In Praise of QR

Jan de Leeuw

Version 003, March 21, 2016

Contents

Note: This is a working paper which will be expanded/updated frequently. Suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/matrix has a pdf copy of this article, the complete Rmd file with all code chunks, the bib file, and the R and C source code. We thank Bill Venables for some corrections in the handling of pivots in version 001.

1 Problem

Consider the linear system $Xb = y$, with X an $n \times m$ matrix. The function `solve()` in `R` handles the case in which X is square and non-singular, while `solve.qr()` handles rectangular systems, providing the least squares solution if appropriate. Of course a solution of $Xb = y$ may not exist, and if it exists it may not be unique. A *least squares solution*, which minimizes the sum of squares of the residuals $y - Xb$, always exists, but again it may not be unique. If $Xb = y$ is solvable, then of course the solutions are the least squares solutions, and the residual sum of squares is zero. In this note we provide software in `R` and `C` for computing a basis for the affine space of all solutions and/or all least squares solutions to the system (including the case where the solution set is empty). In addition we provide software to compute the Moore-Penrose inverse and to compute an orthonormal bases for the null-space of X.

2 The QR decomposition

2.1 Theory

At the core of our treatment of linear systems is the QR decomposition. We state it in a rank revealing form. This is theorem 1.3.4 in Björck (1996), but the proof can be found in most numerical linear algebra books.

Theorem 1: [**QR**] If the $n \times m$ matrix X has rank r then there exist

- An $n \times r$ matrix Q with $Q'Q = I$,
- An $r \times r$ upper-triangular matrix T with positive diagonal elements,
- An $r \times (m - r)$ matrix S,
- An $m \times m$ permutation matrix P,

such that

$$XP = QR, \tag{1}$$

with $R = (T \mid S)$.

The permutation matrix P is used to makes sure that the leading $n \times r$ submatrix of XP is of rank r. The theorem says the QR decomposition is a *full rank decomposition*, because both Q and $R := (T \mid S)$ are of rank $r = \mathbf{rank}(X)$.

We have written the QR decomposition the form in which it is computed by our code. Note that alternatively we can write it in a more verbose way as

$$X = \begin{bmatrix} Q & \mid & Q_\perp \end{bmatrix} \begin{bmatrix} T & \mid & S \\ -- & & -- \\ 0_{(n-r)\times r} & \mid & 0_{(n-r)\times(m-r)} \end{bmatrix} P', \tag{2}$$

with $Q_\perp$ an $n \times (n-r)$ orthonormal basis for the complement of the column space of X (i.e. the *left null space* of X, the set of all y such that $y'X = 0$).

2.2 Computation

In De Leeuw (2015) we computed the QR decomposition using the *modified Gram-Schmidt (MGS) algorithm*, which is described, for example, in section 2.4.2 of Björck (1996). The software uses the good old `.C()` interface in `R`. It is *exceedingly simple* because it can only handle the case where $n \geq m = r$, i.e. A has to be *tall* and *non-singular*. This is fine for some applications, but useless for what we have in mind in this paper.

We have consequently written a new MGS routine, which computes the rank-revealing QR decomposition from theorem 1. The `R` function `gsRC()` sets up the memory, passes the arguments using the `.C()` interface to the `C` function `gsC()`, and then extracts the output from the returned list. The function `gsC()` can be easily incorporated into both `R` and `C` projects. It passes by reference, does not use input and output routines, does not allocate or free memory, does not call `R` functions, does not use `C` arrays, and does not return a value. All I/O and memory management must be handled by the `C` or `R` calling program. This implies we also pass pointers to memory for the results. All `R` matrices are passed as vectors and the `C` routine takes into account they are in column-major order with index starting at one instead of zero. A simple `C` macro `MINDEX()` handles all array indexing.

The rank reducing QR decomposition requires *column-pivoting* to find an appropriate permutation matrix P. In MGS we handle the columns of X in order, so each column becomes the *working column* in turn. We start a counter for the *rank* at m and a counter for the *working number* at 1. The working column is projected on the orthogonal complement of the space spanned by the previous working columns. If the working column is linearly dependent on the previous columns, i.e. if the projection is zero, then we exchange it with a column with a higher column number and decrease the rank by one. The working column number stays the same. If the working column is independent of the previous ones we normalize the projection and add it to Q. The rank counter stays the same, the working number increases by one. In the MGS algorithm A is transformed in place, no extra storage for Q is needed.

