $x = (1.0 - \cos(1.2 * 10^{-5}))/(1.2 * 10^{-5})^2$. Note that this is evaluated in the following steps.

```
a1=1.2e-5
x1=cos(a1)
x2=1.0-x1
x3=a1*a1
x4=x2/x3
```

<span, style="color:blue">Recall the number of binary digits Python uses for the mantissa of floating point numbers. Use **frexp** to demonstrate that one of computations has significantly fewer digits than expected. This is called **cancellation**.

Answer:

4. (10 points) Use the function **phi** from the previous section to round the result of each step to 20 binary digits before continuing with the next step. Again, use **frexp** to show that the computations lose a substantial number of digits.

Answer:

5. (5 points) Compare the rounded result (**x4_rounded**) with the "exact" **x4** and compute the relative error.

Answer:

B. Linear Equations

B1. Matrix operations

We now take a look at linear systems of equations with Python. We start with the *matrix* object which are a subclass of the numpy arrays (ndarray). The matrix objects inherit all the attributes and methods of ndarry. The difference is, that numpy matrices are strictly 2-dimensional, while numpy arrays can be of any dimension. A matrix can be defined as follows:

```
A = np.matrix([[1,2],[3,4]]) #creates a matrix
c = np.matrix([[1],[2]])     #creates a column vector
r = np.matrix([[1,2]])       #creates a row vector
```

The most important advantage of matrices is that they provide convenient notations for the matrix multiplication. If A and B are two matrices then $A * B$ defines the matrix multiplication. While on the other hand, if A and B are ndarrays, $A * B$ define an element-by-element multiplication.

```
A = np.matrix([[1,2],[3,4]])
B = np.matrix([[4,3],[2,1]])
A*B
```

```
A = np.array([[1,2],[3,4]])
B = np.array([[4,3],[2,1]])
A*B
```

If we want to perform matrix multiplication with two numpy arrays (ndarray), we have to use the dot product:

```
np.dot(A,B)
```

Alternatively, we can cast them into matrix objects and use the $*$ operator:

```
np.matrix(A)*np.matrix(B)
```

The size of a matrix `A` can be determined by `A.shape`. The length of a column vector `c` is calculated by `len(c)`. Here are the examples:

```
M = np.matrix([[1,2,3],[4,5,6]])
M.shape
```

Try the above for matrices and arrays of larger size.
You can also find the transpose and inverse of a matrix `A` by using `A.T` and `A.I` respectively.

For a matrix `A`, we may use `A[i,:]` and `A[:,j]` to produce the $(i+1)$-th row and $(j+1)$-th column respectively. In general we may use `A[m:n, p:q]` to produce the sub-matrix consisting of elements $a_{i,j}$ with $i = m+1,\dots,n$ and $j = p+1,\dots,q$. Try out some examples to see these commands in work.

Look into the functions **ones** and **diag** in the numpy module. These are very useful.

B2. LU factorization

A common way to understand (and in fact construct) many linear algebra methods is via matrix decompositions or factorisation techniques. This allows us to rewrite our original problem into a collection of simpler problems.

Gaussian elimination can be considered as the application of a sequence of elementary matrices that can be encoded into a lower triangular matrix L, and leads to an upper triangular matrix U. Python provides the command `lu` to calculate LU decompositions.

Once we have the LU factorisation of a matrix A, say $A = LU$, we can use it to solve the linear system $A\mathbf{x} = \mathbf{b}$ by the following two steps:

1. Solve $L\mathbf{y} = \mathbf{b}$ by forward substitution.
2. Solve $U\mathbf{x} = \mathbf{y}$ by back substitution.

Let's have a look at the LU factorisation. For illustrations, we use the following matrix

```
A = np.matrix([[-2,  1,  0,  0,  0],
             [ 1, -2,  1,  0,  0],
             [ 0,  1, -2,  1,  0],
             [ 0,  0,  1, -2,  1],
            [ 0,  0,  0,  1, -2]])
A
```

By using the Python command **`lu`** we can produce the L and U factors of A. Here is the code:

```
from scipy.linalg import lu
(L,U) = lu(A, permute_l=1)
```

and here are the matrices

```
L
```

```
U
```

Notice that both of these matrices maintain the same banded structure as the original matrix A. Notice that the inverse of A loses the banded structure. Here is the inverse

```
A.I
```

This is typical, and you should try to avoid calculating inverses in preference to LU factorisations.

