Project 1: Orthogonalization techniques for a set of vectors

Stefano Ferioli

October 31, 2023

Introduction

The aim of this project is to implement and analyze three algorithms for the orthonormalization of a set of vectors.
The first section will formalize the definition of QR factorization. The second and third paragraphs will respectively introduce the Classical Gram-Schmidt and Cholesky QR algorithms, considering both their sequential and parallel implementations. The fourth section will present Tall-Skinny QR, initially employing a flat tree for sequential execution and then moving to a binary tree for parallel implementation. The fifth section will examine the numerical stability of the three methods. Lastly, the sixth section will present the runtimes achieved by these methods both in sequential and parallel implementations, while also varying the size of the matrices.

1 QR factorization

Consider a matrix $W \in \mathbb{R}^{m\times n}$ with $m \geq n$. We say that W admits QR factorization if there exist $\tilde{Q} \in \mathbb{R}^{m\times m}$ orthogonal and $\tilde{R} \in \mathbb{R}^{m\times n}$ upper triangular such that $W = \tilde{Q}\tilde{R}$.
Since

$$W = \tilde{Q}\tilde{R} = \begin{pmatrix} Q & \bar{Q} \end{pmatrix} \begin{pmatrix} R \\ \bar{R} \end{pmatrix} = \begin{pmatrix} Q & \bar{Q} \end{pmatrix} \begin{pmatrix} R \\ 0 \end{pmatrix} = QR,$$

we define thin QR factorization $W = QR$, where $Q \in \mathbb{R}^{m\times n}$ orthogonal and $R \in \mathbb{R}^{n\times n}$ upper triangular.
Indeed, finding a QR factorization of a matrix W is the same as orthonormalizing the columns of W. By multiplying Q and R, we see how the columns of W are linear combinations of the columns of Q.

$$\begin{aligned}
W_{:1} &= R_{11}Q_{:1} \\
W_{:2} &= R_{12}Q_{:1} + R_{22}Q_{:2} \\
W_{:3} &= R_{13}Q_{:1} + R_{23}Q_{:2} + R_{33}Q_{:3} \\
&\vdots \\
W_{:n} &= R_{1n}Q_{:1} + R_{2n}Q_{:2} + \cdots + R_{nn}Q_{:n}
\end{aligned}$$

If W has full rank the QR factorization is unique, except for the signs of the elements of the columns of Q and the rows of R. Given $W = QR$, we can obtain an equivalent QR factorization by inverting the signs of $Q_{:i}$ and $R_{i:}$.
The following sections will present three algorithms to numerically compute the QR factorization in the special case when $W \in \mathbb{R}^{m\times n}$ is a tall and skinny matrix, that is, when its height is significantly greater than its width ($m \gg n$).

2 Classical Gram-Schmidt

The first algorithm we analyze is Classical Gram-Schmidt (CGS).
Consider a set of linearly independent vectors $\{v_1, v_2, ..., v_n\}$, the algorithm generates an orthonormal basis $\{u_1, u_2, ..., u_n\}$ of the linear space $\mathrm{span}\,(v_1, v_2, ..., v_n)$. The unit vectors of the orthonormal basis are generated one by one. The i-th unit vector is obtained by subtracting from v_i its components along the directions defined by the previously generated vectors $\{u_1, u_2, ..., u_{i-1}\}$, and then normalizing. CGS can be schematized as follows.

Algorithm 1 Classic Gram-Schmidt

for $i = 1, ..., n$ **do**
  for $j = 1, ..., n-1$ **do**
    $r_{ji} = \langle u_j, v_i \rangle$
  end for
  $x_i = v_i - \sum_{j=1}^{i-1} r_{ji} u_j$
  $r_{ii} = \|x_i\|$
  $u_i = \frac{x_i}{\|x_i\|}$
end for

Where the operator $\langle \cdot, \cdot \rangle$ denotes the standard scalar product.
We are interested in this algorithm taking the vectors as the columns of a full rank matrix $W \in \mathbb{R}^{m \times n}$. In this case, the resulting orthonormal basis can be arranged in another matrix $Q \in \mathbb{R}^{m \times n}$ orthogonal by construction. In addition, the coefficients r_{ij} will define an upper triangular matrix $R \in \mathbb{R}^{n \times n}$ (we set $r_{ij} = 0$ for $i > j$). Also by construction, $QR = W$.
A possible sequential implementation of CGS in Python is the following. Both the inner cycle and the sum have been replaced by matrix multiplications.

Code 1: sequential CGS

```
1  import numpy as np
2  from numpy.linalg import norm
3
4  # Define the matrix W here
5  W = ...
6
7  (m,n) = W.shape
8
9  Q = np.empty((m,n))
10 R = np.zeros((n,n))
11
12 for i in range(n):
13     R[0:i,i] = Q[:,0:i].T @ W[:,i]
14     Q[:,i] = W[:,i] - Q[:,0:i] @ R[0:i,i]
15     qi_norm = norm(Q[:,i])
16     Q[:,i] = Q[:,i]/qi_norm
17     R[i,i] = qi_norm
```

We want now to parallelize the algorithm.
We distribute the matrix W among the s processors by partitioning it into s row blocks, so that each processor owns a block W_r of W. Similarly, every processor will be responsible for generating a row block of Q, Q_r. R will be stored only in processor 0. We assume without loss of generality that m is divisible by s. Indeed, if m is not divisible by s, we can add $s\left(\lceil \frac{m}{s} \rceil - 1\right)$ lines of zeros to W and refer us back to the previous case. For the sake of simplicity, from now on we assume to have only 4 processors, nevertheless, the same reasoning can be generalized to any number of processors.
The matrix multiplication at **code 1 line 13** can be written using blocks as follows.

$$Q[:,0:i]^\top W[:,i] = \begin{bmatrix} Q_1[:,0:i] \\ Q_2[:,0:i] \\ Q_3[:,0:i] \\ Q_4[:,0:i] \end{bmatrix}^\top \begin{bmatrix} W_1[:,i] \\ W_2[:,i] \\ W_3[:,i] \\ W_4[:,i] \end{bmatrix} = \begin{bmatrix} Q_1[:,0:i]^\top & Q_2[:,0:i]^\top & Q_3[:,0:i]^\top & Q_4[:,0:i]^\top \end{bmatrix} \begin{bmatrix} W_1[:,i] \\ W_2[:,i] \\ W_3[:,i] \\ W_4[:,i] \end{bmatrix} =$$

$$= \sum_{r=1}^{4} Q_r[:,0:i]^\top W_r[:,i]$$

Each processor executes one product $Q_r[:, 0:i]^\top W_r[:,i]$ and the results are summed together using `comm.Allreduce`. This allows us to execute the multiplication quicker (especially for higher values of i), and to still have the result stored in all the processors (we need to store this result only for one iteration, except for processor 0).
The operation at **code 1 line 14** can be performed without requiring additional communication among the processors ($R[0:i,i]$ is already stored in all the processors).
To compute the norm of $Q[:,i]$ at **code 1 line 15**, the processors need to communicate again. Each processor computes the norm of $Q_r[:,i]$ squared, the results are then summed together using `comm.allreduce`. Computing the square root, we obtain the norm of $Q[:,i]$ stored in all the processors.
The instructions at **code 1 line 16** and **code 1 line 17** can be executed without requiring additional communication.
Lastly, we assemble Q using `comm.Gatherv` and storing it in processor 0.

A possible implementation of parallel CGS in Python is the following. Only the most important lines are presented here, the complete code can be found in Appendix A.

Code 2: parallel CGS

```
m_local = m//s
W_local = np.empty((m_local, n))
comm.Scatterv(W, W_local, root=0)

for i in range(n):
    ri_local = Q_local[:,0:i].T @ W_local[:,i]
    ri = np.empty((i,1))
    comm.Allreduce(ri_local, ri, op = MPI.SUM)
    Q_local[:,i,None] = W_local[:,i,None] - Q_local[:,0:i] @ ri
    qi_norm_local2 = norm(Q_local[:,i])**2
    qi_norm = comm.allreduce(qi_norm_local2, op = MPI.SUM)**0.5
    Q_local[:,i] = Q_local[:,i]/qi_norm

    if rank == 0:
        R[0:i,i,None] = ri
        R[i,i] = qi_norm

comm.Gatherv(Q_local, Q, root = 0)
```

3 Cholesky QR

The second algorithm we analyze is Cholesky QR (CholQR).
Consider a full rank matrix $W \in \mathbb{R}^{m \times n}$, we notice that, if $W = QR$, $W^\top W = (QR)^\top QR = R^\top Q^\top QR = R^\top IR = R^\top R$. Then, we can generate R as the Cholesky factorization of $A = W^\top W$ and then reconstruct Q as $Q = WR^{-1}$. Since we take W full rank, $W^\top W$ is positive definite and its Cholesky factorization is unique. CholQR can be schematized as follows.

Algorithm 2 Cholesky QR

$A = W^\top W$
$R = \text{Cholesky}(A)$
Solve $QR = W$ for Q

A possible sequential implementation of CholQR in Python is the following. We solve the linear system by back-substitution, since R is upper triangular.

Code 3: sequential CholQR

```
import numpy as np

# Define the matrix W here
W = ...

(m,n) = W.shape

A = W.T @ W
R = np.linalg.cholesky(A).T

Q = np.empty((m,n))

for i in range(n):
    Q[:,i] = (W[:,i] - Q[:,0:i] @ R[0:i,i])/R[i,i]
```

We want now to parallelize the algorithm.
Since we're working with tall and skinny matrices, $A \in \mathbb{R}^{n \times n}$ is relatively small and its Cholesky factorization makes up only a small portion of the overall execution time (roughly 0.1% of the sequential runtime on a 50000×600 matrix). For this reason, we've decided not to parallelize it; it will execute only on processor 0.
Similarly to CGS, assuming again that s divides m, we distribute the matrix W among the s processors by partitioning it into s row blocks, so that each processor owns a block W_r of W. For the sake of simplicity, here we assume to have only 4 processors.

