Title

stata.com

qrd( ) — QR decomposition

Description

`qrd(`*A*`,` *Q*`,` *R*`)` calculates the QR decomposition of *A*: $m \times n$, returning results in *Q* and *R*.

`hqrd(`*A*`,` *H*`,` *tau*`,` R_1`)` calculates the QR decomposition of *A*: $m \times n$ but, rather than returning *Q* and *R*, returns the Householder vectors in *H* and the scale factors *tau*—from which *Q* can be formed—and returns in R_1 an upper-triangular matrix that is a submatrix of *R* when $m \geq n$ or an upper-trapezoidal matrix when $m < n$; see *Remarks and examples* below for its definition. Doing this saves calculation and memory, and other routines allow you to manipulate these matrices:

1. `hqrdmultq(`*H*`,` *tau*`,` *X*`,` *transpose*`)` returns *QX* or $Q'X$ on the basis of the *Q* implied by *H* and *tau*. *QX* is returned if *transpose* $= 0$, and $Q'X$ is returned otherwise.
2. `hqrdmultq1t(`*H*`,` *tau*`,` *X*`)` returns $Q_1'X$ on the basis of the $Q1$ implied by *H* and *tau*. This function requires $m \geq n$.
3. `hqrdq(`*H*`,` *tau*`)` returns the *Q* matrix implied by *H* and *tau*. This function is rarely used.
4. `hqrdq1(`*H*`,` *tau*`)` returns the Q_1 matrix implied by *H* and *tau*. This function requires $m \geq n$ and is rarely used.
5. `hqrdr(`*H*`)` returns the full *R* matrix. This function is rarely used. (It may surprise you that `hqrdr()` is a function of *H* and not R_1. R_1 also happens to be stored in *H*, and there is other useful information there as well.)
6. `hqrdr1(`*H*`)` returns the R_1 matrix. This function is rarely used.
7. `hqrdmultq(`*H*`,` *tau*`,` *X*`,` *transpose*`,` *transform*`)` and `hqrdmultq1t(`*H*`,` *tau*`,` *X*`,` *transform*`)` do the same things as `hqrdmultq(`*H*`,` *tau*`,` *X*`,` *transpose*`)` and `hqrdmultq1t(`*H*`,` *tau*`,` *X*`)`. The difference is that the *transform* argument allows you to transform the input matrices. *transform* can be specified as 0 or 1. When *transform* is specified as 1, the functions will transform *H*, *tau*, and *X* all to complex matrices if any of these matrices is complex. Not specifying *transform* is equivalent to specifying *transform* $= 0$; in this case, the storage type of the input matrices does not change.

`_hqrd(`*A*`,` *tau*`,` R_1`)` does the same thing as `hqrd(`*A*`,` *H*`,` *tau*`,` R_1`)`, except that it overwrites *H* into *A* and so conserves even more memory.

`qrdp(`*A*`,` *Q*`,` *R*`,` *p*`)` is similar to `qrd(`*A*`,` *Q*`,` *R*`)`: it returns the QR decomposition of *A* in *Q* and *R*. The difference is that this routine allows for pivoting. New argument *p* specifies whether a column is available for pivoting, and on output, *p* is overwritten with a permutation vector that records the pivoting actually performed. On input, *p* can be specified as `.` (missing)—meaning all columns are available for pivoting—or *p* can be specified as a $1 \times n$ row vector containing 0s and 1s, with 1 meaning the column is fixed and so may not be pivoted.

`hqrdp(`*A*`,` *H*`,` *tau*`,` R_1`,` *p*`)` is a generalization of `hqrd(`*A*`,` *H*`,` *tau*`,` R_1`)` just as `qrdp()` is a generalization of `qrd()`.

_hqrdp(*A*, *tau*, R_1, *p*) does the same thing as hqrdp(*A*, *H*, *tau*, R_1, *p*), except that _hqrdp() overwrites *H* into *A*.