Verify that **`L*U=A`**. Be careful here, remember that **`*`** has different results depending on **`L,U`** being arrays or matrices.
Also check that the inverse matrix above is in fact the inverse (or close enough).

Now let's use the decomposition to solve a matrix equation. Take $\mathbf{b} = [1 \;\; 1 \;\; \cdots \;\; 1]^T$. To solve $A\mathbf{x} = \mathbf{b}$ we can use

```
#create a vector b
b = np.matrix([[1],[1],[1],[1],[1]])
```

```
from scipy.linalg import solve
y = solve(L,b)
x = solve(U,y)
```

```
#check the output of x
x
```

Actually we should use specially designed forward and backward substitution operators to undertake the L and U solves. In this case, Python uses a slightly different format to store the L and U matrices.

```
from scipy.linalg import lu_factor, lu_solve
Alu = lu_factor(A)
x_new = lu_solve(Alu,b)
```

```
#check the output of x_new and compare to x
x_new
```

6. (10 points) Use the LU factorization from **SciPy** to solve the equation $Ax = b$ where $b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$ and A is the matrix from the example above.

Answer:

B3. Gaussian elimination for linear systems

It is the time for us to write our own python function to implement the Gaussian elimination for solving linear system $A\mathbf{x} = \mathbf{b}$. We start from the Gaussian elimination with no pivoting whose pseudo code is:

```
M = [A  b]
    for k=1:n-1
      for i=k+1:n
         q = M(i,k)/M(k,k)
         for j = k:n+1
             M(i,j) = M(i,j) - q*M(k,j)
         end
      end
    end
    x(n) = M(n,n+1)/M(n,n)
    for i = n-1:-1:1
        z = 0
        for j=i+1:n
            z = z + M(i,j)*x(j)
        end
        x(i) = (M(i,n+1)-z/M(i,i)
    end
```

In the above algorithm, we use the augmented matrix M by augmenting b to the coefficient matrix A. In python this can be realized by using `column_stack` from numpy:

```
A = np.matrix([[1,2,3],[4,5,6],[7,8,9]])
b = np.matrix([[1],[2],[3]])
M = np.column_stack((A,b))
print(M)
```

Below is the python function for implementing the Gaussian elimination method to solve linear systems without pivoting.

```
def GENP(A, b):
    #    Gaussian elimination for solving A x = b with no pivoting.
    n =  len(A)
    M = np.column_stack((A,b))
    for k in range(n-1):
        for i in range(k+1, n):
            multiplier = M[i,k]/M[k,k]
            for j in range(k,n+1):
                M[i,j] = M[i,j] - multiplier*M[k,j]
    x = np.zeros((n,1))
    k = n-1
    x[k] = M[k,n]/M[k,k]
    for i in range(n-1,1,-1):
        z = 0
        for j in range(i+1,n):
            z = z + M[i,j]*x[j]
        x[i] = (M[i,n]-z)/M[i,i]

    #while k >= 0:
    #   x[k] = (M[k,n] - np.dot(M[k,k+1:n],x[k+1:]))/M[k,k]
    #   k = k-1
    return x
```

7. (30 points) Study the above python code and explain what each loop is doing to the matrix M. Use the code above to solve the linear system

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -2 & -2 & -2 \\ 1 & 4 & -4 & 1 \\ 1 & -5 & -5 & 3 \end{bmatrix} x = \begin{bmatrix} 0 \\ 4 \\ 2 \\ -4 \end{bmatrix}$$

Answer:

Since not every square matrix can have an LU factorisation, it is necessary to use pivoting when Gaussian elimination is used to solve linear system. We have included the pseudo code of Gaussian elimination with partial pivoting in the lecture notes; it takes the following form (with slight modification):

```
M = [A  b]
for k=1:n-1
  select i >= k to maximize |M(i,k)|
  M(k,k:n) <--> M(i,k:n)    (interchange two rows)
  for j=k+1:n
     q = M(j,k)/M(k,k)
     M(j,k:n+1) = M(j,k:n+1) - q*M(k,k:n+1)
  end
end
x(n) = M(n,n+1)/M(n,n)
for i = n-1:-1:1
   z = 0
   x(i) = (M(i,n+1)-M(i, i+1:n)*x(i+1:n))/M(i,i)
end
```

8. (30 points) Write a Python function to implement Gaussian elimination with partial pivoting by adapting the above code for Gaussian elimination with no pivoting (GENP) and test your code with the linear system in the previous exercise.
The key part you need to think over is to implement the pivot selection.

Answer: