Numerical and Symoblic Algorithms Modeling

MU4IN901

Implementation work around matrices

Ezzedine CHAHINE - (21220711)
Jad YEHYA - (28609954)

December 29, 2022

Contents

1 Introduction

In this report, we will present the results of the implementation of the QR algorithm to compute the eigenvalues of a matrix. First, we will see the implementation of QR decomposition using the Givens rotation and discuss its complexity. Then, we will address the eigenvalues problem and finally, we will discuss the results, the limitations of our code, and conclude.

In this document, we will talk about QR factorization. This factorization is a very important tool in linear algebra. It is used to solve linear systems, to compute eigenvalues and eigenvectors, to compute the singular value decomposition, least squares problems, and many other things. The goal is to decompose a matrix A into two matrices Q and R such that $A = QR$, with Q orthogonal and R upper triangular. It is also used to compute a Schur decomposition of a matrix.
The code is written in C and is given in the annex with some comments.

2 Rotation matrices, upper Hessenberg form, and eigenvalues

2.1 Mathematical base of the problem

The core of this project is that we want to compute the eigenvalues of a matrix A using the QR factorization, but first we need to know the maths behind it. We will prove that A_n has the same eigenvalues as A.

The eigenvalues of a matrix A are the roots of the characteristic polynomial of A which is defined as follows :

$$p_A(\lambda) = \det(\lambda I - A)$$

The eigenvalues of the matrix A are the roots of this polynomial $p_A(\lambda)$. Now we will prove that A has the same eigenvalues as $Q\,A\,Q^\star$ where Q is an orthogonal matrix. Q being an orthogonal matrix, we have $Q^\star\,Q = I$ and $Q\,Q^\star = I$. So we have : $Q^\star = Q^{-1}$. Having explicit this, we notice that going from A to $Q\,A\,Q^\star$ is a similarity transformation. And we know that the eigenvalues of a matrix are invariant under similarity transformations. So we have : A has the same eigenvalues as $Q\,A\,Q^\star$.

Proof. Let A, B, and Q be matrices such that $B = Q\,A\,Q^\star$, Q being orthogonal.

$$\begin{aligned} B - \lambda I &= Q\,AQ^\star - \lambda I \\ &= Q\,AQ^\star - \lambda QQ^\star \\ &= Q\,AQ^\star - Q\lambda Q^\star && \text{(because } \lambda \text{ is a scalar)} \\ &= Q\,AQ^\star - Q\lambda IQ^\star \\ &= Q\,(A - \lambda I)Q^\star \end{aligned}$$

Therefore,

$$\begin{aligned} det(B - \lambda I) &= det(Q\,(A - \lambda I)Q^\star) \\ &= det(Q)\,det(A - \lambda I)\,det(Q^\star) \\ &= det(Q)\,det(A - \lambda I)\,det(Q^{-1}) && \text{(because } Q^\star = Q^{-1}\text{)} \\ &= det(Q)\,det(A - \lambda I)\,det(Q)^{-1} \\ &= det(A - \lambda I) && \text{(because } det(Q) * det(Q)^{-1} = 1\text{)} \end{aligned}$$

So we have : $det(B - \lambda I) = det(A - \lambda I)$. And we know that the eigenvalues of a matrix are the roots of its characteristic polynomial. So we have : A has the same eigenvalues as $QAQ^\star$. □

2.2 Upper Hessenberg form

An upper Hessenberg matrix is a matrix that has zeros everywhere below the first subdiagonal. We will show that there exists a rotation matrix $G_{n-1,n}$ such that both $G_{n-1,n}A$ and $A' = G_{n-1,n}AG^\star_{n-1,n}$ have a 0 coefficient in position (n,1).
We know that

$$G_{n-1,n} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & c & s \\ 0 & \dots & -s & c \end{pmatrix}$$

When multiplying $G_{n-1,n}A$, we only modify the $n-1$th and nth lines. of A. So we have:

$$G_{n-1,n}A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a'_{n-1,2} & \dots & a'_{n-1,n} \\ a_{n,1} & a'_{n,2} & \dots & a'_{n,2} \end{pmatrix}$$

The goal is to determine c and s such that $a_n = 0$. We get the following equations :

$$\begin{cases} a_{n-1,1}c + a_{n,1}s = \alpha \\ -a_{n-1,1}s + a_{n,1}c = 0 \end{cases}$$

We then multiply the first equation by $a_{n-1,1}$ and the second equation by $a_{n,1}$ and we get :

$$\begin{cases} a_{n-1,1}^2 c + a_{n,1}a_{n-1,1}s = \alpha a_{n-1,1} \\ -a_{n-1,1}a_{n,1}s + a_{n,1}^2 c = 0 \end{cases}$$

We call A_1 the first column of A. Since we must have $\|A_1\| = \|G_{n-1,n}A\|$, we have : $\alpha = \sqrt{a_{n-1,1}^2 + a_{n,1}^2}$. So we get :

$$\begin{cases} c = \frac{a_{n-1,1}}{\alpha} \\ s = \frac{a_{n,1}}{\alpha} \end{cases}$$

