Linear systems and complexity

Question 1 **20 marks**

Consider the following linear system:

$$A\vec{x} = \vec{b}, \text{ where } A = \begin{bmatrix} 3 & -21 \\ -4 & 27 \end{bmatrix} \text{ and } \vec{b} = \begin{bmatrix} -12 \\ 15 \end{bmatrix} \text{ with the exact solution } \vec{x} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Suppose that you find an approximate solution

$$\vec{x}_* = \begin{bmatrix} 3.00712 \\ 1.00102 \end{bmatrix}$$

by some numerical method. Check that the following inequality holds:

$$\boxed{\begin{matrix}\text{relative error} \\ \text{of } \vec{x}_*\end{matrix} \leq \left(\begin{matrix}\text{condition} \\ \text{number}\end{matrix}\right)\left(\begin{matrix}\text{relative residual} \\ \text{of } \vec{x}_*\end{matrix}\right)}$$

The relative error is

$$\frac{\|\vec{x} - \vec{x}_*\|}{\|\vec{x}\|} = 2.2 \times 10^{-3}$$

and the relative residual is

$$\frac{\|A\vec{x}_* - \vec{b}\|}{\|\vec{b}\|} = 4.9 \times 10^{-5}$$

The condition number of the matrix (computed with `np.linalg.cond(.)`) is 398.33…
Indeed

$$2.2 \times 10^{-3} < 398.33 \times 4.9 \times 10^{-5} = 1.95 \times 10^{-2}$$

Question 2 **20 marks**

Let the matrix A be given by

$$A = \begin{bmatrix} 1 & -2 & 5 \\ 3 & 6 & 1 \\ -2 & -2 & 2 \end{bmatrix}.$$

and the vector R by

$$R = \begin{bmatrix} -3 \\ 2 \\ 0 \end{bmatrix}$$

(**a**) Compute the decomposition $PA = LU$ using partial pivoting, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. Write down the intermediary results, following the examples in lecture 7.

$$\begin{bmatrix} 1 & -2 & 5 \\ 3 & 6 & 1 \\ -2 & -2 & 2 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 6 & 1 \\ 1 & -2 & 5 \\ -2 & -2 & 2 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 6 & 1 \\ 0 & -4 & 14/3 \\ 0 & 2 & 8/3 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -2/3 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\underset{U}{\begin{bmatrix} 3 & 6 & 1 \\ 0 & -4 & 14/3 \\ 0 & 0 & 5 \end{bmatrix}} \quad \underset{L}{\begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -2/3 & -1/2 & 1 \end{bmatrix}} \quad \underset{P}{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}}$$

(**b**) Using the result of (a), compute the solution to $Ax = R$.

Remember the order of solving: first use forward substitution to find y from $Ly = PR$ and then use backward substitution to find x from $Ux = y$. For forward substitution, we start from the first equation (first row of L), which we can solve directly as $x_1 = (PR)_1/L_{11}$ (with the first element of the permuted right-hand-side vector on the right-hand side). Moving to the second equation, and substituting this partial solution, we can solve directly for x_2, and so forth. For the second system, with matrix U, the idea is the same, but we start from the last equation and work our way backwards. There algorithms have been implemented in the module `LUP.py` in the course_codes repository. Following this recipe, we find that

$$y = \begin{bmatrix} 2 \\ -11/3 \\ -1/2 \end{bmatrix} \qquad x = \begin{bmatrix} -9/10 \\ 8/10 \\ -1/10 \end{bmatrix}$$

Question 3 **60 marks**

The matrix $A \in \mathbb{R}^{n\times n}$ has the form

$$A = \begin{bmatrix} a_1 & c_1 & & & & & b_1 \\ b_2 & a_2 & c_2 & & & & \\ & b_3 & a_3 & c_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & b_{n-2} & a_{n-2} & c_{n-2} & \\ & & & & b_{n-1} & a_{n-1} & c_{n-1} \\ c_n & & & & & b_n & a_n \end{bmatrix} \tag{1}$$

where $\vec{a}$, $\vec{b}$, $\vec{c} \in \mathbb{R}^n$. The entries of $\vec{a}$, $\vec{b}$, and $\vec{c}$ are all assumed to be nonzero.

