EXERCISE SET 7 – MATH-250 Advanced Numerical Analysis I

There is quiz on this exercise sheet. The exercises marked with ($\star$) are graded homework. The exercises marked with **(Python)** are implementation based and can be solved in the Jupyter notebooks which are available on Moodle/Noto. **The deadline for submitting your solutions to the homework is Friday, April 11 at 10h15.**

Exercises

Problem 1. (Python)

The degree n best approximation in the L^2 norm is the polynomial $p_n^* \in \mathbb{P}_n$ that minimizes the L^2 approximation error

$$\|p_n - f\|_2 = \sqrt{\int_{-1}^{1} (p_n(t) - f(t))^2 \, \mathrm{d}t} \tag{1}$$

among all degree n polynomials $p_n \in \mathbb{P}_n$. It can explictly be expressed as

$$p_n^* = \sum_{k=0}^{n} (\tilde{q}_k, f)_2 \tilde{q}_k, \tag{2}$$

where $\tilde{q}_k$ are the rescaled Legendre polynomials $\tilde{q}_k = q_k\sqrt{(2k+1)/2}, k = 0, 1, \ldots, n$, and $(u, v)_2 = \int_{-1}^{1} u(t)v(t) \, \mathrm{d}t$ denotes the L^2 inner product.

(a) Define a function `rescaled_legendre_polynomial(k)` which returns the k-th rescaled Legendre polynomial. You can use `np.polynomial.legendre.Legendre.basis` which takes as input a natural number k and returns the k-th Legendre polynomial q_k.

(b) Define a function `l_2_inner_product` which takes as input two functions u and v and approximates their L^2 inner product $(u, v)_2$ with a sufficiently accurate quadrature rule of your choice.

(c) Using the two previously defined functions and the expression (2), write a function `l_2_optimal_approximation` which takes as input a function f and a natural number n and returns the L^2 best approximation p_n^*.

(d) For the function $f(x) = 1/(1 + 25x^2)$ and the degrees $n = 1, 2, \ldots, 20$, compare the L^2-error (1) of the best approximation in the L^2 norm with the one for the Chebyshev interpolant of the same degree. Use an appropriate quadrature rule of your choice to approximate the integral.

Hint: Use `np.polynomial.chebyshev.Chebyshev.interpolate` *to compute the Chebyshev interpolant.*

Problem 2.

(a) For each of the following matrices, determine if an LU factorisation exists. If an LU factorisation exists, compute it.

(i) $\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$ (ii) $\begin{pmatrix} 4 & 2 \\ 3 & 4 \end{pmatrix}$ (iii) $\begin{pmatrix} 3 & 3 \\ 1 & 1.5 \end{pmatrix}$ (iv) $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

(b) Compute the LU factorisation with pivoting for the matrix

$$A = \begin{pmatrix} 5 & 1 & 2 \\ 2 & \frac{2}{5} & 1 \\ -4 & 0 & 6 \end{pmatrix}$$

Compare it with the Python function `sp.linalg.lu`. (If you are not working in the notebooks we provided you will need to use `scipy.linalg.lu`.)

(c) Suppose that $PA = LU$ is the LU factorisation with pivoting of A. Find a simple formula for $\det(A)$ and $|\det(A)|$ in terms of L, U, and the permutation determining P.

(d) In your linear algebra course you have seen the following definition of the determinant of a matrix $A \in \mathbb{R}^{n\times n}$:

$$\det(A) = \sum_{\sigma\in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{3}$$

where S_n is the set of all permutations of length n.

Compare the computational complexity of computing the determinant with Equation (3) and computing the determinant via the LU factorisation.

Problem 3.

Let the linear system $Ax = b$ be given by defining

$$A = \begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

Solve the system of linear equations $Ax = b$ via the LU factorisation

(a) in exact arithmetic,

(b) without pivoting in floating point arithmetic $\mathbb{F}(10, 3, -10, 10)$ (i.e. with 3 significant digits), and

(c) with partial pivoting in floating point arithmetic $\mathbb{F}(10, 3, -10, 10)$.

and compare the results.

($\star$) Problem 4.

Let $A \in \mathbb{R}^{n\times n}$ be an invertible matrix, and $u, v \in \mathbb{R}^n$ be two vectors.

(a) Show that the *Sherman-Morrison formula*

$$\left(A + uv^\top\right)^{-1} = A^{-1} - \frac{1}{1 + v^\top A^{-1}u} A^{-1}uv^\top A^{-1}$$

holds true.