Since the pivoting, and the resulting column permutations, may be confusing, we show the results of our function `gsRC()` on a singular example in some detail. The construction and manipulation of the permutation matrix P from the pivot sequence are also illustrated.

```
print (x <- matrix(c(1,1,1,1,1,-1,1,-1,2,0,2,0,1,-1,-1,1,0,2,0,2), 4, 5))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    2    1    0
## [2,]    1   -1    0   -1    2
## [3,]    1    1    2   -1    0
## [4,]    1   -1    0    1    2
```

```
print (h <- gsRC(x))
```

```
## $x
##      [,1] [,2] [,3]
## [1,]  0.5  0.5  0.5
## [2,]  0.5 -0.5 -0.5
## [3,]  0.5  0.5 -0.5
## [4,]  0.5 -0.5  0.5
##
## $r
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    0    0    2    2
## [2,]    0    2    0   -2    2
## [3,]    0    0    2    0    0
##
## $rank
## [1] 3
##
## $pivot
## [1] 1 2 4 5 3
```

```
print (p <- ifelse(outer(1:5, h$pivot, "=="), 1,0))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    0    0    1
## [4,]    0    0    1    0    0
## [5,]    0    0    0    1    0
```

```
x %*% p
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    0    2
## [2,]    1   -1   -1    2    0
## [3,]    1    1   -1    0    2
## [4,]    1   -1    1    2    0
```

```
h$x %*% h$r
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    0    2
## [2,]    1   -1   -1    2    0
## [3,]    1    1   -1    0    2
## [4,]    1   -1    1    2    0
```

```
x[, h$pivot]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    0    2
## [2,]    1   -1   -1    2    0
## [3,]    1    1   -1    0    2
## [4,]    1   -1    1    2    0
```

```
(h$x %*% h$r)[, order(h$pivot)]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    2    1    0
## [2,]    1   -1    0   -1    2
## [3,]    1    1    2   -1    0
## [4,]    1   -1    0    1    2
```

```
tcrossprod (h$x %*% h$r, p)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    2    1    0
## [2,]    1   -1    0   -1    2
## [3,]    1    1    2   -1    0
## [4,]    1   -1    0    1    2
```

2.3 Examples

2.3.1 Regular

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(lapply (gsRC (x), zapsmall), digits = 4)
```

```
## $x
##          [,1]    [,2]    [,3]     [,4]
## [1,]  0.48949 -0.7027  0.2543 -0.04908
## [2,]  0.59310  0.5679  0.5599  0.08871
## [3,] -0.09138 -0.1772  0.3475 -0.79043
## [4,] -0.37912 -0.3036  0.5817  0.56200
## [5,]  0.50651 -0.2452 -0.4034  0.22161
##
## $r
##       [,1]    [,2]   [,3]    [,4]
## [1,] 1.196 -0.8488 0.4097 -0.3691
## [2,] 0.000  1.9959 1.0742 -1.4321
## [3,] 0.000  0.0000 1.7221  0.1276
## [4,] 0.000  0.0000 0.0000  0.8394
##
## $rank
## [1] 4
##
## $pivot
## [1] 1 2 3 4
```

2.3.2 Singular

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
x[,3] <- x[,1] + x[,2]
print(lapply (gsRC (x), zapsmall), digits = 4)
```

```
## $x
##          [,1]    [,2]     [,3]
## [1,]  0.48949 -0.7027 -0.01029
## [2,]  0.59310  0.5679  0.17187
## [3,] -0.09138 -0.1772 -0.72921
## [4,] -0.37912 -0.3036  0.64304
## [5,]  0.50651 -0.2452  0.15846
##
## $r
##       [,1]    [,2]    [,3]   [,4]
## [1,] 1.196 -0.8488 -0.3691 0.3474
## [2,] 0.000  1.9959 -1.4321 1.9959
## [3,] 0.000  0.0000  0.8490 0.0000
##
## $rank
## [1] 3
##
## $pivot
## [1] 1 2 4 3
```

2.3.3 Broad

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(lapply (gsRC (t(x)), zapsmall), digits = 4)
```