We notice that multiplying on the right by $G_{n-1,n}^{\star}$ only modifies the $n-1$th and nth columns of A so the 0 coefficient we created is still there in the same position.
However, multiplying on the left by a matrix and on the right by its transpose is interesting because this is a similarity transformation and we know that two similar matrices have the same eigenvalues.

similarly, we can show that there exists a rotation matrix $G_{n-2,n-1}$ such that both $G_{n-2,n-1}A$ and $A'' = G_{n-2,n-1}A'G_{n-2,n-1}^{\star}$ have a 0 coefficient in position $(n-1,1)$. Thus, we will have :

$$\begin{cases} c = \frac{a_{n-2,1}}{\alpha} \\ s = \frac{a_{n-1,1}}{\alpha} \end{cases} \quad \text{where } \alpha = \sqrt{a_{n-2,1}^2 + a_{n-1,1}^2}$$

If we want $G'_{n-1,n}$ such that A''' has a 0 coefficient in position $(n,2)$, we will have : $\alpha = \sqrt{a_{n-1,2}^2 + a_{n,2}^2}$ and

$$\begin{cases} c = \frac{a_{n-1,2}}{\alpha} \\ s = \frac{a_{n,2}}{\alpha} \end{cases}$$

2.3 Hessenberg Algorithm

Algorithm 1 upperHessenberg(matrix A)

1: $n \leftarrow$ number of rows in A
2: **for** $k \leftarrow 1$ to $n-2$ **do**
3: **for** $i \leftarrow k+2$ to n **do**
4: $\alpha \leftarrow \sqrt{A_{i-1,k}^2 + A_{i,k}^2}$
5: $c \leftarrow \frac{A_{i-1,k}}{\alpha}$
6: $s \leftarrow \frac{A_{i,k}}{\alpha}$
7: $G \leftarrow$ createGivensMatrix$(c, s, i, i+1)$
8: $A \leftarrow G * A$
9: **end for**
10: **end for**
11: **return** A

We start the loop of the rows at $k+2$ because the final matrix will be upper Hessenberg, so we only need to zero elements from the 3rd row and down. similarly, we stop at $n-2$ because we only need to zero elements until the second to last column.
In the inner loop, we proceed to zero each element concerned, then we change column when finished.

3 Givens algorithm

The Givens algorithm is a method to compute the QR factorization of a matrix by doing a series of rotations. The idea is to apply a series of rotations to the matrix A to put it in upper triangular form. This result matrix will be the matrix R and the product of all the matrices G will give us the matrix Q. The rotations are done in such a way that the coefficients on the subdiagonal are zero. The rotations are done using the Givens rotation matrix G which is defined as follows :

$$G(i,j) = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & c & \dots & s & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & -s & \dots & c & 0 \\ 0 & \vdots & \dots & \vdots & 1 \end{pmatrix}$$

where c, $s \in \mathbb{R}$ such that $c^2 + s^2 = 1$ and $G_{i,i} = G_{j,j} = c$. and $G_{i,j} = -G_{j,i} = s$

3.1 Implementation

For all our functions, we chose to use the following convention :

- A is a matrix of size $m \times n$.
- R is a matrix of size $m \times n$.
- Q is a matrix of size $m \times m$.
- G is a matrix of size $m \times m$.

For all our functions, we chose to "return" the result as a parameter of the function. This is done to avoid memory leaks and to facilitate debugging. Moreover, as we will see in section 4 it is useful when doing recursion. We also chose to use the convention that the first index of a matrix is the row and the second index is the column. Thus, the element $A_{i,j}$ is the element in the i-th row and j-th column of the matrix A.
Also, for the eigenvalues to be real, we made our test on symmetric matrices as the complex numbers are not implemented in our code. (More on this in section 4)

Here is the code that computes Q and R using the Givens algorithm. We will explain the code after.

```
1  void Givens3(double **A, int m, int n, double*** Q, double*** R){
2      double**G = (double **) malloc(m * sizeof(double *));
3      double**Gt = (double **) malloc(m * sizeof(double *));
4      double**R_tmp = (double **) malloc(m * sizeof(double *));
5      double**Q_tmp = (double **) malloc(m * sizeof(double *));
```

```
6       for (int l = 0; l < m; ++l) {
7           G[l] = (double *) malloc(m * sizeof(double));
8           Gt[l] = (double *) malloc(m * sizeof(double));
9           R_tmp[l] = (double *) malloc(n * sizeof(double));
10          Q_tmp[l] = (double *) malloc(m * sizeof(double));
11      }
12
13      for (int i = 0; i < m; ++i) {
14          (*Q)[i][i] = 1;
15      }
16
17      // Copying A in R
18      copy(A, *R, m, n);
19
20      for (int j = 0; j < n; ++j) {
21          for (int i = j+1; i < m; ++i) {
22              generate_G(G, i, j, m, *R);
23              multiplymatrices(G, *R, m, &R_tmp);
24              transpose_matrix(G, m, m, &Gt);
25              multiplymatrices(*Q, Gt, m, &Q_tmp);
26              // Emptying G
27              for (int k = 0; k < m; ++k) {
28                  for (int l = 0; l < m; ++l) {
29                      G[k][l] = 0;
30                  }
31              }
32              for (int k = 0; k < m; ++k) {
33                  for (int l = 0; l < n; ++l) {
34                      (*R)[k][l] = R_tmp[k][l];
35                  }
36              }
37              for (int k = 0; k < m; ++k) {
38                  for (int l = 0; l < m; ++l) {
39                      (*Q)[k][l] = Q_tmp[k][l];
40                  }
41              }
42          }
43
44      }
45      // FREES
46      free_matrix(G, m);
47      free_matrix(Gt, m);
48      free_matrix(R_tmp, m);
49      free_matrix(Q_tmp, m);
50
51  }
```

