## 2.12 Compute the *QR* factorization of a general matrix *A*.

In linear algebra, a QR decomposition (also called a QR factorization) of a matrix is a decomposition of a matrix *A* into a product *A=QR* of an orthogonal matrix *Q* and an upper triangular matrix *R*. In linear algebra, an **orthogonal matrix** or **real orthogonal matrix** is a square matrix with real entries whose columns and rows are orthogonal unit vectors (i.e., orthonormal vectors), i.e. $Q^TQ=QQ^T=I$, where is the identity matrix. QR decomposition is often used to solve the linear least squares problem, and is the basis for a particular eigenvalue algorithm, the QR algorithm.

If *A* has *n* linearly independent columns, then the first *n* columns of *Q* form an orthonormal basis for the column space of *A*. More specifically, the first *k* columns of *Q* form an orthonormal basis[1] for the span of the first *k* columns of *A* for any $1 \le k \le n$. The fact that any column *k* of *A* only depends on the first *k* columns of *Q* is responsible for the triangular form of *R*.

The

To compute QR factorization of $m \times n$ matrix *A*, with $m > n$, we annihilate sub diagonal entries of successive columns of A, eventually reaching upper triangular form. Possible methods include

- Householder transformations
- Givens rotations
- Gram-Schmidt orthogonalization

### Householder Transformation

Householder transformation has form $H = I - 2\frac{vv^T}{v^Tv}$ for nonzero vector column *v* and *I* is the identity matrix of order *size(v)*. It is clear that *H* is orthogonal and symmetric: $H=H^T=H^{-1}$. Given vector *a*, we want to choose *v* so that

$$Ha = \left(I - 2\frac{vv^T}{v^Tv}\right)a = \begin{bmatrix}\alpha\\0\\\vdots\\0\end{bmatrix} = \alpha\begin{bmatrix}1\\0\\\vdots\\0\end{bmatrix} = \alpha e_1.$$

Substituting into formula for *H*, we can take $v = a - \alpha e_1$ and $\alpha = \pm\|a\|_2$ with sing chosen to avoid cancellation.

So it is true the following theorem.

**Theorem**.

*If* $e_1 = [1\ 0\ \ldots 0]^T$, *then for every* $x \neq 0$, *if we take* $v_x = x + \sigma e_1$, *with* $\sigma = \mathrm{sing}\left(x^Te_1\right)\sqrt{x^Tx}$ *then* $H(v_x)x = \sigma e_1$.

**Remark 1.**

In this theorem, the *sign* function is introduced for a numerical robustness property and is defend by: $\mathrm{sing}(z) = \begin{cases}-1 & \text{if } z < 0,\\ +1 & \text{if } z \ge 0.\end{cases}$

**Example** If $a = [2 \quad 1 \quad 2]^T$, then we take

[1] Adica produsul scalar al vectorilor coloane este egal cu 0.

$v = a - \alpha e_1 = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} - \alpha \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} - \begin{bmatrix} \alpha \\ 0 \\ 0 \end{bmatrix}$ where $\alpha = \pm \|a\|_2 = \pm 3$. Since $a_1$ is positive, we choose negative sing for $\alpha$ to avoid cancellation, so $v = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} - \begin{bmatrix} -3 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 2 \end{bmatrix}$. To confirm that transformation work, $Ha = a - 2\frac{v^T a}{v^T v} v = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} - 2\frac{15}{30} \begin{bmatrix} 5 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -3 \\ 0 \\ 0 \end{bmatrix}$ because $v^T a = \begin{pmatrix} 5 & 1 & 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} = 15$ and $v^T v = \begin{pmatrix} 5 & 1 & 2 \end{pmatrix} \begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix} = 30$ .

Algorithm of the Householder QR factorization

- To compute QR factorization of *A*, use Hauseholder transformations to annihilate sub diagonal entries to each successive column;
- Each Householder transformation is applied to entire matrix, but does not affect prior columns, so zeros are preserved;
- In applying Householder transformation *H* to arbitrary vector *u*,

$$Hu = \left( I - 2\frac{vv^T}{v^T v} \right) u = u - \left( 2\frac{v^T u}{v^T v} \right) v$$

  which is much cheaper than general matrix-vector multiplication and requires only vector *v*, not full matrix *H*.

- Process just described produces factorization $H_n \cdots H_1 A = \begin{bmatrix} R \\ 0 \end{bmatrix}$ where *R* is *nxn* and upper triangular.
- If $Q = H_1 \cdots H_n$, then $A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$

So, using the Householder transformations we will obtained:

$$Q = H_1 \cdots H_n, \text{ and } R = H_n \cdots H_1 A.$$

- To preserve solution of linear least squares problem, right-hand side *b* is transformed by same sequence of Householder transformations
- Then solve triangular least squares problem $\begin{bmatrix} R \\ 0 \end{bmatrix} x \cong Q^T b$
- For solving linear least squares problem, product *Q* of Householder transformations need not be formed explicitly
- *R* can be stored in upper triangle of array initially containing *A*
- Householder vectors *v* can be stored in (now zero) lower triangular portion of *A* (almost)
- Householder transformations most easily applied in this form anyway.

