Computational Linear Algebra (MATH70024) - Coursework 1

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

November 5, 2021

Abstract

This report has been written as part of the coursework 1 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 `cw1` directory. The relevant commit's hash is `50de4ff497a11eb48a26af17f4bc2897ed0442ff`. 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 Electrical signal measured on a sample of plant tissue when exposed to light — Exercise 1

We study a time series of an electrical signal measured on a "particular type of plant tissue when suddenly exposed to light". 1000 samples were taken, each of them is composed of 100 values of the electrical signal. The values have been taken at equispaced times.

The relevant python files are `exercise1.py` and `test/test_exercise1.py`, located in the `cw1` directory.

1.1 QR factorisation — (a)

We compute the QR factorisation of the 1000×100 matrix containing the 100 values of each 1000 samples. By analyzing R, we notice that for all lines under line 3, the values $r_{i,j}$ are smaller than 10^{-9}. Formally $\forall i < 3, \forall j \; |r_{i,j}| < 10^{-9}$. This means that the column vectors of C are - approximatively - "only" a linear combination of the first three orthonormal basis vectors of Q (the column vectors of Q are spanning $\mathbb{R}^{1000}$). Thus, the values taken by the different samples are strongly related to each other : they are "closely" linearly dependent and only set in a 3 dimensional subspace. Multiplying the samples do not provide much more information on the physical process at stake.

You may run `exercise1.py` to get the computation of the maximum absolute value of $r_{i,j}$ under line 3.

1.2 Compression of C — (b)

We noticed that the column vectors of C approximatively lie in an 3 dimensional subspace. We may compress the information contained in C by focusing on the expression of its column vectors in a basis of this 3 dimensional space. Namely, having $C = QR$, we may drop the lines of R below 3 as they are irrelevantly small, and only keep the 3 first column vectors of Q which span said subspace. We note $\check{Q}$ (a 1000×3 matrix), $\check{R}$ (a 3×1000 matrix) those submatrix, and $\check{C}$ the compressed version of C. $\check{C}$ can be represented (and stored) as the pair $(\check{Q}, \check{R})$. While C needs 800120 bytes of memory space to be stored, $\check{C} \approx (\check{Q}, \check{R})$ needs only 240 bytes of memory space plus a reasonable computational cost to be retrieved. We have $\check{C} = \check{Q}\check{R}$. In the `test/test_exercise1.py` file, we verify that this compression method is efficient by asserting that $||C - \check{C}||_2 < 10^{-8}$.

You may run `exercise1.py` to get the corresponding memory size values. You may also test the script by using `pytest test/test_exercise1.py` while in `cw1` directory.

1.3 Transformation of C — (c)

To better show the linear dependence of the column vectors of C, we may investigate the diagonal coefficients of R, $A = QR$ being the complete factorisation. By reordering - with a permutation matrix for instance - the columns of Q and R such that the $r_{i,i}$ are as $|r_{1,1}| > |r_{2,2}| > \dots > |r_{100,100}|$, we may find the first column k for which $|r_{k,k}| \approx 0$. This gives us that the matrix C spans a subspace of dimension k, which is the maximum degree of freedom in between its column vectors. The first k column vectors of the reordered Q form an orthonormal basis of said subspace.

We may devise a compression algorithm in which we perform the QR factorisation of C, reorder the columns of Q and R such that we obtain $|r_{1,1}| > |r_{2,2}| > \dots > |r_{n,n}|$, and drop all columns for which $|r_{k,k}| < \epsilon$ for a given ϵ. This gives us the reduced pair $(\check{Q}, \check{R})$, storing C in a more memory-efficient manner.

2 Least squares polynomial curve fitting — Exercise 2

In this exercise, we construct a degree 12 polynomial, fitting - by the least squares method - the function f defined as follows :

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \text{ or } x = 0.9803\dots \\ 0 & \text{if } x \in]0,1] \backslash \{0.9803\dots\} \end{cases}$$

The relevant python file here is `exercise2.py`.

