Università di Pisa

Computer Science Master Degree

CM Project 21

Authors: Marco Mazzei and William Simoni

ACADEMIC YEAR 2020/2021

Chapter 1

Introduction

The problem consists in finding a low-rank approximation of a matrix $A \in \mathbb{R}^{m \times n}$, defined by the equation 1.1.

$$\min_{U \in \mathbb{R}^{m\times k}, V \in \mathbb{R}^{k \times n}} f(U, V) = \|A - UV\|_F \tag{1.1}$$

The approximation is computed using an alternating minimization procedure: choose $V = V_0$, compute $U_1 = \arg\min_U \|A - UV_0\|_F$, then use it to compute $V_1 = \arg\min_V \|A - U_1 V\|_F$, and so on until we find a sufficiently good approximation. We found the minimum transforming the problem in the following way:

$$\min_{U \in \mathbb{R}^{m\times k}, V \in \mathbb{R}^{k \times n}} \|A - UV\|_F^2 = \min_{U \in \mathbb{R}^{m\times k}, V \in \mathbb{R}^{k \times n}} tr((A - UV)^t (A - UV))$$

$$= \min_{U \in \mathbb{R}^{m\times k}, V \in \mathbb{R}^{k \times n}} tr(A^t A - A^t UV - V^t U^t A + V^t U^t UV)$$

$$=^{A.1} \min_{U \in \mathbb{R}^{m\times k}, V \in \mathbb{R}^{k \times n}} tr(A^t A) - tr(A^t UV) - tr(V^t U^t A) + tr(V^t U^t UV) \tag{1.2}$$

Deriving the formula 1.2 with respect to U:

$$\frac{\partial \|A - UV\|_F^2}{\partial U} = \frac{\partial\, tr(A^t A) - tr(A^t UV) - tr(V^t U^t A) + tr(V^t U^t UV)}{\partial U}$$

Deriving term by term:

$$\frac{\partial tr(A^t A)}{\partial U} = 0$$
$$\frac{\partial tr(A^t UV)}{\partial U} =^{A.2} \frac{\partial tr(VA^t U)}{\partial U} = (VA^t)^t = AV^t$$
$$\frac{\partial tr(V^t U^t A)}{\partial U} =^{A.2} \frac{\partial tr(U^t AV^t)}{\partial U} = AV^t$$
$$\frac{\partial tr(V^t U^t UV)}{\partial U} = U((VV^t)^t + (VV^t)) = U(VV^t + VV^t) = 2UVV^t$$

Putting all together:

$$\frac{\partial \left\|A - UV\right\|_F^2}{\partial U} = 2UVV^t - 2AV^t \tag{1.3}$$

With similar steps, we obtained the derivation with respect to V:

$$\frac{\partial \left\|A - UV\right\|_F^2}{\partial V} = 2U^tUV - 2U^tA \tag{1.4}$$

To find the minimum we set the derivatives (1.3 and 1.4) to 0. The fact 2 ensures that the point found is a minimum.

$$2UVV^t - 2AV^t = 0 \Leftrightarrow UVV^t = AV^t \tag{1.5}$$

$$2U^tUV - 2U^tA = 0 \Leftrightarrow U^tUV = U^tA \tag{1.6}$$

Substituting V^t with its thin QR factorization in the formula 1.5:

$$Q_1R_1 = V^t$$

$$\begin{aligned} UVV^t = AV^t &\Leftrightarrow UR_1^tQ_1^tQ_1R_1 = AQ_1R_1 \\ &\Leftrightarrow UR_1^tR_1 = AQ_1R_1 \\ &\Leftrightarrow^* UR_1^t = AQ_1 \\ &\Leftrightarrow R_1U^t = Q_1^tA^t \\ &= \underset{k\times k}{\mathrm{R}_1} \times \underset{k\times m}{\mathrm{U}^t} = \underset{k\times n}{\mathrm{Q}_1^t} \times \underset{n\times m}{\mathrm{A}^t} \end{aligned} \tag{1.7}$$

Where $\Leftrightarrow^*$ follows from fact 1 assuming $k \leq rank(A)$.

Substituting U with its thin QR factorization in the formula 1.6:

$$Q_1R_1 = U$$

$$\begin{aligned} U^tUV = U^tA &\Leftrightarrow R_1^tQ_1^tQ_1R_1V = R_1^tQ_1^tA \\ &\Leftrightarrow R_1^tR_1V = R_1^tQ_1^tA \\ &\Leftrightarrow^* R_1V = Q_1^tA \\ &= \underset{k\times k}{\mathrm{R}_1} \times \underset{k\times n}{V} = \underset{k\times m}{Q_1^t} \times \underset{m\times n}{A} \end{aligned} \tag{1.8}$$

Where $\Leftrightarrow^*$ follows from fact 1 assuming $k \leq rank(A)$.

Finally, 1.7 and 1.8 can be solved by computing first the right-hand side of the equations and then solving, via back substitution, an upper triangular system.

Fact 1. If $k \leq rank(A) \leq \min(m,n)$, then we will assume that $rank(V) = rank(U) = k$ in every iteration of the algorithm.

Proof. To prove it by induction we assume that V_0 is initialized at random.
Base case:

- V_0 has rank k since $k \leq n$;

- In the computation of U_1, doing the thin QR factorization of V_0^t, R_1 is full column rank, so it is invertible. Therefore U_1 has rank k because from formula 1.7, $rank(U_1^t) = rank(R_1^{-1}Q_1^t A^t) =^* rank(A^t) = k$.
- In the computation of V_1, doing the thin QR factorization of U_1, R_1 is full column rank, so it is invertible. Therefore V_1 has rank k because from formula 1.8, $rank(V_1) = rank(R_1^{-1}Q_1^t A) =^* rank(A) = k$.

Where $=^*$ is correct almost all the time because V_0 is initialized at random.

Inductive case: computing U_i, by inductive hypothesis $rank(V_{i-1}) = k$. So, making the same steps already explained, $rank(U_i) = k$. The same holds for V_i. □

Fact 2. If $k \leq rank(A)$, then the function 1.1 is strictly convex in U only or V only. So, f is biconvex.

Proof. The second derivative of 1.1 with respect to U is VV^t. If $k \leq rank(A)$, then from fact 1, $rank(VV^t) = rank(V) = k$. Therefore, VV^t is positive definite because V^t is full column rank. The same can be done deriving with respect to V concluding that U^tU is positive definite. Thus, the problem is strictly convex. □

Chapter 2

Algorithm's Properties

For the rest of this report, we will consider the following assumptions:

Assumption 1. $k \leq rank(A)$ (see fact 1.).

Assumption 2. $\forall i$, every subproblem:

$$U_{i+1} = \arg\min_{U} \|A - UV_i\|_F \tag{2.1}$$

$$V_{i+1} = \arg\min_{V} \|A - U_{i+1}V\|_F \tag{2.2}$$

has one unique solution. This assumption holds if $\forall i$ U_i and V_i have rank k and, as a consequence, the subproblems are strictly convex (as fact 2. assures).

Assumption 2. is very likely to be correct, but unfortunately, there could be cases in which, from an iteration to another, the rank becomes smaller than k.

2.1 Algorithm pseudocode

Code 2.1 shows the pseudocode of the algorithm, called RESETQR[1], described in the previous chapter.

Code 2.1: pseudocode of RESETQR

```
function RESETQR(A, k, min_error, max_iterations)
    Initialize V and U at random
    num_iterations = 0
    do
        f_old = f(U,V)       //compute the function value
        Q,R = qr(V')         //compute thin QR factorization of V'
        R * U' = Q' * A'     //solve linear system to find U
        Q,R = qr(U)          //compute thin QR factorization of U
        R * V = Q' * A       //solve linear system to find V
        num_iterations += 1
    while (f_old - f(U,V) > min_error) OR (num_iterations <
       max_iterations)
    return U,V
```

The algorithm stops whenever:

[1] alte**R**nating l**E**ast **S**quare solv**E**d wi**T**h **QR**

- The function value does not change too much: $f\_old - f(U,V) \leq$ `min_error`. Fact 4 guarantees that after an arbitrary number of steps, the algorithm will surely reach this condition.
- It has executed `max_iterations` iterations.

2.2 Convergence

Fact 3. The sequence $\{f(z_i)\}_{i\in\mathbb{N}}$ generated by the algorithm is monotonically decreasing.

Proof. Intuitively, at every step, we compute an approximation that is at least good as the previous one:

- U_0 and V_0 are initialized at random and $f(U_0,V_0) = \|A - U_0V_0\|_F$.
- $U_1 =^* \arg\min_U \|A - UV_0\|_F$, so $f(U_1,V_0) = \|A - U_1V_0\|_F \leq \|A - U_0V_0\|_F$. They are equal only if $U_1 = U_0$.
- $V_1 =^* \arg\min_V \|A - U_1V\|_F$, so $f(U_1,V_1) = \|A - U_1V_1\|_F \leq \|A - U_1V_0\|_F$. They are equal only if $V_1 = V_0$. Therefore, $f(U_1,V_1) \leq f(U_0,V_0)$

$=^*$ because of assumption 2.
So, in general we have that:

$$f(U_{i+1},V_{i+1}) \leq f(U_{i+1},V_i) \leq f(U_i,V_i)) \tag{2.3}$$

□

Fact 4. The sequence $\{f(z_i)\}_{i\in\mathbb{N}}$ generated by the algorithm converges monotonically.

Proof. As shown above, the sequence $\{f(z_i)\}_{i\in\mathbb{N}}$ is monotonically decreasing and f is bounded below ($f \geq 0$). Then, the sequence $\{f(z_i)\}_{i\in\mathbb{N}}$ generated by the algorithm converges monotonically. □

Fact 5. Given $f : D \to R$, where D is a convex set in R^n, if f is convex, then it is easy to prove that f is also quasi-convex.

Since the two subproblems 2.1 and 2.2 are strictly convex (under assumption 2), they are also quasi-convex.

Fact 6. Given $f : D \to R$, where D is a convex set in R^n, if f is strictly convex, then it is also strictly quasi-convex.

Proof. Looking at the definitions (definitions 1 and 2 in Appendix A), it is straightforward to see that a strictly convex function f is also convex. We recall the convexity definition (see also definition 1 in Appendix A):

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \quad (x,y \in D)(\lambda \in [0,1]) \tag{2.4}$$

Assuming that $f(x) > f(y)$, the right-hand side in inequality 2.4 is maximum when $\lambda = 1$, and the maximum is equal to $f(x)$. Therefore, for every $\lambda \in (0,1)$, we have that:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) < f(x) \tag{2.5}$$

So, we proved that for every x and y in D and for every $\lambda \in (0,1)$:

$$\text{if } f(x) > f(y) \text{ then } f(\lambda x + (1-\lambda)y) < f(x)$$

That is the definition of strictly quasi convexity (4). Therefore, f is also strictly quasi-convex. □

Theorem 1 (Grippo and Sciandrone [2]). If f is differentiable and strictly quasi-convex over each block, then the limit point is a stationary point.

Since f is biconvex, the stationary point may be either a saddle point or a local minimum. Without more assumptions, it is not possible to prove that the stationary point is a global/local minima.

The algorithm 2.1 does not guarantee that there will be a limit point. But we could slightly change the updating rules in such a way that the generated sequence lies into a compact set to ensure the limit point existence. To be compact, a set must be bounded and closed. The next section will describe a possible solution.

2.3 CRESETQR

At the end of step i, the CRESETQR[2] changes the computed U_{i+1} and V_{i+1} matrices as follows:

$$Q_1, R_1 = qr(U_{i+1}) \tag{2.6}$$
$$\hat{U}_{i+1} = Q_1 \tag{2.7}$$
$$\hat{V}_{i+1} = R_1 V \tag{2.8}$$

And at iteration $i+1$, instead of using U_{i+1} and V_{i+1}, it uses $\hat{U}_{i+1}$ and $\hat{V}_{i+1}$. Code 2.2 depicts the two additional operations.

