This file consists of Chapter 6 of **MacAnova User's Guide** by Gary W. Oehlert and Christopher Bingham, issued as Technical Report Number 617, School of Statistics, University of Minnesota, revised August 1998, describing Version 4.07 of MacAnova.

Fonts used in this manual are Palatino, Courier, and Symbol.

For information concerning MacAnova, write University of Minnesota, Department of Applied Statistics, 352 Classroom Office Building, 1994 Buford Avenue, St. Paul, MN 55108-6042.

6. Other functions

6.1 **Linear model computations using swp()** Although the linear models commands such as `anova()`, `manova()` and `regress()` provide a powerful facility for fitting least squares models and summarizing the results, you may sometimes want to do the computations yourself, perhaps to convince yourself that a result is correct, or because you are writing a macro for which `regress()` and/or `anova()` is inadequate. Function `swp()` is an asymmetric implementation of the Beaton SWP operator (Beaton 1964) that is particularly useful for least squares computations starting with a cross product matrix.

Suppose $A = [a_{ij}]$ is a m by n matrix and k is an integer between 1 and min(m ,n). Then, if the *pivotal element* $a_{kk} \neq 0$, an asymmetric Beaton SWP on row and column k of A produces a m by n matrix $D = [d_{ij}]$ = SWP(A, k) where

$$\begin{aligned} d_{ij} &= a_{ij} - a_{ik}a_{kj}/a_{kk}, \quad && i \neq k, j \neq k \\ d_{ik} &= a_{ik}/a_{kk}, && i \neq k \\ d_{kj} &= -a_{kj}/a_{kk}, && j \neq k \\ d_{kk} &= 1/a_{kk} \end{aligned}$$

If $k_1, k_2, ..., k_r$ are r integers between 1 and min(m ,n), the matrix SWP(A, k_1, k_2, ..., k_r) is defined to be SWP(... (SWP(SWP(A, k_1),k_2)...),k_r). We say that A is *swept* on rows and columns $k_1, k_2, ..., k_r$.

It can be shown mathematically that the order of the k_i's doesn't matter, so that, for example, SWP(A,k_2, k_1, ..., k_r) = SWP(A,k_1, k_2, ..., k_r). Moreover, if $k_i = k_j$, with $i \neq j$, the effects of their SWP's cancel out. For example, SWP(A, 1, 1, 2, 3, 3, 4) = SWP(A, 2, 4), since both 1 and 3 are repeated, and SWP(A, 1, 1, 2, 3, 3, 3) = SWP(A, 2, 3) since after the second 3 has cancelled the first, A is once again swept on row and column 3.

Now suppose that $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ is partitioned, where A_{ij} is m_i by n_j, with $m_1 = n_1$ (that is, A_{11} is square), $m_1 + m_2 = m$ and $n_1 + n_2 = n$. Then if $D = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} =$ SWP(A,1,2,..,m_1) is partitioned the same way as A, it can be shown that, provided A_{11} is non singular,

$$D_{11} = A_{11}^{-1},\ D_{12} = -A_{11}^{-1}A_{12},\ D_{21} = A_{21}A_{11}^{-1}, \text{ and } D_{22} = A_{22} - A_{21}A_{11}^{-1}A_{12}.$$

Although it is difficult to describe it compactly, a similar result holds whenever A is swept on any other set of distinct rows and columns.

These last identities are what makes SWP useful for regression and ANOVA computations. Suppose X and Y are respectively N by m_1 and N by m_2 data matrices, and let $A = [X \quad Y]'[X \quad Y] = \begin{bmatrix} X'X & X'Y \\ Y'X & Y'Y \end{bmatrix}$ be the matrix of sums of squares and products of all the $m_1 + m_2$ variables. Then

$$\mathrm{SWP}(A,1,2,\ldots,m_1) = \begin{bmatrix} (X'X)^{-1} & -(X'X)^{-1}X'Y \\ Y'X(X'X)^{-1} & Y'Y - Y'X(X'X)^{-1}X'Y \end{bmatrix}$$

The submatrices of this matrix are important quantities in the least squares regression of the columns of Y as dependent variables on the columns of X as independent variables. The least squares regression coefficients for column j of Y are in column j of the m_1 by m_2 matrix $\hat{\beta} = (X'X)^{-1}X'Y$, the negative of the upper right hand block of SWP(A,1,2,..,m_1) and the transpose of the lower left hand block; and the matrix of sums of squares and products of the least squares residuals in these regressions is

$$(Y - X\hat{\beta})'(Y - X\hat{\beta}) = Y'Y - Y'X(X'X)^{-1}X'Y,$$

the lower right hand block of SWP(A,1,2,..,m_1). Moreover, if the errors in the regression of column j of Y are independent with variance σ_j^2, the covariance matrix of the estimated regression coefficients is $\sigma_j^2(X'X)^{-1}$, which is proportional to the upper left hand block of SWP(A,1,2,..,m_1). Thus from a single application of SWP you can estimate the regression coefficients, find residual sums of squares and products, and compute the factors necessary to compute standard errors of the coefficients. If the regression has a constant term (intercept) then the first column of X should be a column of 1's.

MacAnova function `swp()` implements SWP and its usage is essentially the same as the mathematical notation. If `a` is an m by n `REAL` matrix, and `k1`, `k2`, ..., `kr` are integers with values between 1 and min(m,n), `d <- swp(a,k1,k2,...,kr)` computes the swept version of `a`. If, in the course of the computation, a pivotal element is found to be too close to zero, that row and column is not swept and an advisory message is printed.

Actually, arguments `k1`, `k2`, ... to `swp()` can be *vectors* of integers each of whose elements designates a row and column to sweep. Thus, in particular, `swp(a, vector(k1,k2,...,kr))` yields the same result as `swp(a,k1,k2,...,kr)` and `swp(a,run(k))` yields the same result as `swp(a,1,2,...,k)`.

