Computational Linear Algebra (MATH70024) - Coursework 3

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

January 13, 2022

Abstract

This report has been written as part of the coursework 3 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 `cw3` directory. The relevant commit's hash is `eb366c110b40c0c2b7b2552d821aa83f1c9da8fd`. The report goes through 3 exercises. Each section corresponds to one exercise. The relevant python files will be recalled at the beginning of each section.

1 Tridiagonal matrix eigenvalue computation through QR decomposition — Exercise 2

We study a $2n \times 2n$ tridiagonal matrix A of the form $\begin{cases} A_{ij} = -1 & \text{if } j = i-1 \\ A_{ij} = 1 & \text{if } j = i+1 \\ A_{ij} = 0 & \text{otherwise} \end{cases}$

The relevant python files are `exercise2.py` and `test/test_exercise2.py`, located in the `cw3` directory.

1.1 Pure QR algorithm applied to matrix A — (a)

We compute the defined matrix A for various values of n. We further apply the pure QR algorithm to said matrix. We notice that the output matrix R of the QR factorization is not upper triangular as expected, as the first lower sub-diagonal is not filled with zeros. The matrix R has roughly the same tridiagonal structure than matrix A, opposite values are set on either side of the diagonal : $A_{k+1,k} = -A_{k,k+1}$. Thus, eigenvalues are not accessible directly through the use of pure QR factorization.

To better understand this phenomenon, we compute the eigenvalues of matrix A and we observe that if $\lambda_i \in \Lambda(A)$ is an eigenvalue of A, then $\bar{\lambda_i} \in \Lambda(A)$, i.e. the complex conjugate is also an eigenvalue of A. By *Theorem 5.23* of lecture notes, we have that the pure QR algorithm and the simultaneous iteration algorithm with I are equivalent. Yet, we know that the $\hat{Q}^{(k)}$ matrix - as noted in the lecture notes - involved in the simultaneous iteration method, converges toward the eigenvectors of A under the condition that the eigenvalues of A are *distinct in absolute value*, which is not the case here. Thus we do not obtain the relevant R.

You may find the corresponding tridiagonal matrix A generating function, as well as the relevant computations of the pure QR algorithm of said matrix in the `exercise2.py` file located in the `cw3` directory.

1.2 Custom method for eigenvalue computation of matrix A — (b)

Through the pure QR algorithm, we have that A and R are similar, thus accessing R's eigenvalues is accessing A's eigenvalues.

In our previous computations, we have seen that R is made of n diagonally aligned 2×2 block matrix. Those block matrix are of the form :

$$\begin{pmatrix} 0 & A_{k,k+1} \\ -A_{k,k+1} & 0 \end{pmatrix}, \text{ where } A_{k,k+1} \in \mathbb{C}.$$

The block determinant formula gives us that the eigenvalues of R are the ones of its n diagonally aligned sub-matrix. Furthermore, those sub-matrix have two eigenvalues each that are $\pm iA_{k,k+1}$.

From this analysis we devise a method for computing the eigenvalues of matrix A. Namely, we first perform a pure QR factorization of A, before extracting all non zero entries in the first upper-diagonal of R. The eigenvalues of A are exactly said extracted entries multiplied by $\pm i$.

1.3 Custom method for eigenvalue computation implementation — (c)

We implement the algorithm devised in question 1.2. We provide several numerical investigations with various values of n.

You may find the python implementation of said algorithm in the `exercise2.py` file and you may also run the automatic testing of the function by using `pytest test/test_exercise2.py` while in `cw3` directory.

1.4 Extending custom method for a new tridiagonal matrix — (d)

We now consider a $2n \times 2n$ tridiagonal matrix B of the form $\begin{cases} B_{ij} = -1 & \text{if } j = i - 1 \\ A_{ij} = 2 & \text{if } j = i + 1 \\ A_{ij} = 0 & \text{otherwise} \end{cases}$

We compute the matrix B and further apply the pure QR algorithm. We observe that in this new case the output matrix R of the QR factorization is not upper triangular neither. The first lower sub-diagonal is not filled with zeros, as what we had previously. However, by contrast with the QR factorization of matrix A, the upper-triangular entries are here not all close to zero.