**Example.** Compute the QR decomposition of $A = \begin{pmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \\ 1 & -1 & 0 \end{pmatrix}$

Here we index $A$ with superscript, and let $A^{(0)} = A$.

1. Compute the reflector $v_1 = a_1 - sign(a_{11})||a_1||e_1$

$$v_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} - 2\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \text{ because } ||a_1||=2$$

2. Householder matrix $H_1 = I - 2\dfrac{v_1 v_1^T}{v_1^T v_1}$

$$H_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} - \frac{2}{4}\begin{pmatrix} -1 \\ 1 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} -1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 & -1/2 \\ 1/2 & -1/2 & 1/2 & -1/2 \\ 1/2 & -1/2 & -1/2 & 1/2 \end{pmatrix},$$

because $v_1^T v_1 = 4$.

3. Apply the Householder matrix $H_1$ to matrix $A^{(0)}$ we obtain

$$H_1 A^{(0)} = A^{(0)} - \frac{1}{2} v v^T A^{(0)} = \begin{pmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \\ 1 & -1 & 0 \end{pmatrix} - \frac{1}{2}\begin{pmatrix} -1 \\ 1 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} 2 & 8 & -4 \end{pmatrix} \text{ because}$$

$v^T A^{(0)} = \begin{pmatrix} 2 & 8 & -4 \end{pmatrix}$. And finally, because $\begin{pmatrix} -1 \\ 1 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} 2 & 8 & -4 \end{pmatrix} = \begin{pmatrix} -2 & -8 & 4 \\ 2 & 8 & -4 \\ 2 & 8 & -4 \\ 2 & 8 & -4 \end{pmatrix}$ we

obtained $A^{(1)} = H_1 A^{(0)} = \begin{pmatrix} 2 & 3 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & -5 & 2 \end{pmatrix}$.

4. Now we only consider the submatrix $A^{(1)}(2:4,2:3) = \begin{pmatrix} 0 & 0 \\ 0 & 4 \\ -5 & 2 \end{pmatrix}$ (using Matlab's notation: *from line 2 to line 4 and from column 2 to column 3*), $A^{(1)}(2:4,2:3) = \left(a_1^{(1)}, a_2^{(1)}\right)$.

5. Let $v_2 = a_1^{(1)} + \sin g\left(A^{(1)}(2,2)\right)\left\|a_1^{(1)}\right\| e_1$ and because $\left\|a_1^{(1)}\right\| = \left\| \begin{pmatrix} 0 \\ 0 \\ -5 \end{pmatrix} \right\| = 5$ we obtained

$$v_2 = \begin{pmatrix} 0 \\ 0 \\ -5 \end{pmatrix} - \begin{pmatrix} 5 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -5 \\ 0 \\ -5 \end{pmatrix}.$$

6. Similar step 2 we have,

$$H_2 = I - 2\frac{v_2 v_2^T}{v_2^T v_2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{1}{25}\begin{pmatrix} -5 \\ 0 \\ -5 \end{pmatrix}\begin{pmatrix} -5 & 0 & -5 \end{pmatrix} = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$ because

$v_2^T v_2 = 50$.

7. Similar step 3 we have

$$H_2 A^{(1)}(2:4,2:3) = A^{(1)}(2:4,2:3) - \frac{1}{25} v_2 v_2^T A^{(1)}(2:4,2:3) =$$

$$= \begin{pmatrix} 0 & 0 \\ 0 & 4 \\ -5 & 2 \end{pmatrix} - \frac{1}{25}\begin{pmatrix} -5 \\ 0 \\ -5 \end{pmatrix}\begin{pmatrix} 25 & -10 \end{pmatrix} = \begin{pmatrix} 5 & -2 \\ 0 & 4 \\ 0 & 0 \end{pmatrix}$$ because

$v_2^T A^{(1)}(2:4,2:3) = \begin{pmatrix} 25 & -10 \end{pmatrix}$. Finally we have

$A^{(2)} = H_2 A^{(1)} = H_2 H_1 A^{(0)} = \begin{pmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{pmatrix}$. So in matrix

$A^{(1)} = H_1 A^{(0)} \begin{pmatrix} 2 & 3 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & -5 & 2 \end{pmatrix}$ block (2:4,2:3) is replaced

with $H_2 A^{(1)}(2:4,2:3) = \begin{pmatrix} 5 & -2 \\ 0 & 4 \\ 0 & 0 \end{pmatrix}$.

8. The $R$ factor is $\begin{pmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \end{pmatrix}$, but where is the $Q$ factor?