The matrix multiplication at **code 3 line 9** can be written using blocks as follows.

$$W^\top W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix}^\top \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix} = \begin{bmatrix} W_1^\top & W_2^\top & W_3^\top & W_4^\top \end{bmatrix} \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix} = = \sum_{r=1}^{4} W_r^\top W_r$$

Each processor executes one product $W_r^\top W_r$ and the results are summed together using `comm.Reduce` having processor 0 as root.
As we have anticipated, we execute the Cholesky factorization of A only on processor 0. Processor 0 then shares the result with the other processors using `comm.bcast`.
Then, we solve the linear system by back-substitution, as in **code 3 lines 13-14**. Every processor is responsible for generating a row block of Q, Q_r.
Lastly, we assemble Q using `comm.Gatherv` and store it in processor 0.
A possible implementation of parallel CholQR in Python is the following. Only the most important lines are presented here, the complete code can be found in Appendix B.

Code 4: parallel CholQR

```
1  m_local = m//s
2  W_local = np.empty((m_local, n))
3  comm.Scatterv(W, W_local, root=0)
4
5  A_local = W_local.T @ W_local
6  A = np.empty((n,n))
7  comm.Reduce(A_local, A, op = MPI.SUM, root = 0)
8
9  if rank == 0:
10     R = np.linalg.cholesky(A).T
11 else:
12     R = None
13
14 R = comm.bcast(R, root = 0)
15
16 for i in range(n):
17     W_local[:,i] = (W_local[:,i] - W_local[:,0:i] @ R[0:i,i])/R[i,i]
18
19 comm.Gatherv(W_local, Q, root = 0)
```

Finally, we remark that this particular algorithm fails when $A = W^\top W$ is numerically singular. Usually, this happens when W is very badly conditioned. One way to solve this problem is to introduce a small perturbation to A by adding $\delta\, I_n$ before executing the Cholesky factorization. This can be done with increasing values of δ until A becomes non-singular. The process can be implemented as follows.

Code 5: Cholesky factorization with perturbation

```
1  def cholesky_perturbed(A, delta = 1e-12, delta_step = 10):
2      try:
3          return np.linalg.cholesky(spd)
4      except:
5          I_n = np.identity(n)
6          while True:
7              try:
8                  return np.linalg.cholesky(A + delta*I_n).T
9              except:
10                 delta *= delta_step
```

In general, such a function can be quite time-consuming, as it may take many attempts to successfully execute the Cholesky factorization. However, it is important to notice that one execution takes very little time over the total execution time. Moreover, by choosing `delta_step` large enough (10 has proven to be a good choice), $A + \delta\, I_n$ typically becomes non-singular in very few iterations.

4 TSQR

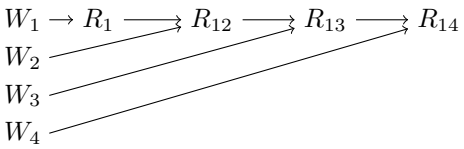
The third algorithm we analyze is Tall-Skinny QR (TSQR). The content and the notation of this section are mostly taken from [4] and [3].
For this algorithm, we assume to have available a numerically stable method able to compute the thin QR factorization

of a matrix in sequential programming. For the Python implementations we use `numpy.linalg.qr`, that interfaces to the LAPACK routine dgeqrf that is implemented using Householder QR.
For the sequential implementation, our aim will be to present how the QR factorization of a matrix can be computed when we have limited fast memory available. For the parallel implementation, we will compute the QR factorization using a binary tree.
The idea behind the sequential implementation is to make sure that all the computations are done in fast memory and that the matrix is read from slow memory only once. In order to achieve this, we partition the matrix W into p row blocks W_r so that each block can be contained in the fast memory. For the sake of simplicity, here we assume to cut the matrix in only 4 blocks. The algorithm starts by factorizing W_1 as $Q_{11}R_{11}$. Afterward, it stacks R_{11} and W_2 together and factorizes this new matrix as $Q_{12}R_{12}$. Then, it stacks R_{12} and W_3 together and factorizes the resulting matrix as $Q_{13}R_{13}$. The process continues until we obtain $Q_{14}R_{14}$. Our claim is that R_{14} is the R factor in the QR factorization of W. This can be shown, by writing down the factorization as follows.

$$A = \begin{pmatrix} (W_1) \\ W_2 \\ W_3 \\ W_4 \end{pmatrix} = \begin{pmatrix} Q_{11}R_{11} \\ W_2 \\ W_3 \\ W_4 \end{pmatrix} = \begin{pmatrix} Q_{11} & & & \\ & I_{m/4} & & \\ & & I_{m/4} & \\ & & & I_{m/4} \end{pmatrix} \begin{pmatrix} \begin{pmatrix} R_{11} \\ W_2 \end{pmatrix} \\ W_3 \\ W_4 \end{pmatrix} = \begin{pmatrix} Q_{11} & & & \\ & I_{m/4} & & \\ & & I_{m/4} & \\ & & & I_{m/4} \end{pmatrix} \begin{pmatrix} Q_{12}R_{12} \\ W_3 \\ W_4 \end{pmatrix} =$$

$$= \cdots = \begin{pmatrix} Q_{11} & & & \\ & I_{m/4} & & \\ & & I_{m/4} & \\ & & & I_{m/4} \end{pmatrix} \begin{pmatrix} Q_{12} & & & \\ & I_{m/4} & & \\ & & I_{m/4} & \\ & & & I_{m/4} \end{pmatrix} \begin{pmatrix} Q_{13} & & \\ & I_{m/4} & \\ & & I_{m/4} \end{pmatrix} Q_{14}R_{14}$$

The algorithm can be visualized as follows.



Where the notation has the following meaning: when k arrows point to a R matrix, this is the R factor in the QR factorization of the k matrices stacked one above the other [3].
In order to compute Q, we can either execute the multiplication of the orthogonal matrices above or solve the linear system $QR = W$ for Q as in CholQR. To execute the multiplication efficiently, we want to exploit the structure of the matrices in order to avoid to multiply by 1 or 0. For example, we notice that

$$\begin{pmatrix} Q_{13} & \\ & I_{m/4} \end{pmatrix} Q_{14} = \begin{pmatrix} Q_{13}\, Q_{14}[0:n,:] \\ Q_{14}[n:(n+m/4),:] \end{pmatrix}.$$

This same reasoning can be applied to all the products in order to greatly reduce the number of multiplication executed. A possible implementation of sequential TSQR in Python is the following. Only the most important lines are presented here, the complete code can be found in Appendix C.

Code 6: sequential TSQR

```
p = 4

Q = np.empty((m,n))
R = Q[0:0,:]

Q_list = []
m_step = m//p
toFactor = W[0:m_step,:]

for step in range(p):
    toFactor = np.vstack((R, W[(step*m_step):((step+1)*m_step),:]))
    (Q_step, R) = np.linalg.qr(toFactor, mode='reduced')
    Q_list.append(Q_step)

for step in reversed(range(1,p)):
    Q_list[step-1] = Q_list[step-1] @ Q_list[step][0:n,:]
    Q[(step*m_step):((step+1)*m_step),:] = Q_list[step][n:,:]
Q[0:m_step] = Q_list[0]
```

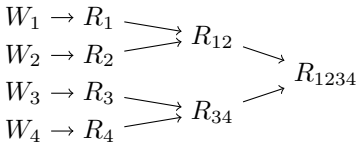
We move now to parallel TSQR.
Without loss of generality, we assume that m is divisible by s. We distribute the matrix W among the s processors

by partitioning it into s row blocks, so that each processor gets a block W_r of W. For the sake of simplicity, here we assume again to have only 4 processors.
As a first step, each processors performs the QR decomposition of its assigned block $W_r = Q_r R_r$ for $r = 1, 2, 3, 4$. Then, the R factors are paired, stacked together as $\begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$ and $\begin{pmatrix} R_3 \\ R_4 \end{pmatrix}$ and factorized as $Q_{12}R_{12}$ and $Q_{34}R_{34}$ on processors 1 and 3, respectively. Finally, $\begin{pmatrix} R_{12} \\ R_{34} \end{pmatrix}$ is factorized as $Q_{1234}R_{1234}$ on processor 1. Our claim is that R_{1234} is the R factor in the QR factorization of W. This can be shown by writing down the factorization as follows.

$$W = \begin{pmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{pmatrix} = \begin{pmatrix} Q_1 R_1 \\ Q_2 R_2 \\ Q_3 R_3 \\ Q_4 R_4 \end{pmatrix} = \begin{pmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{pmatrix} \left(\begin{pmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{pmatrix} \right) = \begin{pmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{pmatrix} \begin{pmatrix} Q_{12}R_{12} \\ Q_{34}R_{34} \end{pmatrix} =$$

$$= \begin{pmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{pmatrix} \begin{pmatrix} Q_{12} & \\ & Q_{34} \end{pmatrix} \begin{pmatrix} R_{12} \\ R_{34} \end{pmatrix} = \begin{pmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{pmatrix} \begin{pmatrix} Q_{12} & \\ & Q_{34} \end{pmatrix} Q_{1234} R_{1234}$$

The algorithm can be visualized as follows.



Moreover, after this process is finished the $Q...$ factors are located as follows among the 4 processors.

Processor 1	Q_1	Q_{12}	Q_{1234}
Processor 2	Q_2		
Processor 3	Q_3	Q_{34}	
Processor 4	Q_4		

Like for sequential TSQR, in order to compute Q, we can either execute the multiplication of the orthogonal matrices above or solve the linear system $QR = W$ for Q.
To execute the multiplication efficiently, we want to exploit the structure of the matrices in order to minimize the number of multiplications and possibly take advantage of the locality of the different matrices. For example, we notice that

$$\begin{pmatrix} Q_{12} & \\ & Q_{34} \end{pmatrix} Q_{1234} = \begin{pmatrix} Q_{12}Q_{1234}[0\!:\!n, :] \\ Q_{34}Q_{1234}[n\!:\!2n, :] \end{pmatrix}$$

Since, Q_{12} and Q_{34} are located in processors 1 and 3 respectively, this first multiplication can be easily parallelized by executing the two matrix multiplications directly there. This same reasoning can be extended to the other multiplications generating a sort of reversed tree multiplication.
A possible implementation of parallel TSQR in Python can be found in Appendix C.