```
## $x
##            [,1]     [,2]    [,3]     [,4]
## [1,]  0.28143  0.44828 -0.6526 -0.54221
## [2,] -0.87379 -0.03325 -0.4763  0.09223
## [3,] -0.05587  0.85010  0.1408  0.50436
## [4,]  0.39264 -0.27435 -0.5722  0.66567
##
## $r
##        [,1]    [,2]    [,3]     [,4]    [,5]
## [1,] 2.081 -0.8005 0.05967  0.53164  1.1330
## [2,] 0.000  2.0852 0.36622 -0.05908 -0.4178
## [3,] 0.000  0.0000 0.44481 -0.13676 -0.2341
## [4,] 0.000  0.0000 0.00000  1.22809 -0.5930
##
## $rank
## [1] 4
##
## $pivot
## [1] 1 2 3 4 5
```

2.3.4 Weird

```
x <- matrix(1, 1, 6)
print(lapply (gsRC (x), zapsmall), digits = 4)
```

```
## $x
##      [,1]
## [1,]    1
##
## $r
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    1    1    1    1    1
##
## $rank
## [1] 1
##
## $pivot
## [1] 1 3 4 5 6 2
```

3 Least Squares

3.1 Theory

Least squares solutions are minimizers of the sum of squares, which we write as **SSQ** $(y-Xb)$. For solvable systems $Xb=y$ the set of least squares solutions and the set of solutions to the system are obviously identical.

Theorem 2: [**LS**] Suppose X has QR decomposition QRP' of theorem 1. Then

- the minimum of **SSQ** $(y-Xb)$ is $y'(I-QQ')y$,
- the least squares solutions are all are of the form

$$b=P\left\{\begin{bmatrix}T^{-1}Q'y\\0\end{bmatrix}+\begin{bmatrix}-T^{-1}S\\I\end{bmatrix}v\right\}. \tag{3}$$

Proof: We can write $y-Xb=Q(Q'y-R\tilde{b})+(I-QQ')y$ with $\tilde{b}:=P'b$, and thus

$$\mathbf{SSQ}\ (y-Xb)=y'(I-QQ')y+\mathbf{SSQ}\ (Q'y-T\tilde{b}_1-S\tilde{b}_2).$$

The second term can be always made equal to zero by choosing $\tilde{b}_2$ arbitrarily and setting $\tilde{b}_1=T^{-1}(Q'y-Sb_2)$. **QED**

The theorem has a simple special case, in which the residual sum of squares is zero.

Corollary 1: [**Linear**] Suppose X has the QR decomposition QRP' of theorem 1. Then

- $Xb=y$ is solvable if and only if $QQ'y=y$.
- If $Xb=y$ is solvable, then all solutions are of the form

$$b=P\left\{\begin{bmatrix}T^{-1}Q'y\\0\end{bmatrix}+\begin{bmatrix}-T^{-1}S\\I\end{bmatrix}v\right\}. \tag{4}$$

Proof: Directly from theorem 2. **QED**

3.2 Computation

Our function `lsRC()` computes least squares solutions by using the result in theorem 2. It can, of course, also be used to solve consistent systems of linear equations.

3.3 Examples

3.3.1 Regular

```r
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(lapply (lsRC (x, rep(1, 5)), zapsmall), digits = 4)
```

```
## $solution
## [1]  0.09947 -0.82045  0.77524  0.03908
##
## $residuals
## [1] -0.49160  0.07219  0.50991  0.36487  0.75564
##
## $minssq
## [1] 1.211
##
## $nullspace
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
## [4,]    0
##
## $rank
## [1] 4
##
## $pivot
## [1] 1 2 3 4
```

3.3.2 Singular

```r
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
x[,3] <- x[,1] + x[,2]
print(lapply (lsRC (x, rep(1,5)), zapsmall), digits = 4)
```

```
## $solution
## [1]  0.8543 -0.2336  0.0000  0.2754
##
## $residuals
## [1] -0.1500  0.7852  1.1202  1.0124  0.1853
##
## $minssq
## [1] 2.953
##
## $nullspace
##      [,1]
## [1,]   -1
## [2,]   -1
## [3,]    1
## [4,]    0
##
```