9. Since $H_2 H_1 A = \begin{pmatrix} R \\ 0 \end{pmatrix}$, $Q = (H_2 H_1)^{-1} = (H_2 H_1)^T = H_1^T H_2^T = H_1 H_2 =$

$$= \begin{pmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 & -1/2 \\ 1/2 & -1/2 & 1/2 & -1/2 \\ 1/2 & -1/2 & -1/2 & 1/2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1/2 & -1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & -1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 & 1/2 \end{pmatrix}.$$

10. Verified:

$$A = QR = \begin{pmatrix} 1/2 & -1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & -1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 & 1/2 \end{pmatrix} \begin{pmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \\ 1 & -1 & 0 \end{pmatrix}.$$

## Description of the block QR factorization

The algorithm of the QR factorization that is implemented in LAPACK is based on accumulating a number of Householder transformations in what is called a panel factorization, which are, then, applied all at once by means of high-performance.
We describe the matrix form of the Householder transformation which will allow reducing in one step the last component of several vectors. This will lead to a paralleled version of the QR algorithm where only $\log_2(n)$ steps will be necessary.

Len us consider the full column rank matrix V and if we introduce the matrix defined by:

$$H(V) = I_n - 2V(V^T V)^{-1} V^T \qquad (*)$$

which appears as a matrix extension of the usual Householder transformation, we have the following result:

**Theorem.** *For every (nxr) matrix V, such that rank(V)=r, then H(V) is symmetric orthogonal.*

Also it is true the following theorem.

**Theorem.** *For any (mxn) matrix A, such that rank(A)=r, there exists a matrix Householder transformation $H_A$ (*), such that* $H_A A = \begin{pmatrix} \tilde{A} \\ O_{((m-r)\times n)} \end{pmatrix}$, where $\tilde{A}$ is $(r \times n)$.

Given an $M \times N$ matrix $A$ we seek the factorization $A=QR$ where $Q$ is an $M \times M$ orthogonal matrix and $R$ is an $M \times N$ upper triangular matrix. At the $k$-th step of the computation we partition this factorization to the $m \times n$ submatrix of $A^{(k)}$ as follows

$$A^{(k)} = (A_1 \quad A_2) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

where the block $A_{11}$ is $n_b \times n_b$, $A_{12}$ is $n_b \times (n-n_b)$, and $A_{21}$ is $(m-n_b) \times n_b$, $A_{22}$ is $(m-n_b) \times (n-n_b)$. $A_1$ is $m \times n_b$ matrix, containing the first $n_b$ columns of the matrix $A^{(k)}$, and $A_2$ is an $m \times (n-n_b)$ matrix, containing the last $(n-n_b)$ columns of $A^{(k)}$ (that is $A_1 = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ and $A_2 = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$). $R_{11}$ is $n_b \times n_b$ upper triangular matrix. A QR factorization is performed on the first $m \times n_b$ panel of $A^{(k)}$ (i.e., $A_1$). In practice, $Q$ is computed by applying a series of Householder transformation to $A_1$ of the form $H_i = I - \tau_i v_i v_i^T$ where $i=1,\ldots,n_b$. The vector $v_i$ is of length $m$ with $0$'s for the first $i-1$ entries and $1$ for $i$-th entry, and $\tau_i = \frac{2}{v_i^T v_i}$. During the QR factorization, the vector $v_i$ overwrites the entries of $A$ below the diagonal, and $\tau_i$ is stored in a vector.

Furthermore, it can be shown that $Q = H_1 H_2 \cdots H_{n_b} = I - VTV^T$, where $T$ is $n_b \times n_b$ upper triangular and $i$-th column of $V$ equals $v_i$. This is indeed a block version of the QR factorization, and is rich in matrix-matrix operations.

The block equation can be rearranged as $\tilde{A}_2 = \begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} \leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = Q^T A_2 = \left(I - VT^T V^T\right) A_2$. During the computation, the sequence of the Householder vectors $V$ is computed, and the row panel $R_{11}$ and $R_{12}$, and the trailing submatrix $A_{22}$ are updated. The factorization can be done by recursively applying the steps outlined above to the $(m-n_b) \times (n-n_b)$ matrix $\tilde{A}_{22}$.
The computation of the above steps in the LAPACK routine **DGEQRF** involves the following operations

1. **DGEQF2:** Apply the QR factorization on an $m \times n_b$ column panel of $A^{(k)}$ (i. e. $A_1$)
   - [ Repeat $n_b$ times ($i = 1 \cdots n_b$) ]
     - **DLARFG** generate the elementary reflector $v_i$ and $\tau_i$
     - **DLARF** update the trailing submatrix $\tilde{A}_1 \leftarrow H_i^T A_1 = \left(I - \tau_i v_i v_i^T\right) A_1$
2. **DLARFT** Compute the triangular factor $T$ of the block reflector $Q$
3. **DLARFB** Apply $Q^T$ to the rest of the matrix from the left $\tilde{A}_2 = \leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = Q^T A_2 = \left(I - VT^T V^T\right) A_2$.
   - **DGEMM:** $W \leftarrow V^T A_2$
   - **DTRMM:** $W \leftarrow T^T W$
   - $\tilde{A}_2 = \leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW$