2.1 Constructing the polynomial — (a)

We first introduce the following Vandermonde matrix :

$$V = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{12} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{12} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^{12} \end{pmatrix}$$

Where $(x_1, ..., x_m)$ are the values on which the function f is computed ($f(x_i) = f_i$). Programmatically speaking those values are in the `arange(0., 1.0001, 1./51)` array. We also introduce

x, the vector containing the coefficients of the polynomial we are building, and b, the vector containing the function's values f_i for $i \leq m$:

$$x = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{12} \end{bmatrix}, \quad b = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_m \end{bmatrix}$$

In particular we have : $Vx = \begin{bmatrix} a_0 + a_1 x_1 + \cdots + a_{12} x_1^{12} \\ a_0 + a_1 x_2 + \cdots + a_{12} x_2^{12} \\ \vdots \\ a_0 + a_1 x_m + \cdots + a_{12} x_m^{12} \end{bmatrix}$.

We are here seeking to minimise the 2-norm of the "residual" $r = b - Vx$. Thus, we are looking for $\arg\min_x ||Vx - b||^2$. Using the arguments given in Section 2.7 of the lectures notes, the nearest vector to b that is in the range of V is the orthogonal projection of b on the space spanned by V. Theorem 1.28 gives us that this projection is represented by the matrix $P = \hat{Q}\hat{Q}^*$, where $\hat{Q}$ is the matrix given by the reduced QR factorisation of V. As in the relevant section of the lecture notes, we get that the minimizing x is such that $\hat{R}x = \hat{Q}^* b$. Thus, we need to perform a reduced QR factorisation, followed by triangular system solving, to get our polynomial's coefficients.

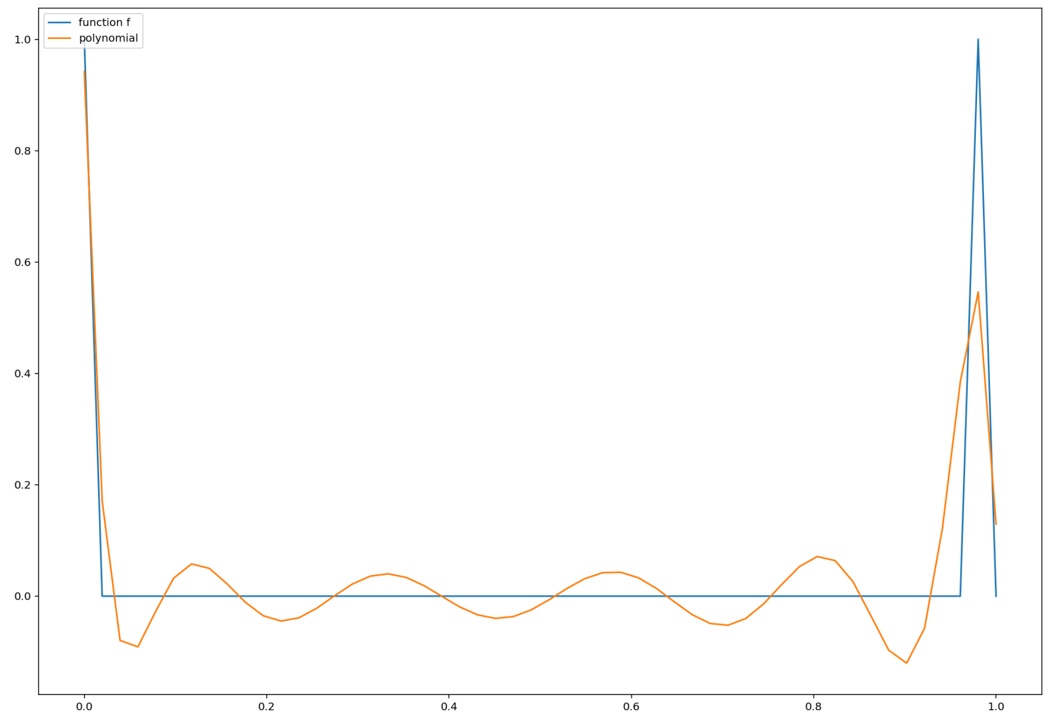

Figure 1: Polynomial of degree 12 fitting the function f, using the least squares method, performed by a Householder QR factorisation.

We construct our degree 12 polynomial by implementing a QR factorisation, using Gram-Schmidt, modified Gram-Schmidt and the Householder method. You may run `exercise2.py`, in the `cw1` directory, to get the corresponding computations and plot.