The first 19 lines of the code are only used to allocate the memory for the

matrices. After that, we initialize Q to the identity matrix. Then we do a deep copy of A in R using the function `copy`. After that, we start the main loops by iterating over R. For each row, we generate the Givens rotation matrix G using the function `generate_G`. We multiply G by R and store the result in R using the function `multiplymatrices`. We transpose G and store it in G^T using the function `transpose_matrix`. We multiply Q by G^T and store the result in Q. We empty G by setting all its elements to 0. We copy R_tmp and Q_tmp in R and Q using the function `copy`. Finally, we use the function `free_matrix` to free the memory.

The function `generate_G` is the following :

```
1  void generate_G(double **G, int i,
   ↪  int j, int m, double** R){
2      double c =
   ↪  R[j][j]/sqrt(R[j][j]*R[j][j] +
   ↪  R[i][j]*R[i][j]);
3      double s =
   ↪  R[i][j]/sqrt(R[j][j]*R[j][j] +
   ↪  R[i][j]*R[i][j]);
4
5      for (int k = 0; k < m; ++k) {
6          if (k == i || k == j) {
7              G[k][k] = c;
8          } else {
9              G[k][k] = 1;
10         }
11     }
12     G[i][j] = -s;
13     G[j][i] = s;
14 }
```

This function allows us to generate the Givens rotation matrix G for the row i and the column j of the matrix R. The matrix G has been defined in 3. The function takes as input the matrix R and the row and column indices i and j. The function returns the matrix G. where $c = \frac{R_{jj}}{\sqrt{R_{jj}^2+R_{ij}^2}}$ and $s = \frac{R_{ij}}{\sqrt{R_{jj}^2+R_{ij}^2}}$.

We also have tested our program using a memory checker. We have used the tool `leaks` to check for memory leaks (as `Valgrind` wasn't working on our computer). The tool `leaks` is a tool that allows us to check for memory leaks. It is a part of the Xcode developer tools. We have used the command `leaks -atExit -- ./main_given` to check for memory leaks. The result of the command is as follows :

```
Process:         main_given [37303]
Path:            /Users/USER/*/main_given
Load Address:    0x10008c000
Identifier:      main_given
Version:         0
Code Type:       X86-64
Platform:        macOS
Parent Process:  leaks [37302]
```

```
Date/Time:        2022-12-25 11:25:32.208 +0100
Launch Time:      2022-12-25 11:25:31.862 +0100
OS Version:       macOS 13.0.1 (22A400)
Report Version:   7
Analysis Tool:    /usr/bin/leaks

Physical footprint:         1624K
Physical footprint (peak):  1624K
Idle exit: untracked
----

leaks Report Version: 4.0, multi-line stacks
Process 37303: 181 nodes malloced for 14 KB
Process 37303: 0 leaks for 0 total leaked bytes.
```

3.2 Complexity

To test the efficiency of the algorithm, we computed the time it takes to compute the QR factorization of a matrix of size $n \times n$, $n \in [2; 50]$ with random coefficients in $\mathbb{R}$. We repeated this process 2000 times for each size n. Plotting the average time taken to compute QR decomposition using Givens algorithm, we get figure 1.

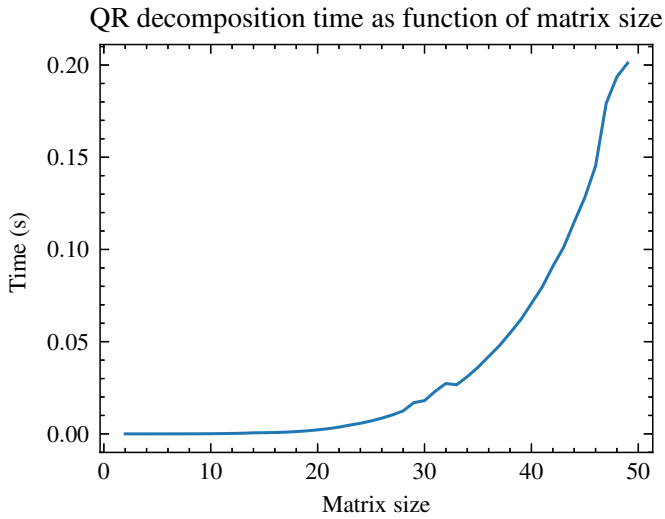

Figure 1: Complexity of Givens algorithm

Regarding the complexity of the algorithm, we can see that the complexity is

$O(n^3)$ because we have to iterate over n rows and n columns. For each row and column, we have to compute the Givens rotation matrix G which has a complexity of $O(n^2)$. Then we have to multiply G by R and Q by G^T which has a complexity of $O(n^3)$. So, the complexity of the algorithm is $O(n^3)$.
In the memory complexity, we have to allocate memory for the matrices G, G^T, R_tmp, and Q_tmp. The size of these matrices is $m \times m$. So the memory complexity is $O(m^2)$.