Here are some examples illustrating the properties stated above.

```
Cmd> a <- matrix(vector(9,9,8, 4,11,13, 20,2,12),3); a
(1,1)           9           4          20
(2,1)           9          11           2
(3,1)           8          13          12

Cmd> swp(a,2) # sweep on row and column 2
(1,1)      5.7273     0.36364      19.273
(2,1)    -0.81818    0.090909    -0.18182
(3,1)     -2.6364      1.1818      9.6364
```

We can check this by doing the sweep "by hand." The first line creates a new matrix `d` the same size as `a` and computes $d_{ij} = a_{ij} - a_{i2}a_{2j}/a_{22}$, $i \neq 2$, $j \neq 2$. The next line computes $d_{i2} = a_{i2}/a_{22}$, $i \neq 2$, and $d_{2j} = -a_{2j}/a_{22}$, $j \neq 2$, and the next computes $d_{22} = 1/a_{22}$ and displays `d`.

```
Cmd> d <- 0*a; d[-2,-2] <- a[-2,-2] - a[-2,2] %*% a[2,-2]/a[2,2]
```

```
Cmd> d[-2,2] <- a[-2,2]/a[2,2]; d[2,-2] <- -a[2,-2]/a[2,2]

Cmd> d[2,2] <- 1/a[2,2]; d # same as swp(a,2)
(1,1)       5.7273      0.36364       19.273
(2,1)     -0.81818     0.090909     -0.18182
(3,1)      -2.6364       1.1818       9.6364

Cmd> # swp() on several k's is the same as successive swps():

Cmd> swp(a,run(3)) - swp(swp(swp(a,1),2),3)
(1,1)            0            0            0
(2,1)            0            0            0
(3,1)            0            0            0

Cmd> # Repeated columns cancel out:

Cmd> swp(a,1,2,1) # equivalent to  swp(a,2) above
(1,1)       5.7273      0.36364       19.273
(2,1)     -0.81818     0.090909     -0.18182
(3,1)      -2.6364       1.1818       9.6364
```

We use `swp()` to carry another regression analysis of the Hald data analyzed in Sec. 3.20.

```
Cmd> setoptions(format:"9.5g") # change the default format

Cmd> hald <- matread("macanova.dat","halddata",quiet:T)

Cmd> # Add labels to hald (See Sec. 8.4.1)

Cmd> setlabels(hald,structure("@",enterchars(x1,x2,x3,x4,y)))

Cmd> augmented <- hconcat(rep(1,13),hald)#add col of 1's at left

Cmd> setlabels(augmented,structure("@",\
    vector("CONSTANT", getlabels(hald, 2))))# See Sec. 8.4.1

Cmd> cp <- augmented' %*% augmented; cp # Matrix of SS and SP
          CONSTANT        x1        x2        x3        x4         y
CONSTANT        13        97       626       153       390    1240.5
x1              97      1139      4922       769      2620     10032
x2             626      4922     33050      7201     15739     62028
x3             153       769      7201      2293      4628     13982
x4             390      2620     15739      4628     15062     34733
y           1240.5     10032     62028     13982     34733 1.2109e+05
```

Element `cp[1,1]` is $N = 13$; the remaining elements of the first row and column are the sums of variables `x1` through `y`. Everything else are sums of squares and products of elements from these vectors. For example 10032 is `sum(x1*y)` and 15062 is `sum(x4^2)`.

```
Cmd> ans <- swp(cp,run(5));ans # sweep on indep vars including const
          CONSTANT        x1        x2        x3        x4         y
CONSTANT    820.65   -8.4418   -8.4578   -8.6345   -8.2897   -62.405
x1         -8.4418   0.09271  0.085686  0.092637  0.084455   -1.5511
x2         -8.4578  0.085686   0.08756  0.087867  0.085598  -0.51017
x3         -8.6345  0.092637  0.087867  0.095201  0.086392  -0.10191
x4         -8.2897  0.084455  0.085598  0.086392  0.084031   0.14406
y           62.405    1.5511   0.51017   0.10191  -0.14406    47.864
```

```
Cmd> xxinv <- ans[-6,-6]; beta <- -ans[-6,6]; rss <- ans[6,6]

Cmd> mse <- rss/(13 - 5);mse # mean square error
          y
y     5.983

Cmd> stderr <- sqrt(mse*diag(xxinv))# std errors of coefficients

Cmd> matrix(hconcat(beta,stderr,beta/stderr),\
       labels:structure(getlabels(cp,1)[-6],\
       vector("Coef", "StdErr","t stat")))# See Sec. 8.4.1
              Coef    StdErr    t stat
CONSTANT    62.405    70.071    0.8906
x1          1.5511   0.74477    2.0827
x2         0.51017   0.72379   0.70486
x3         0.10191   0.75471   0.13503
x4        -0.14406   0.70905  -0.20317

Cmd> # Now check results using regress()

Cmd> makecols(hald,x1,x2,x3,x4,y)

Cmd> regress("y=x1+x2+x3+x4") # See Sec. 3.8
Model used is y=x1+x2+x3+x4
              Coef    StdErr         t
CONSTANT    62.405    70.071    0.8906
x1          1.5511   0.74477    2.0827
x2         0.51017   0.72379   0.70486
x3         0.10191   0.75471   0.13503
x4        -0.14406   0.70905  -0.20317

N: 13,  MSE: 5.983, DF: 8,  R^2: 0.98238
Regression F(4,8): 111.48, Durbin-Watson: 2.0526
To see the ANOVA table type 'anova()'
```

6.1.1 **Computing a more accurate cross product matrix – bcprd()** When column 1 of a matrix is the constant 1, sweeping row and column 1 of the cross product matrix is essentially the so called shortcut formula for computing mean corrected sums of squares and products $\sum_{i=1}^{N}\left(x_{ij}-\bar{x}_j\right)\left(x_{ik}-\bar{x}_k\right)=\sum_{i=1}^{N}x_{ij}x_{ik}-\frac{1}{N}\sum_{i=1}^{N}x_{ij}\sum_{i=1}^{N}x_{ik}$. Use of this formula (and hence this use of `swp()`) can lead to serious cancellation and thus loss of accuracy. You can use function `bcprd()` (**b**order **c**ross **pr**o**d**uct) to avoid this problem. `bcprd()` also makes it easier to set up things for `swp()` when doing regression or ANOVA computations.