2.2 Investigating the different QR factorisation methods — (b)

Using Gram-Schmidt, modified Gram-Schmidt and the Householder method, we get different coefficients for the built polynomial. The polynomial coefficients given by the Householder method and the modified Gram-Schmidt method differ at maximum from 0.1458. Both methods give very close results, that fit the curve of f as seen in Figure 1. However, the coefficients given by the Gram-Schmidt method differ from the ones of the Householder method from a maximum of 3854464. The coefficients given by the Gram-Schmidt method are not close to those of the other methods, and they do not provide a polynomial that fits "well" the curve of f. By performing $Q_{GS}^{*}Q_{GS} - I$, we notice that the matrix lacks of orthogonality compared - for instance - to the Q matrix of the modified Gram-Schmidt method. Precisely, we have $||Q_{GS}^{*}Q_{GS} - I|| \approx 7.60$, where $||Q_{GS\text{modified}}^{*}Q_{GS\text{modified}} - I|| \approx 10^{-8}$ As recalled in Section 2.4 of the lecture notes, the computation of Gram-Schmidt method may propagate round-off errors, giving inexact imprecise results.

You may run `exercise2.py`, in the `cw1` directory, to get the corresponding computations.

3 QR factorisation — Exercise 3

We consider a new way of performing the Householder QR factorisation by storing the ongoing useful vectors v directly at the bottom of R.

The relevant python files are `exercise4.py` and `test/test_exercise4.py`, located in the `cw1` directory.

3.1 Discussion on reduced QR factorisation efficiency — (a)

If $m >> n$ and one is seeking only to compute the reduced $\hat{Q}$ of the $A = QR$ factorisation, the extended Householder method proves inefficient since we compute an $m \times m$ unreduced Q matrix, by stacking the identity matrix at the end of A and performing several implicit multiplications. Those multiplications involve an $m \times (n+m)$ matrix (the ongoing transformation of A stacked with I) and an $m \times m$ matrix. This means having a high computational cost for computing irrelevant $m - n$ column vectors.

3.2 Storing v vectors in R — (b)

In the Householder method, we compute R "in place" by changing the values of A. We use several v column vectors to compute R. At each step of the for loop (pseudo-code is described in Section 2.6 of the lecture notes), the vector v is of length $m - k$ (where k is the step, between 1 and n). As the relevant values of R - stored in the k-th column of A - are of length k, we can add the $\frac{r_{k,k}}{v_1}v$ vector - where its first value overlapping the diagonal of A has been scaled accordingly - in the remaining space of the column, that is to say in $A_{k:m,k}$.

3.3 R-v arrays algorithm — (c)

You may find the python implementation of the above described algorithm (stacking v vectors in the bottom of R) in the `exercise3.py` file. You may also run the automatic testing of the function by using `pytest test/test_exercise3.py` while in `cw1` directory. The new algorithm globally gives the same results as the one implemented during the weekly exercises.

3.4 Computing Q^*b using R-v — (d)

Using the same scheme as in the Householder algorithm, we may perform implicit multiplications on b by using the v vectors stored in the lower part of R-v. We retrieve v by taking the slice R-$v_{k:n,k}$, rescale it, and, as done in the Householder method, we modify b "in place" by subtracting $2v(v^*b_{k:n})$ to it. This process will transform b into Q^*b.

You may find the python implementation of this algorithm in the `exercise3.py` file. You may also run the automatic testing of the function by using `pytest test/test_exercise3.py` while in `cw1` directory. As it closely uses the same Householder process as in the weekly exercises, the efficiency and precision of said new method will be comparable to what we had previously.

3.5 Least squares problems using R-v — (e)

As seen in Section 2.7 of the lecture notes, solving a least squares problem is finding an x vector minimizing $||Ax - b||^2$, where A and b are given. We have seen that, by using the reduced QR factorisation of A ($A = \hat{Q}\hat{R}$), a minimizing x is such that $\hat{R}x = \hat{Q}^*b$. Thus by using the above described algorithm (3.4), we may use R-v to compute $\hat{Q}^*b$, before solving the resulting upper triangular system.

