Computational Linear Algebra (MATH70024) - Coursework 2

Amaury Francou — CID : 01258326
MSc Applied Mathematics — Imperial College London
amaury.francou16@imperial.ac.uk

December 7, 2021

Abstract

This report has been written as part of the coursework 2 assessment, required for the completion of the "Computational Linear Algebra" (MATH70024) module, taught in the Department of Mathematics at Imperial College London. It comes along with several python codes (`.py` files), findable in my personal Github Classroom repository (https://github.com/Imperial-MATH96023/clacourse-2021-amauryfra/), in the `cw2` directory. The relevant commit's hash is `324c4266ff62346c3a521c84764f51a6668b1575`. The report goes through 4 exercises. Each section corresponds to one exercise. The relevant python files will be recalled at the beginning of each section.

1 Comparing matrix multiplication algorithms operations counts — Exercise 2

We devise, study and compare several custom matrix multiplication algorithms in terms of operations counts.

1.1 Custom $\mathcal{O}(n^2)$ algorithm for computing $C = (xy^T)^k$ — (a)

We consider computing the matrix $C = (xy^T)^k$, where $x, y \in \mathbb{R}^n$ and $k \in \mathbb{N}$. By expanding C as $C = \underbrace{xy^Txy^T \dots xy^T}_{k \text{ times}}$, we further identify $k-1$ dot products y^Tx. By noting $\lambda = y^Tx \in \mathbb{R}$, we have $C = \lambda^{k-1}xy^T$. Thus, by computing λ^{k-1} only once, the operations count is here $\mathcal{O}(n^2)$ as forming xy^T becomes the central costly computation. We compute xy^T by performing $n \times n$ operations, as the outer product of vectors is obtained by multiplying each element of x by each element of y.

1.2 Analysis of $D = A^TBA$ computation — (b)

We study the operations count of forming $D = A^TBA$, where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{m\times m}$.

First consider the computation of the matrix product $C^{(1)} = A^TB$. We compute one coefficient of $C^{(1)}$ using the formula $c^{(1)}_{i,j} = \sum_{k=1}^{m} a_{k,i}b_{k,j}$. Thus, for one coefficient we perform m multiplications and $m-1$ additions, which gives $2m-1$ operations. As we have $n \times m$ coefficients to compute, we have a number of operations that is $nm(2m-1)$. Using same arguments, computing $C^{(1)}A$ is made under $n^2(2m-1)$ operations ($C^{(1)} \in \mathbb{R}^{n\times m}$ and $A \in \mathbb{R}^{m\times n}$), which gives that the computing of D is made in $nm(2m-1) + n^2(2m-1)$ operations, which is here $\mathcal{O}(n^2m + nm^2)$.

Now consider $C^{(2)} = BA$, as $B \in \mathbb{R}^{m\times m}$ and $A \in \mathbb{R}^{m\times n}$, using the same analysis we have that $C^{(2)}$ is computed through $nm(2m-1)$ operations. Computing $A^TC^{(2)}$ is made through

$n^2(2m-1)$ operations as $A^T \in \mathbb{R}^{n\times m}$ and $C^{(2)} \in \mathbb{R}^{m\times n}$. Hence, the computing of D is also made in $nm(2m-1)+n^2(2m-1)$ operations, which is $\mathcal{O}(n^2m+nm^2)$. The both methods require the same number of operations and no specific values for m and n would give better results for one method.

1.3 Computation of the real and imaginary parts of $A = (P+iQ)(R+iS)$ — (c)

We consider four real matrix $P, Q, R, S \in \mathbb{R}^{m\times m}$. We consider the matrix $A = (P+iQ)(R+iS)$, which is $A = PR + iPS + iQR - QS$.

We also consider the matrix $T = (P+Q)(R-S) = PR - PS + QR - QS$. We notice that $A = (T + PS - QR) + i(QR + PS)$. Thus, computing A and its real and imaginary parts is computing three $m \times m$ matrix multiplications : $T = (P+Q)(R-S)$, PS and QR.