When `data` is a `REAL` matrix, the assignment `cp1 <- bcprd(data)` is mathematically equivalent to

```
@tmp <- hconcat(rep(1,nrows(data)),data);cp1 <- swp(@tmp %c% @tmp,1),
```

except it uses a more accurate algorithm. Specifically, when X is N by k and $\bar{x}$ is the k by 1 vector of column means, then `bcprd(x)` computes the matrix $k+1$ by $k+1$ matrix $\begin{bmatrix} 1/N & -\bar{x}' \\ \bar{x} & (X-1_N\bar{x}')'(X-1_N\bar{x}') \end{bmatrix}$, where $X-1_N\bar{x}'$ is the matrix of the residuals from the sample mean and $(X-1_N\bar{x}')'(X-1_N\bar{x}')$ is the matrix of sums of squares and products of the these residuals, computed directly by subtracting means, multiplying and summing. Thus the following sequence also computes `ans` in Sec. 6.1.

```
Cmd> cp1 <- bcprd(hald)
          CONSTANT        x1        x2        x3        x4         y
CONSTANT  0.076923   -7.4615   -48.154   -11.769       -30   -95.423
x1          7.4615    415.23    251.08   -372.62      -290    775.96
x2          48.154    251.08    2905.7   -166.54     -3041      2293
x3          11.769   -372.62   -166.54    492.31        38   -618.23
x4              30      -290     -3041        38      3362   -2481.7
y           95.423    775.96      2293   -618.23   -2481.7    2715.8

Cmd> # This result is equivalent to swp(cp, 1); See. Sec. 6.1

Cmd> ans <- swp(cp1,run(2,5)); ans # like swp(cp,run(5))
          CONSTANT        x1        x2        x3        x4         y
CONSTANT    820.65   -8.4418   -8.4578   -8.6345   -8.2897   -62.405
x1         -8.4418   0.09271  0.085686  0.092637  0.084455   -1.5511
x2         -8.4578  0.085686   0.08756  0.087867  0.085598  -0.51017
x3         -8.6345  0.092637  0.087867  0.095201  0.086392  -0.10191
x4         -8.2897  0.084455  0.085598  0.086392  0.084031   0.14406
y           62.405    1.5511   0.51017   0.10191  -0.14406    47.864
```

We sweep only on columns 2 through 5 since `bcprd()` has already implicitly swept on column 1.

An alternate usage for `bcprd()` is `bcprd(x1,x2,...)`. This is equivalent to `bcprd(hconcat(x1,x2,...))`. All the arguments must be `REAL` vectors or matrices with the same number of rows and no `MISSING` values.

6.2 **Computation of eigenvalues and eigenvectors** MacAnova has several commands for computing eigenvectors and eigenvalues of symmetric matrices.

6.2.1 **Ordinary eigenvalues and eigenvectors – eigenvals() and eigenvectors()** Let A be a m by m square matrix. Then if a m by 1 vector u satisfies $Au = \lambda u$ for some scalar λ, the vector u is said to be an *eigenvector* of A with *eigenvalue* λ. Other names for these concepts are *proper vector* and *proper value* or *characteristic vector* and *characteristic value*. With arbitrary square matrices, eigenvalues and eigenvectors may be complex (involving imaginary numbers). However, when $A = A'$ is symmetric, there are always m linearly independent eigenvectors $u_1, u_2, ..., u_m$ with associated *real* eigenvalues $\lambda_1, \lambda_2, ..., \lambda_m$. By convention we can assume that the subscripts are assigned in such a way that $\lambda_{max} = \lambda_1 \geq \lambda_2 \geq ... \geq \lambda_m = \lambda_{min}$. The eigenvectors $u_1, u_2, ..., u_m$ can always be found so that, when $U = [u_1, u_2, ..., u_m]$, U satisfies the following identities:

$$U'AU = \Lambda \equiv \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \lambda_m \end{bmatrix}, \text{ and } UU' = U'U = I_m = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix},$$

That is U is a orthogonal matrix, with each u_i satisfying $u_i'u_i = \|u_i\|^2 = 1$ and $u_i'u_j = 0$, $i \neq j$. Because $U'U = I_m$, $U^{-1} = U'$ and $AU = \Lambda U$.

Some of the important properties of eigenvectors and values are the following.

(i) A has rank r if and only if there are exactly r non-zero eigenvalues.
(ii) A is positive definite if and only if $\lambda_i > 0$, $i = 1, ..., m$.
(iii) $\max_{\|u\|^2=1} u'Au = \lambda_1 = \lambda_{\max}$ and $\min_{\|u\|^2=1} u'Au = \lambda_m = \lambda_{\min}$
(iv) $\text{tr}(A)$ = trace of $A = \sum_{i=1}^{m} a_{ii} = \sum_{i=1}^{m} \lambda_{ii}$
(v) $\det(A) = \prod_{i=1}^{m} \lambda_i$, where $\det(A)$ is the determinant of A.

In MacAnova, if a is a square m by m symmetric square matrix, then `eigenvals(a)` computes the eigenvalues of a in *decreasing* order in a REAL vector of length m. If you want the diagonal matrix Λ, you compute it as `dmat(eigenvals(a))`. To compute both eigenvalues and eigenvectors of a, use `eigen(a)`, which returns a structure with components `values` (the eigenvalues in a vector) and `vectors` (the m by m matrix U whose columns are the eigenvectors of a). You cannot use either `eigenvals()` or `eigen()` to compute the eigenvalues and eigenvectors of a matrix that is *not* symmetric.