_hqrdp_la() is the interface to the LAPACK routine that performs the QR calculation; it is used by all the above routines. Direct use of _hqrdp_la() is not recommended.

Syntax

void qrd(*numeric matrix A*, *Q*, *R*)

void hqrd(*numeric matrix A*, *H*, *tau*, R_1)

void _hqrd(*numeric matrix A*, *tau*, R_1)

numeric matrix hqrdmultq(*numeric matrix H*, *rowvector tau*, *numeric matrix X*, *real scalar transpose*)

numeric matrix hqrdmultq(*numeric matrix H*, *rowvector tau*, *numeric matrix X*, *real scalar transpose*, *real scalar transform*)

numeric matrix hqrdmultq1t(*numeric matrix H*, *rowvector tau*, *numeric matrix X*)

numeric matrix hqrdmultq1t(*numeric matrix H*, *rowvector tau*, *numeric matrix X*, *real scalar transform*)

numeric matrix hqrdq(*numeric matrix H*, *numeric matrix tau*)

numeric matrix hqrdq1(*numeric matrix H*, *numeric matrix tau*)

numeric matrix hqrdr(*numeric matrix H*)

numeric matrix hqrdr1(*numeric matrix H*)

void qrdp(*numeric matrix A*, *Q*, *R*, *real rowvector p*)

void hqrdp(*numeric matrix A*, *H*, *tau*, R_1, *real rowvector p*)

void _hqrdp(*numeric matrix A*, *tau*, R_1, *real rowvector p*)

void _hqrdp_la(*numeric matrix A*, *tau*, *real rowvector p*)

Remarks and examples

stata.com

Remarks are presented under the following headings:

QR decomposition

The decomposition of square or nonsquare matrix A can be written as

$$A = QR \tag{1}$$

where Q is an orthogonal matrix ($Q'Q = I$) and R is upper triangular or upper trapezoidal. A matrix is an upper-trapezoidal matrix if its nonzero elements are found only in the upper triangle of the matrix, including the main diagonal. `qrd(A, Q, R)` will make this calculation:

```
: A
       1   2
    ┌──────────┐
  1 │   4   8  │
  2 │   4   6  │
  3 │   1   0  │
  4 │   2   4  │
  5 │   2   1  │
    └──────────┘

: Q = R = .
: qrd(A, Q, R)
: Ahat = Q*R
: mreldif(Ahat, A)
  2.53765e-16
```

Avoiding calculation of Q

In fact, you probably do not want to use `qrd()`. Calculating the necessary ingredients for Q is not too difficult, but going from those necessary ingredients to form Q is devilish. The necessary ingredients are usually all you need, which are the Householder vectors and their scale factors, known as H and *tau*. For instance, one can write down a mathematical function $f(H, tau, X)$ that will calculate QX or $Q'X$ for some matrix X.

Also, QR decomposition is often carried out on violently nonsquare matrices A: $m \times n$, $m \gg n$. We can write

$$\underset{m\times n}{A} = \begin{bmatrix} \underset{m\times n}{Q_1} & \underset{m\times m-n}{Q_2} \end{bmatrix} \begin{bmatrix} \underset{n\times n}{R_1} \\ \underset{m-n\times n}{R_2} \end{bmatrix} = \underset{m\times n}{Q_1 R_1} + \underset{m\times n}{Q_2 R_2}$$

R_2 is zero, and thus

$$\underset{m\times n}{A} = \begin{bmatrix} \underset{m\times n}{Q_1} & \underset{m\times m-n}{Q_2} \end{bmatrix} \begin{bmatrix} \underset{n\times n}{R_1} \\ \underset{m-n\times n}{0} \end{bmatrix} = \underset{m\times n}{Q_1 R_1}$$

Thus, it is enough to know Q_1 and R_1. Rather than defining QR decomposition as

$$A = QR \qquad Q : m \times m \quad R : m \times n \tag{1}$$

We can better define it as

$$A = Q_1 R_1 \qquad Q_1 : m \times n \quad R_1 : n \times n \tag{1'}$$

To appreciate the savings, consider the reasonable case where $m = 4{,}000$ and $n = 3$,

$$A = QR \qquad Q : 4{,}000 \times 4{,}000 \quad R : 4{,}000 \times 3$$

versus

$$A = Q_1 R_1 \qquad Q_1 : 4{,}000 \times 3 \qquad R_1 : 3 \times 3$$

Memory consumption is reduced from 125,094 kilobytes to 94 kilobytes, a 99.92% saving!