3.3 Comparing with MPFR

The MPFR implementation was a bit more tricky as it is a new library that we never used before. Somehow, because of our programming choices with the `double` implementation, and the fact that all the functions are `void` functions and the result is stored in a matrix whose address is passed as a parameter, we were able to adapt the code to the MPFR library without too much trouble. Moreover, this facilitates the comprehension of the structure of the code, as it is very similar to the `double` implementation.
The code is the following :

```
1  void Givens_mpfr(mpfr_t ** A, int m, int n, mpfr_t ** * Q, mpfr_t ** * R) {
2    // Initialization of G, Gt, R_tmp, Q_tmp
3    mpfr_t ** G, ** Gt, ** R_tmp, ** Q_tmp;
4    G = (mpfr_t ** ) malloc(m * sizeof(mpfr_t * ));
5    Gt = (mpfr_t ** ) malloc(m * sizeof(mpfr_t * ));
6    R_tmp = (mpfr_t ** ) malloc(m * sizeof(mpfr_t * ));
7    Q_tmp = (mpfr_t ** ) malloc(m * sizeof(mpfr_t * ));
8    for (int i = 0; i < m; i++) {
9      G[i] = (mpfr_t * ) malloc(m * sizeof(mpfr_t));
10     Gt[i] = (mpfr_t * ) malloc(m * sizeof(mpfr_t));
11     R_tmp[i] = (mpfr_t * ) malloc(n * sizeof(mpfr_t));
12     Q_tmp[i] = (mpfr_t * ) malloc(m * sizeof(mpfr_t));
13     for (int j = 0; j < m; j++) {
14       mpfr_init2(G[i][j], 128);
15       mpfr_init2(Gt[i][j], 128);
16       mpfr_init2(Q_tmp[i][j], 128);
17     }
18     for (int j = 0; j < n; j++) {
19       mpfr_init2(R_tmp[i][j], 128);
20     }
21   }
22
23   // Initialization of Q to identity matrix
24   for (int i = 0; i < m; i++) {
25     for (int j = 0; j < m; j++) {
26       if (i == j) mpfr_set_d(( * Q)[i][j], 1, MPFR_RNDN);
```

```
27          else mpfr_set_d(( * Q)[i][j], 0, MPFR_RNDN);
28        }
29      }
30
31      // Copying A into R
32      copy_mpfr_matrix(A, m, n, R);
33
34      for (int j = 0; j < n; ++j) {
35        for (int i = j + 1; i < m; ++i) {
36          // ================ Givens rotation Matrix ================
37          // Initialization of G to identity matrix
38          for (int k = 0; k < m; k++) {
39            for (int l = 0; l < m; l++) {
40              if (k == l) mpfr_set_d(G[k][l], 1, MPFR_RNDN);
41              else mpfr_set_d(G[k][l], 0, MPFR_RNDN);
42            }
43          }
44          // Initialization of c, s, tmp, tmp2
45          mpfr_t c, s, tmp, tmp2;
46          mpfr_init2(c, 128);
47          mpfr_init2(s, 128);
48          mpfr_init2(tmp, 128);
49          mpfr_init2(tmp2, 128);
50
51          // Calculating c and s
52          mpfr_mul(tmp, ( * R)[j][j], ( * R)[j][j], MPFR_RNDN);
53          mpfr_mul(tmp2, ( * R)[i][j], ( * R)[i][j], MPFR_RNDN);
54          mpfr_add(tmp, tmp, tmp2, MPFR_RNDN);
55          mpfr_sqrt(tmp, tmp, MPFR_RNDN);
56
57          mpfr_div(c, ( * R)[j][j], tmp, MPFR_RNDN);
58          mpfr_div(s, ( * R)[i][j], tmp, MPFR_RNDN);
59
60          // Calculating G
61          mpfr_set(G[j][j], c, MPFR_RNDN);
62          mpfr_set(G[i][i], c, MPFR_RNDN);
63
64          mpfr_set(G[i][j], s, MPFR_RNDN);
65          mpfr_neg(s, s, MPFR_RNDN);
66          mpfr_set(G[j][i], s, MPFR_RNDN);
67
68          mpfr_clear(c);
69          mpfr_clear(s);
70          mpfr_clear(tmp);
71          mpfr_clear(tmp2);
72          // ========================================================
73
74          multiply_mpfr_matrices(G, * R, m, m, & R_tmp);
75          transpose_mpfr_matrix(G, m, m, & Gt);
```

```
76        multiply_mpfr_matrices( * Q, Gt, m, m, & Q_tmp);
77
78        // Emptying G
79        for (int k = 0; k < m; k++) {
80          for (int l = 0; l < m; l++) {
81            mpfr_set_d(G[k][l], 0, MPFR_RNDN);
82          }
83        }
84        // Copying R_tmp into R without using copy_matrix_mpfr and emptying R_tmp
85        for (int k = 0; k < m; k++) {
86          for (int l = 0; l < n; l++) {
87            mpfr_set(( * R)[k][l], R_tmp[k][l], MPFR_RNDN);
88            mpfr_set_d(R_tmp[k][l], 0, MPFR_RNDN);
89          }
90        }
91        // Copying Q_tmp into Q without using copy_matrix_mpfr and emptying Q_tmp
92        for (int k = 0; k < m; k++) {
93          for (int l = 0; l < m; l++) {
94            mpfr_set(( * Q)[k][l], Q_tmp[k][l], MPFR_RNDN);
95            mpfr_set_d(Q_tmp[k][l], 0, MPFR_RNDN);
96          }
97        }
98        // If an element of the subdiagonal is below the threshold, we set it to
    ↪  0
99        mpfr_t threshold;
100       mpfr_init2(threshold, 128);
101       mpfr_set_d(threshold, 1e-23, MPFR_RNDN);
102
103       mpfr_t absRij;
104       mpfr_init(absRij);
105       mpfr_abs(absRij, ( * R)[i][j], MPFR_RNDN);
106       if (mpfr_cmp(absRij, threshold) < 0) mpfr_set_d(( * R)[i][j], 0,
    ↪  MPFR_RNDN);
107       mpfr_clear(absRij);
108       mpfr_clear(threshold);
109     }
110   }
111   // Freeing memory
112   free_mpfr_matrix(G, m, m);
113   free_mpfr_matrix(Gt, m, m);
114   free_mpfr_matrix(R_tmp, m, n);
115   free_mpfr_matrix(Q_tmp, m, m);
116 }
```