Code 2.2: pseudocode of CRESETQR

```
function RESETQR(A, k, min_error, max_iterations)
    Initialize V and U at random
    num_iterations = 0
    do
        f_old = f(U,V)       //compute the function value
        Q,R = qr(V')         //compute thin QR factorization of V'
        R * U' = Q' * A'     //solve linear system to find U
        Q,R = qr(U)          //compute thin QR factorization of U
        R * V = Q' * A       //solve linear system to find V

        //additional steps
        U = Q
        V = R * V

        num_iterations += 1
    while (f_old - f(U,V) > min_error) OR (num_iterations <
       max_iterations)
    return U,V
```

[2]Compact RESETQR

Fact 7. Under the assumptions we have stated at the beginning of the chapter (assumptions 1 and 2), the sequence $z_i = \{(\hat{U}_i, \hat{V}_i)\}_{i \in N}$ generated by the CRESETQR algorithm lies into a compact set.

Proof. We will prove that z_i is compact by proving that the norms of U_i and V_i are bounded for every step i:

- $\|\hat{U}_i\|_F$ is constant for every i because U_i is equal to the Q_1 factor of the thin QR factorization of U_i (2.7), which is an orthogonal matrix.
- Since $\hat{U}_i$ is orthogonal, we have that $\|\hat{V}_i\|_F = \|\hat{U}_i\hat{V}_i\|_F \, \forall i$. Moreover, from fact 3 we know that:

  $$0 \le \|A - \hat{U}_i\hat{V}_i\|_F \le \|A - \hat{U}_1\hat{V}_1\|_F \ \ \forall i$$

  Using the triangle inequality:

  $$\|\hat{U}_i\hat{V}_i\|_F - \|A\|_F \le \|\hat{U}_i\hat{V}_i - A\|_F = \|A - \hat{U}_i\hat{V}_i\|_F \le \|A - \hat{U}_1\hat{V}_1\|_F \ \ \forall i \tag{2.9}$$

  Then, summing $\|A\|_F$ over all the terms in 2.9:

  $$\|\hat{U}_i\hat{V}_i\|_F \le \|\hat{U}_i\hat{V}_i - A\|_F + \|A\|_F \le \|A - \hat{U}_1\hat{V}_1\|_F + \|A\|_F \ \ \forall i \tag{2.10}$$

  Therefore $\|\hat{V}_i\|_F$ is bounded by $\|A - \hat{U}_1\hat{V}_1\|_F + \|A\|_F \ \ \forall i$.

□

2.4 Complexity

Every iteration of the RESETQR algorithm consists of four steps:

Calculate the QR factorization of $V^t \in \mathbb{R}^{n \times k}$, using Householder reflectors, that costs $2nk^2 + \frac{2}{3}k^3 + O(nk) = O(nk^2)$

Solve the upper triangular linear system $RU^t = Q^tA^t$:

- calculate $W = Q^tA^t$ $(O(kmn))$
- solve $RU^t = W$ $(O(mk^2))$

that costs $O(kmn)$

Calculate the QR factorization of $U \in \mathbb{R}^{m \times k}$, using Householder reflectors, that costs $2mk^2 + \frac{2}{3}k^3 + O(mk) = O(mk^2)$

Solve the upper triangular linear system $RV = Q^tA$:

- calculate $W = Q^tA$ $(O(kmn))$
- solve $RV = W$ $(O(nk^2))$

that costs $O(kmn)$

Thus, the total cost per iteration is:

- $O(k^2 \times max\{m, n\})$ if $k = \Theta(\min\{m, n\})$.
- $O(kmn)$ if $k << min\{m, n\}$.

CRESETQR adds another step, which requires $O(nk^2)$, and therefore it has the same asymptotic complexity of RESETQR.

2.5 Other properties

- QR factorization is a numerical stable algorithm. Also back-substitution, using the R factor of the QR factorization, is numerical stable. Hence, every iteration is numerical stable.

Chapter 3

Algorithm Implementation

This chapter documents a Julia implementation of the algorithm described in the previous chapters. The implementation consists of three modules:

- **resetqr.jl** that is the implementation of the main algorithm.
- **QR_factorization.jl** that contains the implementation of the thin QR factorization. Beside the QR factorization, that returns the factor R and the reflectors v, there is a method to allocate the memory for the matrices involved in the factorization, and a method that given a matrix A and the reflectors v, returns Q*A.
- **back_substitution.jl** that implements the back-substitution algorithm.

3.1 Implementation choices

To reduce the space occupied by the QR factorization of the matrix $A \in \mathbb{R}^{m \times n}$, we decided to store both the factor R and the reflectors v in the same matrix $QR \in \mathbb{R}^{m+1 \times n}$, where:

- the upper triangular part contains R. Therefore, `R = triu(QR)[1:n,:]`
- the sub-diagonal part contains the reflectors v. Hence, `V = tril(QR,-1)[2:n+1,:]`

The matrix 3.1 shows the structure of the QR matrix when $A \in \mathbb{R}^{7 \times 6}$.

$$\begin{bmatrix} R_{1,1} & R_{1,2} & R_{1,3} & R_{1,4} & R_{1,5} & R_{1,6} \\ v_{1,1} & R_{2,2} & R_{2,3} & R_{2,4} & R_{2,5} & R_{2,6} \\ v_{1,2} & v_{2,1} & R_{3,3} & R_{3,4} & R_{3,5} & R_{3,6} \\ v_{1,3} & v_{2,2} & v_{3,1} & R_{4,4} & R_{4,5} & R_{4,6} \\ v_{1,4} & v_{2,3} & v_{3,2} & v_{4,1} & R_{5,5} & R_{5,6} \\ v_{1,5} & v_{2,4} & v_{3,3} & v_{4,2} & v_{5,1} & R_{6,6} \\ v_{1,6} & v_{2,5} & v_{3,4} & v_{4,3} & v_{5,2} & v_{6,1} \\ v_{1,7} & v_{2,6} & v_{3,5} & v_{4,4} & v_{5,3} & v_{6,2} \end{bmatrix} \tag{3.1}$$

Julia automatically performs bounds checking to ensure program safety when accessing arrays. In order to improve the efficiency, and considering the fact that the implementation is specifically made for the `resetqr` execution, we use the `@inbounds` macro to avoid this behaviour.

For the same reasons, and to reduce the space allocated during the execution, the QR factorization requires matrices and vectors already allocated. Therefore, also the QR matrix is allocated at the beginning of the `resetqr` algorithm. These allocations are performed by a method in `qr_factorization.jl`.

3.2 How to execute

You need Julia installed in your PC. Then:

1. Open the Julia console.
2. Exec `include("resetqr.jl")` for RESETQR and `include("cresetqr.jl")` for CRESETQR.
3. Create a matrix A for which you want to find the UV decomposition. For instance, `A = rand(10,20);`.
4. Call the `resetqr/cresetqr` function with the following parameters:
   - **A**: matrix
   - **k**: the rank of the UV decomposition.
   - **initV**: V_0 ([optional] default value is `rand(k, n)`).
   - **min_error**: the algorithm will stop if $\|f_{old} - f\|_F \leq min\_error$ ([optional] default value is $1e-8$).
   - **max_iterations**: the algorithm will perform at most max_iteration steps ([optional] default value is 100).
   - **print_error**: if it is true, the algorithm will print the error at every iteration ([optional] default value is *true*).
   - **save_errors**: if it is true, the algorithm will return an array containing the history of the errors, besides U and V ([optional] default value is *false*).

   The function returns two matrices that represent a UV decomposition of rank k of the matrix A. For example:

```
U,V=resetqr(A, 5, min_error=1e-12, max_iterations=500)

step      ||A - UV||_f     ||f_old-f||
1)       2.098604347324  5.852418782100
2)       1.866288188078  0.232316159246
3)       1.794141049717  0.072147138361
4)       1.757460260626  0.036680789091
...
58)      1.7247659213348 0.0000000000026
```

```
59)     1.7247659213330 0.0000000000018
60)     1.7247659213318 0.0000000000012
61)     1.7247659213310 0.0000000000008
Reached good approximation with 61 iterations
```

3.3 Correctness test

To assure that the algorithm will work as expected, we have tested that every function implemented by us returns approximately the same results to corresponding Julia functions.

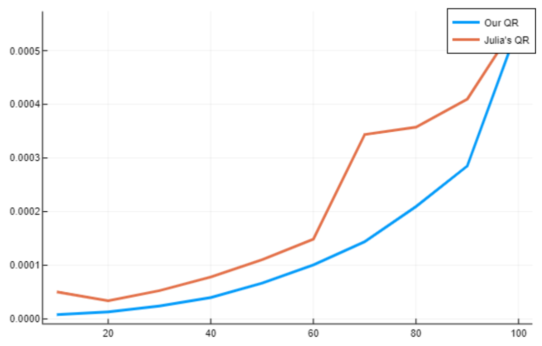
`test_qr.jl` tests the qr_factorization functions:

- `test_qr_A(num_tests)` verifies `num_tests` times that $\|Q * R - A\| < 1e-13$ by generating, for each test, a matrix A with random shape (max 100×100 and min 10×10) and random rank.

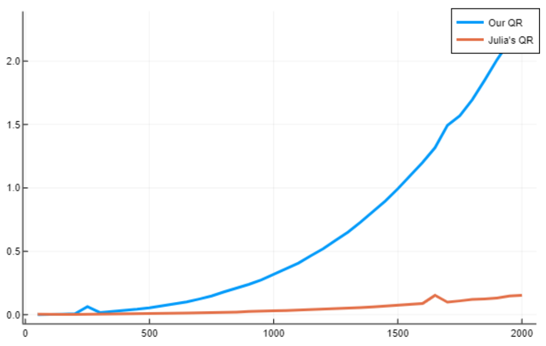
- `test_qr_R(num_tests)` checks `num_tests` times whether the R factor returned by our implementation is the same returned by the Julia QR factorization (allowing an error of at most $1e-10$).

- `test_qr_multiplication(num_tests)` tests `num_tests` times the multiplication between Q^t and a matrix of random size (max 100×100) and random rank (allowing an error of at most $1e-10$).

- `test_time_qr(min_dimension, max_dimension; step = 15)` draws a plot that compares the time (in seconds) required by our implementation and the one built-in in Julia. For simplicity, the matrix has a square shape in every iteration. At step k, the matrix shape will be $(min\_dimension + ((k-1) * step) \times min\_dimension + ((k-1) * step))$. The algorithm terminates when the matrix shapes reach `max_dimension`. Figure 3.1 shows one run of the function.



(a) Test with min dimension 10, max dimension 100 and step 10



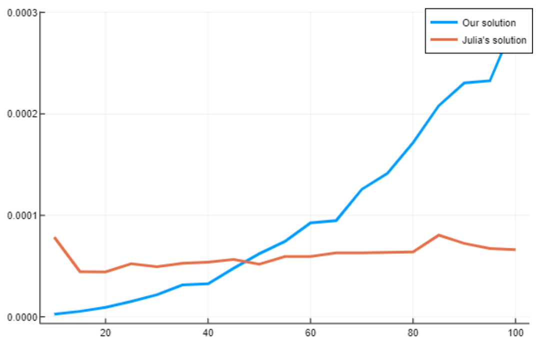
(b) Test with min dimension 50, max dimension 2000 and step 50

Figure 3.1: Time comparison between our implementation and the one built-in in Julia. We performed the test on a PC with CPU i7-7700k and 16 Gb of RAM. For small matrices (roughly up to 100x100) performs better than the Julias's one. As expected, our implementation asymptotically behaves worse than the Julias's one

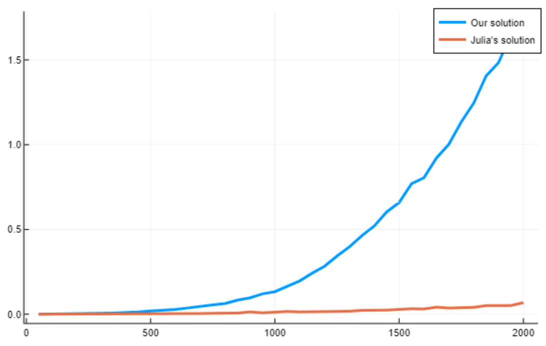
`test_back_substitution.jl` tests the back substitution functions:

- `test_back_sub(num_tests)` verifies the correctness of our back-substitution algorithm `num_tests` times (allowing an error of at most $1e-13$). Each test uses a randomly generated matrix.

- `test_time_back_sub(min_dimension, max_dimension; step = 15)` does the same of `test_time_qr)` for the back_substitution algorithm. Figure 3.2 exposes one run of the function.



(a) Test with min dimension 10, max dimension 100 and step 10



(b) Test with min dimension 50, max dimension 2000 and step 50

Figure 3.2: Time comparison between our implementation and the one built-in in Julia. We performed the test on a PC with CPU i7-7700k and 16 Gb of RAM. For small matrices (roughly up to 50x50) performs better than the Julias's one. As expected, our implementation asymptotically behaves worse than the Julias's one

3.4 Complexity test

In this section, we compare the actual floating-point operations (FLOPS) with the expected ones. We have measured the number of floating-point operations using GFlops.jl[1].

The following table shows the number of FLOPS for QR and back-substitution with some matrix shapes:

Shape	QR	Ideal
500 x 500	125.249.499	125.000.000
1000 x 1000	1.000.998.999	1.000.000.000
2000 x 100	29.415.000	20.000.000
5000 x 100	74.265.000	50.000.000

[1]https://github.com/triscale-innov/GFlops.jl

Shape R	Shape V and W	Back-substitution	Ideal
100 x 100	100 x 500	5.000.000	5.000.000
200 x 200	200 x 1000	40.000.000	40.000.000
30 x 30	30 x 2000	1.800.000	1.800.000

3.5 Convergence test

In this section, we check whether the algorithms converge towards a critical point and in case how fast they reach it. We compute the optimal solution using the following function:

```
function approx_matrix_svd(A,k)
    (m,n) = size(A)
    F = svd(A)
    A_SVD = F.U[1:m,1:k] * Diagonal(F.S)[1:k,1:k] * F.V'[1:k,1:n]
    return A_SVD
end
```

That using the truncated SVD finds the optimum (as stated by the Eckart–Young–Mirsky Theorem [1]).

We evaluate the convergence towards the optimum, measuring, at every step, the relative error between $U_i \times V_i$ and the optimal solution:

$$relative\_error\_x(i) = \frac{\|U_i V_i - A\_SVD\|_F}{\|A\_SVD\|_F} \tag{3.2}$$

Where A_SVD is the optimal solution computed by the function approx_matrix_svd(A).

And measuring, at every step, the relative error between $f^i = \|A - U_i V_i\|_F$ and $f^* = \|A - A\_SVD\|_F$:

$$relative\_error\_f(i) = \frac{f^i - f^*}{f^*} \tag{3.3}$$

In addition to 3.2 and 3.3, we compute the **order of convergence** to evaluate how fast the algorithm converges to A_SVD.

$$R(i) = \frac{\|A - U_i V_i\|_F - \|A - A\_SVD\|_F}{\|(A - U_{i-1} V_{i-1}\|_F - \|A - A\_SVD\|_F)^p} = \frac{f^i - f^*}{(f^{i-1} - f^*)^p} \tag{3.4}$$

Figure 3.3 exhibits the rate of convergence with $p = 2$ for a 500×1000 random matrix using $k = 25$. The plot indicates that the algorithms do not converge quadratically since the rate of convergence is clearly diverging.

Figure 3.4 shows formula 3.2 and 3.3 values in 3221 iterations (the algorithm stopped once $f^{i-1} - f^i <= 1e - 16$) for a 500×1000 random matrix using $k = 50$. We can notice that:

- The algorithm does not converge exactly to A_SVD since we reach a minimum residual error of $2.330177e - 6$. Therefore the algorithm does not reach the global optimum.

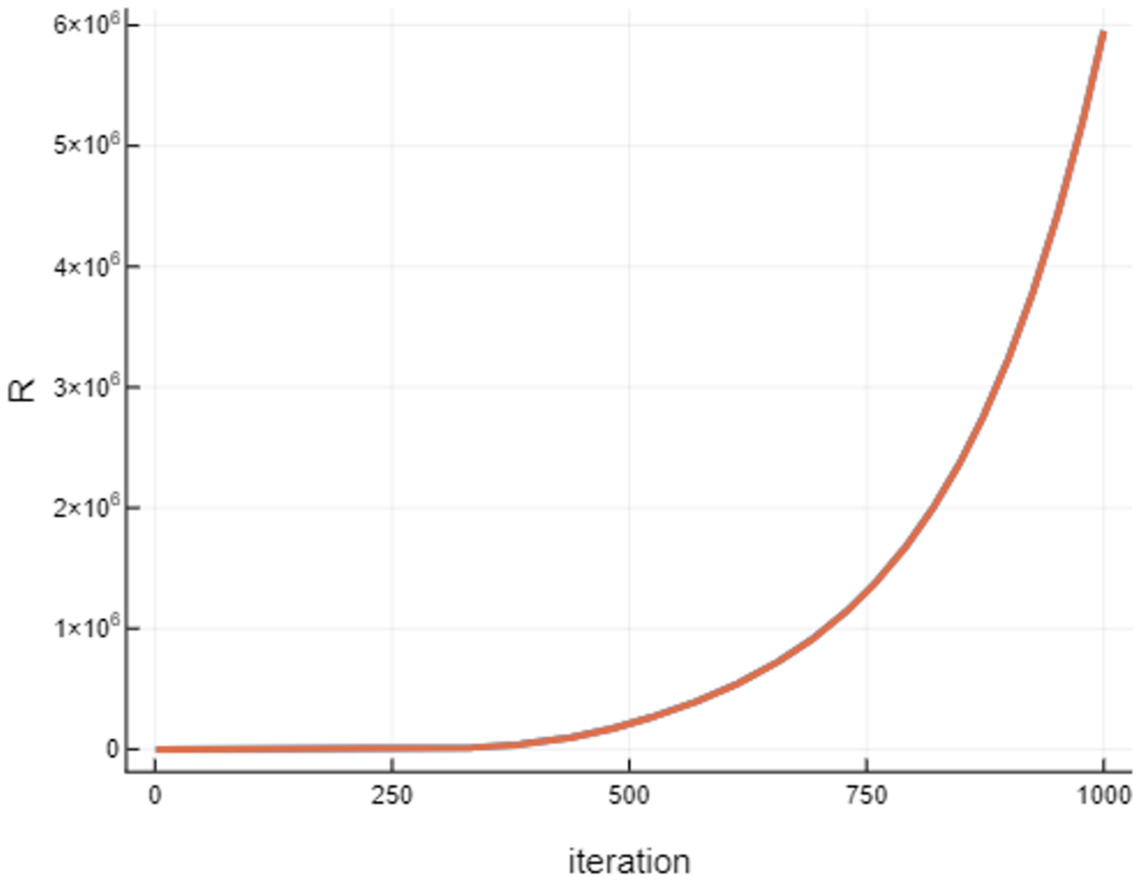


Figure 3.3: Rate of convergence with $p = 2$ computed in one trial with random 500×1000 matrix and $k = 25$