5 Numerical stability

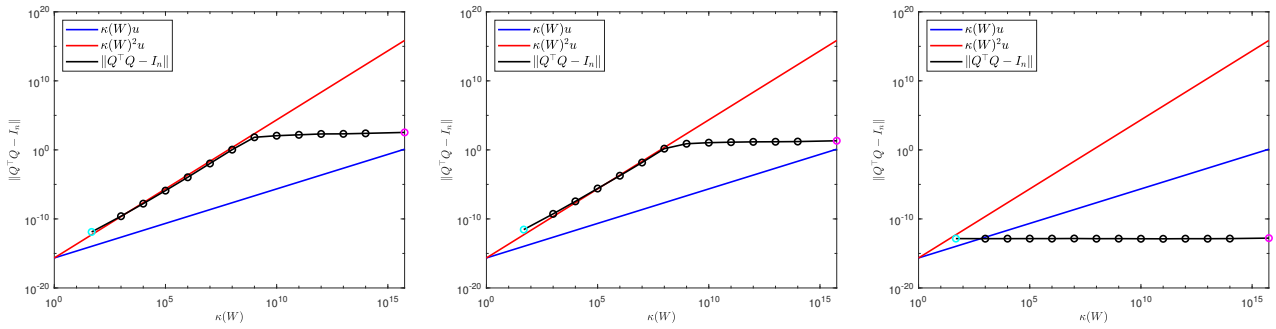
In this section, we investigate the numerical stability of the three algorithms. We define loss of orthogonality $\|I_n - Q^\top Q\|$ and use it to assess the numerical stability of the algorithms.
We start by presenting the theoretical results on the loss of orthogonality for the three algorithms.

- CGS: $\|I_n - Q^\top Q\| \leq C\,\kappa^2(W)\,u$ [5]
- CholQR: $\|I_n - Q^\top Q\| = O(\kappa^2(W)\,u)$ [2]
- TSQR: $\|I_n - Q^\top Q\| = O(u)$ [2]

Here u is the machine precision (for NumPy floating point numbers with double precision, $u = 2.220446049250313e{-}16$). For the numerical tests, we decided to use synthetic matrices instead of matrices from real-world problems. This allows us to control key properties of the matrices such as size, condition number and density. We define 3 types of synthetic matrices:

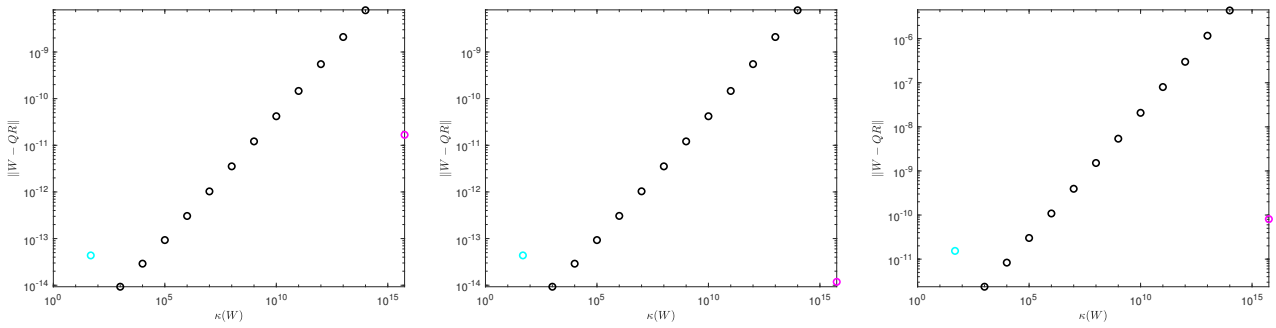
1. We define $f(x,y) = \frac{\sin(10(y+x))}{\cos(100(y-x))+1.1}$ and we take $W^1 \in \mathbb{R}^{m\times n}$ as $W^1_{ij} = f\left(\frac{i-1}{m-1}, \frac{j-1}{n-1}\right)$ for $i = 1...m$ and $j = 1...n$ [1].
2. We take $W^2 =$ `numpy.random.rand(m,n)`, that means that every entry of $W^2 \in \mathbb{R}^{m\times n}$ is sampled uniformly in $[0, 1)$.
3. We take $W^{3,\kappa} = UDV^*$, where κ is a priori fixed, $D \in \mathbb{R}^{n\times n}$ is diagonal and $U \in \mathbb{R}^{m\times n}$ and $V \in \mathbb{R}^{n\times n}$ are random orthogonal matrices obtained respectively with
   `U,_ = np.linalg.qr(np.random.rand(m,n), mode='reduced')` and
   `V,_ = np.linalg.qr(np.random.rand(n,n), mode='reduced')`.
   To generate the elements $d_1, ...d_n$ of the diagonal of D, we sample $x_1, ...x_n$ uniformly in $[-\frac{1}{2}, \frac{1}{2})$, we rescale them as $y_i = (x_i - \min_i x_i)/(\max_i x_i - \min_i x_i)$ and we compute $d_i = \kappa^{y_i}$. We remark that the values $1/\sqrt{\kappa}$ and $\sqrt{\kappa}$ are always sampled. Defining $W^{3,\kappa}$ this way allows us to control its condition number via its singular value decomposition.