To compute $T = (P+Q)(R-S)$, two times m^2 additions to form $(P+Q)$ and $(R-S)$ are needed, and one additional matrix multiplication to finish the computation is needed, which - given the arguments of question 1.2 - requires $m^2(2m-1)$ operations. Thus, forming T requires a total of $m^2(2m+1)$ operations.

To compute $A = (T + PS - QR) + i(QR + PS)$, two $m \times m$ matrix multiplications are needed, which each require $m^2(2m-1)$ operations. In addition, the $m^2(2m+1)$ cost of computing T is also needed. Furthermore, four matrix additions are required, which represents a total of $4m^2$ floating point operations. By this method, computing A requires a total of $4m^2+2m^2(2m-1)+m^2(2m+1) = 6m^3 + 3m^2$ operations, which is $\mathcal{O}(m^3)$.

2 Stability of eigenvalue computation algorithms — Exercise 3

The relevant python file is `exercise3.py` located in the `cw2` directory.

We consider a matrix $A \in \mathbb{C}^{2\times 2}$. We compute the eigenvalues of A by using the roots of its characteristic polynomial. Let $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$. Its characteristic polynomial is defined as $\chi_A = \det(XI_2 - A) = (X-a)(X-d) - bc = X^2 - (a+d)X + ad - bc$. This polynomial has two roots in $\mathbb{C}$ which are the eigenvalues of A :

$$\lambda_1 = \frac{a+d+\sqrt{(a+d)^2-4(ad-bc)}}{2} \text{ and } \lambda_2 = \frac{a+d-\sqrt{(a+d)^2-4(ad-bc)}}{2} \tag{1}$$

2.1 Roots computing algorithm stability — (a)

Consider the problem f of finding the eigenvalues of matrix $A \in \mathbb{C}^{2\times 2}$. We have $f : \mathbb{C}^{2\times 2} \to \mathbb{C}^2$, that maps $A \mapsto (\lambda_1, \lambda_2)$ using equation 1. Further consider $\hat{f}$, the floating point implemented algorithm of f.

The algorithm $\hat{f}$ is said to be stable if for a "small" perturbation δA of matrix A, we have $\hat{f}(A) - f(A+\delta A)$ also "small".

Namely, for some perturbation δA such that $\frac{||\delta A||}{||A||} = \mathcal{O}(\epsilon)$, we have $\frac{||\hat{f}(A) - f(A+\delta A)||}{||f(A+\delta A)||} = \mathcal{O}(\epsilon)$, where ϵ is the unit roundoff of the machine used.

2.2 Roots computing algorithm backward stability — (b)

Consider the same problem function f and its corresponding implementation $\hat{f}$.

The algorithm $\hat{f}$ is said to be backward stable if for all computations $\hat{f}(A)$ we can find a "small" perturbation δA, such that we have $f(A+\delta A) = \hat{f}(A)$, which represents the problem the algorithm actually solved.

Namely, for all $A \in \mathbb{C}^{2\times 2}$, there exists a perturbation δA verifying $\frac{||\delta A||}{||A||} = \mathcal{O}(\epsilon)$ such that $\hat{f}(A) = f(A + \delta A)$.

2.3 Python implementation of eigenvalue computation algorithm — (c)

We consider the python implementation of $\hat{f}$ and apply it to the following matrix : $A_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $A_2 = \begin{pmatrix} 1+10^{-14} & 0 \\ 0 & 1 \end{pmatrix}$. As diagonal matrix, we directly obtain the corresponding eigenvalues : $\lambda_1^{(1)} = \lambda_2^{(1)} = 1$ for A_1, and $\lambda_1^{(2)} = 1 + 10^{-14}$, $\lambda_2^{(2)} = 1$ for A_2.

