Linear systems and complexity

Question 1 **10 marks**

Consider the following matrices and vector:

$$A = \begin{pmatrix} 1 & 2 & -4 \\ 2 & 4 & 1 \\ -2 & -1 & 3 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 0 & -4 \\ 2 & -4 & 1 \\ 2 & 10 & 3 \end{pmatrix} \qquad C = \begin{pmatrix} 2 & 0 \\ 50 & 1 \end{pmatrix} \qquad R = \begin{pmatrix} 2 \\ -3 \\ 1 \end{pmatrix}$$

(**a**) Compute the decomposition $PA = LU$. Show the intermediary steps like we did in lecture 6.

Solution:

$$\begin{pmatrix} 1 & 2 & -4 \\ 2 & 4 & 1 \\ -2 & -1 & 3 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 4 & 1 \\ 1 & 2 & -4 \\ -2 & -1 & 3 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 4 & 1 \\ 0 & 0 & -9/2 \\ 0 & 3 & 4 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \quad P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 4 & 1 \\ 0 & 3 & 4 \\ 0 & 0 & -9/2 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1/2 & 0 & 1 \end{pmatrix} \quad P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

(**b**) Use the decomposition you found at (**a**) to solve $Ax = R$.

Solution:

$$Ly = PR \;\Rightarrow\; y = \begin{pmatrix} -3 \\ -2 \\ 7/2 \end{pmatrix} \quad \text{(by forward substitution)}$$

$$Ux = y \;\Rightarrow\; x = \begin{pmatrix} -50/27 \\ 10/27 \\ -7/9 \end{pmatrix} \quad \text{(by backward substitution)}$$

(**c**) Use Gaussian elimination to solve $Bx = R$. Show intermediary steps like we did in lecture 5.

Solution:

$$\left(\begin{array}{ccc|c} 2 & 0 & -4 & 2 \\ 2 & -4 & 1 & -3 \\ 2 & 10 & 3 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 2 & 0 & -4 & 2 \\ 0 & -4 & 5 & -5 \\ 0 & 10 & 7 & -1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 2 & 0 & -4 & 2 \\ 0 & -4 & 5 & -5 \\ 0 & 0 & 39/2 & -27/2 \end{array}\right) \rightarrow x = \begin{pmatrix} -5/13 \\ 5/13 \\ -9/13 \end{pmatrix}$$

using backward substitution in the last step.

(**d**) Compute the condition number of C. If we solve $Cx = Q$, where Q is a 2-vector of which we know the entries up to 5 digits of precision, then how accurately can we compute x?

Solution: the singular values of C are the square roots of the eigenvalues of C^tC:

$$C^tCv = \lambda v \;\text{ gives } \lambda_1 = 2.5 \times 10^3,\; \lambda_2 = 1.6 \times 10^{-3},\; \text{so that } K(C) = \frac{\sqrt{\lambda_1}}{\sqrt{\lambda_2}} = \frac{\sigma_1}{\sigma_2} \approx 1.25 \times 10^3$$

This means that we can compute x up to two digits of precision.

Question 2 **10 marks**

The banded, upper triangular matrix $A \in \mathbb{R}^{n\times n}$ can be written as

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

where

$$\vec{a} = (a_1, a_2, \ldots, a_{n-2}, a_{n-1}, a_n) \in \mathbb{R}^n,$$
$$\vec{b} = (b_1, b_2, \ldots, b_{n-2}, b_{n-1}) \in \mathbb{R}^{n-1},$$
$$\vec{c} = (c_1, c_2, \ldots, c_{n-2}) \in \mathbb{R}^{n-2}.$$

The entries of $\vec{a}$, $\vec{b}$, and $\vec{c}$ are all assumed to be nonzero.

(**a**) Write a pseudocode for computing the matrix-vector product of A with a vector x. That is, given vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$ that define A as in definition (1), and given a vector $x \in \mathbb{R}^n$, your algorithm should compute the matrix-vector product $\vec{y} = A\vec{x}$.
Your pseudocode should have the following form:

Input: vector $\vec{x} \in \mathbb{R}^n$ and vectors $\vec{a} \in \mathbb{R}^n$, $\vec{b} \in \mathbb{R}^{n-1}$ and $\vec{c} \in \mathbb{R}^{n-2}$.

1. $n \leftarrow \text{length}(x)$
2. $y = 0$
2. for $i = 1 : n-2$
    a. $y_i = a_i x_i + b_i x_{i+1} + c_i x_{i+2}$

    end
3. $y_{n-1} \leftarrow a_{n-1}x_{n-1} + b_{i-1}x_n$
4. $y_n \leftarrow a_n x_n$

Output: vector $\vec{y} \in \mathbb{R}^n$ such that $\vec{y} = A\vec{x}$

(**b**) Analyse the complexity of the algorithm from part (**a**). That is, determine how many flops are required to compute the product $\vec{y} = A\vec{x}$ with your algorithm. In terms of "Big-Oh" notation, what is the asymptotic behaviour of your algorithm as n increases?

$$\#\text{flops} = 1 + 3 + \sum_{i=1}^{n-2} 5 = 5n - 6$$

so this matrix-vector product is $O(n)$.