In order to be able to run different tests on the same matrices, before generating every matrix, we reset the state of the random number generator to a specific value using `np.random.set_state`.
All the tests have been executed on a 24-core Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz (only 4 cores available) via a virtual desktop interface provided by EPFL. At the beginning of each testing code, `OMP_NUM_THREADS` has been set to 1 in order to prevent the BLAS library (that is used in most of the NumPy functions) from autonomously run on multiple processors.
To numerically evaluate the bounds on the loss of orthogonality, we fixed $m = 50000$ and $n = 600$, and we selected a set of 14 matrices for algorithm testing. The matrices we took are W^1, W^2, $W^{3,1e3}$, $W^{3,1e4}$, ..., $W^{3,1e14}$.
We executed CGS, CholQR, and TSQR on these matrices and we computed the loss of orthogonality. The results are collected in the following graphs in logarithmic scale. The light blue and pink circles represent W^1, W^2, respectively, while the black circles represent the other matrices.



(a) CGS (b) CholQR (c) TSQR

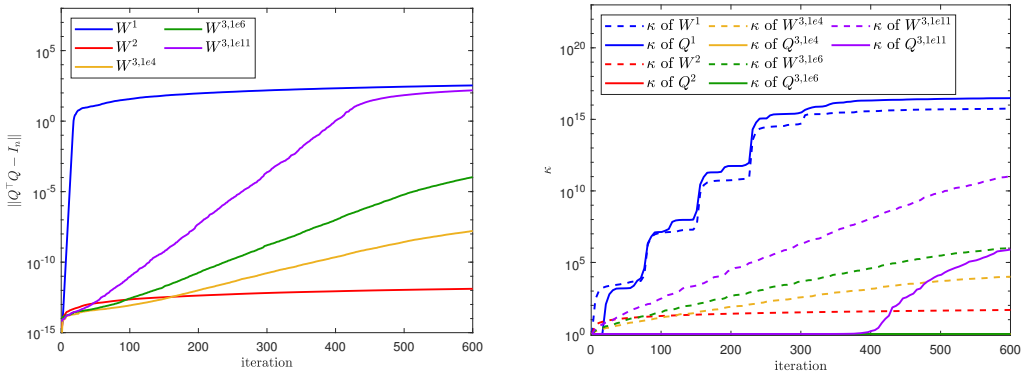
We also plot the norm of the difference between each matrix W and its factorization QR. The graphs are in logarithmic scale.



(d) CGS (e) CholQR (f) TSQR

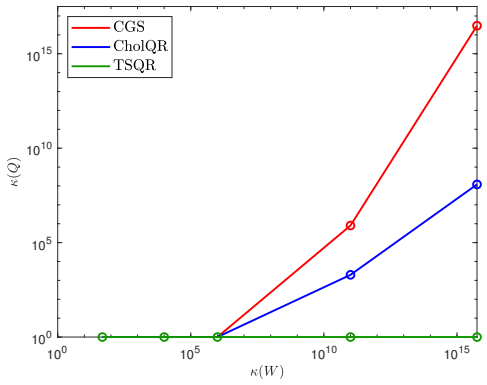
We point out that when $\kappa \geq 10^9$ CholQR fails, since $A = W^\top W$ becomes numerically singular. The results reported here are obtained using `cholesky_perturbed` in **code 5** with default parameters. From (e), we see that this perturbation

does not significantly impact the quality of the factorization.
From (a) and (b), we see that in both cases the loss of orthogonality follows the correct trend until $\kappa = 10^8$ (the line is parallel to $\kappa(W)^2 u$). Starting from $\kappa = 10^9$, the loss of orthogonality reaches a plateau. One possible explanation for this fact is that right after $\kappa = 10^8$ the matrix becomes numerically singular and this improves the performance of the two algorithms. Intuitively, this makes sense, since a rank smaller than n means that there are fewer vectors to normalize. A similar behavior for CGS has been observed (but not explained) also in [6]. Nevertheless, the theoretical bounds are satisfied. From (c), we confirm that the loss of orthogonality is constant with respect to $\kappa(W)$, as affirmed by the theoretical bound.
From (d), (e) and (f), we notice that the first two methods give in general a better decomposition of W in Q and R, this is however compensated by the better loss of orthogonality provided by TSQR. Moreover, we see how the error in the decomposition does not depend directly on the condition number of W. Indeed, W^1 and W^2 don't follow at all the trend provided by the type 3 matrices, proving that this trend is more likely related to how the matrices were generated rather than to a numerical property of the methods with regards to the condition number.
We now compare the three methods in terms of their final loss of orthogonality and the condition number of the Q factor generated. Indeed, for some applications it's not only important that Q is orthogonal but also well conditioned. For CGS, we plot the evolution of the loss of orthogonality through the iterations, when factorizing W^1, W^2, $W^{3,1e4}$, $W^{3,1e6}$, and $W^{3,1e11}$. We also provide a graph that shows how the condition numbers of Q and $W[:, 0:i]$ evolve through the iterations.



We see here that the loss of orthogonality quickly reaches a plateau when factorizing W^1 and W^2, instead it increases quite steadily with matrices of the third type. In this last case, for $W^{3,1e11}$, we notice a plateau in the last iterations. This is correlated with an increase in the condition number of its Q factor $Q^{3,1e11}$. About the condition numbers, we can say that, in general, CGS leads to a decrease in the condition number; this is however not true if the factorized matrix is very badly conditioned, like W^1.
Finally, we have collected the final results obtained by CholQR and TSQR on the same 5 matrices in the following table and graph (in logarithmic scale).