We denote $\check{\lambda}_i^{(k)}$ the different python computed eigenvalues using $\hat{f}$. Our computation gives : $\check{\lambda}_1^{(1)} = \check{\lambda}_2^{(1)} = 1$ and $\check{\lambda}_1^{(2)} \approx 1 + (1.49 \cdot 10^{-8})i$, $\check{\lambda}_2^{(2)} \approx 1 + (1.49 \cdot 10^{-8})i$.

We notice that the python algorithm outputs an exact solution for the computation of A_1 eigenvalues as $||\lambda^{(1)} - \check{\lambda}^{(1)}|| = 0$, where $\lambda^{(1)} = (\lambda_1^{(1)}, \lambda_2^{(1)})$ and $\check{\lambda}^{(1)} = (\check{\lambda}_1^{(1)}, \check{\lambda}_2^{(1)})$.

However an error appears in the computation of A_2 eigenvalues. We have $||\lambda^{(2)} - \check{\lambda}^{(2)}|| \approx 2.10 \cdot 10^{-8}$, where $\lambda^{(2)} = (\lambda_1^{(2)}, \lambda_2^{(2)})$ and $\check{\lambda}^{(2)} = (\check{\lambda}_1^{(2)}, \check{\lambda}_2^{(2)})$. Thus, our implementation induces an error of order 10^{-8}.

You may find the python implementation of said algorithm, as well as the corresponding computations, in the `exercise3.py` file located in the `cw2` directory.

2.4 Scaling of computation errors with machine epsilon — (d)

As we compute the roots of the characteristic polynomial, it is to expect to have errors of order $\epsilon_{\text{machine}}$ appearing in the coefficients computations. As we are computing an order 2 polynomial, the root formula used involves a square root which will ultimately propagate a dominant error of order $\sqrt{\epsilon_{\text{machine}}}$. Indeed, as the computation has been made in a 64-bits (double precision) operating system, we have $\epsilon_{\text{machine}} \approx 1.11 \cdot 10^{-16}$ and further $\sqrt{\epsilon_{\text{machine}}} \approx 10^{-8}$, which is the order of the error we previously computed.

You may find the corresponding $\epsilon_{\text{machine}}$ obtaining in the `exercise3.py` file located in the `cw2` directory.

3 Analysis of an overlapping blocks matrix — Exercise 4

We study an $(4n+1) \times (4n+1)$ overlapping blocks matrix of the form $A = I + \epsilon \sum_{k=1}^{n} B^k$, where $B_{i,j}^k = \begin{cases} \nu_{ijk} & \text{if } 4(k-1) < i, j \leq 4k+1 \\ 0 & \text{otherwise} \end{cases}$

The relevant python files are `exercise4.py` and `test/test_exercise4.py`, located in the `cw2` directory.

3.1 LU factorization of the overlapping blocks matrix — (a)

We compute a prototype of an overlapping blocks matrix A. We first notice that A is made of 5×5 block matrix diagonally aligned, with one value of two consecutive blocks overlapping on the diagonal, and zeros everywhere else. Indeed, we have $4k + 1 - 4k + 4 = 5$, the k-th and $(k + 1)$-th blocks overlapping at coordinate $(4k + 1, 4k + 1)$.

We further compute the LU factorization of the studied matrix A. We notice, for instance, that the corresponding L is a lower triangular matrix made of diagonally aligned lower triangular sub-matrix blocks, those triangular blocks also overlapping on the diagonal. In the same way, the upper triangular matrix U is made of diagonally aligned upper triangular sub-matrix overlapping only on one diagonal coefficient. The L and U matrix acquire the same structure as A through our LU decomposition algorithm.

You may find the corresponding overlapping blocks matrix prototype generating function, as well as the relevant computations of $A = LU$ in the `exercise4.py` file located in the `cw2` directory.

3.2 Leveraging overlapping blocks matrix structure for LU decomposition — (b)

As A is made of 5×5 diagonally aligned block matrix, it is banded. However, it contains more zeros than the general form of banded matrix as the diagonally aligned square blocks leave two "triangular set" of zeros in between their lower and upper bandwidth lines and their overlapping coefficient. Thus, we have that the corresponding coefficients in the L_k matrix involved in the LU decomposition will be set to zero, which further induces fruitless computations by null values.