The process is the same as the one we used in `double` precision. The only difference is that we initialize the G rotation matrix directly in the function instead of calling an external function.

After running some tests, we notice that there is a "reconstruction error" when

computing $Q\,R$, we don't get the original matrix A. This is due to the fact that we are using `mpfr` numbers and not `double` numbers. The error is of the order of 10^{-23} for matrices of size 20×20, which is acceptable for our purposes, as we can see in the following graph. The process of calculating the error is : $\sum_{i=0}^{M}\sum_{j=0}^{N}(A_{ij}-(QR)_{ij})^2$.

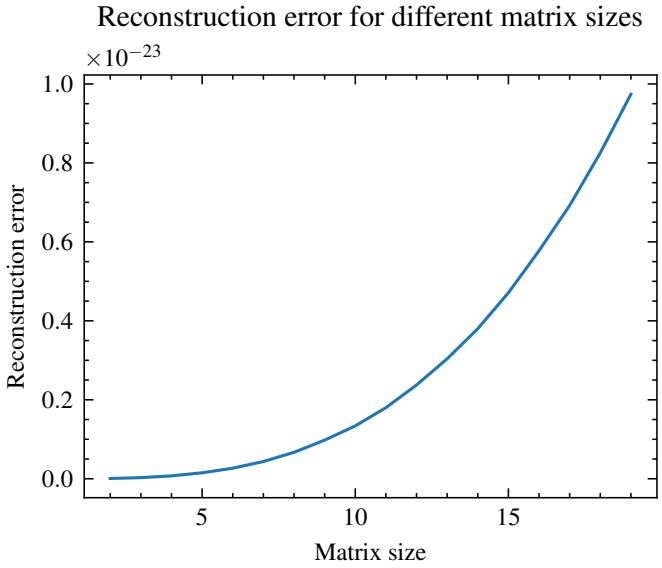

Figure 2: Quadratic reconstruction error according to the size of the matrix

However, when we compute the reconstruction error for `double` precision, we get 0.000 error because it is not possible to have a reconstruction error of 0 with `double` precision. This is due to the fact that the `double` precision is not enough to store the exact value of the error.

Furthermore, the MPFR implementation takes much more time than the `double` implementation. This is due to the fact that we are using `mpfr` numbers, which are slower to compute than `double` numbers. We can see this in the following graph.

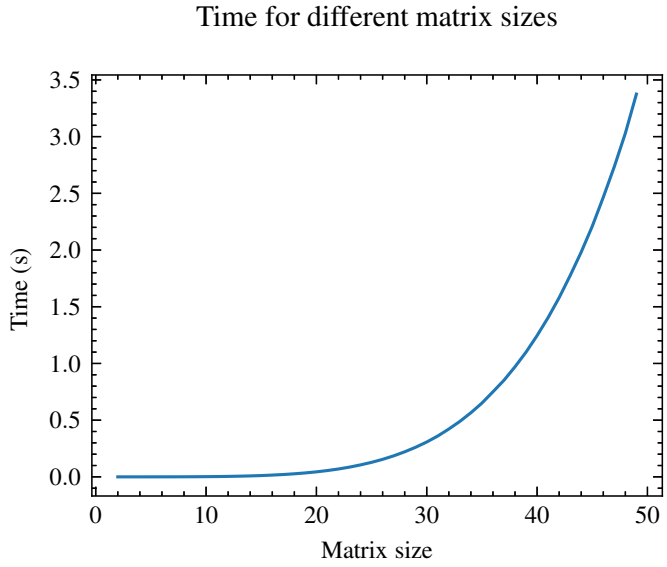

Figure 3: Average time taken to compute the QR factorization of a matrix using Givens rotations with `MPFR` precision according to the size of the matrix

This test was done by doing 1000 iterations of the QR factorization of a random matrix of size $n \times n$. It took more than 8 hours to be done ! It is approximately 17.5 times slower than the `double` implementation. (see Figure 1).