Moreover, we notice that n diagonally aligned 2×2 block matrix also arise in the structure of this newly given R. Those block matrix are of the form :

$$\begin{pmatrix} 0 & A_{k,k+1} \\ A_{k+1,k} & 0 \end{pmatrix}, \text{ where } A_{k,k+1}, A_{k,k+1} \in \mathbb{C} \text{ are not necessarily equal.}$$

Again, by using the block determinant formula we have that the eigenvalues of R are the ones of those n diagonally aligned sub-matrix. The corresponding eigenvalues are $\pm\sqrt{A_{k,k+1}A_{k,k+1}}$.

We extend our previous method in order to compute the eigenvalues of matrix B. Namely, we again perform a pure QR factorization of B, and extract all non zero entries in the first upper-diagonal and first lower-diagonal of R. The eigenvalues of B are given by the positive and negative square root of their product.

You may find the python implementation of this new method in the `exercise2.py` file and you may also run the automatic testing of the function by using `pytest test/test_exercise2.py` while in `cw3` directory.

2 QR algorithm and reduction to tridiagonal form — Exercise 3

The relevant python files are `exercise3.py` and `test/test_exercise3.py`, located in the `cw3` directory. We here also modify the `exercises9.pure_QR` function located in the `cla_utils` directory.

2.1 Pure QR algorithm combined with tridiagonal reduction — (a)

We consider the pure QR algorithm. We prove that said algorithm preserves the structure of inputed symmetric tridiagonal matrix.

First consider any symmetric input matrix $A^{(0)} \in \mathscr{M}_n(\mathbb{R})$. By induction consider that at step k, $A^{(k-1)}$ is symmetric. At said step k of the process, we have $A^{(k-1)} = Q^{(k)}R^{(k)}$. The next iteration of the matrix is then computed as follows : $A^{(k)} = R^{(k)}Q^{(k)}$. Using those two products, we can write $A^{(k)} = {Q^{(k)}}^T A^{(k-1)} Q^{(k)}$, as $Q^{(k)}$ is orthonormal. We consider real matrix here. Furthermore, we directly have that ${A^{(k)}}^T = {Q^{(k)}}^T {A^{(k-1)}}^T ({Q^{(k)}}^T)^T = {Q^{(k)}}^T A^{(k-1)} Q^{(k)} = A^{(k)}$, which proves that $A^{(k)}$ is also symmetric. Thus, we show that the pure QR algorithm preserves the symmetry of any such inputed matrix.

Secondly, we show that if $A^{(k-1)} = Q^{(k)}R^{(k)}$ is tridiagonal, then $A^{(k)} = R^{(k)}Q^{(k)}$ is also tridiagonal. We show that $A^{(k)}_{ij} = \langle e_i, A^{(k)} e_j \rangle = 0$ for all $|i-j| > 1$. Considering the column vectors $q^{(k)}_i$ of $Q^{(k)}$, we have that $q^{(k)}_i \in \text{vect}\{e_1, \dots, e_{i+1}\}$. It follows that for $j > i+2$, $(A^{(k-1)} q^{(k)}_i)_j = 0$. We also have that $q^{(k)}_i$ is not a linear combination of the $i-1$ columns of $A^{(k-1)}$, and so that $A^{(k-1)} q^{(k)}_i \in \text{vect}\{e_i, e_{i+1}, e_{i+2}\}$. This gives us that for all i and j such that $i - j < 1$ we have $\langle q^{(k)}_j, A^{(k-1)} q^{(k)}_i \rangle = 0$, which further gives that for $i - j > 1$, $\langle e_j, A^{(k)} e_i \rangle = 0$. However, having $A^{(k)}$ symmetric give us finally that $A^{(k)}_{ij} = \langle e_i, A^{(k)} e_j \rangle = 0$ for all $|i-j| > 1$.

2.2 Modifying previous implementation of the QR algorithm — (b)

We modify our previously implemented function `exercises9.pure_QR`, to add a new termination criteria that is such that the inputed tridiagonal matrix T verifies $|T_{m,m-1}| < 10^{-12}$. To activate this new termination criteria, we add a new parameter condition `cw3Condition` that has to be set to `True`.