We illustrate the use of `eigenvals()` and `eigen()` with some computations using the matrix of corrected sums and squares of products of the Hald data.

```
Cmd> a <- bcprd(hald)[-1,-1]# leave off row and col 1

Cmd> eigenvals(a) # compute the eigenvalues of a
(1)      8344.5        1341      184.13      18.666      2.6534

Cmd> eigs <- eigen(a) # compute the eigenvalues and vectors of a

Cmd> eigs
component: values
(1)      8344.5        1341      184.13      18.666      2.6534
component: vectors columns are the eigenvectors
             (1)         (2)         (3)         (4)         (5)
x1      0.095089    -0.48809     -0.2635     0.59666     0.57209
x2       0.57346     0.26733     0.59386   -0.080656     0.49041
x3     -0.061069     0.53759    -0.62282    -0.29039      0.4848
x4      -0.61793    -0.33676     0.32387    -0.45501     0.43912
y        0.52587    -0.53655    -0.29171    -0.58833    -0.06595
```

```
Cmd> (a %*% eigs$vectors) / eigs$vectors # defining property
            (1)          (2)          (3)          (4)          (5)
x1       8344.5         1341       184.13       18.666       2.6534
x2       8344.5         1341       184.13       18.666       2.6534
x3       8344.5         1341       184.13       18.666       2.6534
x4       8344.5         1341       184.13       18.666       2.6534
y        8344.5         1341       184.13       18.666       2.6534

Cmd> eigs$vectors' %*% eigs$vectors # eigenvectors are orthonormal
             (1)          (2)          (3)          (4)          (5)
(1)            1  -2.6114e-17   1.4412e-16   1.0589e-17   2.3842e-16
(2)  -2.6114e-17            1  -2.1556e-16  -1.3628e-16   1.9262e-16
(3)   1.4412e-16  -2.1556e-16            1  -2.1329e-16   1.2139e-16
(4)   1.0589e-17  -1.3628e-16  -2.1329e-16            1  -6.7158e-18
(5)   2.3842e-16   1.9262e-16   1.2139e-16  -6.7158e-18            1

Cmd> vector(sum(eigs$values),trace(a)) # trace(a) = sum(eigs$values)
(1)        9891        9891

Cmd> vector(prod(eigs$values),det(a)) # det(a) = prod(eigs$values)
(1)  1.0205e+11  1.0205e+11
```

An important statistical application of eigenvalues is in computing the so called *principal components* of a data set. These are linear combinations of variables whose vectors of coefficients are the eigenvectors of the estimated covariance matrix S. These eigenvectors are also the eigenvectors of the matrix, $(X - 1_N \bar{x}')'(X - 1_N \bar{x}')$, of corrected sums of squares and products. For example, if X is a data matrix and u_1 is the first eigenvector of S or of $(X - 1_N \bar{x}')'(X - 1_N \bar{x}')$, the vector Xu_1 is the vector of values of the first principal component, and the columns XU are all m principal components. If x_i is the data vector for case i (x_i' is row i of X), then u_1'x_i is the value of the first principal component for case i. For the Hald data, we can compute the principal components as follows:

```
Cmd> princomps <- hald %*% eigs$vectors # compute princ. components
```

6.2.2 **Eigenvalues and eigenvectors of a tridiagonal matrix – trideigen()** In certain specialized situations (for example, multitaper spectrum estimation; see Sec. 5.4.7), you may need the eigenvalues and vectors of a symmetric *tridiagonal* matrix, that is a symmetric matrix whose only non-zero values are on or immediately above or below the diagonal. MacAnova function `trideigen()` computes the eigenvalues and eigenvectors of a such a matrix. By default, `trideigen()`, like `eigen()`, returns a structure with components `values` and `vectors`.

The simplest usage is `trideigen(d1,d2)`, where `d1` is the REAL vector of diagonal values and `d2` is a REAL vector of the sub-diagonal and super-diagonal values. You must have either `length(d2)` = `length(d1) - 1`, or `length(d1)` = `length(d2)`. In the latter case, `d2[1]`, the first element of `d2`, is ignored.

```
Cmd> d1 <- run(5);d2 <- rep(1,4) # diagonal and super/sub diagonal

Cmd> trideigen(d1,d2) # length(d1) = 5, length(d2) = 4 = 5-1
component: values
(1)       5.7462       4.2077            3       1.7923      0.25384
```

```
component: vectors
(1,1)     0.014027      0.10388      0.30151      0.54249     -0.77705
(2,1)     0.066575      0.33322      0.60302       0.4298       0.5798
(3,1)      0.23537      0.63178      0.30151     -0.63178     -0.23537
(4,1)       0.5798       0.4298     -0.60302      0.33322     0.066575
(5,1)      0.77705     -0.54249      0.30151     -0.10388    -0.014027

Cmd> # Explicitly construct the tridiagonal matrix and use eigen()

Cmd> # See Sec. 2.8.17 for the use of matrix subscripts

Cmd> w <- matrix(rep(0,25),5) # make an all zero matrix

Cmd> i <- run(5);w[hconcat(i,i)]<-d1 # set diagonal

Cmd> # next set sub diagonal and super diagonal

Cmd> i <- run(4); w[hconcat(i,i+1)] <- w[hconcat(i+1,i)] <- d2;w
(1,1)           1           1           0           0           0
(2,1)           1           2           1           0           0
(3,1)           0           1           3           1           0
(4,1)           0           0           1           4           1
(5,1)           0           0           0           1           5

Cmd> eigen(w) # eigen gets same results except for eigenvector signs
component: values
(1)       5.7462       4.2077            3       1.7923      0.25384
component: vectors
(1,1)     0.014027     -0.10388     -0.30151     -0.54249      0.77705
(2,1)     0.066575     -0.33322     -0.60302      -0.4298      -0.5798
(3,1)      0.23537     -0.63178     -0.30151      0.63178      0.23537
(4,1)       0.5798      -0.4298      0.60302     -0.33322    -0.066575
(5,1)      0.77705      0.54249     -0.30151      0.10388     0.014027
```

You can suppress the computation of either component by keyword phrases `values:F` or `vectors:F`.

```
Cmd> trideigen(d1,d2,vectors:F) # just compute eigenvalues
(1)       5.7462       4.2077            3       1.7923      0.25384
```

In certain applications, you don't need for all the eigenvalues and/or eigenvectors. You can limit the eigenvalues and eigenvectors computed by including 1 or 2 additional integer arguments:

```
Cmd> trideigen(d1,d2,1,3,vectors:F) # eigen values 1 through 3
(1)       5.7462       4.2077            3

Cmd> trideigen(d1,d2,3,vectors:F) # equivalent to the preceding
(1)       5.7462       4.2077            3
```

6.2.3 **Relative eigenvectors and eigenvalues of a symmetric matrix – releigenvals() and releigen()** Certain multivariate statistical procedures require what are sometimes called *relative* eigenvalues and eigenvectors. If A and B are m by m symmetric square matrices and B is positive definite, a vector u is an *eigenvector of A relative to B* with *relative eigenvalue* λ if $Au = \lambda Bu$. There are always m linearly independent real relative eigenvectors with real relative eigenvalues. Moreover, the m relative eigenvectors can always be found to satisfy

$$U'AU = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \lambda_m \end{bmatrix}, \text{ and } U'BU = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix},$$

where the columns of U are the relative eigenvectors associated with relative eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_m$. If u_i is column i of U, these matrix equations are equivalent to

$$u_i'A\,u_i = \lambda_i,\ u_i'Bu_i = 1, \text{ and } u_i'A\,u_j = u_i'Bu_j = 0,\ i \neq j.$$

Relative eigenvalues and eigenvectors are also ordinary eigenvalues and eigenvectors of $B^{-1}A$. However, because this matrix is usually non-symmetric, `eigen()` *cannot* be used. Instead, MacAnova provides functions `releigenvals()` and `releigen()`.