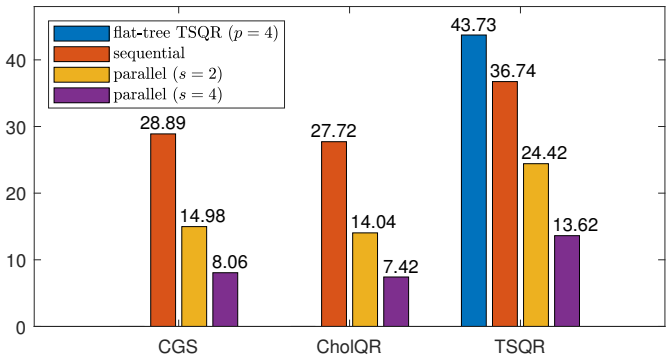
	$\kappa(W)$	$\kappa(Q)$	$\|I_n - Q^\top Q\|$	
W^1	5.6692e15	1.2049e8	20.877	CholQR
W^2	47.662	1+2.307e–13	2.8443e–12	
$W^{3,1e4}$	1e4	1+8.9272e–9	3.5038e–08	
$W^{3,1e6}$	1e6	1+5.2303e–5	1.8063e–4	
$W^{3,1e11}$	1e11	1.9642e3	13.083	
W^1	5.6692e15	1+1.5e–14	1.6551e–13	TSQR
W^2	47.662	1+6.4e–15	1.4099e–13	
$W^{3,1e4}$	1e4	1+9.3e–15	1.3859e–13	
$W^{3,1e6}$	1e6	1+6.9e–15	1.4025e–13	
$W^{3,1e11}$	1e11	1+1e–14	1.3134e–13	



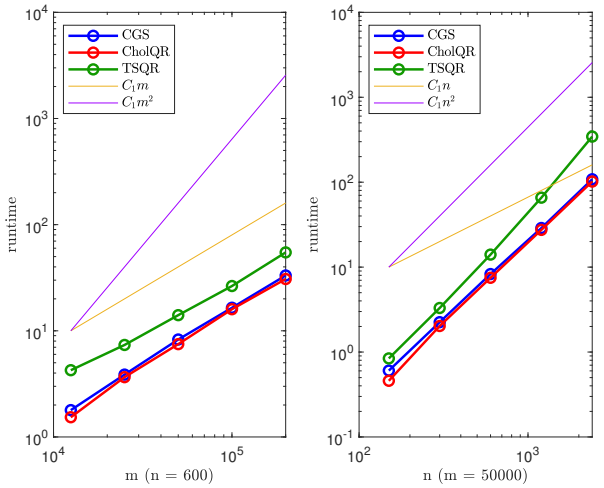
Similarly to CGS, when W is badly conditioned, CholQR generates a badly conditioned Q factor, when this happens also the loss of orthogonality is very high. On the other hand, TSQR proves to be very numerically stable. Independently from the initial condition number of W, the output Q is always very close to orthogonal and well conditioned.

6 Execution time

In this section we present the execution times obtained by the three algorithms and we investigate their relations with m and n. All the performance testing will be conducted using matrix W^2.
We start by plotting the sequential and parallel runtimes obtained by the three algorithms in the following graph. Each test has been run 3 times on W^2 with $m = 50000$ and $n = 600$; the average runtime is reported here. For sequential TSQR, we timed also the NumPy function `numpy.linalg.qr`.



We have observed that TSQR is the slowest among the three algorithms, while CGS and CholQR exhibit similar runtimes. These two algorithms also show the most significant improvements from parallelization. Indeed, the runtimes are almost cut in half when doubling the number of processors used, proving that we did a good job avoiding communication. The runtime for TSQR is roughly cut in half only when going from 2 to 4 processors. The flat-tree TSQR implementation also proves to be quite efficient, being only 16% worse than the NumPy QR function.
We continue by executing an array of tests varying the size of the matrix and keeping the number of processors at 4. We consider $m = 50000$ and $n = 600$ as our central case. We then proceed fixing one of the dimensions of the matrix and varying the other. We take $m = 12500, 25000, 50000, 100000, 200000$, and then $n = 150, 300, 600, 1200, 2400$. We plot the resulting run times in the following graph in logarithmic scale.



From these graphs we can empirically infer that the runtimes for the three algorithms is proportional to n^2 fixing m and to m fixing n. Generalizing, we have that the runtimes are $O(mn^2)$ for the three algorithms. This is confirmed to be also the theoretical bound for example in [4].