Combining the arguments, we need not save Q because Q_1 is sufficient, we need not calculate Q_1 because H and *tau* are sufficient, and we need not store R because R_1 is sufficient.

That is what `hqrd(`*A*`, `*H*`, `*tau*`, `*R1*`)` does. Having used `hqrd()`, if you need to multiply the full Q by some matrix X, you can use `hqrdmultq()`. Having used `hqrd()`, if you need the full Q, you can use `hqrdq()` to obtain it, but by that point, you will be making the devilish calculation you sought to avoid, and so you might as well have used `qrd()` to begin with. If you want Q_1, you can use `hqrdq1()`. Finally, having used `hqrd()`, if you need R or R_1, you can use `hqrdr()` and `hqrdr1()`:

```
: A
       1   2
  +-----------+
1 |   4   8   |
2 |   4   6   |
3 |   1   0   |
4 |   2   4   |
5 |   2   1   |
  +-----------+

: H = tau = R1 = .
: hqrd(A, H, tau, R1)
: Ahat = hqrdq1(H, tau) * R1                  // i.e.,  Q1*R1
: mreldif(Ahat, A)
  2.53765e-16
```

Note that Q_1 is obtained only when $m \geq n$. When $m < n$, we can write

$$\underset{m\times n}{A} = \underset{m\times m}{Q} \quad \underset{m\times n}{R}$$

where R, which is also called R_1 for consistency when $m \geq n$, is upper trapezoidal. For example,

```
: B
       1   2   3
  +---------------+
1 |   4   4   1   |
2 |   8   6   0   |
  +---------------+

: Q = R = .
: qrd(B, Q, R)
: Bhat = Q * R
: mreldif(Bhat, B)
  4.44089e-16
: R
                  1              2              3
  +-------------------------------------------------+
1 |  -8.94427191   -7.155417528   -.4472135955     |
2 |             0    -.894427191    -.894427191     |
  +-------------------------------------------------+
```

Pivoting

The QR decomposition with column pivoting solves

$$AP = QR \tag{2}$$

or, if you prefer,

$$AP = Q_1 R_1 \tag{2'}$$

for $m \geq n$, where P is a permutation matrix; see [M-1] **Permutation**. We can rewrite this as

$$A = QRP' \tag{3}$$

and

$$A = Q_1 R_1 P' \tag{3'}$$

for $m \geq n$.

Column pivoting can improve the numerical accuracy. The functions `qrdp(`*A*`, `*Q*`, `*R*`, `*p*`)` and `hqrdp(`*A*`, `*H*`, `*tau*`, `*R1*`, `*p*`)` perform pivoting and return the permutation matrix P in permutation vector form:

```
: A
       1   2
    ┌──────────
  1 │  4   8
  2 │  4   6
  3 │  1   0
  4 │  2   4
  5 │  2   1
    └──────────

: Q = R = p = .
: qrdp(A, Q, R, p)
: Ahat = (Q*R)[., invorder(p)]                  // i.e., QRP'
: mreldif(Ahat, A)
  1.97373e-16
: H = tau = R1 = p = .
: hqrdp(A, H, tau, R1, p)
: Ahat = (hqrdq1(H, tau)*R1)[., invorder(p)]    // i.e., Q1*R1*P'
: mreldif(Ahat, A)
  1.97373e-16
```

Before calling `qrdp()` or `hqrdp()`, we set p equal to missing, specifying that all columns could be pivoted. We could just as well have set p equal to (0, 0), which would have stated that both columns were eligible for pivoting.

When pivoting is disallowed, and when A is not of full-column rank, the order in which columns appear affects the kind of generalized solution produced; later columns are, in effect, omitted. When pivoting is allowed, the columns are reordered based on numerical accuracy considerations. In the rank-deficient case, you no longer know ahead of time which columns will be omitted, because you do not know in what order the columns will appear. Generally, you do not care, but there are occasions when you do.

In such cases, you can specify which columns are eligible for pivoting and which are not—you specify p as a vector, and if p_i==1, the ith column may not be pivoted. The p_i==1 columns are (conceptually) moved to appear first in the matrix, and the remaining columns are ordered optimally after that. The permutation vector that is returned in p accounts for all of this.

Least-squares solutions with omitted columns

Least-square solutions are one popular use of QR decomposition. We wish to solve for x

$$Ax = b \qquad (A : m \times n, \quad m \geq n) \tag{4}$$

The problem is that there is no solution to (4) when $m > n$ because we have more equations than unknowns. Then, we want to find x such that $(Ax - b)'(Ax - b)$ is minimized.

If A is of full-column rank, then it is well known that the least-squares solution for x is given by `solveupper(`R_1`,` $Q_1'b$`)` where `solveupper()` is an upper-triangular solver; see [M-5] **solvelower( )**.

If A is of less than full-column rank and we do not care which columns are omitted, then we can use the same solution: `solveupper(`R_1`,` $Q_1'b$`)`.

Adding pivoting to the above hardly complicates the issue; the solution becomes `solveupper(`R_1`,` $Q_1'b$`)[invorder(`p`)]`.

For both cases, the full details are

```
: A
        1   2   3
    +-------------+
  1 |  3   9   1  |
  2 |  3   8   1  |
  3 |  3   7   1  |
  4 |  3   6   1  |
    +-------------+

: b
        1
    +---------+
  1 |   7     |
  2 |   3     |
  3 |  12     |
  4 |   0     |
    +---------+

: H = tau = R1 = p = .
: hqrdp(A, H, tau, R1, p)
: q1b = hqrdmultq1t(H, tau, b)                  // i.e., Q1'b
: xhat = solveupper(R1, q1b)[invorder(p)]
: xhat
                    1
    +---------------+
  1 | -1.166666667  |
  2 |          1.2  |
  3 |            0  |
    +---------------+
```

The A matrix in the above example has less than full-column rank; the first column contains a variable with no variation, and the third column contains the data for the intercept. The solution above is correct, but we might prefer a solution that included the intercept. To do that, we need to specify that the third column cannot be pivoted:

```
: p = (0, 0, 1)
: H = tau = R1 = .
: hqrdp(A, H, tau, R1, p)
: q1b = hqrdmultq1t(H, tau, b)
```

```
: xhat = solveupper(R1, q1b)[invorder(p)]
: xhat
          1
    ┌──────┐
  1 │    0 │
  2 │  1.2 │
  3 │ -3.5 │
    └──────┘
```

Conformability

`qrd(`A`, `Q`, `R`)`:

input:

A:	$m \times n$	

output:

Q:	$m \times m$	
R:	$m \times n$	

`hqrd(`A`, `H`, `*tau*`, `R_1`)`:

input:

A:	$m \times n$	

output:

H:	$m \times n$	
tau:	$1 \times n$	
R_1:	$n \times n$	

`_hqrd(`A`, `*tau*`, `R_1`)`:

input:

A:	$m \times n$	

output:

A:	$m \times n$	(contains H)
tau:	$1 \times n$	
R_1:	$n \times n$	

`hqrdmultq(`H`, `*tau*`, `X`, `*transpose*`, `*transform*`)`:

H:	$m \times n$	
tau:	$1 \times n$	
X:	$m \times c$	
transpose:	1×1	
transform:	1×1	(optional)
result:	$m \times c$	

`hqrdmultq1t(`H`, `*tau*`, `X`, `*transform*`)`:

H:	$m \times n$,	$m \geq n$
tau:	$1 \times n$	
X:	$m \times c$	
transform:	1×1	(optional)
result:	$n \times c$	

`hqrdq(`*H*`,` *tau*`)`:

H:	$m \times n$	
tau:	$1 \times n$	
result:	$m \times m$	

`hqrdq1(`*H*`,` *tau*`)`:

H:	$m \times n$,	$m \geq n$
tau:	$1 \times n$	
result:	$m \times n$	

`hqrdr(`*H*`)`:

H:	$m \times n$
result:	$m \times n$

`hqrdr1(`*H*`)`:

H:	$m \times n$
result:	$n \times n$

`qrdp(`*A*`,` *Q*`,` *R*`,` *p*`)`:

input:

A:	$m \times n$	
p:	1×1	or $1 \times n$

output:

Q:	$m \times m$
R:	$m \times n$
p:	$1 \times n$

`hqrdp(`*A*`,` *H*`,` *tau*`,` R_1`,` *p*`)`:

input:

A:	$m \times n$	
p:	1×1	or $1 \times n$

output:

H:	$m \times n$
tau:	$1 \times n$
R_1:	$n \times n$
p:	$1 \times n$

`_hqrdp(`*A*`,` *tau*`,` R_1`,` *p*`)`:

input:

A:	$m \times n$	
p:	1×1	or $1 \times n$

output:

A:	$m \times n$ (contains *H*)
tau:	$1 \times n$
R_1:	$n \times n$
p:	$1 \times n$

_hqrdp_la(*A*, *tau*, *p*):

input:

A:	$m \times n$		
p:	1×1	or	$1 \times n$

output:

A:	$m \times n$	(contains *H*)
tau:	$1 \times n$	
p:	$1 \times n$	

Diagnostics

qrd(*A*, ...), hqrd(*A*, ...), _hqrd(*A*, ...), qrdp(*A*, ...), hqrdp(*A*, ...), and _hqrdp(*A*, ...) return missing results if *A* contains missing values. That is, *Q* will contain all missing values. *R* will contain missing values on and above the diagonal. *p* will contain the integers 1, 2, ...

_hqrd(*A*, ...) and _hqrdp(*A*, ...) abort with error if *A* is a view.

hqrdmultq(*H*, *tau*, *X*, *transpose*, *transform*) and hqrdmultq1t(*H*, *tau*, *X*, *transform*) return missing results if *X* contains missing values.

Vera Nikolaevna Kublanovskaya (1920–2012) was born in Krokhino, Russia, a small fishing village east of St. Petersburg. After finishing her secondary studies, Vera started her training to become a primary school teacher, but her grades were so outstanding that her mentors encouraged her to pursue a career in mathematics.

After graduation, she started working on computational algorithms for the Soviet nuclear program, from which she retired in 1955. She also participated in the development of numerical linear-algebra operations in the computer language PRORAB for the first electronic computer in the Soviet Union, BESM. In 1955, she received her PhD. She then published numerous papers, in particular on the topic of numerical linear algebra. Her most acclaimed contribution is as one of the inventors of the QR algorithm for computing eigenvalues of matrices.

Alston Scott Householder (1904–1993) was born in Rockford, Illinois, and grew up in Alabama. He studied philosophy at Northwestern and Cornell, and then mathematics, earning a doctorate in the calculus of variations from the University of Chicago. Householder worked on mathematical biology for several years at Chicago, but in 1946 he moved on to Oak Ridge National Laboratory, where he became the founding director of the Mathematics Division in 1948. There he moved into numerical analysis, specializing in linear equations and eigensystems and helping to unify the field through reviews and symposia. His last post was at the University of Tennessee.

Also see

[M-5] **qrinv( )** — Generalized inverse of matrix via QR decomposition

[M-5] **qrsolve( )** — Solve AX=B for X using QR decomposition

[M-4] **Matrix** — Matrix functions