You may find the python implementation of said algorithm in the `exercise3.py` file and you may also run the automatic testing of the function by using `pytest test/test_exercise3.py` while in `cw1` directory. Instead of creating an appropriate extended matrix and performing the Householder algorithm on it, we here performed the Householder process on A directly, and held the several v vectors that we used. The v vectors were then used to compute $\hat{Q}^*b$. Overall the same implicit multiplication process is at stake, which in the end gives similar performance than what we had in the weekly exercises implementations.

4 Minimizing $||Ax - b||^2$ having $||x|| = 1$ — Exercise 4

We examine minimizing the least squares problem $||Ax - b||^2$ under the $||x|| = 1$ constraint.

The relevant python files are `exercise4.py` and `test/test_exercise4.py`, located in the `cw1` directory.

4.1 Reformulation of the problem — (a)

We reformulate the problem by introducing the function $\phi(x, \lambda) = a(x) + \lambda b(x)$ with $a(x) = ||Ax - b||^2$ and $b(x) = ||x|| - 1$. We are thus looking for a stationary point of ϕ.

4.2 Finding an equation on x — (b)

A stationary point is such that $\frac{\partial \phi}{\partial x} = 0$. We have $\frac{\partial \phi}{\partial x} = 2A^TAx - 2A^Tb + 2\lambda x$. Provided $(A^TA + \lambda I)$ is invertible, this gives us $x = (A^TA + \lambda I)^{-1}A^Tb$.

4.3 Solver algorithm — (c)

Computing the QR factorisation of A, we have $x = (R^TR + \lambda I)^{-1}R^TQ^Tb$.

The Woodbury identity gives us : $(R^TR + \lambda I)^{-1} = \frac{1}{\lambda}I - \frac{1}{\lambda^2}R^T(I + \frac{1}{\lambda}RR^T)^{-1}R$.

We then have $x = \frac{1}{\lambda}R^TQ^Tb - \frac{1}{\lambda^2}R^T(I + \frac{1}{\lambda}RR^T)^{-1}RR^TQ^Tb$.

If we suppose that $n >> m$, this new formulation of x is highly beneficial as RR^T is only an $m \times m$ matrix to invert, as we had R^TR to invert before, which was $n \times n$. To compute $(I + \frac{1}{\lambda}RR^T)^{-1}RR^TQ^Tb = (I + \frac{1}{\lambda}RR^T)^{-1}RA^Tb$, we set $Y = (I + \frac{1}{\lambda}RR^T)^{-1}RA^Tb$, which implies having $(I + \frac{1}{\lambda}RR^T)Y = RA^Tb$. We are then able to compute and invert the expression using a Householder method based solver.

Under our hypothesis, we avoid a "heavier" matrix inversion, and "only" have to compute matrix multiplications and one Householder solving to get the solution. You may find the python computation of x in the `exercise4.py` file. You may also run the automatic testing of the function by using `pytest test/test_exercise4.py` while in `cw1` directory.

4.4 λ search algorithm — (d)

We are now able to compute a minimizing vector x for any given λ. We aim to find a λ such that $||x|| = 1$. We may start with an arbitrary value of λ, compute x, compare $||x||$ to 1, and iteratively increase or decrease λ to get a norm closer to 1. Practically speaking, we may use recursive programming to perform this task. The increase or decrease of λ must be proportional to $|||x|| - 1|$. If our stopping condition (namely having $\lambda < \epsilon$ for some ϵ) is not achieved, than we perform the increase or decrease on λ depending on the sign of $||x|| - 1$, before returning the function itself, but with the new λ as input. After a certain number of iterations the function returns a relevant λ.

You may find the python implementation of this algorithm in the `exercise4.py` file. You may also run the automatic testing of the function by using `pytest test/test_exercise4.py` while in `cw1` directory.

4.5 Investigating λ search algorithm with various A — (e)

We may tryout said λ search algorithm with several random matrix A of increasing size. We monitor the execution time of the function for those increasing A. We notice that, as the size of A increases, the execution time increases exponentially, which proves that our implementation is inefficient. One may also consider a different λ search algorithm, in which we would use two starting values λ_1 and λ_2, compute a specific x using the midpoint $\lambda = \frac{\lambda_1+\lambda_2}{2}$ and iteratively "refresh" λ by setting $\lambda_1 := \lambda$ or $\lambda_2 := \lambda$, depending on the sign of $||x|| - 1$.

You may find the python implementation of this execution-time monitoring in the `exercise4.py` file.