We apply this new stopping condition to the 5×5 matrix A, defined by $A_{ij} = \frac{1}{i+j+1}$. This matrix being symmetric, we apply the Hessenberg algorithm to reduce it to a tridiagonal form. We further apply the pure QR algorithm with our new stopping condition. As the output is roughly upper-triangular, we directly access the eigenvalues. We measure the convergence of those eigenvalues by computing $\det(A - \lambda_i I)$. We notice that the 3 last diagonal entries of R converged reasonably well to relevant eigenvalues, as the corresponding determinant is small (approximately lower than 10^{-12}). However, the first 2 entries are not converged.

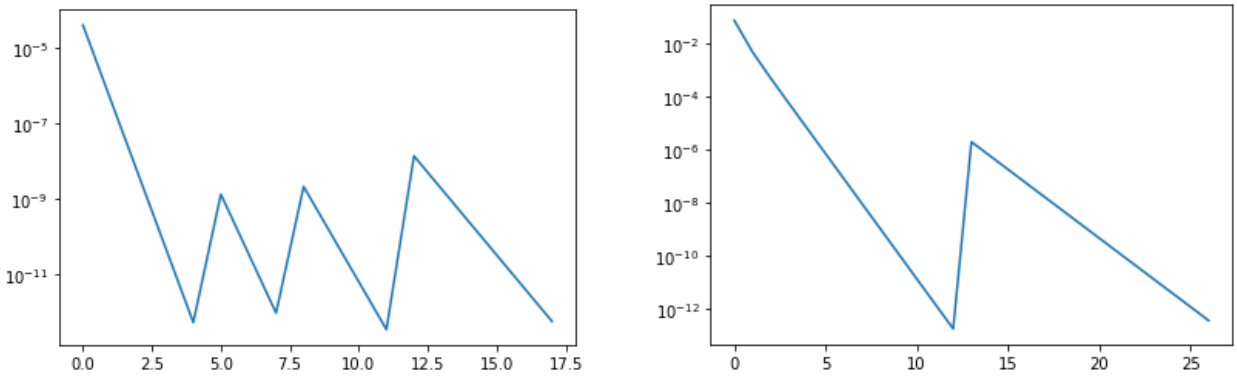
You may find the python implementation of said investigations in the `exercise3.py` file and you may also run related automatic testings by using `pytest test/test_exercise3.py` while in `cw3` directory.

2.3 Further modifications of the QR algorithm — (c)

We implement the procedure described in the subject. We again modify the pure QR algorithm using the `cw3Condition` condition. We apply the method to the 5×5 previously defined matrix, as well as to several random symmetric matrix.

We first investigate the aspect of the eigenvalues we obtain through this algorithm. We notice that the eigenvalues converge only for small size matrix. Namely, for $m > 10$ the computed values are distant from those obtain through `np.linalg.eig` from a factor greater than 10^{-4}.

Moreover, we investigate the residuals' evolution that we obtain through the several iterations of the pure QR algorithm. We concatenate these residuals and plot the convergence scheme of the given procedure. We obtain a sawtooth pattern. In particular, we notice that the residual fastly decreases toward the stopping condition of 10^{-12}, before giving place to the following sequence of residuals, related to the next iteration sub-matrix. As the first element of the following sequence is higher than the stopping condition, we observe said sawtooth pattern.



(a) Previously defined 5×5 symmetric matrix residuals sequences.

(b) Random 3×3 symmetric matrix residuals sequences.

Figure 1: Convergence scheme of the given procedure : evolution of the residuals sequences in log scale.

We measure the computation times of our unmodified and modified pure QR algorithm for computing eigenvalues of symmetric matrix. We consider our 5×5 matrix. We observe that the unmodified QR related method is truly faster than the procedure implemented in this question. However, we recall that the unmodfied method converges well for only the first few eigenvalues of the matrix, while the modified method gives more accurate values for all the entries.