(**a**) Write a pseudo-code for computing the product of this matrix with a vector. That is, given vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$ that describe the matrix A and given a vector $x \in \mathbb{R}^n$, we want to have an algorithm to compute the matrix-vector product $\vec{y} = A\vec{x}$ that avoids multiplying by, and adding, zeros.

```
Input: vector $\vec{x} \in \mathbb{R}^n$; vectors $\vec{a}$, $\vec{b}$, $\vec{c} \in \mathbb{R}^n$ that define matrix $A$ according to definition (1).
    1. Initialize $y$ as a vector of length $n$.
    2. $y_1 = a_1x_1 + c_1x_2 + b_1x_n$
    3. $y_n = c_nx_1 + b_nx_{n-1} + a_nx_n$
    4. Do for $i = 2, \dots, n-1$:
      a. $y_i = b_ix_{i-1} + a_ix_i + c_ix_{i+1}$
Output: vector $\vec{y} \in \mathbb{R}^n$ such that $\vec{y} = A\vec{x}$
```

Note, that only the first and the last element of the product follow a different pattern, that is why I wrote them separately. Some of you have chosen to keep only the loop and write some additional code to map indices in a cyclic fashion, which is fine.

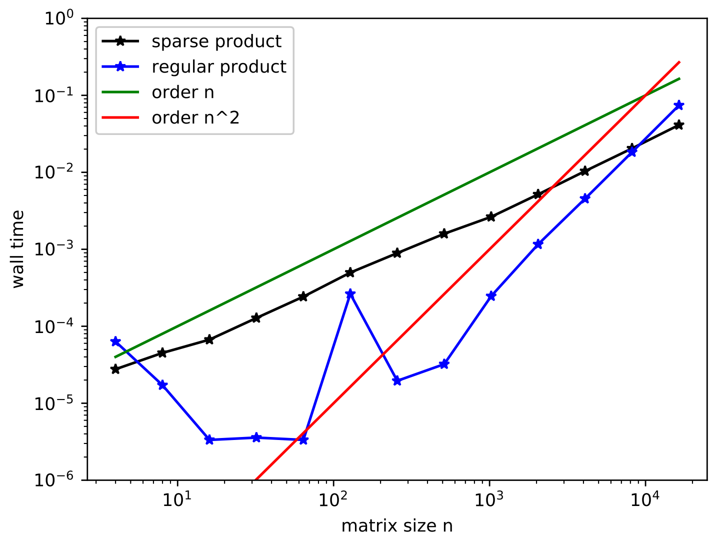
(**b**) Analyse the complexity of the algorithm from part (**b**). That is, determine how many flops are required to compute the product. In terms of "Big-Oh" notation, what is the asymptotic behaviour of your algorithm as n increases? How is that different from ordinary matrix-vector multiplication?

There are 5 FLOPS in lines 2 and 3 and there are 5 in line 4a, inside the loop. All together

$$\#\text{FLOPs} = 5 + 5 + \sum_{i=2}^{n-1} 5 = 10 + 5(n-2) = 5n$$

Thus, the complexity is of order $\mathrm{O}(n)$, whereas the complexity of the general matrix-vector product is $\mathrm{O}(n^2)$.

(**c**) Implement your pseudo-code following the starter code in the repository.



(d) Write a script that does the following:

1. it generates random vectors $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{x}$ of size n;
2. it calls your matrix-vector multiplication function and measures and stores the wall time it takes to complete;
3. it computes the matrix-vector product using the built-in function `@` and measures and stores the wall time it takes to complete;
4. it repeats steps 1-3 for matrix sizes $n = 2^k$, starting from $k = 2$.
5. it plots the wall times for each method as a function of n on the appropriate scale to verify the order of complexity you found at (b);
6. for comparison, it plots lines corresponding to n and n^2.

Make sure your script uses matrices large enough to clearly see the scaling (order of complexity). Follow the starter code in the repository. **Save the plot that your script produces and include it in your LaTeX/PDF document.**

Note, that a log-log scale is the best choice here since we expect the wall time to grow algebraically with the matrix size so that it should show up as a straight line on that scale. Also, on my laptop I needed to go up to $n = 2^{13}$ to clearly see the scaling of my function as n and that of the built-in function as n^2. The result is shown below.
The measurements are a bit noisy for matrix sizes below $n = 500$, where the wall time drops below millisecond. For larger matrix sizes, the predicted orders of complexity are indeed observed.
As explained in class, always carefully consider what information you want your plot to convey, and check such issues as

1. Is the amount of data in my plot appropriate (is it too sparse or too crowded)?
2. Are the axis scales (linear or logarithmic) and limits appropriate (not too much empty space)?
3. Are there spurious data (e.g. values close to zero that get mapped out off the graph on a logarithmic scale, obvious noise, NaN of Inf entries, ...) and can I safely remove them?
4. Are my axes labeled and is there a legend or caption to explain the different colours, line styles and objects in the graph?

There are many more considerations that go into proper visual representation of data, but this is a good checklist to start with!