```
## $rank
## [1] 3
##
## $pivot
## [1] 1 2 4 3
```

3.3.3 Broad

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(lapply (lsRC (t(x), rep(1,4)), zapsmall), digits = 4)
```

```
## $solution
## [1]  0.2368  1.0762 -3.3275  0.5863  0.0000
##
## $residuals
## [1] 0.00e+00 0.00e+00 2.22e-16 0.00e+00
##
## $minssq
## [1] 4.93e-32
##
## $nullspace
##          [,1]
## [1,] -0.65057
## [2,]  0.09553
## [3,]  0.67480
## [4,]  0.48286
## [5,]  1.00000
##
## $rank
## [1] 4
##
## $pivot
## [1] 1 2 3 4 5
```

3.3.4 Weird

```
x <- matrix(1, 1, 6)
print(lapply (lsRC (x, 1), zapsmall), digits = 4)
```

```
## $solution
## [1] 1 0 0 0 0 0
##
## $residuals
## [1] 0
```

```
##
## $minssq
## [1] 0
##
## $nullspace
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   -1   -1   -1   -1   -1
## [2,]    0    0    0    0    1
## [3,]    1    0    0    0    0
## [4,]    0    1    0    0    0
## [5,]    0    0    1    0    0
## [6,]    0    0    0    1    0
##
## $rank
## [1] 1
##
## $pivot
## [1] 1 6 2 3 4 5
```

4 The Null-space

4.1 Theory

Of course $Xb = 0$ always has the solution $b = 0$. The set of all z such that $Xz = 0$ is the *right null space* of X, which is equal to the left null space of the transpose of X. The QR decomposition allows us to give a more complete description of these null spaces. Note that `Null()` in the package MASS (Venables and Ripley (2002)) computes the right null space, using the singular value decomposition.

Theorem 3: [**Null**] Suppose X has QR decomposition QRP' of theorem 1. Then the columns of

$$P\begin{bmatrix}-T^{-1}S\\ I\end{bmatrix} \tag{5}$$

are a basis for the left null-space of X.

Proof: Special case of theorem 2. **QED**

4.2 Computation

The basis in theorem 3 is generally not orthonormal. It can easily be made orthonormal by applying the QR decomposition a second time, this time to the basis. Our function `nullRC()` does exactly this.

4.3 Examples

4.3.1 Regular

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(zapsmall(nullRC (x)))
```

```
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
## [4,]    0
```

```
print(zapsmall(nullRC (t(x))))
```

```
##            [,1]
## [1,] -0.4467204
## [2,]  0.0655973
## [3,]  0.4633603
## [4,]  0.3316631
## [5,]  0.6866594
```

4.3.2 Singular

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
x[,3] <- x[,1] + x[,2]
x[,4] <- x[,1] + x[,2]
print(zapsmall(nullRC (x)), digits = 4)
```

```
##         [,1]    [,2]
## [1,] -0.5774 -0.2582
## [2,] -0.5774 -0.2582
## [3,]  0.0000  0.7746
## [4,]  0.5774 -0.5164
```

4.3.3 Broad

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(zapsmall(nullRC (t(x))), digits = 4)
```

```
##         [,1]
## [1,] -0.4467
## [2,]  0.0656
## [3,]  0.4634
## [4,]  0.3316
## [5,]  0.6867
```

4.3.4 Weird

```
x <- matrix(1, 1, 6)
print(zapsmall(nullRC (x)), digits = 4)
```

```
##         [,1]    [,2]    [,3]    [,4]    [,5]
## [1,] -0.7071 -0.4082 -0.2887 -0.2236 -0.1826
## [2,]  0.0000  0.0000  0.0000  0.0000  0.9129
## [3,]  0.7071 -0.4082 -0.2887 -0.2236 -0.1826
## [4,]  0.0000  0.8165 -0.2887 -0.2236 -0.1826
## [5,]  0.0000  0.0000  0.8660 -0.2236 -0.1826
## [6,]  0.0000  0.0000  0.0000  0.8944 -0.1826
```