The corresponding parallel implementation of the ScaLAPACK routine **PDGEQRF** proceeds as follows

1. **PDGEQR2:** The current column of processes performs the QR factorization on an $m \times n_b$ panel of $A^{(k)}$ (i e $A_1$)

- [Repeat $n_b$ times ($i = 1 \cdots n_b$) ]
  - **PDLARFG** generate elementary reflector $v_i$ and $\tau_i$
  - **PDLARF** update the trailing submatrix

2. **PDLARFT:** The current column of processes, which has a sequence of the Householder vector $V$, computes $T$ only in the current process row.

4. **PDLARFB** Apply $Q^T$ to the rest of the matrix from the left
   - **PDGEMM:** $V$ is broadcast row wise across all columns of processes. The transpose of V is locally multiplied by $A_2$, then the products are added to the current process row $W \leftarrow V^T A_2$
   - **PDTRMM:** $T$ is broadcast row wise in the current process row to all columns of processes and multiplied with the sum $(W \leftarrow T^T W)$.
   - **PDGEMM:** $W$ is broadcast column wise down all rows of processes. Now, processes have their own portions of $V$ and $W$, then they update the local portions of the matrix

$$A_2 \left( \tilde{A}_2 =\leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW \right).$$

**Purpose**

These subroutines compute the *QR* factorization of a general matrix *A*, where, in this description:

- *A* represents the global general submatrix *A(ia:ia+m-1, ja:ja+n-1)* to be factored.
- For PDGEQRF, *Q* is an orthogonal matrix.
- For PZGEQRF, *Q* is a unitary matrix.
- For $m \geq n$, *R* is an upper triangular matrix.
- For $m < n$, *R* is an upper trapezoidal matrix.

If $m=0$ or $n=0$, no computation is performed and the subroutine returns after doing some parameter checking.

Table 1. Data Types

| **Data Types** | |
|---|---|
| ***A, tau, work*** | **Subroutine** |
| Long-precision real | PDGEQRF |
| Long-precision complex | PZGEQRF |

**Syntaxes of the routine for compute the *QR* factorization of a general matrix *A*.**

```
CALL PDGEQRF | PZGEQRF (m, n, a, ia, ja, desc_a, tau, work,
                         lwork,info)
pdgeqrf_ | pzgeqrf_(int *m, int *n, double *a, int *ia, int *ja,
                    int *desc_a, double *tau, double *work, int
                   *lwork, int *info)
```

**On Entry**

m

is the number of rows in submatrix *A* used in the computation.
Scope: **global**

Specified as: a fullword integer; m≥0.

`n`

is the number of columns in submatrix *A* used in the computation.
Scope: **global**
Specified as: a fullword integer; n ≥ 0.

`a`

is the local part of the global general matrix *A*. This identifies the **first element** of the local array A. This subroutine computes the location of the first element of the local subarray used, based on ia, ja, desc_a, p, q, myrow, and mycol; therefore, the leading LOCp(ia+m-1) by LOCq(ja+n-1) part of the local array A must contain the local pieces of the leading ia+m-1 by ja+n-1 part of the global matrix.
Scope: **local**
Specified as: an LLD_A by (at least) LOCq(N_A) array, containing numbers of the data type indicated in Table 1. Details about the block-cyclic data distribution of global matrix *A* are stored in desc_a.

`ia`

is the row index of the global matrix *A*, identifying the first row of the submatrix *A*.
Scope: **global**
Specified as: a fullword integer; 1≤ia≤M_A and ia+m-1≤M_A.

`ja`

is the column index of the global matrix *A*, identifying the first column of the submatrix *A*.
Scope: **global**
Specified as: a fullword integer; 1 ≤ ja ≤ N_A and ja+n-1 ≤ N_A.

`desc_a`

is the array descriptor for global matrix A,

`tau`

See On Return.

`work`

has the following meaning:
If lwork = 0, work is ignored.
If lwork ≠ 0, work is the work area used by this subroutine, where:
If lwork ≠ -1, its size is (at least) of length lwork.
If lwork = -1, its size is (at least) of length 1.
Scope: **local**
Specified as: an area of storage containing numbers of data type indicated in Table 1.