In the multivariate analysis of variance (MANOVA), typically A is a hypothesis matrix and B is an error matrix. See Sec. 10.16 for the use of `releigen()` in this context.

Relative eigenvalues satisfy the following identities

$$\sum_{i=1}^{m} \lambda_i = \text{tr}(B^{-1}A) = \text{trace of } B^{-1}A \text{ and } \prod_{i=1}^{m} (1+\lambda_i) = \det(A+B)/\det(B).$$

Here is a short example of the use of releigen.

```
Cmd> a <- matrix(vector(15.25,-0.89, -0.89,30.90),2) # symmetric
Cmd> b <- matrix(vector(120.74,12.97, 12.97,120.28),2) # symmetric
Cmd> print(a,b)
a:
(1,1)        15.25        -0.89
(2,1)        -0.89         30.9
b:
(1,1)       120.74        12.97
(2,1)        12.97       120.28

Cmd> releigs <- releigen(a,b); releigs #relative eigen things
component: values
(1)      0.26618      0.12312
component: vectors
(1,1)    -0.023317     0.088519
(2,1)     0.090687     0.013681
Cmd> (a %*% releigs$vectors)/(b %*% releigs$vectors)
(1,1)      0.26618      0.12312
(2,1)      0.26618      0.12312
Cmd> releigs$vectors' %*% a %*% releigs$vectors # U'AU
(1,1)      0.26618  -4.7762e-18
(2,1)  -6.8887e-19      0.12312
```

```
Cmd> releigs$vectors' %*% b %*% releigs$vectors # U'BU
(1,1)             1    3.672e-17
(2,1)   4.7331e-17             1
```

6.3 **Singular value decomposition – svd()** If X is an n by m matrix with $n \geq m$, it can always be represented as $X = LSR'$, where L is n by m, S is m by m diagonal with non-negative diagonal elements, and R is m by m, and where $L'L = R'R = I_m$, the m by m identity matrix. When the diagonal elements of S are in decreasing order, this is the *singular value decomposition* or SVD of X. The diagonal elements of S are the *singular values* of X, the columns of L are the *left singular vectors* of X, and the columns of R are the *right singular vectors* of X. The restrictions on L and R imply that the sets of left and right singular vectors are each orthonormal. That is, if $L = [l_1, ..., l_m]$ and $R = [r_1, ..., r_m]$, then $\|l_i\|^2 = l_i'l_i = 1$, $\|r_i\|^2 = r_i'r_i = 1$ and $l_i'l_j = r_i'r_j = 0, i \neq j$.

The singular values and left singular vectors are, respectively, the square roots of the first m eigenvalues and the first m eigenvectors of XX'. In addition, the singular values and right singular vectors are, respectively, the square roots of the eigenvalues and the eigenvectors of $X'X$. Since $XR = LS$, a matrix whose columns are proportional to those of L, the columns of L are closely related to the principal components of X considered as a data matrix.

These definitions can be extended to the case when $m > n$, in which case the last $m - n$ singular values are 0.

MacAnova function `svd()` computes the elements of the SVD of a matrix `x` with `nrows(x)` ≥ `ncols(x)`.

```
Cmd> x <- matrix(vector(12,2,12,9,10, 1,7,3,2,9, 5,11,12,14,8),5);x
(1,1)           12            1            5
(2,1)            2            7           11
(3,1)           12            3           12
(4,1)            9            2           14
(5,1)           10            9            8

Cmd> svd(x) # just computes the singular values
(1)       32.108       9.5365         6.72

Cmd> sqrt(eigenvals(x' %*% x)) # sqrt of eigen values of x'x
(1)       32.108       9.5365         6.72

Cmd> svd(x,left:T) # singular values and left singular vectors
component: values          Singular values
(1)       32.108       9.5365         6.72
component: leftvectors     Left singular vectors
(1,1)      -0.3587      0.63489      0.12448
(2,1)      -0.3475     -0.73774     0.017009
(3,1)     -0.53179      0.19903     -0.20279
(4,1)     -0.50696    -0.072113     -0.58736
(5,1)     -0.45909    -0.088545      0.77337

Cmd> svd(x,right:T,left:T) #entire SVD
component: values
(1)       32.108       9.5365         6.72
```

```
component: leftvectors     Left singular vectors
(1,1)      -0.3587      0.63489      0.12448
(2,1)      -0.3475     -0.73774     0.017009
(3,1)     -0.53179      0.19903     -0.20279
(4,1)     -0.50696    -0.072113     -0.58736
(5,1)     -0.45909    -0.088545      0.77337
component: rightvectors    Right singular vectors
(1,1)     -0.63955      0.73371      0.22943
(2,1)     -0.29689     -0.51102      0.80667
(3,1)     -0.70911     -0.44779     -0.54466

Cmd> tmp <- svd(x,right:T,left:T)

Cmd> tmp$leftvectors %*% dmat(tmp$values) %*% tmp$rightvectors'
(1,1)           12            1            5  Reproduce x as
(2,1)            2            7           11  L S R'
(3,1)           12            3           12
(4,1)            9            2           14
(5,1)           10            9            8
```

You can also use keyword phrase `all:T` to get all the pieces of the SVD, suppressing unwanted components with phrases like `values:F`.

```
Cmd> svd(x,all:T,values:F,right:F) # just left singular vectors
(1,1)      -0.3587      0.63489      0.12448
(2,1)      -0.3475     -0.73774     0.017009
(3,1)     -0.53179      0.19903     -0.20279
(4,1)     -0.50696    -0.072113     -0.58736
(5,1)     -0.45909    -0.088545      0.77337
```

6.4 **QR decomposition – qr()** Another important matrix decomposition that is useful in regression analysis is the QR decomposition. Any full rank m by n matrix X with $m \geq n$ can be decomposed as $X = Q\,R$,

$$X = QR = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & q_{22} & \cdots & q_{2n} \\ q_{31} & q_{32} & \cdots & q_{3n} \\ & & & \\ & & & \\ q_{m1} & q_{m2} & \cdots & q_{mn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1,n-1} & r_{1,n} \\ 0 & r_{22} & \cdots & r_{2,n-1} & r_{2,n} \\ & & & & \\ 0 & 0 & & r_{n-1,n-1} & r_{n-1,n} \\ 0 & 0 & & 0 & r_{n,n} \end{bmatrix}$$

where Q is m by n with orthonormal columns (that is $Q'Q = I_n$), and $R = [r_{ij}]$ is a n by n upper triangular matrix, that is $r_{ij} = 0$, $i > j$.