(b) Given that $v^\top A^{-1}u = -1$ holds, argue whether or not $M = A + uv^\top$ can be invertible. If you find that M can be invertible, give an example for A, u, and v. Otherwise, prove that M cannot be invertible.

(c) Let A be the tridiagonal matrix with diagonals $(a_1, a_2, \dots, a_n)$ and $(b_1, b_2, \dots, b_{n-1})$

$$A = \begin{pmatrix} a_1 & b_1 & & & & \\ b_1 & a_2 & b_2 & & & \\ & b_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-2} & a_{n-1} & b_{n-1} \\ & & & & b_{n-1} & a_n \end{pmatrix},$$

and suppose that A possesses a unique LU factorisation $A = LU$. From Algorithm 4.9 one can verify that L and U are given by

$$L = \begin{pmatrix} 1 & & & & & \\ \ell_1 & 1 & & & & \\ & \ell_2 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & \ell_{n-2} & 1 & \\ & & & & \ell_{n-1} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_1 & b_1 & & & & \\ & u_2 & b_2 & & & \\ & & u_3 & b_3 & & \\ & & & \ddots & \ddots & \\ & & & & u_{n-1} & b_{n-1} \\ & & & & & u_n \end{pmatrix}$$

with the individual entries $u_1 = a_1,\ \ell_k = \frac{b_k}{u_k},\ u_k = a_k - \ell_{k-1}b_{k-1},\ k > 1$. Hence, computing the LU factorisation of A only requires $\mathcal{O}(n)$ arithmetic operations. Furthermore, forwards and backwards substitution in Algorithms 4.4 and 4.3, respectively, also require $\mathcal{O}(n)$ operations each. In total, we can therefore solve the system $Ax = b$ in $\mathcal{O}(n)$ arithmetic operations.

Using the facts above, give an algorithm which solves the system $Cx = b$ in $\mathcal{O}(n)$ operations for a vector $b \in \mathbb{R}^n$ and the matrix $C \in \mathbb{R}^{n \times n}$ given by

$$C = \begin{pmatrix} \alpha_1 & \beta_1 & & & & \beta_n \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \beta_2 & \alpha_3 & \beta_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ \beta_n & & & & \beta_{n-1} & \alpha_n \end{pmatrix}.$$

(d) Let's assume that $b = [1, 1, \dots, 1]^\top$ is given and that the values of C satisfy

$$\alpha_1 = \alpha_n = -2, \quad \alpha_2 = \alpha_3 = \dots = \alpha_{n-1} = -4, \quad \beta_1 = \beta_2 = \dots = \beta_n = 2.$$

Implement a Python function `solve(n)` for the algorithm you developped in task (c) that returns the solution x of the linear system and the relative error $\frac{\|Cx-b\|_2}{\|b\|_2}$. For $n = 10$, print the output x of your algorithm and the relative error $\frac{\|Cx-b\|_2}{\|b\|_2}$. For $n \in$ `np.logspace(2, 7, num=6, dtype=int)`, plot the elapsed computational times of your algorithm and the relative errors depending on n in a doubly logarithmic plot with `plt.loglog` (or `matplotlib.pyplot.loglog` if you are not using the Jupyter notebook provided on Moodle).

Hints:

(i) This code will be excessively slow if you use NumPy to construct the matrix. Instead, use SciPy's `sparse` submodule. We recommend using `sps.diags_array` and `sps.coo_array` (or instead use, respectively, `scipy.sparse.diags_array` and `scipy.sparse.coo_array` if you are not using the Jupyter notebook we provided on Moodle).

(ii) SciPy's `sparse` submodule has a few ways of solving sparse linear systems. Use SciPy `sparse`'s `linalg.splu` to compute the LU decomposition and solve the linear system, or `scipy.sparse.linalg.splu` if you are not using the Jupyter notebook provided on Moodle. This function returns an object `lu = sps.linalg.splu(A)` with a `solve` function such that `x = lu.solve(b)` returns the solution of the linear system.

(iii) When assembling the $uv^\top$ matrix, make sure you use a sparse matrix and not a dense NumPy array.

Remember to upload a scan `homework07.pdf` of your solutions and the completed Jupyter notebook `homework07.ipynb` corresponding to the homework to the submission panel on Moodle until Friday, April 11 at 10h15. To download your notebook from Noto, use `File > Download`. Only your submissions to Moodle will be considered for grading.