`lwork`

is the number of elements in array WORK.
Scope:
If lwork ≥ 0, lwork is **local**
If lwork = -1, lwork is **global**
Specified as: a fullword integer; where:
If lwork = 0, PDGEQRF and PZGEQRF dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the ScaLAPACK standard.
If lwork = -1, PDGEQRF and PZGEQRF perform a work area query and return the minimum size of work in work1. No computation is performed and the subroutine returns after error checking is complete.
Otherwise, it must have the following value:
lwork ≥ nb (mp0 + nq0 + nb)

where:
mb = MB_A
nb = NB_A
iroff = mod(ia-1, mb)
icoff = mod(ja-1, nb)
iarow = mod(RSRC_A + (ia-1)/mb, nprow)
iacol = mod(CSRC_A + (ja-1)/nb, npcol)
mp0 = NUMROC(m+iroff, mb, myrow, iarow, nprow)
nq0 = NUMROC(n+icoff, nb, mycol, iacol, npcol)

`info`

See On Return.

**On Return**

`a`

is the updated local part of the global general matrix *A*, containing the results of the computation. The elements on and above the diagonal of A[*ia:ia+m-1, ja:ja+n-1]* contain the *min(m, n)* × *n* upper trapezoidal matrix *R* (*R* is upper triangular if $m \geq n$). The elements below the diagonal with the array TAU represent the matrix *Q* as a product of elementary reflectors. (see *Further Details)*.
Scope: **local**
Returned as: an LLD_A by (at least) LOCq(N_A) array, containing numbers of the data type indicated in Table 1. Details about the block-cyclic data distribution of global matrix *A* are stored in desc_a.

`tau`

is the updated local part of the global matrix *tau*, where: *tau*(*ja:ja+min(m, n)-1)* contains the scalar factors of the elementary reflectors. This identifies the **first element** of the local array *tau*. This subroutine computes the location of the first element of the local subarray used, based *on ja, desc_a, p, q, myrow*, and *myco*l; therefore, the leading 1 by *LOCq(ja+min(m, n)-1)* part of the local array tau must contain the local pieces of the leading 1 by *ja+min(m, n)-1* part of the global matrix *tau*.
A copy of the vector *tau*, with a block size of NB_A and global index ja, is returned to each row of the process grid. The process column over which the first column of *tau* is distributed is CSRC_A.
Scope: **local**
Returned as: a 1 by (at least) LOCq(ja+min(m, n)-1) array, containing numbers of the data type indicated in Table 1.

`work`

is the work area used by this subroutine if lwork ≠ 0, where:
If lwork ≠ 0 and lwork ≠ -1, its size is (at least) of length lwork.
If lwork = -1, its size is (at least) of length 1.
Scope: **local**
Returned as: an area of storage, where:
If lwork ≥ 1 or lwork = -1, then work1 is set to the minimum lwork value and contains numbers of the data type indicated in Table 1. Except for work1, the contents of work are overwritten on return.

`info`

indicates that a successful computation occurred.

Scope: **global**
Returned as: a fullword integer; info = 0.

### *Notes and Coding Rules*

1. In your C program, argument `info` must be passed by reference.
2. Matrix *`A`*, `tau`, and *`work`* must have no common elements; otherwise, results are unpredictable.
3. The *`NUMROC`* utility subroutine can be used to determine the values of *`LOCp(M_)`* and *`LOCq(N_)`* used in the argument descriptions above..
4. There is no array descriptor for `tau`. `tau` is a row-distributed vector with block size *`NB_A`*, local array of dimension 1 by *`LOCq(ja+min(m, n)-1)`*, and global index *`ja`*. A copy of *`tau`* exists on each row of the process grid, and the process column over which the first column of `tau` is distributed is `CSRC_A`.
5. If `lwork=-1` on any process, it must equal -1 on all processes. That is, if a subset of the processes specifies -1 for the work area size, they must all specify -1.

*FURTHER DETAILS*
The matrix *Q* is represented as a product of elementary reflectors *Q = H(ja) H(ja+1) . . . H(ja+k-1)*, where *k = min(m,n)*. Each *H(i)* has the form *H(j) = I - tau * v * v'* where *tau* is a real scalar, and *v* is a real vector with *v(1:i-1) = 0* and *v(i) = 1; v(i+1:m)* is stored on exit in *A(ia+i:ia+m-1,ja+i-1)*, and *tau* in *TAU(ja+i-1)*.

*Example 2.12 The using of function* `pdgeqrf`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <sys/time.h>
#include "mpi.h"
using namespace std;
#define A(i,j) A[(i)*m+(j)]
#define Q(i,j) Q[(i)*m+(j)]
#define U(i,j) U[(i)*m+(j)]
#define A distr(i,j) A distr[(i)*k+(j)]
#define Q distr(i,j) Q distr[(i)*k+(j)]
#define U distr(i,j) U distr[(i)*k+(j)]
static int MAX( int a, int b ){
        if (a>b) return(a); else return(b);
}
static int MIN( int a, int b ){
        if (a<b) return(a); else return(b);
}
extern "C"
{
void Cblacs pinfo( int* mypnum, int* nprocs);
void Cblacs_get( int context, int request, int* value);
```