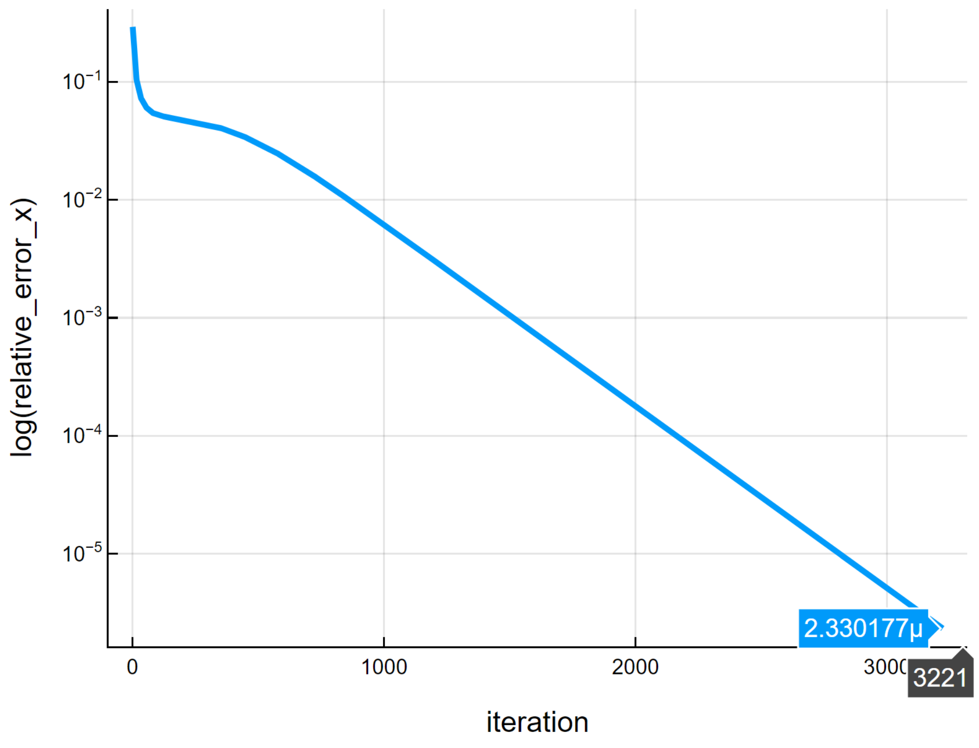
- On the other hand, 3.3 converges towards 0 reaching a minimum value of $2.059445e - 14$. Therefore, even though the final solution is not too close to A_SVD, the algorithm reaches a good approximation of A with rank k. From the previous point, and from Theorem 1, (U_{3221}, V_{3221}) is a saddle point or a local minima.

From Figure 3.5 it is also clear that the rate of convergence is converging towards 1 or towards a number close to 1. Therefore we can conclude that the algorithm converged linearly (with R very close to 1) or sub-linearly.

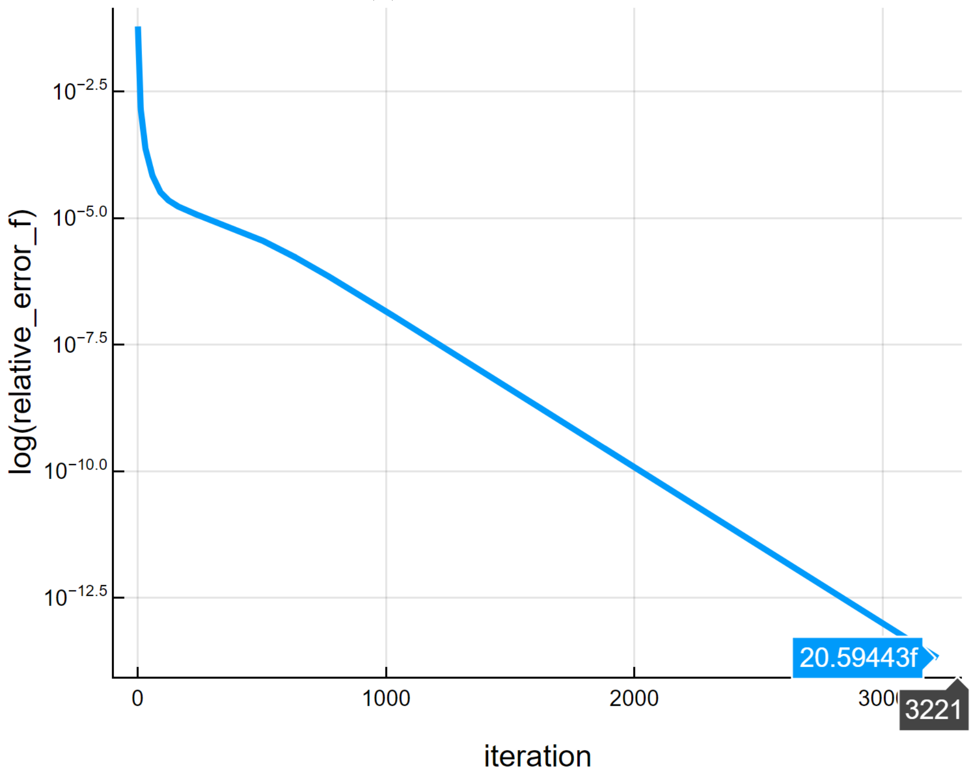
Figure 3.8 shows indeed that the ratio between the numerator and the denominator of 3.4 is almost 1 at every iteration.

We have done these tests dozen of times using random matrices of random size and random density. Figures 3.6 and 3.7 show the rate of convergence computed in 10 trials with matrices of random shape, scale and density, with $p = 1$.

We stop the executions at the 1000th iteration because after that number of iterations the denominator of 3.4 becomes very close to 0.