Hence, we may devise a modified version of the Gaussian elimination algorithm for banded matrix, that is specifically tailored for our custom overlapping blocks matrix. We focus on said square blocks, applying the elimination process uniquely to them :

$U \leftarrow A$
$L \leftarrow I$
for $k = 1$ to $m - 1$ **do**
    $b \leftarrow \lfloor \frac{k}{4} \rfloor + 1$ ▷ the sub-matrix block currently processed
    $n \leftarrow 4b + 1$ ▷ the maximum index of the current sub-matrix block's elements
    for j = k + 1 to n **do**
        $l_{jk} \leftarrow u_{jk}/u_{kk}$
        $u_{j,k:n} \leftarrow u_{j,k:n} - l_{jk}u_{k,k:n}$

For an $m \times m$ matrix, we know that the LU decomposition has an operation count of $\frac{2m^3}{3}$. Moreover, we have that our $(4n+1) \times (4n+1)$ matrix is made of n 5×5 square sub-matrix blocks, on each of which we perform an LU decomposition. We obtain that our modified algorithm has an operation count that is approximately $\frac{2 \cdot 5^3}{3} n \approx 83n$, which is $\mathcal{O}(n)$.

3.3 Python implementation of modified banded matrix Gaussian elimination algorithm — (c)

We implement the devised algorithm in question 3.2. We use the Numpy slice operations and compute the LU decomposition of matrix A "in-place".

You may find the python implementation of said algorithm in the `exercise4.py` file and you may also run the automatic testing of the function by using `pytest test/test_exercise4.py` while in `cw2` directory.

4 Finite difference method for solving the stationary advection-reaction-diffusion equation — Exercise 5

We study the equation $\mathbf{b} \cdot \nabla u - \mu \nabla^2 u + cu = S(x, y)$ (3) on $[0, 1]^2$, with the following bounding conditions : $u = 0$ when $x = 0$, $x = 1$, $y = 0$ and $y = 1$. We discretize this square into a grid made of the $(i\Delta x, j\Delta x)$ points, where $\Delta x = \frac{1}{n}$ for a positive integer n and $(i, j) \in [\![0, n]\!]^2$. We consider the discrete translation of the studied equation, using the finite difference method, given by (4) (see subject).

The relevant python file is `exercise5.py`, located in the `cw2` directory.

4.1 Formulating (4) as a matrix-vector equation — (a)

We consider the vector $\mathbf{v} \in \mathbb{R}^{(n-1)^2}$ such that $v_{(n-1)(i-1)+j} = u_{i,j}$ for $(i, j) \in [\![1, n-1]\!]^2$.

We notice that $\mathbf{v}$ is the "flatten" adaptation of the matrix storing the values of $u_{i,j}$, with related bounding values coefficients left out. Namely we have $\mathbf{v} = (u_{1,1}, u_{1,2}, \dots, u_{1,n-1}, u_{2,1}, u_{2,2}, \dots, u_{n-1,n-1})$. We may rewrite equation (4) as $A\mathbf{v} = \hat{\mathbf{S}}$, where $\hat{\mathbf{S}}$ is the vector representing the flatten version of the matrix storing the values of $S_{i,j}$, with the bounding related coefficients left out.

The matrix A is the sum of three matrix representing the three terms in the equation (4). We have $A = \frac{1}{2\Delta x} A_1 - \frac{\mu}{(\Delta x)^2} A_2 + cI_{(n-1)^2}$.