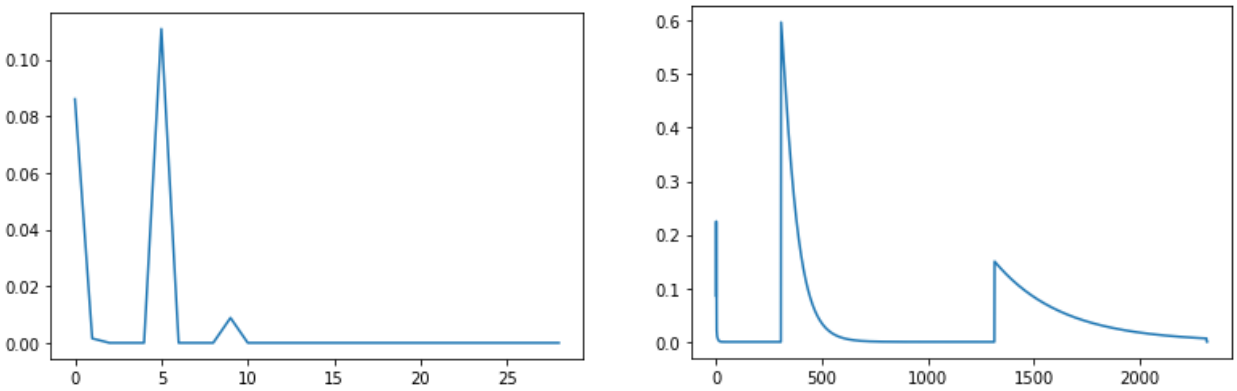
You may find the python implementation of the presented computations in the `exercise3.py` file and you may also run the corresponding automatic testings by using `pytest test/test_exercise3.py` while in `cw3` directory.

2.4 Wilkinson shift implementation — (d)

We implement the practical QR algorithm as described in section 5.11 of the lecture notes. We use the Wilkinson shift parameter defined in the subject. We modify the pure QR algorithm using a new `shiftCondition` condition. We apply a modified version of the procedure described in question 2.3, first tridiagonalizing, then applying the practical QR algorithm. This new method is

tested on the 5×5 previously defined matrix, as well as on several random symmetric matrix.

We observe that the concatenated residuals decreases faster than the unshifted version of the procedure, and the process needs less iterations to reach the stopping condition. For instance consider the following comparison of the convergence schemes with and without the Wilkinson shift implementation of a 15×15 random symmetric matrix :



(a) Random symmetric matrix residuals sequences with Wilkinson shift.

(b) Random symmetric matrix residuals sequences without Wilkinson shift.

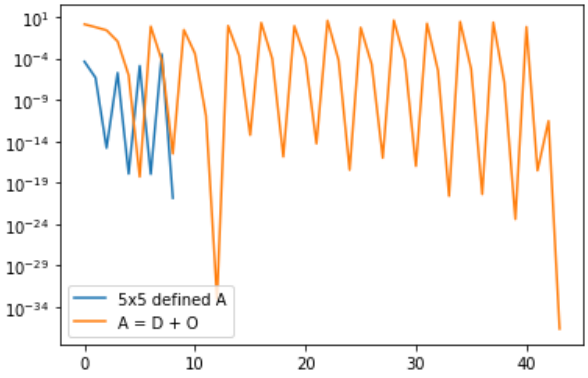
Figure 2: Evolution of the residuals sequences for a random 15×15 symmetric matrix with and without the use of the Wilkinson shift, in normal scale.

We observe here that the method using the Wilkinson shift uses 28 iterations, while the method without Wilkinson shift use 2316 iterations. This efficiency gain is a consequence of the eigenvalue estimate given by the Wilkinson shift formula.

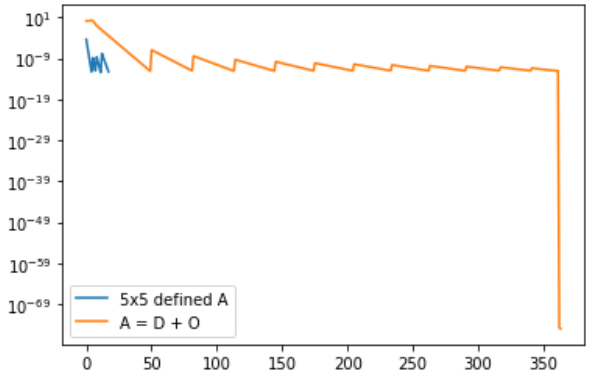
You may find the python implementation of those computations in the `exercise3.py` file. We ensured that the modified version the `exercises9.pure_QR` passes the corresponding automatic testing, that has been copied from the `comp-lin-alg-course/test/test_exercises9.py` file to the `cw3/test/test_exercise3.py` file. The testing function has been slightly modified, in order for it to make use of the Wilkinson shift. We notice that the automatic testing is truly faster with this new implementation.