(a) relative error 3.2



(b) relative error 3.3

Figure 3.4: Convergence plots for CRESETQR starting from same V_0 and with $k = 50$

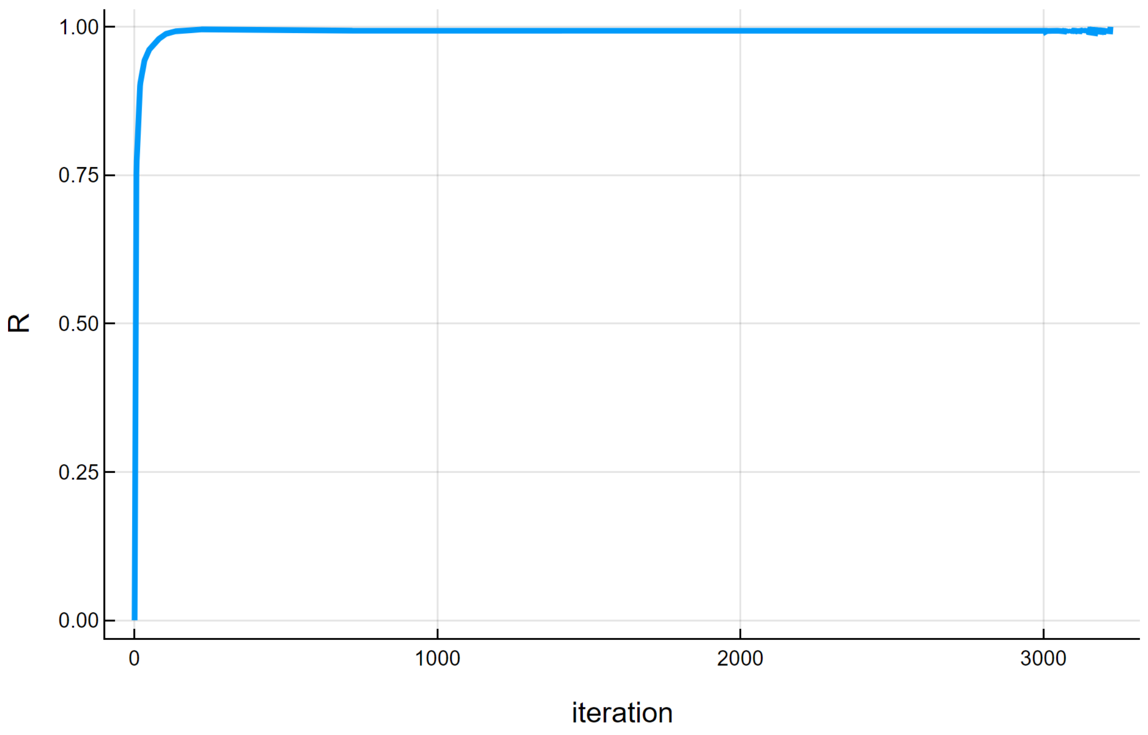


Figure 3.5: Rate of convergence with $p = 1$ computed in one trial (same execution of Figure 3.4)

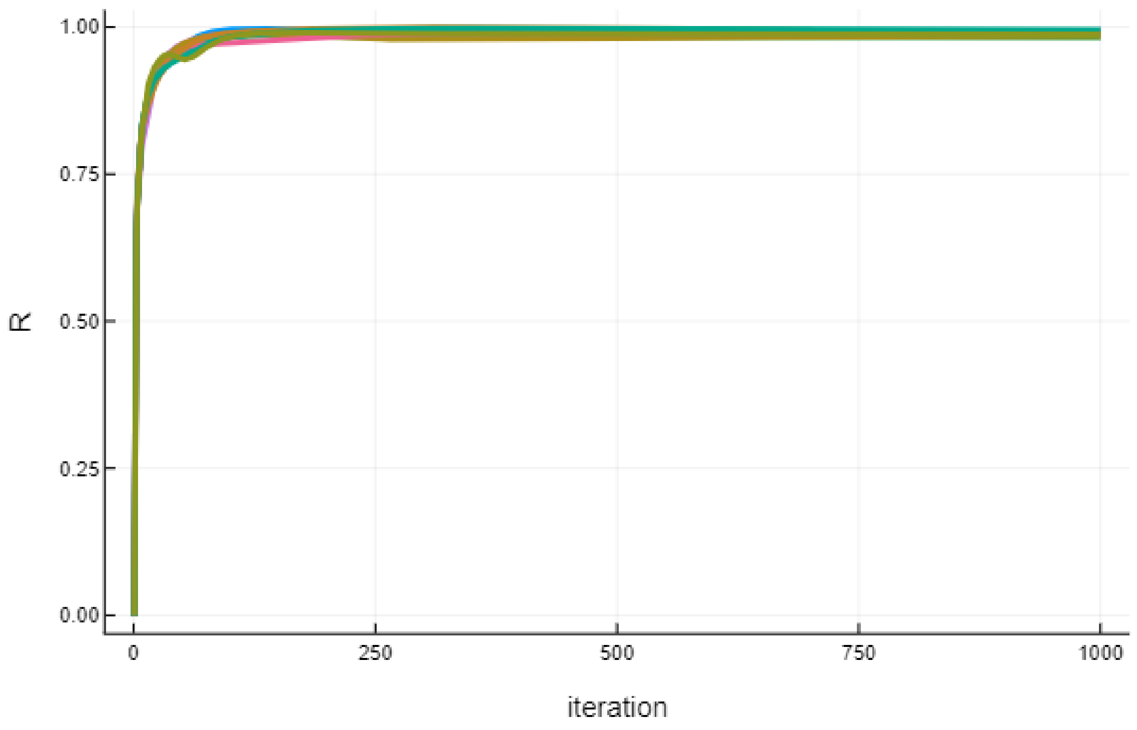


Figure 3.6: Rate of convergence computed in 10 trials with matrices of random shape, scale and density, with $p = 1$ and $k = 25$