4 Problems around eigenvalues

We will build a sequence of matrices $A_0, A_1, \ldots, A_n$ such that A_n is upper triangular and $A_{k+1} = Q_k \, A_k \, Q_k^\star$ where Q_n is an orthogonal matrix.

4.1 Computing eigenvalues using QR factorization

In order to compute the eigenvalues of a matrix, a common approach is to use the QR factorization of the matrix. As we have previously shown, if Q is an orthogonal matrix, then $Q \, A \, Q^\star$ has the same eigenvalues as A. In that matter, a way of computing the eigenvalues of a matrix recursively is to compute A_i such that

$$A_{i+1} = Q A_i Q^\star$$

and stop when A_i is upper triangular. Being on a computer, the coefficients of A_i on the subdiagonal won't be exactly zero. So we will stop when the coefficients are smaller than a given threshold.

4.2 Implementation

The implementation is straightforward. We use the Givens algorithm presented in Section 2 to compute Q and then compute $A_{i+1} = QA_iQ^\star$. We stop when the coefficients on the subdiagonal are smaller than a given threshold.

```
void compute_eigenvalues3(double ***A, int m, int n, long* nb_iter){
    if((thresh((*A), m, n, 1e-12) == 0) || (*nb_iter > 1000000)){
        printf("End of the recursion\n");
        printf("Number of iterations : %li\n", *nb_iter);
        printf("Eigenvalues : \n");
        for (int i = 0; i < m; ++i) {
            printf("%f\n", (*A)[i][i]);
        }
        return;
    }

    double **Q      = (double **) malloc(m * sizeof(double *));
    double **R      = (double **) malloc(m * sizeof(double *));
    double **Qt     = (double **) malloc(m * sizeof(double *));
    double **A2     = (double **) malloc(m * sizeof(double *));
    double **A_temp = (double **) malloc(m * sizeof(double *));

    for (int l = 0; l < m; ++l) {
        Q[l] =      (double *) malloc(m * sizeof(double));
        R[l] =      (double *) malloc(n * sizeof(double));
        Qt[l] =     (double *) malloc(m * sizeof(double));
        A2[l] =     (double *) malloc(n * sizeof(double));
        A_temp[l] = (double *) malloc(n * sizeof(double));
    }
    Givens3((*A), m, n, &Q, &R);
    transpose_matrix(Q, m, m, &Qt);
    multiplymatrices(Q, (*A), m, &A2);
    multiplymatrices(A2, Qt, m, &A_temp);

    free_matrix(Q, m);
    free_matrix(Qt, m);
    free_matrix(R,  m);
    free_matrix(A2, m);

    (*nb_iter)++;

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            (*A)[i][j] = A_temp[i][j];
        }
    }
    free_matrix(A_temp, m);

    compute_eigenvalues3(A, m, n, nb_iter);
}
```

The function `thresh` is used to check if the coefficients on the subdiagonal are less than a given threshold. It returns 0 if the threshold is reached and 1 otherwise. In this function, the base case is given in the first condition : if the threshold is reached, we stop the recursion. If not, we are in recursion. We compute Q and R using the Givens algorithm. Then, we transpose Q and compute $A_{i+1} = QA_iQ^\star$ in two steps. We free the memory used for Q, R, and A_i. We increment the number of iterations and call the function again with our new A. The `long nb_iter` is used to count the number of iterations. We will use it later to compare the precision of the algorithm with the precision of the MPFR library and to see the complexity of the algorithm.

Like in the previous section, we also did a memory check using `Leaks` and we didn't find any memory leak. The report is the following :

```
Process:         main_deb [97786]
Path:            /Users/USER/*/main_deb
Load Address:    0x10b53b000
Identifier:      main_deb
Version:         0
Code Type:       X86-64
Platform:        macOS
Parent Process:  leaks [97785]

Date/Time:       2022-12-23 16:48:23.224 +0100
Launch Time:     2022-12-23 16:48:22.807 +0100
OS Version:      macOS 13.0.1 (22A400)
Report Version:  7
Analysis Tool:   /usr/bin/leaks

Physical footprint:         2512K
Physical footprint (peak):  2512K
Idle exit: untracked
----

leaks Report Version: 4.0, multi-line stacks
Process 97786: 176 nodes malloced for 13 KB
Process 97786: 0 leaks for 0 total leaked bytes.
```

4.3 Complexity

We have done some statistics on this algorithm. We have computed the eigenvalues of a matrix of size 2×2 to 10×10 with a threshold of $1e^{-2}$.