```
int  Cblacs gridinit( int* context, char * order, int np row, int np col);
void Cblacs gridinfo( int context, int*  np row, int* np col, int*  my row,
int*  my col);
void Cblacs gridexit( int context);
void Cblacs barrier(int, const char*);
void Cblacs exit( int error code);
void Cblacs pcoord(int, int, int*, int*);
int  numroc ( int *n, int *nb, int *iproc, int *isrcproc, int *nprocs);
int indxl2g (int*, int*, int*, int*, int*);
void descinit (int *desc, int *m, int *n, int *mb, int *nb, int *irsrc, int
*icsrc, int *ictxt, int *lld, int *info);
void pdgeadd (char *TRANS,int *M, int *N,double * AQPHA,double *A,int *IA,int
*JA,int *DESCA,double *BETA,double *C,
              int *IC,int *JC,int *DESCC);
void pdgeqrf (int *m, int*n, double *A distr, int* i one, int* i one, int*
descA, double *tau, double *work,
              int *lwork, int *info);
void pdorgqr (int *m, int *n, int *k, double *a, int *ia, int *ja, int
*desca, double *tau, double *work,
              int *lwork, int *info);
void pdgemm ( char *TRANSA,char *TRANSB,int *M,int *N,int *K,double
*AQPHA,double *A,int *IA,int *JA,int *DESCA,
             double * B, int * IB, int * JB, int * DESCB,double * BETA,double
* C, int * IC, int * JC, int * DESCC );

} // extern "C"
int main(int argc, char **argv) {
// Useful constants
int i one = 1, i negone = -1, i zero = 0;
double zero=0.0E+0, one=1.0E+0;
int descA[9],descQ[9],descU[9],descA distr[9],descQ distr[9],descU distr[9];
int iam,nprocs,nprow,npcol,myrow,mycol;
int m,n,mb,nb,mp,nq;
int i, j,mypnum;
int lld,lld distr;
int ictxt,info,lwork;
      int iloc,jloc;
      int ZERO=0,ONE=1;
      double alpha, beta;
m=6; n=4;
mb=2; nb=2;
nprow=2; npcol=2; // Astfel, programul se executa pe 4 procese
lwork=-1;//MAX(1,n); //HB
double *A, *Q,*U, *A distr,*Q distr,*U distr,*work,*tau;
// Part with invoking of ScaQAPACK routines. Initialize process grid, first
Cblacs pinfo(&iam,&nprocs);
Cblacs get( -1, 0, &ictxt );
Cblacs gridinit(&ictxt, "R", nprow, npcol );
Cblacs gridinfo(ictxt, &nprow, &npcol, &myrow, &mycol );
// Matricea A se initializeaza numai pentru procesul cu rankul 0
//if ( iam==0 )
  {//#1
A = new double[m*n];
Q = new double[m*n];
U = new double[m*n];
tau = new double[m];
work = new double[n];
/*A =  (double*)malloc(m*n*sizeof(double));
```

```
Q =  (double*)malloc(m*n*sizeof(double));
U = (double*)malloc(m*n*sizeof(double));
tau=(double*)malloc(m*sizeof(double));
work=(double*)malloc(n*sizeof(double));
//double *tau = new double[MIN(m,n)];
//double *work = new double[lwork];
//input matrix A here
/*for(i=0;i<m;i++ )
     for(j=0;j<n;j++)
            A[i*m+j]=(i+j);
           } */
A(0,0)=1.44;  A(0,1)=-7.84;  A(0,2)=-4.39;  A(0,3)= 4.53;
A(1,0)=-9.96;  A(1,1)=-0.28;  A(1,2)=-3.24;  A(1,3)= 3.83;
A(2,0)= -7.55;  A(2,1)= 3.24;  A(2,2)= 6.27;  A(2,3)=-6.64;
A(3,0)= 8.34;  A(3,1)= 8.09;  A(3,2)= 5.28;  A(3,3)= 2.06;
A(4,0)=7.08;  A(4,1)= 2.52;  A(4,2)= 0.74;  A(4,3)=-2.47;
A(5,0)=-5.45; A(5,1)=-5.70;  A(5,2)=-1.19; A(5,3)= 4.70;
if ( iam==0 )     {
 printf("============ REZULT OF THE PROGRAM %s \n",argv[0]);
  cout << "Global matrix A:\n";
              for (i = 0; i < m; ++i) {
                  for (j = 0; j < n; ++j) {
                      cout << setw(3) << A(i,j) << "   "; //*(A + m*i + j) <<
" ";
                                          }
                  cout << "\n";
              }
              cout << endl;   }
  }//#1
/*          else {
            A = NULL;

            Q = NULL;
            U = NULL;
               };
*/
//other processes don't contain parts of A, Q and U

Cblacs barrier(ictxt, "All");