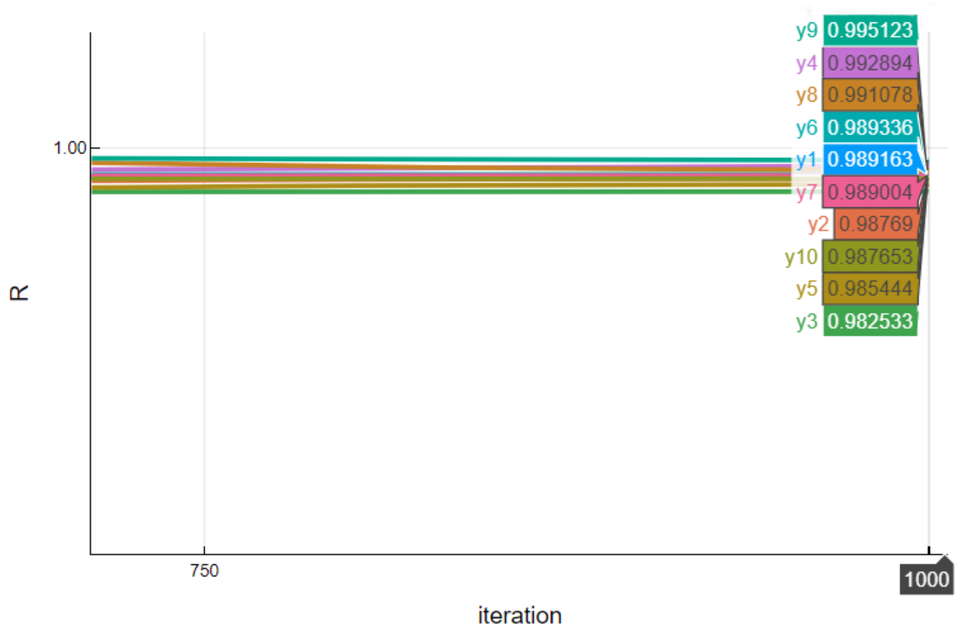


Figure 3.7: Rate of convergence computed in 10 trials with matrices of random shape, scale and density, with $p = 1$ and $k = 25$

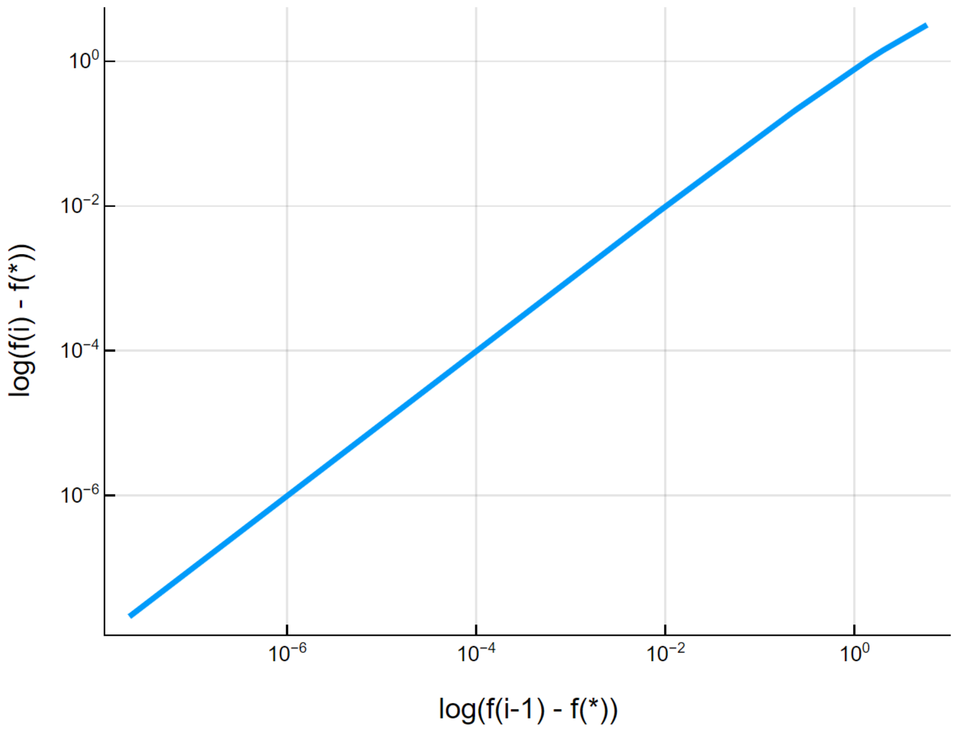


Figure 3.8: On the y axis the values of the numerator of 3.4 at a step i, on the x axis the value of the denominator at the same step. Values are computed in 1000 iterations using a 500×1000 random matrix with $k = 25$.

Appendix A

Appendix

A.1 Traces

Trace of a sum:

$$tr(A \pm B) = tr(A) \pm tr(B) \tag{A.1}$$

Cyclic permutation:

$$tr(ABC) = tr(CAB) = tr(BCA) \tag{A.2}$$

A.2 Convexity

Definition 1 (Convexity)**.** The function $f : D \to R$, where D is a convex set in R^m and R denotes the real line, is *convex* on D provided that for every x and y in D and for every $\lambda \in [0, 1]$:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

Definition 2 (Strictly convexity)**.** The function $f : D \to R$, where D is a convex set in R^m and R denotes the real line, is *strictly convex* on D provided that for every x and y in D and for every $\lambda \in (0, 1)$:

$$f(\lambda x + (1 - \lambda)y) < \lambda f(x) + (1 - \lambda)f(y)$$

Definition 3 (Quasi-convexity)**.** The function $f : D \to R$, where D is a convex set in R^m and R denotes the real line, is *quasi-convex* on D provided that for every x and y in D and for every $\lambda \in [0, 1]$:

$$f(\lambda x + (1 - \lambda)y) \leq \max(f(x), f(y))$$

Definition 4 (Strictly quasi-convexity)**.** The function $f : D \to R$, where D is a convex set in R^m and R denotes the real line, is *strictly quasi-convex* on D provided that for every x and y in D and for every $\lambda \in (0, 1)$:

$$f(y) < f(x) \text{ implies } f(\lambda x + (1 - \lambda)y) < f(x)$$

Bibliography

[1] Carl Eckart and Gale Young. “The approximation of one matrix by another of lower rank”. In: *Psychometrika* 1.3 (1936), pp. 211–218.

[2] Luigi Grippo and Marco Sciandrone. “On the convergence of the block nonlinear Gauss–Seidel method under convex constraints”. In: *Operations research letters* 26.3 (2000), pp. 127–136.