A_1 is such that :

$$A_1 = \begin{pmatrix} 0 & b^2_{1,1} & 0 & \cdots & a^{(1)}_{1,n} = b^1_{1,1} & & \cdots & & 0 \\ -b^2_{1,2} & 0 & b^2_{1,2} & 0 & \cdots & b^1_{1,2} & \cdots & & 0 \\ 0 & -b^2_{1,3} & 0 & b^2_{1,3} & & & \ddots & \cdots & 0 \\ & & \ddots & \ddots & \ddots & & & b^1_{2,1} & 0 \\ a^{(2)}_{n,1} = -b^1_{2,1} & & & & & & & & \ddots \\ & -b^1_{2,2} & & & & & & & \\ & & \ddots & & & & & & \\ & & & & & & & & \end{pmatrix}$$

The $b^{(k)}_{i,j}$ are arranged in the same way as vector $\mathbf{v}$: $(b^{(k)}_{1,1}, b^{(k)}_{1,2}, \dots, b^{(k)}_{1,n-1}, b^{(k)}_{2,1} \dots)$.

A_2 is such that : $A_2 =$

$$\begin{pmatrix} -4 & 1 & 0 & \cdots & a^{(2)}_{1,n} = 1 & & \dots & & 0 \\ 1 & -4 & 1 & 0 & \cdots & 1 & \cdots & & 0 \\ 0 & 1 & -4 & 1 & & & 1 & \cdots & 0 \\ & & \ddots & \ddots & \ddots & & & \ddots & 0 \\ a^{(2)}_{n,1} = 1 & & & & & & & & 1 \\ & 1 & & & & & & & \\ & & \ddots & & & & & & \\ & & & & & & & & \end{pmatrix}$$

As A is made of three banded matrix, it is banded. We can notice that the upper and lower bandwidths are both $n-1$ here. The operation count for an $m \times m$ banded matrix LU decomposition algorithm - with bandwidths of size p and q - is $\mathscr{O}(mpq)$. In this specific case, the operation count for the LU decomposition of A is $\mathscr{O}(n^4)$, as $m = (n-1)^2$ and $p = q = (n-1)$.

4.2 Banded matrix LU decomposition algorithm — (b)

We consider applying the banded matrix specific LU factorisation algorithm to matrix with upper and lower bandwidths of length $n-1$. We start by generating random banded matrix with said bandwidths, n being a given input. We than implement the banded LU decomposition algorithm as stated in Section 4.4 of lecture notes.

We further monitor the execution time of the banded matrix specific factorisation, which is approximately an "image" of the operations count. We investigate the computation time divided by n^4, for increasing values of n. We obtain :

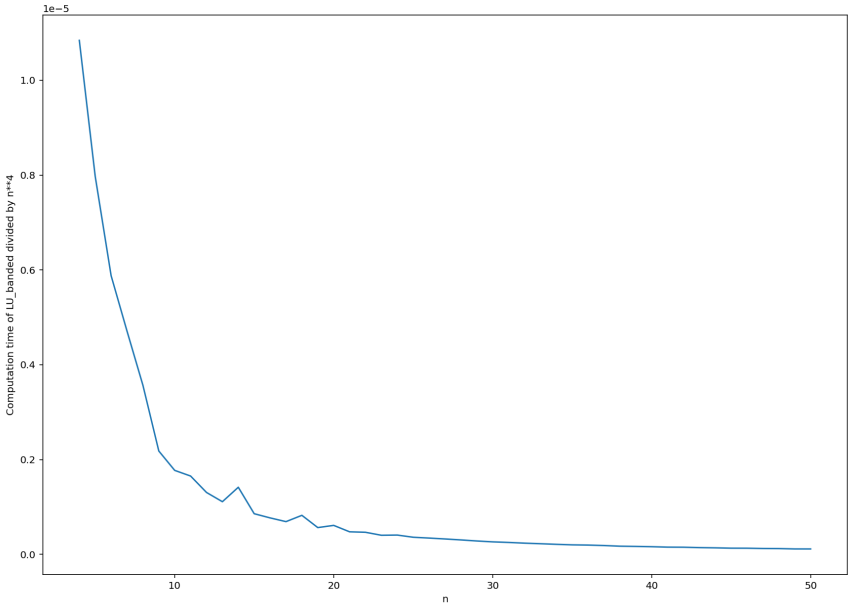

Figure 1: Computation time of the banded matrix specific LU decomposition algorithm divided by n^4, plotted for increasing values of $n \in [\![4, 50]\!]$.