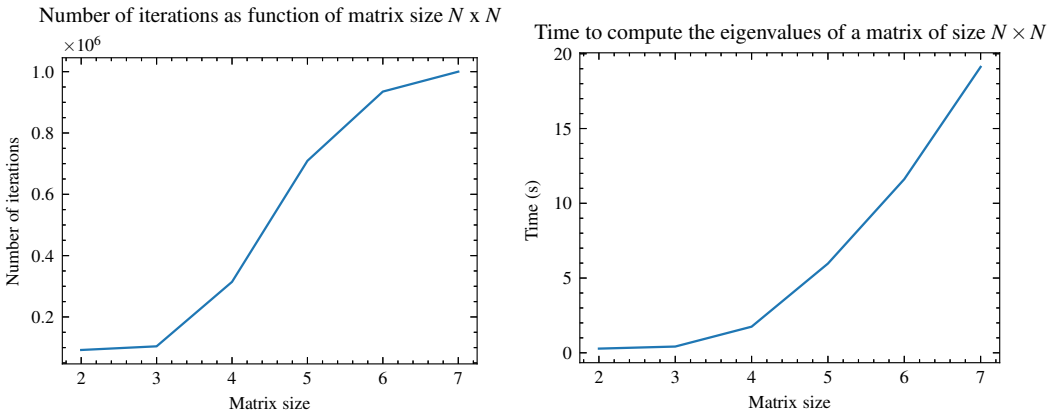

Figure 4: Average number of iterations and computation time as function of the size of the matrix.

The algorithm seems to have a complexity of $O(n^3)$.

The precision of the algorithm is not bad at all, we have tested the `double` for matrices of size 2×2 to 10×10 and ran the test 100 times for each size. On the other hand, we have used a `numpy` routine to compute the eigenvalues of the same matrices. We have done the same test and we have compared the results. The quadratic error is given in the graph below.

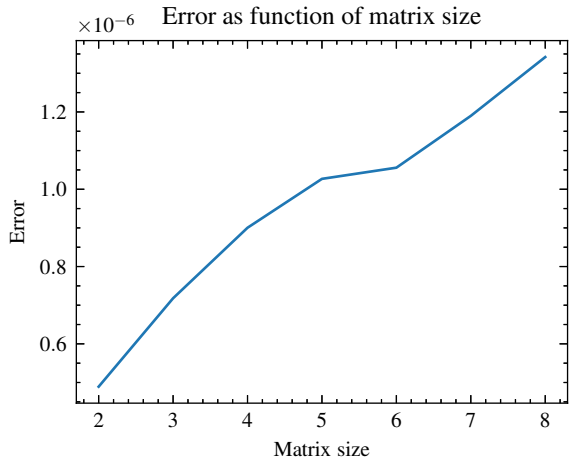

Figure 5: Quadratic error as function of the size of the matrix.

In `Numpy`, the numbers were encoded as `numpy.float128` which offers a precision of 128 bits, comparable to the `MPFR` one. We can see that our algorithm is precise, and the error is small, around a factor 10^{-6}.

4.4 Comparing with MPFR precision

Unfortunately, we weren't able to run reliable tests with the `MPFR` library for this part.

5 Conclusion

5.1 Improvements, limitations, and future work

Other method :

$$A_{k+1} = Q^\star A_k Q$$

This is an alternative method that we see more in research papers. Assume $A_0 = Q_0 R_0$ be the QR factorization of A. Let $A_1 = R_0 Q_0$. We can do the following computations :

$$\begin{aligned} A_1 &= R_0\, Q_0 \\ &= Q_0^\star\, Q_0\, R_0\, Q_0 \\ &= Q_0^\star\, A_0\, Q_0 \end{aligned}$$

We can generalize this process to get the following equation :

$$\begin{aligned} A_{i+1} &= R_i\, Q_i \\ &= Q_i^\star\, Q_i R_i\, Q_i \\ &= Q_i^\star\, A_i\, Q_i \end{aligned} \tag{1}$$

At each step, $A_{k+1} = R_k\, Q_k$ and we do a QR factorization on A_{k+1}. The core difference is just a similarity transformation added (or we can see it a substitution of Q by $Q^\star$ since $(Q^\star)^\star = Q$).We have implemented this algorithm, and we have done some statistics on it. The results are given in the graph below. We can see that the algorithm is (way) more efficient than the previous one. The complexity is $O(n^3)$ and the number of iterations is smaller. Also, for a given matrix size, it is approximately 10x faster.We have also done a memory check using `Leaks` and we didn't find any memory leak.

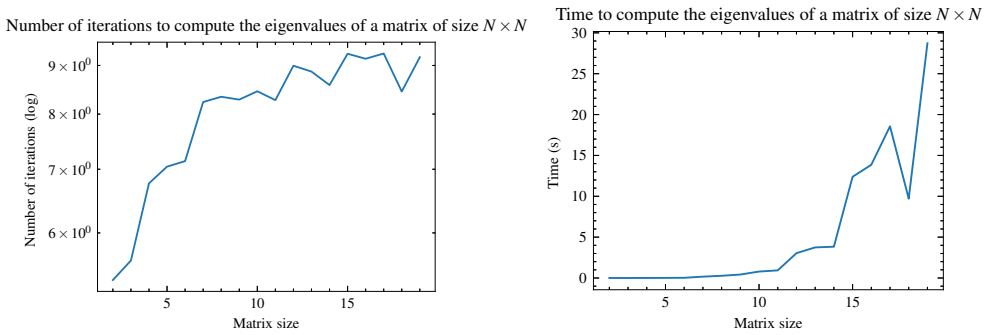

Figure 6: Average number of iterations and computation time as function of the size of the matrix.

