R. Bergmann, E. Çokaj, M. Ludvigsen

Submission Deadline:
Wednesday 7. September, 12:00 (noon)

NTNU

Exercise #1

31. August 2022

Note. The exercises can be submitted in groups of three. All solutions have to be written into *one* Jupyter notebook (use Markdown cells for the mathematical answers). This notebook has to state the exercise number and *all group members* (again, up to 3) in the very beginning. One of the group members then submits this Jupyter notebook on Blackboard.

Problem 1.

a) Prove that, for any nonsingular matrix $A \in \mathbb{R}^{n\times n}$,

$$\kappa_2(A) = \left(\frac{\lambda_{\max}}{\lambda_{\min}}\right)^{\frac{1}{2}},$$

where $\lambda_{\min}$ is the smallest and $\lambda_{\max}$ is the largest eigenvalue of the matrix $A^\top A$.

b) Show that the condition number $\kappa_2(Q)$ of an orthogonal matrix Q is equal to 1.

c) Conversely, if $\kappa_2(A) = 1$ for the matrix A, show that all the eigenvalues of $A^\top A$ are equal; deduce that A is a scalar multiple of an orthogonal matrix.

Problem 2.

We want to compute the LU factorisation with pivoting of a matrix A

$$PA = LU$$

where P is a permutation matrix, L is a lower-triangular matrix with unit diagonal, and U is an upper-triangular matrix. We represent the matrices in question as follows: The permutation matrix P is $n \times n$, but is represented as a vector P such that row number k in P is the canonical unit vector e_{P_k}. Let us illustrate this by an example

$$\mathrm{P} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} \quad \Rightarrow \quad P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

We stipulate that a Python function takes a two-dimensional numpy-array A as input, and returns an *over-written* A which contains L and U in the following sense upon return:

$$\begin{aligned} \mathrm{A}[\mathrm{P}[i], j] &= L_{ij} \quad \text{for } i < j \\ \mathrm{A}[\mathrm{P}[i], j] &= U_{ij} \quad \text{for } i \geq j \end{aligned}$$

That L has 1 on the diagonal is always the case, so the diagonal of L needs not be stored. The remaining elements of L and U are zero and need not be stored either. The algorithm can be formulated as follows (compare to text book):

- Input: A of size $n \times n$.
- Initialisation:
  - Let $P_i = i, \; i = 0, \ldots, n-1$ be a vector (array) with n components.
- for k **in** range (n-1):
  a) Find index P_ℓ such that $|\mathrm{A}_{P_\ell,k}| = \max_{k \leq i \leq n-1} |\mathrm{A}_{P_i,k}|$, i.e. scan column k from the diagonal and down for the largest element in absolute value.
  b) Swap P_k by P_ℓ.
  c) Find multipliers $A_{P_i,k} \leftarrow \frac{A_{P_i,k}}{A_{P_k,k}}, \; i = k+1, \ldots, n-1$.
  d) Perform elimination, i.e. $A_{P_i,j} \leftarrow A_{P_i,j} - A_{P_i,k} \cdot A_{P_k,j}, \; i, j = k+1, \ldots, n-1$

- Output: A,P.

There are – of course – implementations of these, often highly optimised for special cases (e.g. when A is sparse) but here we first want to learn how to code it ourselves. Let's also use this to our advantage.

a) Write a function for LU-factorisation with row-wise pivoting as indicated above. A template could be

```
def mylu (A):
```

and it should return the pivot vector (permutation vector) P, and over-written version of A. You can also choose to copy A into some other matrix LU from the beginning using e.g.

```
LU = A.copy().
```

b) Use the function `scipy.linalg.lu` to test your implementation from the first part. Write a test function `def mylutest(A)` that compares the result of your implementation to the one from `SciPy`. Call this function for example with

$$A = \begin{pmatrix} 2 & 5 & 8 & 7 \\ 5 & 2 & 2 & 8 \\ 7 & 5 & 6 & 6 \\ 5 & 4 & 4 & 8 \end{pmatrix}$$

(or `np.array ([[2, 5, 8, 7], [5, 2, 2, 8], [7, 5, 6, 6], [5, 4, 4, 8]])` to easier copy it).

c) Test your function with a matrix `A` that does not meet our assumption of having full rank, for example by repeating a column. What happens?

Problem 3.

a) Implement a

```
def forward_substitution(A,b):
```

function that takes a lower triangular matrix `A` and some vector `b` to solve $A\mathbf{x} = \mathbf{b}$.

b) Implement a

```python
def backward_substitution(A,b):
```

function that takes an upper triangular matrix `A` and some vector `b` to solve $A\mathbf{x} = \mathbf{b}$ for this case.

c) Combine the last two parts with Problem 2 and implement a function

```python
def my_solve(A,b):
```

for a square matrix `A` and a right hand side `b` that computes the LU decomposition of `A` and then uses the first two parts of this problem to compute the solution to $A\mathbf{x} = \mathbf{b}$.

d) Let's compare our implementation for two cases as well as to the original. To be precise:

- fix some n, say `n=100`,
- generate a reguar square matrix `A` (Hint: maybe create `L` and `U` here to be sure `A` is regular),
- generate `m`, say `m=200` right hand sides `b_k`,
- run an experiment where you time:
    1) calling `m` times `my_solve(A,b)` once for each `b\_k`,
    2) only computing the LU decomposition once and use backward and forward substitutions `m` times,
    3) using `np.linalg.solve`.

Where do the time differences from (1) to (2) come from and where the ones from (3) to (2)?