Instructor: Dr. S. Hughes

Due date: Sun 4th Feb, 2023 (midnight)

Lab 3 - Matrices and Systems of Linear Equations

Keywords: Matrix Methods, NumPy, System of Linear Equations, Back Substitution, Gaussian Elimination, Partial Pivoting

Task: Go through all the questions and coding exercises below and hand in a single modularized Python code where each part (or cell) runs each part of the question(s). Remember and include your surname at the beginning of the code name.

Code Submissions on onQ: Attach a single (preferable, or no more than two) .py (called LabX_YourLastName.py) **and** a matching .py.txt (same code, same name, just a different extension).

Marks: Coding Skills, Efficiency, Correct Results and Clear Graphs and Good Presentation of Results (6); Good use of Comments (2); Good Coding Structure and Readability (2). Total: (10)

Late Assigments: Late assignments will not be marked unless accompanied with a usable slip day, please indicate how many you are using and how many you have left.

Reminder: All codes must run under Python 3.x. and the Spyder IDE (check this before submitting). Section off the code by using #%% (and label section in code), and add useful comments, so we can step through it easily.

Acknowledgements: Please comment on any help that you received from your group members or others concerning this Lab assignment.

Some Background (in addition to lecture notes):

Gaussian Elimination - Wikipedia
Python Matrices
NumPy - Linear Algebra
SciPy - Linear Algebra

Question 1

(a) Use NumPy to create a 2d square matrix, $\boldsymbol{A}$, of size $n \times n$, where n is an input integer, and the array elements will begin at 21 and increase by 1 on each subsequent element, e.g., $A[0,0] = 21, A[0,1] = 22, \cdots$. Check the function returns the following for $n = 4$ (print it out):

$$\boldsymbol{A} = \begin{bmatrix} 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \end{bmatrix}. \tag{1}$$

Next, use your own Python code to (i) find the lower ($\boldsymbol{L}$) and upper ($\boldsymbol{U}$) triangular part of a given matrix (without affecting $\boldsymbol{A}$), as well as (ii), the *Euclidean norm* (or Frobenius norm),

$$||\boldsymbol{A}|| \equiv ||\boldsymbol{A}||_E = \sqrt{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} |A_{ij}|^2}, \tag{2}$$

and (iii), the *infinity norm* (or maximum row-sum norm):

$$||\boldsymbol{A}||_\infty = \max_{0\le i\le n-1} \sum_{j=0}^{n-1} |A_{ij}|\,. \tag{3}$$

Apply these to the example above, print out the answers. Note for (ii) and (iii), these can be done in one line of code without any `for` loops using basic operations from the numpy module, such as `sum`, `max`, and `sqrt`.

You can check the solutions to (ii) and (iii) using NumPy's functions, namely `np.linalg.norm(A)` and `np.linalg.norm(A,np.inf)`.

(b) The solution to certain linear equations and large matrix problems in general may be ill-posed numerically, in the sense that small fluctuations of input numbers can have a drastic effect on the solution. One way to quantify the degree of "ill-conditioning" is through the *condition number*:

$$\kappa(\boldsymbol{A}) = ||\boldsymbol{A}||\,||\boldsymbol{A}^{-1}||, \tag{4}$$

where $\mathbf{A}^{-1}$ is the inverse matrix which you can also compute from NumPy (for this question), through `np.linalg.inv(A)`. For $\kappa \gg 1$, then the problem is generally ill-conditioned, and may not be numerically stable with respect to small fluctuations of the input variables. We will explore this is more detail below.

(i) Study the NumPy documentation to set up the following matrix,

$$\boldsymbol{A} = \begin{bmatrix} 1 & -1 & -1 & \cdots & -1 \\ 0 & 1 & -1 & \cdots & -1 \\ 0 & 0 & 1 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{5}$$

and, of course, your code should be written so as to be able to work for any value of n.

(ii) For $n = 4$, solve the system of linear equations $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, using the NumPy routine `np.linalg.solve(A)` (we will use this later to check our own code), when $\boldsymbol{b}$ is made up of alternating $+1$'s and -1's, with and without a small perturbation of -0.001 in the bottom-left element of A. Print and compare the first 3 x solutions, namely $x[0], x[1], x[2]$ (with and without the small perturbation).

(iii) Repeat for $n = 16$, evaluating the *condition number* in both cases.

Question 2

(a) In the lecture notes, we were shown an example Python code for *back substitution*:

```
def backsub1(U, bs):
    n = bs.size
    xs = np.zeros(n)

    xs[n-1] = bs[n-1]/U[n-1, n-1] # bb is not needed for this case
    for i in range(n-2, -1,-1):
        bb = 0
        for j in range (i+1, n):
            bb += U[i, j]*xs[j]
        xs[i] = (bs[i] - bb)/U[i, i]
    return xs
```

Unfortunately this code is not very efficient. To quantify this, test the speed of the code, using the following code snippet (which uses the `timeit` module in Python) which might look something like this

```
from timeit import default_timer
timer_start = default_timer()
mysolve(backsub1,U,bs)
timer_end = default_timer()
time1 = timer_end-timer_start
print('time1:', time1)
```

and set up an example $\boldsymbol{U}(n)$ from Q1(a), with $n = 5000$, and set $\boldsymbol{b}$ equal to the first *row* of $\boldsymbol{U}$.

Next, write your own subroutine for back substitution, say `backsub2`, and try and make it more efficient using vectorization. Time your new code and compare it with the one above. You should find orders of magnitude improvement in the run-time. You should only need one loop now, for i. For my tests, the run-time changed from ≈ 2.6 s to ≈ 0.01 s, with $n = 5000$. For both subroutines, write out the first three elements of x and check they are the same. Of course, you could also check this using NumPy (which runs in about 0.3 s, so you can do better!).

To make your code clear, use a small subroutine where you can pass the back substitution function, so your subroutine might look something like this:

```
def mysolve(f,A,bs):
    xs = f(A,bs);
    print('my solution is:',xs[0],xs[1],xs[2])
```

Note `f` is your chosen function, for example, you could call with `mysolve(backsub1,U,bs)`.

(b) Write a subroutine that implements Gaussian Elimination (see lecture notes), and employs your `backsub2` routine. Use it to solve the following system of linear questions, in matrix form:

$$\boldsymbol{A} = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ -2 \\ 2 \end{bmatrix} \tag{6}$$

and confirm the solution is $\boldsymbol{x}^T = \begin{bmatrix} 4 & -2 & 2 \end{bmatrix}$.

(c) Next, modify your Gaussian Elimination code to account for partial pivoting, and use it to confirm the solution to the following matrix problem:

$$\boldsymbol{A} = \begin{bmatrix} 2 & 1 & 1 \\ 2 & 1 & -4 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ -2 \\ 2 \end{bmatrix}, \tag{7}$$

is the same $\boldsymbol{x}^T$ as in (b). Note, if you use without pivoting, the solution will return $[nan]$ "not a number", so pivoting is required here.