2.5 Application to $A = D + O$ — (e)

We compute the convergence schemes with and without the Wilkinson shift method, for our previously defined 5×5 matrix and for the $A = D + O$ matrix - where $D = \text{diag}(15, 14, \ldots, 1)$ and O is the 15×15 matrix filled with ones. We plot the following results :



(a) Comparison of residuals sequences with Wilkinson shift.



(b) Comparison of residuals sequences without Wilkinson shift.

Figure 3: Evolution of the residuals sequences for our previously defined 5×5 symmetric matrix, and for our new matrix $A = D + O$, with and without the use of the Wilkinson shift, in log scale.

We notice that for both implementations the process needs less iterations for the 5×5 matrix, which is consistent with the difference in size between the two matrix. However, the difference between the number of iterations required is much higher in the case where the Wilkinson shift is not used (roughly 300 iterations more needed between the two matrix without the shift against approximately 30 more needed with the shift). Moreover, we notice that the last residual of the $A = D + O$ matrix is much smaller than the corresponding one of the 5×5 matrix (10^{-69} against 10^{-12} for the no shift version). Those differences arise in particular since the $A = D + O$ is well-conditioned with high diagonal entries, compared to the 5×5 matrix, where the diagonal coefficients are not dominant.

You may find the python implementation of the presented plots in the `exercise3.py` file.

3 Image denoising algorithm — Exercise 4

We here consider denoising an image stored as an $(n + 2) \times (n + 2)$ matrix, filled with brightness levels $u_{ij} \in [0, 1]$. We perform this task using an iterative method (see subject).

The relevant python files are `exercise4.py` and `test/test_exercise4.py`, located in the `cw3` directory. We also modify the `exercises10.GMRES` function located in the `cla_utils` directory.

3.1 Serializing relevant matrix — (a)

We implement a serializing function that flattens a given matrix into a 1D vector, by stacking the column entries one after the other. We use the `numpy.flatten` method. We apply it to the transpose of the inputed matrix, as it works row-wise by default. We also implement a deserializing function that applies to opposite process, i.e. taking a flatten vector and computing back the corresponding matrix. We retrieve the matrix size using the fact that any $m \times m$ inputed matrix outputs an m^2-length vector. We use the `numpy.reshape` method to perform this task.

You may find the python implementation of said functions in the `exercise4.py` file and you may also run related automatic testings by using `pytest test/test_exercise4.py` while in `cw3` directory.

3.2 Applying $H = \mu I + \lambda A$ — (b)

We implement a function that takes the serialized vector v and applies $Hv = (\mu I + \lambda A)v$, for the inputed parameters μ and λ. We follow the provided matrix-free procedure.

You may find this python implementation in the `exercise4.py` file and you may also run the corresponding automatic testings by using `pytest test/test_exercise4.py` while in `cw3` directory.

3.3 Modifying GMRES implementation — (c)

We modify our GMRES implementation in order to have the possibility to provide a function as argument, instead of the initial matrix A. As the GMRES algorithm solves $Ax = b$, we may provide a function that acts on the vector x in the same way as the matrix product Ax. This black box function may prove more efficient than the full-length product in certain situations. Namely, we provide an optional argument `func` in our GMRES python implementation which - when not set to `None` - replaces all `numpy` products of the type `np.dot(A,y)`.

You may find the python modification of the GMRES function in the `cla_utils/exercise10.py` file and you may also run specific automatic testings by using `pytest test/test_exercise4.py` while in `cw3` directory.

3.4 Sweeping algorithm — (d)

We implement the described sweeping algorithm. You may find the related python function in the `exercise4.py` file.

3.5 Preconditioned GMRES algorithm — (e)

We modify our GMRES algorithm to allow some optional preconditioning. You may find the python modification of the GMRES function in the `cla_utils/exercise10.py` file and you may also run specific automatic testings by using `pytest test/test_exercise4.py` while in `cw3` directory.