```
print(zapsmall(nullRC (t(x))), digits = 4)
```

```
##      [,1]
## [1,]    0
```

5 The Moore-Penrose Inverse

5.1 Theory

The Moore-Penrose inverse of the $n \times m$ matrix X is the $m \times n$ matrix X^+, uniquely defined by the four Penrose conditions

- XX^+ is symmetric,
- X^+X is symmetric,
- $XX^+X = X$
- $X^+XX^+ = X^+$.

We use MacDuffee's theorem, in combination with the full-rank version of QR from theorem 1. A discussion of the origin of the lemma, and a proof, are in Ben-Israel and Greville (2001), page 37.

Theorem 4: [**MacDuffee**] If $X = FG$ is a full-rank decomposition, then $X^+ = G'(F'XG')^{-1}F'$.

Corollary 2: [**General Inverse**] If $X = QR$ is a full-rank QR decomposition, then $X^+ = R'(RR')^{-1}Q'$.

Proof: Simply substitute $X = QR$ in theorem 4. **QED**

Alternatively, from Cline (1964), using representation (2) and $U := T^{-1}S$,

$$X^+ = P \begin{bmatrix} (I - U(I+U'U)^{-1}U')T^{-1} & 0_{r\times(n-r)} \\ (I+U'U)^{-1}U')T^{-1} & 0_{(m-r)\times(n-r)} \end{bmatrix} \begin{bmatrix} Q' \\ Q'_\perp \end{bmatrix}.$$

5.2 Computation

The `ginv()` function in the package `MASS` (Venables and Ripley (2002)) computes the Moore-Penrose inverse by using the singular value decomposition. This is somewhat wasteful from the computational point of view. In our function `ginvRC()` we use the result in corollary 2.

5.3 Examples

5.3.1 Regular

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(ginvRC(x), digits = 4)
```

```
##           [,1]   [,2]    [,3]     [,4]    [,5]
## [1,]  0.001437 0.5543 -1.1062 -0.08611  0.7360
## [2,] -0.475830 0.1896 -0.9106  0.17322  0.2032
## [3,]  0.152025 0.3173  0.2716  0.28814 -0.2538
## [4,] -0.058472 0.1057 -0.9417  0.66952  0.2640
```

5.3.2 Singular

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
x[,3] <- x[,1] + x[,2]
print(ginvRC(x), digits = 4)
```

```
##          [,1]      [,2]    [,3]       [,4]    [,5]
## [1,]  0.21990  0.432249 -0.3261 -0.0008035  0.3222
## [2,] -0.29032 -0.001226 -0.1895  0.1960648 -0.1556
## [3,] -0.07043  0.431023 -0.5156  0.1952613  0.1666
## [4,] -0.01212  0.202422 -0.8589  0.7573697  0.1866
```

5.3.3 Broad

```
set.seed(12345)
x <- matrix (rnorm(20), 5, 4)
print(ginvRC( t(x)), digits = 4)
```

```
##           [,1]    [,2]    [,3]     [,4]
## [1,]  0.001437 -0.4758  0.1520 -0.05847
## [2,]  0.554303  0.1896  0.3173  0.10568
## [3,] -1.106171 -0.9106  0.2716 -0.94165
## [4,] -0.086113  0.1732  0.2881  0.66952
## [5,]  0.736010  0.2032 -0.2538  0.26401
```

5.3.4 Weird

```
x <- matrix(1, 1, 6)
print(ginvRC(x), digits = 4)
```

```
##        [,1]
## [1,] 0.1667
## [2,] 0.1667
## [3,] 0.1667
## [4,] 0.1667
## [5,] 0.1667
## [6,] 0.1667
```

6 Appendix: Code

6.1 R code

```
dyn.load("matrix.so")

gsRC <- function (x, eps = 1e-10) {
  n <- nrow (x)
  m <- ncol (x)
  h <-
    .C(
      "gsC",
      x = as.double(x),
      r = as.double (matrix (0, m, m)),
      n = as.integer (n),
      m = as.integer (m),
      rank = as.integer (0),
      pivot = as.integer (1:m),
      eps = as.double (eps)
    )
  rank = h$rank
  return (list (
    x = matrix (h$x, n, m)[, 1:rank, drop = FALSE],
    r = matrix (h$r, m, m)[1:rank, , drop = FALSE],
    rank = rank,
    pivot = h$pivot
  ))
}

lsRC <- function (x, y, eps = 1e-10) {
  n <- length (y)
  m <- ncol (x)
  h <- gsRC (x, eps)
  l <- h$rank
  p <- order (h$pivot)
  k <- 1:l
  q <- h$x
  a <- h$r[, k, drop = FALSE]
  v <- h$r[, -k, drop = FALSE]
  u <- crossprod (q, y)
  b <- solve (a, u)
  res <- drop (y - q %*% u)
  s <- sum (res ^ 2)
  b <- c(b, rep (0, m - l))[p]
  if (l == m) {
    e <- matrix(0, m, 1)
  } else {
    e <- rbind (-solve(a, v), diag(m - l))[p, , drop = FALSE]
  }
  return (list (
    solution = b,
    residuals = res,
    minssq = s,
    nullspace = e,
    rank = l,
    pivot = p
  ))
}