We notice that, as n increases, the execution time divided by n^4 decreases towards 0. We thus empirically have that the computation time does not grow faster than n^4.

You may find the python implementation of said time monitoring in the `exercise5.py` file, in `cw2` directory.

4.3 Iterative methods analysis — (c)

We now consider iterative methods to compute the $(u_{i,j})$ given in the discrete translation of the studied equation. The iterative solution is computed using an initial guess $u^0_{i,j}$, that is further updated by equations (6) and (7) (see subject).

We use the same vector $\mathbf{v}$ - that is the "flatten" adaptation of the matrix storing the values of $u_{i,j}$ - as seen previously (see question 4.1). We also introduce the vector $\check{\mathbf{v}} \in \mathbb{R}^{(n-1)^2}$ such that $v_{(n-1)(j-1)+i} = u_{i,j}$ for $(i,j) \in [\![1, n-1]\!]^2$. We have $\check{\mathbf{v}} = (u_{1,1}, u_{2,1}, \dots, u_{n-1,1}, u_{1,2}, u_{2,2}, \dots, u_{n-1,n-1})$. The vector $\check{\mathbf{v}}$ is also a "flatten" adaptation of the matrix storing the values $u_{i,j}$, but with a different ordering.

The equation (6) turns into the following matrix based translation : $C_1\check{\mathbf{v}} = \check{\mathbf{S}} - C_2\mathbf{v}$, where $\check{\mathbf{S}}$ is the vector representing the flatten version of the matrix storing the values of $S_{i,j}$, with the same ordering as in $\check{\mathbf{v}}$. By this relevant ordering of $\mathbf{v}$ and $\check{\mathbf{v}}$, C_1 and C_2 are tridiagonal matrix here.

In the same way, the equation (7) turns into the following matrix based translation : $D_1\mathbf{v} = \hat{\mathbf{S}} - D_2\check{\mathbf{v}}$, where $\hat{\mathbf{S}}$ is as defined in question 4.1. Similarly, D_1 and D_2 are also being tridiagonal here.

Thus, the updating process becomes :

$$\begin{cases} C_1\check{\mathbf{v}}^{k+\frac{1}{2}} = \check{\mathbf{S}} - C_2\mathbf{v}^k & (6) \\ D_1\mathbf{v}^{k+1} = \hat{\mathbf{S}} - D_2\check{\mathbf{v}}^{k+\frac{1}{2}} & (7) \end{cases}$$

Using this method, we devise an algorithm which :

- First correctly flattens the matrix $U = (u_{i,j})$ and $S = (s_{i,j})$. This is mainly a memory operation.
- Then computes the right hand side of the matrix translation updating equations, by selecting the right coefficients, and computing the relevant right hand side vector. A full matrix multiplication can be avoided here, the computation can be made by going through one axis of U, which will induce $\mathcal{O}(n^2)$ operations.
- Further performs the banded LU factorization of the left hand side tridiagonal matrix in $\mathcal{O}(n^2)$, as the bandwidths are equal to 1 (see Section 4.4 of lecture notes).
- Finally executes banded specific backward and forward substitutions, which are both $\mathcal{O}(n^2)$, also because the bandwidths are equal to 1 (see Section 4.4 of lecture notes).

Overall, for one iteration from k to $k+1$, the operation count will not grow faster than n^2 ($\mathcal{O}(n^2)$).

4.4 Python implementation of the iterative method algorithm — (d)

We implement the previously discussed process using several helper functions. The main function `exercise5_main` takes the grid parameter n, as well as a tolerance parameter, as inputs, and returns

a matrix containing the $(u_{i,j})$ solution coefficients. The user must also provide the parameters s_0, r_0, c, α and μ to the function.

For the tested values, the iterative solution does not seem to converge, disclosing a probable logic error.

You may find the relevant python implementations in the `exercise5.py` file, in `cw2` directory.