

Title

[M-5] `qrd()` — QR decomposition

Syntax

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

void          hqrd(numeric matrix A, H, tau, R1)
void          _hqrd(numeric matrix A, tau, R1)

numeric matrix hqrdmultq(numeric matrix H, numeric matrix tau,
                          numeric matrix X, real scalar transpose)
numeric matrix hqrdmultq1t(numeric matrix H, numeric matrix tau,
                           numeric matrix X)
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 colvector p)
void          hqrdp(numeric matrix A, H, tau, R1, real colvector p)
void          _hqrdp(numeric matrix A, tau, R1, real colvector p)
void          _hqrdp_la(numeric matrix A, tau, real colvector p)
```

Description

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

`hqrd(A, H, tau, R1)` calculates the QR decomposition of $A: m \times n, m \geq n$, 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 an upper-triangular matrix in R_1 that is a submatrix of R ; see *Remarks* 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 $Q'X$ based on 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 $Q_1'X$ based on the Q_1 implied by H and tau .
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 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.

`_hqrd(A, tau, R1)` does the same thing as `hqrd(A, H, tau, R1)`, 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 an $n \times 1$ column vector containing 0s and 1s, with 1 meaning the column is fixed and so may not be pivoted.

`hqrpd(A, H, tau, R1, p)` is a generalization of `hqrd(A, H, tau, R1)` just as `qrdp()` is a generalization of `qrd()`.

`_hqrpd(A, tau, R1, p)` does the same thing as `hqrpd(A, H, tau, R1, p)`, except that `_hqrpd()` overwrites H into A .

`_hqrpd_la()` is the interface into the [M-1] **LAPACK** routine that performs the QR calculation; it is used by all the above routines. Direct use of `_hqrpd_la()` is not recommended.

Remarks

Remarks are presented under the headings

QR decomposition
Avoiding calculation of Q
Pivoting
Least-squares solutions with dropped columns

QR decomposition

The decomposition of square or nonsquare matrix A can be written

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

where Q is an orthogonal matrix ($Q'Q = I$), and R is upper triangular. `qrd(A, Q, R)` will make this calculation:

```
: A
      1  2
      1  7  4
      2  9  6
      3  9  6
      4  7  2
      5  3  1
: Q = R = .
: qrd(A, Q, R)
: Ahat = Q*R
: mreldif(Ahat, A)
4.44089e-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. In most cases, the necessary ingredients are all you need. Those necessary ingredients 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 .

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

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

It turns out that R_2 is zero, and thus

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

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

$$A = QR, \quad Q : m \times m, \quad R : m \times n \quad (1)$$

it is better to define it as

$$A = Q_1 R_1 \quad Q_1 : m \times n \quad R_1 : n \times n \quad (1')$$

To appreciate the savings, consider the reasonable case where $m = 4000$ and $n = 3$:

$$A = QR, \quad Q : 4000 \times 4000, \quad R : 4000 \times 3$$

versus,

$$A = Q_1 R_1 \quad Q_1 : 4000 \times 3 \quad R_1 : 3 \times 3$$

Memory consumption is reduced from 125,094 kilobytes to 94 kilobytes, a 99.92 percent 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 `hqrmultq()`. Having used `hqrd()`, if you need the full Q , you can use `hqrdaq()` 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 `hqrdaq1()`. Finally, having used `hqrd()`, if you need R or R_1 , you can use `hqrdr()` and `hqrdr1()`:

```

: A
      1  2
1  [ 7  4 ]
2  [ 9  6 ]
3  [ 9  6 ]
4  [ 7  2 ]
5  [ 3  1 ]

: H = tau = R1 = .
: hqrd(A, H, tau, R1)
: Ahat = hqrdq1(H, tau) * R1           // i.e., Q1*R1
: mreldif(Ahat, A)
  4.44089e-16

```

Pivoting

The QR decomposition with column pivoting solves

$$AP = QR \quad (2)$$

or, if you prefer

$$AP = Q_1 R_1 \quad (2')$$

where P is a permutation matrix; see [M-1] **permutation**. We can rewrite this as

$$A = QRP' \quad (3)$$

and

$$A = Q_1 R_1 P' \quad (3')$$

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  [ 7  4 ]
2  [ 9  6 ]
3  [ 9  6 ]
4  [ 7  2 ]
5  [ 3  1 ]

: Q = R = p = .
: qrdp(A, Q, R, p)
: Ahat = (Q*R)[., invorder(p)]       // i.e., QRP'
: mreldif(Ahat, A)
  3.55271e-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)
  3.55271e-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, dropped. 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 dropped, 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 i th 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 dropped columns

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

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

The problem is that there is no solution to (4) when $m > n$ because we have more equations than unknowns. In that case, 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] `solveupper()`.

If A is of less than full column rank and we do not care which columns are dropped, 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 of these 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

$\text{qrd}(A, Q, R)$:

input:

A : $m \times n$, $m \geq n$

output:

Q : $m \times m$

R : $m \times n$

$\text{hqrd}(A, H, \text{tau}, R_1)$:

input:

A : $m \times n$, $m \geq n$

output:

H : $m \times n$

tau : $1 \times n$

R_1 : $n \times n$

$\text{_hqrd}(A, \text{tau}, R_1)$:

input:

A : $m \times n$, $m \geq n$

output:

A : $m \times n$ (contains H)

tau : $1 \times n$

R_1 : $n \times n$

`hqrddmultq(H, tau, X, transpose):`

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

`hqrddmultqt(H, tau, X):`

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

`hqrddq(H, tau):`

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

`hqrddq1(H, tau):`

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

`hqrddr(H, tau):`

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

`hqrddr1(H, tau):`

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

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

input:

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

output:

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

`hqrddp(A, H, tau, R1, p):`

input:

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

output:

H: $m \times n$
tau: $1 \times n$
R₁: $n \times n$
p: $1 \times n$

`_hqrqp(A, tau, R1, p)`:

input:

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

output:

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

`_hqrqp_la(A, tau, p)`:

input:

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

output:

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

Diagnostics

`qrd(A, ...)`, `hqrqp(A, ...)`, `_hqrqp(A, ...)`, `qrdp(A, ...)`, `hqrqp(A, ...)`, and `_hqrqp(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, ...

`_hqrqp(A, ...)` and `_hqrqp(A, ...)` abort with error if A is a view.

`hqrqp_multq(H, tau, X, transpose)` and `hqrqp_multqt(H, tau, X)` return missing results if X contains missing values.

Also See

[M-5] `qrsolve()`, [M-5] `qrinv()`; [M-4] **matrix**