Function `qr(x)` uses a version of Linpack subroutine *dqrdc* to compute the QR decomposition of a matrix `x`. It returns a structure with components `qr` and `qraux` as described in the Linpack manual (Dongarra et al. 1979) This is not quite the QR decomposition in the form just described. Although R can be directly read off as the elements `result$qr[i,j]` with `i ≤ j`, Q is in a coded form. Macro `qrdcomp` in file MacAnova.mac distributed with MacAnova uses `qr()` and then decodes its results to compute Q and R. We illustrate with the same matrix `x` as in Sec. 6.3.

```
Cmd> qr(x)
component: qr
(1,1)      -21.749      -7.8166      -19.863
(2,1)      0.09196       -9.105      -6.8903
(3,1)      0.55176    -0.014819      -10.391
(4,1)      0.41382    -0.038572      0.74483
(5,1)       0.4598      0.70155     -0.58675
component: qraux
(1)       1.5518       1.7114       1.3177

Cmd> getmacros(qrdcomp,quiet:T) # retrieve macro qrdcomp

Cmd> qrd <- qrdcomp(x); qrd # true QR decomposition
component: r
(1,1)      -21.749      -7.8166      -19.863
(2,1)            0       -9.105      -6.8903
(3,1)            0            0      -10.391
component: q
(1,1)     -0.55176      0.36386      0.33229
(2,1)     -0.09196     -0.68986     -0.42538
(3,1)     -0.55176       0.1442     -0.19572
(4,1)     -0.41382       0.1356     -0.64619
(5,1)      -0.4598     -0.59373      0.50276

Cmd> qrd$q %*% qrd$r # reproduce x as q times r
(1,1)           12            1            5
(2,1)            2            7           11
(3,1)           12            3           12
(4,1)            9            2           14
(5,1)           10            9            8
```

For reasons of numerical stability, it is sometimes desirable to reorder the columns of `x` according to the magnitudes of certain pivotal elements. This can be done implicitly by adding a second argument to `qr()` and/or `qrdcomp`.

```
Cmd> qrdcomp(x,T)
component: r
(1,1)      -23.452      -18.421      -9.2956
(2,1)            0       11.562      -0.1063
(3,1)            0            0       7.5882
component: q
(1,1)      -0.2132       0.6982     -0.11961
(2,1)     -0.46904     -0.57429      0.33986
(3,1)     -0.51168      0.22267     -0.22834
(4,1)     -0.59696     -0.17266     -0.47013
(5,1)     -0.34112      0.32142      0.77268
component: pivot
(1)            3            1            2
```

Component `pivot` gives the reordering. Components `q` and `r` are actually the QR decomposition of `x[,vector(3,1,2)]`.

```
Cmd> qrdcomp(x[,vector(3,1,2)])$r
(1,1)     -23.452     -18.421     -9.2956
(2,1)           0      11.562     -0.1063
(3,1)           0           0      7.5882
```

6.5 **Cholesky decomposition – cholesky()** When A is a n by n positive semi-definite (all eigenvalues ≥ 0) symmetric matrix ($A' = A$) , then A can always be factored in the form $A = R'R$, where $R = [r_{ij}]$ is a n by n upper triangular matrix, that is, with $r_{ij} = 0$, when $j < i$. This is known as the *Cholesky decomposition* of A. When A is positive definite (all eigenvalues > 0), R is unique except for multiplying by –1. If $A = X'X$, where X is m by n has the QR decomposition $X = Q\tilde{R}$, then, because $Q'Q = I_n$, $X'X = \tilde{R}'Q'Q\tilde{R} = \tilde{R}'\tilde{R}$, a Cholesky decomposition of $X'X$. Thus the upper triangular part of the QR decomposition of X also defines the Cholesky decomposition of $X'X$.

If MacAnova variable `a` represents a positive definite symmetric matrix, `cholesky(a)` computes the upper triangular factor in the Cholesky decomposition.

```
Cmd> a <- x' %*% x; a # same x as in Sec. 6.3 and 6.4
(1,1)          473          170          432
(2,1)          170          144          218
(3,1)          432          218          550

Cmd> r <- cholesky(a);r # compare -r with QR decomp in Sec. 6.4
(1,1)       21.749       7.8166       19.863
(2,1)            0        9.105       6.8903
(3,1)            0            0       10.391

Cmd> r' %*% r # same as a within rounding error.
(1,1)          473          170          432
(2,1)          170          144          218
(3,1)          432          218          550
```

It is an error if the argument to `cholesky()` has any negative eigenvalues:

```
Cmd> b <- a - dmat(3,60); eigenvals(b)
(1)        970.9       30.944      -14.842

Cmd> cholesky(b)
ERROR: argument to cholesky() is not positive definite
       Problem found while pivoting column 2
```

If `a` has an eigenvalue that is 0 or close to 0, the result of `cholesky(a)` is unreliable.

6.6 **Working with triangular matrices – triupper(), trilower() and triunpack()** There are three functions that make it easier to work with triangular matrices such as are produced by `qrdcomp` and `cholesky()`. `triupper()` and `trilower()` extract the diagonal and elements above it (`triupper()`) or below it (`trilower()`); `triunpack()` creates triangular matrices and symmetric matrices from a vector of length m (m + 1)/2.

`triupper(a)` returns a matrix `d` of the same size and shape as `a` with `d[i,j]` = `a[i,j]` for `i` ≤ `j` (on or above the diagonal) and `d[i,j]` = 0 for `i` > `j` (below the diagonal).