nullRC <- function (x, eps = 1e-10) {
  h <- gsRC (x, eps = eps)
  rank <- h$rank
  p <- order (h$pivot)
  r <- h$r
  m <- ncol (x)
  t <- r[, 1:rank, drop = FALSE]
  s <- r[, -(1:rank), drop = FALSE]
  if (rank == m)
    return (matrix(0, m, 1))
  else {
    nullspace <- rbind (-solve(t, s), diag (m - rank))[p, , drop = FALSE]
    return (gsRC (nullspace)$x)
  }
}

ginvRC <- function (x, eps = 1e-10) {
  h <- gsRC (x, eps)
  p <- order(h$pivot)
  q <- h$x
  s <- h$r
  z <- crossprod (s, (solve (tcrossprod(s), t(q))))
  return (z[p, , drop = FALSE])
}
```

6.2 C code

```
#include <math.h>

#define MINDEX(i, j, n) (((j) - 1) * (n) + (i) - 1)

void gsC(double *, double *, int *, int *, int *, int *, double *);

void gsC(double *x, double *r, int *n, int *m, int *rank, int *pivot,
         double *eps) {
  int i, j, ip, nn = *n, mm = *m, rk = *m, jwork = 1;
  double s = 0.0, p;
  for (j = 1; j <= mm; j++) {
    pivot[j - 1] = j;
  }
  while (jwork <= rk) {
    for (j = 1; j < jwork; j++) {
      s = 0.0;
      for (i = 1; i <= nn; i++) {
        s += x[MINDEX(i, jwork, nn)] * x[MINDEX(i, j, nn)];
      }
      r[MINDEX(j, jwork, mm)] = s;
      for (i = 1; i <= nn; i++) {
        x[MINDEX(i, jwork, nn)] -= s * x[MINDEX(i, j, nn)];
      }
    }
    s = 0.0;
    for (i = 1; i <= nn; i++) {
      s += x[MINDEX(i, jwork, nn)] * x[MINDEX(i, jwork, nn)];
    }
    if (s > *eps) {
      s = sqrt(s);
      r[MINDEX(jwork, jwork, mm)] = s;
      for (i = 1; i <= nn; i++) {
        x[MINDEX(i, jwork, nn)] /= s;
      }
      jwork += 1;
    }
    if (s <= *eps) {
      ip = pivot [rk - 1];
      pivot[rk - 1] = pivot[jwork - 1];
      pivot[jwork - 1] = ip;
      for (i = 1; i <= nn; i++) {
        p = x[MINDEX(i, rk, nn)];
        x[MINDEX(i, rk, nn)] = x[MINDEX(i, jwork, nn)];
        x[MINDEX(i, jwork, nn)] = p;
      }
      for (j = 1; j <= mm; j++) {
        p = r[MINDEX(j, rk, mm)];
        r[MINDEX(j, rk, mm)] = r[MINDEX(j, jwork, mm)];
        r[MINDEX(j, jwork, mm)] = p;
      }
      rk -= 1;
    }
  }
  *rank = rk;
}
```

7 NEWS

001 02/28/16

- First release

002 03/20/16

- Corrections and clarifications suggested by Bill Venables

003 03/21/16

- Example with explicit permutation (to set my mind at ease)
- Computed both left and right null spaces

References

Ben-Israel, A., and T.N.E. Greville. 2001. *Generalized Inverses: Theory and Applications*. Second Edition. New York: Springer.

Björck, Å. 1996. *Numerical Methods for Least Squares Problems*. Philadelphia, PA: SIAM.

Cline, R.E. 1964. "Representations for the Generalized Inverse of a Partitioned Matrix." *Journal of the Society for Industrial and Applied Mathematics* 12 (3): 588–600.

De Leeuw, J. 2015. "Exceedingly Simple Gram-Schmidt Code." http://deleeuwpdx.net/pubfolders/gs/gs.pdf.

Venables, W.N., and B.D. Ripley. 2002. *Modern Applied Statistics with S*. Fourth Edition. New York: Springer. {http://www.stats.ox.ac.uk/pub/MASS4}.