// Compute dimensions of local part of distributed matrix A distr
mp = numroc ( &m, &mb, &myrow, &i zero, &nprow );
nq = numroc ( &n, &nb, &mycol, &i zero, &npcol );
A distr = (double *)calloc(mp*nq,sizeof(double)) ;
if (A distr==NULL){ printf("error of memory allocation A distr on proc
%dx%d\n",myrow,mycol); exit(0); }
Q distr = (double *)calloc(mp*nq,sizeof(double)) ;
if (Q distr==NULL){ printf("error of memory allocation Q distr on proc
%dx%d\n",myrow,mycol); exit(0); }
U distr = (double *)calloc(mp*nq,sizeof(double)) ;
if (U distr==NULL){ printf("error of memory allocation U distr on proc
%dx%d\n",myrow,mycol); exit(0); }
// Initialize discriptors (local matrix A is considered as distributed with
blocking parameters
// m, n,i.e. there is only one block - whole matrix A - which is located on
process (0,0) )

lld = MAX( numroc_( &m, &mb, &myrow, &i_zero, &nprow ), 1 );
```

```
descinit (descA, &m, &n, &mb, &nb, &i zero, &i zero, &ictxt, &lld, &info );

descinit (descQ, &m, &n, &mb, &nb, &i zero, &i zero, &ictxt, &lld, &info );
descinit (descU, &m, &n, &mb, &nb, &i zero, &i zero, &ictxt, &lld, &info );
lld distr = MAX( mp, 1);
descinit (descA distr, &m, &n, &mb, &nb, &i zero, &i zero, &ictxt,
&lld distr, &info );
descinit (descQ distr, &m, &n, &mb, &nb, &i zero, &i zero, &ictxt,
&lld distr, &info );
descinit (descU distr, &m, &n, &mb, &nb, &i zero, &i zero, &ictxt,
&lld distr, &info );
/*
// Atentie! descA distr practic este descriptorul matricei glabale daca ea ar
fi fost difizata in blocuri mb*nb

// Call pdgeadd  to distribute matrix (i.e. copy A into A distr)
pdgeadd ( "N", &m, &n, &one, A, &i one, &i one, descA, &zero, A distr,
&i one, &i one, descA distr );
// Tipar A distr
//for (int id = 0; id < nprocs; ++id)
{
//Cblacs barrier(ictxt, "All");
if  (iam) //(id == iam)
  {
cout << "A distr " << iam << endl;
for (i = 0; i < mp; ++i)
      {
  for (j = 0; j < nq; ++j)
  cout << setw(3) << *(A distr+mp*j+i) << " "; //A distr(i,j) << " ";
  cout << endl;
      }
//Cblacs barrier(ictxt, "All");
  }
}
*/
//=== se complecteaza cu valori matricele locale
for(iloc=0;iloc<mp;iloc++)
for(jloc=0;jloc<nq;jloc++){
int fortidl = iloc + 1;
int fortjdl = jloc + 1;
i = indxl2g (&fortidl, &nb, &myrow, &i zero, &nprow)-1;
j = indxl2g (&fortjdl, &nb, &mycol, &i zero, &npcol)-1;
A distr[iloc*mp+jloc]=A(i,j);
                            }
// === se tiparesc matricele locale
sleep(iam);
printf("For process (%2d%2d) np=%2d,nq %2d and
A distr:\n",myrow,mycol,mp,nq);
for (i = 0; i < mp; ++i)
 {
  for (j = 0; j < nq; ++j)
  cout << setw(3) << *(A distr+mp*i+j) << "   ";
  cout << endl;
 }
pdgeqrf (&m, &n, A distr, &i one, &i one, descA, tau, work, &lwork, &info);
//pdgeqrf (&m, &n, A distr, &i one, &i one, descA distr, tau, work, 0,
&info);
```

```
pdorgqr (&m, &n, &n, A distr, &i one, &i one, descA, tau, work, &lwork,
&info);

// Tipar Q distr  dupa factorizare

sleep(iam);
cout << "Q distr dupa QR factorizare " << iam << endl;
for (i = 0; i < mp; ++i)
 {
  for (j = 0; j < nq; ++j)
  cout << setw(3) << *(A distr+mp*i+j) << " ";
  cout << endl;
  }
Cblacs barrier(ictxt, "All");


// Copy result into local matrix (adica "restabilirea matrice A)
pdgeadd ( "N", &m, &n, &one, A distr, &i one, &i one, descA distr, &zero, A,
&i one, &i one, descA );
// Tipar A
if (iam==0)
  {
   cout << "Global matrix A (dupa QR factorizare):\n";
              for (i = 0; i < m; ++i) {
                  for (j = 0; j < n; ++j) {
                      cout << setw(3) << *(A + m*i + j) << " ";
                  }
                  cout << "\n";
              }
              cout << endl;