`triupper(a,square:T)` returns a triangular m by m square matrix, where m = `min(nrows(a),ncols(a))`.

`trilower(a)` returns a matrix `d` of the same size and shape as `a` with `d[i,j]` = `a[i,j]` for `i` ≤ `j` (on or below the diagonal) and `d[i,j]` = 0 for `i` < `j` (above the diagonal). Keyword `square` cannot be used with `trilower()`.

For both `triupper(a)` and `trilower(a)`, argument `a` must be a matrix but need not be square. If `a` is a CHARACTER matrix, elements above or below the diagonal are set to empty quoted strings ("") instead of 0's.

A m by m upper or lower triangular matrix is determined by the $m(m+1)/2$ elements consisting of m diagonal elements and $m(m-1)/2$ elements above or below the diagonal. `triupper(a,pack:T)` and `trilower(a,pack:T)` return these elements in a vector of length $m(m+1)/2$. For this usage, `a` must be square.

```
Cmd> a <- matrix(run(12),4); a
(1,1)            1            5            9
(2,1)            2            6           10
(3,1)            3            7           11
(4,1)            4            8           12

Cmd> triupper(a)
(1,1)            1            5            9
(2,1)            0            6           10
(3,1)            0            0           11
(4,1)            0            0            0

Cmd> triupper(a,square:T)
(1,1)            1            5            9
(2,1)            0            6           10
(3,1)            0            0           11

Cmd> triupper(a,pack:T)
ERROR: input matrix must be square when pack is T

Cmd> v <- triupper(a[run(3),],pack:T); v # a[run(3),] is square
(1)              1            5            6            9           10
(6)             11

Cmd> trilower(a)
(1,1)            1            0            0
(2,1)            2            6            0
(3,1)            3            7           11
(4,1)            4            8           12
```

Similarly a m by m symmetric matrix is determined by the $m(m+1)/2$ elements on and above the diagonal. `triunpack(v)` creates a square symmetric matrix from vector `v` whose length must be of the form $m(m+1)/2$. The result is a m by m *symmetric* matrix whose upper triangle (including the diagonal) is determined by `v`. `triunpack(v,lower:T)` and `triunpack(v,upper:T)` produce lower and upper triangular matrices, respectively.

```
Cmd> triunpack(v)
(1,1)            1            5            9
(2,1)            5            6           10
(3,1)            9           10           11

Cmd> triunpack(v,upper:T)
(1,1)            1            5            9
(2,1)            0            6           10
(3,1)            0            0           11
```

```
Cmd> triunpack(v,lower:T)
(1,1)            1            0            0
(2,1)            5            6            0
(3,1)            9           10           11
```

When v is a CHARACTER vector, empty strings ("") are used instead of 0's in the other half.

6.7 **Cluster analysis** cluster() computes a hierarchical agglomerative cluster analysis of the rows of a data matrix or of objects whose inter-object dissimilarity or similarity coefficients are provided. kmeans() computes a *k*-means analysis directly on the data, starting with randomly assigned clusters, pre-specified clusters, or a matrix of means vectors used as "seeds."

6.7.1 **Hierarchical analysis – cluster()** When using cluster(), you can specify any of 7 methods for computing the distance between clusters. cluster() prints a *class table*, that is a table of cluster membership, and a dendrogram showing the history of agglomeration. You can save the class table and/or the values of the distance or similarity coefficient between clusters at each agglomerative step. The cluster code was adapted from Fortran program hcl written by F. Murtagh.

cluster(x,method:Meth), where x is a *n* by *p* matrix, clusters the rows of x considered as multivariate observations and Meth is a quoted string or CHARACTER scalar specifying the agglomeration method used. Legal values for Meth are "ward", "single", "complete", "average", "mcquitty", "median" and "centroid". When method:Meth is omitted, method:"average" is assumed. "average", "complete" and "single" specify the average, complete and single linkage methods, respectively; "ward" and "mcquitty" specify Ward's incremental sums of squares method and McQuitty's weighted average link method, respectively; and "centroid" and "median" specify the centroid and median methods, respectively. See Gordon (1987) for details and further references.

By default, the columns of x are standardized by subtracting column means and dividing by column standard deviations before distances are computed. If this is not desired, use cluster(x,method:Meth,standard:F).

The join points of the dendrogram are labelled in the left margin with the value of the criterion used, either derived from the Euclidean distances among the rows of x or from dissimilarity matrix d or similarity matrix s (see below). When you are clustering the rows of x and distance:"euclidsq" is an argument, these will be labelled with *squared* Euclidean distance. This affects only output, not the clustering itself.

cluster() displays only the final stages of the agglomeration process. By default this is is the history from a stage with 9 (or fewer) clusters to the next to final stage at which there remain 2 clusters. Keyword phrase nclust:m sets the maximum number of clusters displayed to m, that is it shows the history starting with the stage when there are m clusters.

```
Cmd> setseeds(67871,32211) # set values of seeds (see Sec. 2.13.1)
```

```
Cmd> x <- rnorm(5); x# generate miniscule data set
(1)      0.32218   -0.0020892      -1.3853     -0.47216      0.60574

Cmd> cluster(x,method:"ward")
 Case  Number of Clusters  Class table
 No.   2  3  4  5
 ---- -- -- -- --
   1   1  1  1  1
   2   1  3  3  3
   3   2  2  2  2
   4   1  3  4  4
   5   1  1  1  5

    Criterion             Dendrogram or tree
                  +
       1.7166     +-----------+
      0.89782     +-----+     |
      0.42566     |     +--+  |
      0.25677     +--+  |  |  |
  Cluster No.     1  5  3  4  2
                  Clusters 1 to 5 (Top 4 levels of hierarchy).
                  Clustering method: Ward's minimum variance
                  Distance: Euclidian (standardized)

Cmd> cluster(x,nclust:3,method:"ward") #limit no of clusters shown
 Case  Number of Clusters
 No.   2  3
 ---- -- --
   1   1  1              1st 2 columns of class table
   2   1  3              above
   3   2  2
   4   1  3
   5   1  1

    Criterion
                  +       This is equivalent to the top
       1.7166     +-----+ of the preceding dendrogram
      0.89782     +--+  |
  Cluster No.     1  3  2
                  Clusters 1 to 3 (Top 2 levels of hierarchy).
                  Clustering method: Ward's minimum variance
                  Distance: Euclidian (standardized)
```