Some clarification about the Hessenberg algorithm

Section 2.3 was finished last, so a lot of further optimizations could've been made but weren't due to a lack of time. We are aware that the optimal way to implement this project was first to transform a matrix A into an upper Hessenberg matrix H, then to multiply on the right by $G^\star$ symmetrically to the left, creating a similarity transformation of A that we'll call $\hat{A}$ and then to compute the eigenvalues of $\hat{A}$ using the QR algorithm. This would've been a more efficient than the current implementation because the shifting to an upper Hessenberg matrix is not very costly when comparing to all the process that we did in section 4.

5.2 Conclusion

In conclusion, throughout our tests we saw that in general, the `double` float precision is faster than the `MPFR` one but as expected is not as precise. With the `MPFR` version, we were able to detect errors up to $1e^{-23}$ consistently and could've gone way more precise, but our infrastructure didn't permit it. Also, we chose to run our tests on symmetric matrices as they have the property of having real eigenvalues.

6 Annex

6.1 Double implementation

`void print_matrix(double **A,int m, int n);` Print the matrix `A` of size $m \times n$.
`void generate_G(double **G, int i, int j, int m, double** R);` Generate the matrix $G_{i,j}$.
`void transpose_matrix(double ** M, int m, int n, double*** Mt);` Transpose the matrix `M` of size $m \times n$ and store the result in `Mt`.
`void copy(double **A, double **B, int m, int n);` Deep copy the matrix `A` of size $m \times n$ into `B`.
`void free_matrix(double **A, int m);` Free the matrix `A` of size $m \times n$.
`void multiplymatrices(double **A, double **B, int n, double ***C);` Multiply the matrix `A` of size $n \times n$ by the matrix `B` of size $n \times n$ and store the result in `C`.
`void compute_eigenvalues3(double ***A, int m, int n, long* nb_iter);` Compute the eigenvalues of the matrix `A` of size $m \times n$ using the QR algorithm as presented in the subject.
`void compute_eigenvalues5(double ***A, int m, int n, long* nb_iter, FILE* f);` Compute the eigenvalues of the matrix `A` of size $m \times n$ using the QR algorithm as presented in section 5.1. Store the eigenvalues in the file `f`.
`void Givens3(double **A, int m, int n, double*** Q, double*** R);` Compute the QR factorization of the matrix `A` of size $m \times n$ using the Givens algorithm. Store the result in `Q` and `R`.
`void generate_random_matrix(double ***A, int m, int n, int a, int b);` Generate a random matrix of size $m \times n$ with random numbers between a and b.
`void generate_symmetric_matrix(double ***A, int m, int n, int a, int b);` Generate a random symmetric matrix of size $m \times n$ with random numbers between a and b.

6.2 MPFR implementation

`void generate_random_matrix_mpfr(mpfr_t *** matrix, int m, int n, int a, int b);` Generate a random matrix of size $m \times n$ with random numbers between a and b.
`void generate_symetric_matrix_mpfr(mpfr_t *** matrix, int m, int n, int a, int b);` Generate a random symmetric matrix of size $m \times n$ with random numbers between a and b.
`void print_mpfr_matrix(mpfr_t ** matrix, int m, int n);` Print a matrix of size $m \times n$.
`void multiply_mpfr_matrices(mpfr_t **A, mpfr_t **B, int m, int n, mpfr_t ***C);` Multiply two matrices of size $m \times n$.
`void generate_givens_matrix(mpfr_t *** G, int m, int i, int j, mpfr_t **R);` Generate a Givens matrix $G(i, j)$ of size $m \times m$.
`void free_mpfr_matrix(mpfr_t ** matrix, int m, int n);` Free a matrix of size $m \times n$.
`void transpose_mpfr_matrix(mpfr_t **M, int m, int n, mpfr_t ***Mt);` Transpose the matrix M of size $m \times n$ and store it in Mt.
`void copy_mpfr_matrix(mpfr_t **A, int m, int n, mpfr_t ***B);` Copy matrix `A` of size $m \times n$ into matrix `B`.
`void Givens_mpfr(mpfr_t ** A, int m, int n, mpfr_t*** Q, mpfr_t*** R);` Compute the QR factorization of matrix `A` of size $m \times n$ using the Givens method and store the result in `Q` and `R`.
`int thresh_mpfr(mpfr_t ** A, int m, int n, mpfr_t threshold);` Check if the subdiagonal of matrix `A` of size $m \times n$ is below the threshold `threshold`.
`void compute_eigenvalues_mpfr(mpfr_t*** A, int m, int n, int* nb_iter, mpfr_t threshold);` Compute the eigenvalues of matrix `A` of size $m \times n$ using the QR algorithm, store the number of iterations in `nb_iter`.
`void compute_eigenvalues_mpfr2(mpfr_t*** A, int m, int n, int* nb_iter, mpfr_t threshold);` Same as `compute_eigenvalues_mpfr` but using the alternative algorithm presented in Section 5.1.