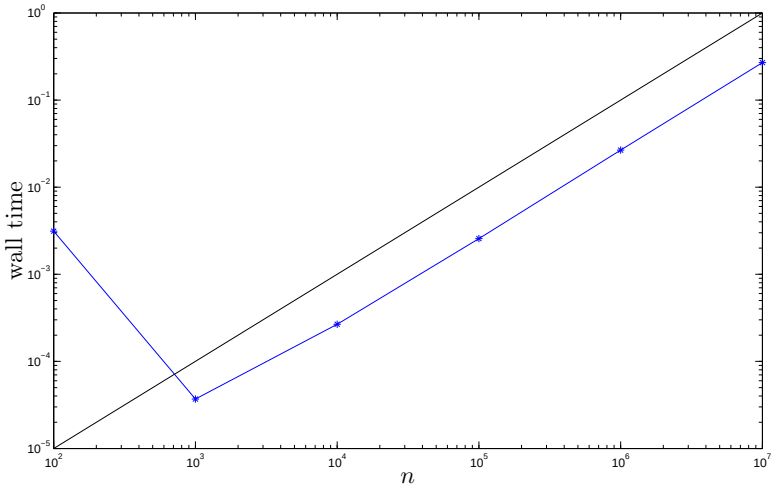
(**c**) Implement your pseudo-code in as a MATLAB function called `tridag_matvec.m`. Generate a tridiagonal test matrix and a test vector for a $n = 10^k$, $k = 2, \ldots, 7$. and check that your answer at (**c**) is correct. Produce a plot of the time taken versus n on a logarithmic scale, along with your prediction.

```
function [ y ] = tridiag( a,b,c,x )
%tridag: matrix-vector product with upper tridiagonal matrix
%   Input: diagonal a, first super diagonal b and second super diagonal c,
%   vector x
```

```
n=length(x);
y=zeros(n,1);
for i=1:n-2
    y(i,1)=a(i)*x(i)+b(i)*x(i+1)+c(i)*x(i+2);
end;
y(n-1,1)=a(n-1)*x(n-1)+b(n-1)*x(n);
y(n,1)=a(n)*x(n);

end
```

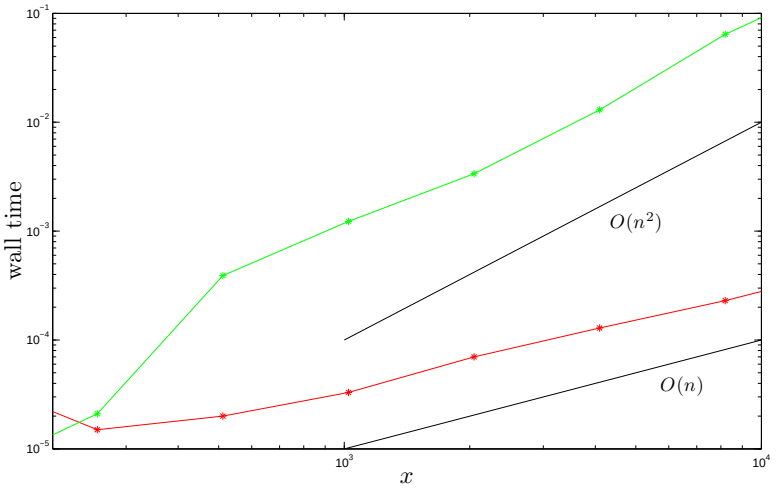
Measuring the wall time give the following picture (black is $O(n)$ scaling):



Note, that in the code we initialize y. This is not strictly necessary, but speeds up the function considerably. This is purely because of memory management. If you leave the initialization out, Matlab has to *dynamically allocate* memory for the variable y, i.e. it has to add memory space as it goes through iterations in the loop. This is slow. For that reason, it is always a good idea to initialize large variables, telling Matlab beforehand how much memory it needs to reserve.

(**d**) Repeat the test, but now using the Matlab matrix-vector product and the matrix A defined as an $n \times n$ matrix with mostly zeros. Plot the time taken in the same plot as for (**d**). Which algorithm is faster? Is the difference as great as you expected?

Of course, the matrix-vector multiplication with a full matrix is slower, namely $O(n^2)$. Moreover, for some $n > 10^4$ (depending on your computer), Matlab runs out of memory. In the figure below you see the wall time for the function we programmed in red, and for the full matrix-vector product in green.


wall time
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{3}
10^{4}
x
$O(n^2)$
$O(n)$