Appendices

A Code for parallel Classical Gram Schmidt

Code 7: parallel CGS (complete)

```
from mpi4py import MPI
import numpy as np
from numpy.linalg import norm

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
s = comm.Get_size()

(m, n) = (None, None)

if rank == 0:
    # Define the matrix W here
    W = ...

    (m,n) = W.shape

(m,n) = comm.bcast((m, n), root = 0)
m_local = m//s
W_local = np.empty((m_local, n))
comm.Scatterv(W, W_local, root=0)

if rank == 0:
    Q = np.empty((m,n))
    R = np.zeros((n,n))
else:
    Q = None
    R = None

Q_local = np.empty((m_local, n))

for i in range(n):
    ri_local = Q_local[:,0:i].T @ W_local[:,i]
    ri = np.empty((i,1))
    comm.Allreduce(ri_local, ri, op = MPI.SUM)
    Q_local[:,i,None] = W_local[:,i,None] - Q_local[:,0:i] @ ri
    qi_norm_local2 = norm(Q_local[:,i])**2
    qi_norm = comm.allreduce(qi_norm_local2, op = MPI.SUM)**0.5
    Q_local[:,i] = Q_local[:,i]/qi_norm

    if rank == 0:
        R[0:i,i,None] = ri
        R[i,i] = qi_norm

comm.Gatherv(Q_local, Q, root = 0)
```

B Code for parallel Cholesky QR

Code 8: parallel CholQR (complete)

```python
from mpi4py import MPI
import numpy as np
from numpy.linalg import norm
import math

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
s = comm.Get_size()

(m, n) = (None, None)

if rank == 0:
    # Define the matrix W here
    W = ...

    (m,n) = W.shape

(m,n) = comm.bcast((m, n), root = 0)
m_local = m//s
W_local = np.empty((m_local, n))
comm.Scatterv(W, W_local, root=0)

A_local = W_local.T @ W_local
A = np.empty((n,n))
comm.Reduce(A_local, A, op = MPI.SUM, root = 0)

if rank == 0:
    R = np.linalg.cholesky(A).T
else:
    R = None

R = comm.bcast(R, root = 0)

for i in range(n):
    W_local[:,i] = (W_local[:,i] - W_local[:,0:i] @ R[0:i,i])/R[i,i]

Q = np.empty((m,n))
comm.Gatherv(W_local, Q, root = 0)
```

C Codes for Tall-Skinny QR

Code 9: sequential TSQR (complete)

```python
import numpy as np

p = 4

# Define the matrix W here
W = ...

(m,n) = W.shape

Q = np.empty((m,n))
R = Q[0:0,:]

Q_list = []
m_step = m//p
toFactor = W[0:m_step,:]

for step in range(p):
    toFactor = np.vstack((R, W[(step*m_step):((step+1)*m_step),:]))
    (Q_step, R) = np.linalg.qr(toFactor, mode='reduced')
    Q_list.append(Q_step)

for step in reversed(range(1,p)):
    Q_list[step-1] = Q_list[step-1] @ Q_list[step][0:n,:]
    Q[(step*m_step):((step+1)*m_step),:] = Q_list[step][n:,:]
Q[0:m_step] = Q_list[0]
```

Code 10: parallel TSQR

```
from mpi4py import MPI
import numpy as np
from numpy.linalg import norm

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
s = comm.Get_size()

(m, n) = (None, None)

if rank == 0:
    # Define the matrix W here
    W = ...

    (m,n) = W.shape

(m,n) = comm.bcast((m, n), root = 0)
m_local = m//s
toFactor = np.empty((m_local, n))
comm.Scatterv(W, toFactor, root=0)

Q_list = []

is_active = True
n_steps = int(math.log2(s))
for step in range(n_steps):
    if is_active:
        activeComm = comm.Split(color = 1 + rank/2**(step+1), key = rank)
        active_rank = activeComm.Get_rank()
        (Q_step, R_step) = np.linalg.qr(toFactor, mode='reduced')
        Q_list.append(Q_step)

        if active_rank == 0:
            R_rec = activeComm.recv(source = 1)
            toFactor = np.vstack((R_step,R_rec))
        else:
            activeComm.send(R_step, dest = 0)
            is_active = False
    else:
        activeComm = comm.Split(color = 0, key = rank)
if rank == 0:
    (Q_step, R) = np.linalg.qr(toFactor, mode='reduced')
    Q_list.append(Q_step)

    Q_local = Q_step

for step in range(n_steps):
    is_active = rank % (2**(n_steps-step-1)) == 0
    if is_active:
        activeComm = comm.Split(color = 1 + rank/2**(n_steps-step), key = rank)
        active_rank = activeComm.Get_rank()
        if active_rank == 0:
            activeComm.send(Q_local[n:,:], dest = 1)
            Q_local = Q_list[n_steps-step-1] @ Q_local[0:n,:]
        else:
            Q_local = activeComm.recv(source = 0)
            Q_local = Q_list[n_steps-step-1] @ Q_local
    else:
        activeComm = comm.Split(color = 0, key = rank)

Q = np.empty((m,n))
comm.Gatherv(Q_local, Q, root = 0)
```

References

[1] Oleg Balabanov and Laura Grigori. Randomized gram-schmidt process with application to gmres. 2022.

[2] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication–optimal parallel and sequential qr and lu factorizations: theory and practice. page 44, 2008.

[3] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Implementing communication-optimal parallel and sequential qr factorizations. *arXiv*, 2008.

[4] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[5] Luc Giraud, Julien Langou, Miroslav Rozložník, and Jasper van den Eshof. Rounding error analysis of the classical gram-schmidt orthogonalization process. *Numerische Mathematik*, 101(1):87–100, 2005.

[6] Katarzyna Swirydowicz, Julien Langou, Shreyas Ananthan, Ulrike Yang, and Stephen Thomas. Low synchronization gram–schmidt and generalized minimal residual algorithms. 28(2), 2020.