 }

delete( A distr );
delete( Q distr );
delete( U distr );
delete( A );delete( Q );delete( U );
/*
if( myrow==0 && mycol==0 ){
free( A );free( Q );free( U );
                     }
*/
// End of ScaQAPACK part. Exit process grid.
Cblacs gridexit(ictxt );
//Cblacs exit( 0);

}
```

Program's result

```
/*
[Hancu B S@hpc ScaLAPACK Exemple Curs Online]$ clear
[Hancu B S@hpc ScaLAPACK Exemple Curs Online]$ ./mpiCC ScL -o Example2.12.exe
Example2.12.cpp
[Hancu B S@hpc ScaLAPACK Exemple Curs Online]$ /opt/openmpi/bin/mpirun -n 4 -
host compute-0-0  Example2.12.exe
```

```
============ REZULT OF THE PROGRAM Example2.12.exe
Global matrix A:
1.44   -7.84   -4.39   4.53
-9.96  -0.28   -3.24   3.83
-7.55   3.24   6.27   -6.64
8.34    8.09   5.28   2.06
7.08    2.52   0.74   -2.47
-5.45  -5.7   -1.19   4.7

For process ( 0 0) np= 4,nq  2 and A distr:
1.44   -7.84
-9.96   -0.28
7.08   2.52
-5.45   -5.7
For process ( 0 1) np= 4,nq  2 and A distr:
-4.39   4.53
-3.24   3.83
0.74   -2.47
-1.19   4.7
For process ( 1 0) np= 2,nq  2 and A distr:
-7.55   3.24
8.34   8.09
For process ( 1 1) np= 2,nq  2 and A distr:
6.27   -6.64
5.28   2.06
Q distr dupa QR factorizare 0
1.44 -7.84
-9.96 -0.28
7.08 2.52
-5.45 -5.7
Q distr dupa QR factorizare 1
-4.39 4.53
-3.24 3.83
0.74 -2.47
-1.19 4.7
Q distr dupa QR factorizare 2
-7.55 3.24
8.34 8.09
Q distr dupa QR factorizare 3
6.27 -6.64
5.28 2.06
Global matrix A (dupa QR factorizare):
1.44  -7.84   0   0
  0   0 -3.24 3.83
-7.55 3.24 6.27 -6.64
8.34 8.09 5.28 2.06
7.08 2.52 0.74 -2.47
-5.45 -5.7 -1.19 4.7

[Hancu_B_S@hpc ScaLAPACK_Exemple_Curs_Online]$
```

## Helpful routines

```
call psorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

**Purpose:** Generates the orthogonal matrix Q of the QR factorization formed by pvgeqrf.

The pvorgqr routine generates the whole or part of m-by-n real distributed matrix Q denoting A(ia:ia+m-1, ja:ja+n-1) with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m Q= H(1)*H(2)*…*H(k) as returned by pvgeqrf.

***Input Parameters***

`m (global) INTEGER`

The number of rows in the submatrix sub(Q) (m ⩾ 0).

`n (global) INTEGER.`

The number of columns in the submatrix sub(Q) (m ⩾ n ⩾ 0).

`k (global) INTEGER`

The number of elementary reflectors whose product defines the matrix Q (n ⩾ k ⩾ 0).

`a (local) REAL for psorgqr DOUBLE PRECISION for pdorgqr`

Pointer into the local memory to an array of local dimension (lld_a, LOCc(ja+n-1)). The j-th column must contain the vector which defines the elementary reflector H(j), ja⩽j⩽ja +k-1, as returned by p?geqrf in the k columns of its distributed matrix argument A(ia:*, ja:ja+k-1).

`ia, ja (global) INTEGER.`

The row and column indices in the global array a indicating the first row and the first column of the submatrix A(ia:ia+m-1,ja:ja+n-1), respectively.

`desca (global and local) INTEGER array, dimension (dlen_).`

The array descriptor for the distributed matrix A.

`tau (local) REAL for psorgqr DOUBLE PRECISION for pdorgqr Array, DIMENSION LOCc(ja+k-1).`

Contains the scalar factor tau (j) of elementary reflectors H(j) as returned by p?geqrf. tau is tied to the distributed matrix A.

`work (local) REAL for psorgqr DOUBLE PRECISION for pdorgqr`

Workspace array of dimension of lwork.

`lwork (local or global) INTEGER, dimension of work.`

Must be at least lwork ⩾ nb_a*(nqa0 + mpa0 + nb_a), where
iroffa = mod(ia-1, mb_a), icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL);
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If lwork = -1, then lwork is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

`a`

Contains the local pieces of the m-by-n distributed matrix Q.

`work(1)`

On exit, work(1) contains the minimum value of lwork required for optimum performance.

`Info (global) INTEGER.`

= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-entry had an illegal value, then info = - (i* 100+j), if the i-th argument is a scalar and had an illegal value, then info = -i.