`cluster(dissim:d,method:Meth)`, where `d` is a *n* by *n* matrix of dissimilarity coefficients (distances) between *n* objects, clusters the objects. Likewise, when *s* is a *n* by *n* matrix of similarity coefficients between the objects, you can cluseter them by `cluster(similar:s,method:Meth)`. The elements of `d` are treated computationally as if they were (non-squared) Euclidean distances. The use of `similar:s` is algorithmically equivalent to the use of `dissim:sqrt(2*(max(vector(s))-s))`.

```
Cmd> d <- abs(x-x') # Euclidean distances because ncols(x) = 1
```

```
Cmd> cluster(dissim:d, method:"ward")
 Case  Number of Clusters
 No.   2  3  4  5
 ---- -- -- -- --
   1   1  1  1  1
   2   1  3  3  3
   3   2  2  2  2
   4   1  3  4  4
   5   1  1  1  5

    Criterion
                +
       1.3405   +-----------+
      0.70108   +-----+     |
      0.33239   |     +--+  |
       0.2005   +--+  |  |  |
  Cluster No.   1  5  3  4  2
                Clusters 1 to 5 (Top 4 levels of hierarchy).
                Clustering method: Ward's minimum variance
                Distance: Input dissimilarity matrix
```

Because `d` is identical to the Euclidean distances between the cases as computed from non-standardized data, this last example gives output the identical to that of `cluster(x,method:"ward",standard:F)`.

For all agglomeration methods, the clustering algorithm initially forms n clusters, each consisting of a single object. Over the course of $n - 1$ stages at each of which two clusters are merged, all objects are combined into a single cluster. At each stage, the two nearest or most similar clusters are merged. The various methods differ in how they determine the distance between two clusters.

Nothing is printed until the clustering is complete. Then `cluster()` assigns numbers to clusters in such a way that that at each stage one of the clusters that is merged has the hightest number remaining. Thus, when there are k clusters, their numbers are 1, 2, ..., k, and then cluster k and cluster $j < k$ are merged into a new cluster j. The cluster membership table above shows that initially clusters C_1 through C_5 are $C_1 = \{1\}$, $C_2 = \{3\}$, $C_3 = \{2\}$, $C_4 = \{4\}$, and $C_5 = \{5\}$. The first merge combines C_1 and C_5 to form a new cluster $C_1 = \{1,5\}$. Then C_3 and C_4 are merged to form a new cluster $C_3 = \{2,4\}$. At the next stage, C_1 and C_3 are merged to form a new cluster $C_1 = \{1,2,4,5\}$. The final stage is not shown but consists of a new cluster $C_1 = \{1,2,3,4,5\}$. There is little relationship between case number and cluster number, even when there are `n` clusters.

The dendrogram or tree also shows the merging pattern. Reading from the bottom up, C_1 and C_5 whose dissimilarity is 0.2568 merge to form a new C_1. Then C_3 and C_4 whose dissimilarity is 0.4257 merge to form a new C_3, and so on.

By default, `cluster()` prints both the class table and dendrogram. You can suppress printing the former by `class:F` and the latter by `tree:F`.

```
Cmd> cluster(x,method:"ward",tree:F) # suppress dendrogram
 Case  Number of Clusters
 No.   2  3  4  5
 ---- -- -- -- --
   1   1  1  1  1
   2   1  3  3  3
   3   2  2  2  2
   4   1  3  4  4
   5   1  1  1  5
```

The lines in the class table are normally in the same order as the cases or objects being clustered. Keyword phrase `reorder:T` changes the order printed so that similar cases are together. It does not change the dendrogram.

```
Cmd> cluster(x,method:"ward",tree:F,reorder:T)
 Case  Number of Clusters
 No.   2  3  4  5
 ---- -- -- -- --
   1   1  1  1  1
   5   1  1  1  5
   2   1  3  3  3
   4   1  3  4  4
   3   2  2  2  2
```

Ordinarily `cluster()` returns a `NULL` value. However, when a keyword phrase of the form `keep:charVec` is an argument, where `charVec` is a `CHARACTER` vector, printed output is suppressed and certain results are returned as value. Legal values for the elements of `charVec` are the following:

Value of `keep`	What is saved
`"distances"`	Computed distances
`"classes"`	n by (`nclust-1`) class membership matrix
`"crit"`	vector of criterion values at each of the last `nclust` - 1 merges
`"all"`	Distances, class membership matrix, and criterion values.

When more than one type of result is requested, they are saved in a structure with components `distances`, `classes`, and/or `criterion`. The order of rows in the class table that is returned is not affected by `reorder:T`.

When `keep` is used, nothing is normally printed. You can force printing of both the class table and dendrogram by keyword phrase `print:T`, or just one or the other by `class:T` or `tree:T`. These keyword phrases should follow `keep` if it is used.

```
Cmd> result <- cluster(x,keep:"all",method:"ward"); result
component: distances
(1,1)            0      0.41527       2.1866       1.0172      0.36312
(2,1)      0.41527            0       1.7713      0.60198      0.77839
(3,1)       2.1866       1.7713            0       1.1693       2.5497
(4,1)       1.0172      0.60198       1.1693            0       1.3804
(5,1)      0.36312      0.77839       2.5497       1.3804            0
```

```
component: classes
(1,1)           1           1           1           1
(2,1)           1           3           3           3
(3,1)           2           2           2           2
(4,1)           1           3           4           4
(5,1)           1           1           1           5
component: criterion
(1)      1.7166      0.89782      0.42566      0.25677
```

```
Cmd> result<-cluster(x,keep:vector("classes","criterion"),\
method:"ward",tree:T)
```

```
    Criterion
                +
       1.7166   +-----------+
      0.89782   +-----+     |
      0.42566   |     +--+  |
      0.25677   +--+  |  |  |
  Cluster No.   1  5  3  4  2
                Clusters 1 to 5 (Top 4 levels of hierarchy).
                Clustering method: Ward's minimum variance
                Distance: Euclidian (standardized)
```

```
Cmd> compnames(result) # result has two components
(1) "classes"
(2) "criterion"
```

6.7.2 **K-means analysis – kmeans()** Function `kmeans()` performs *k*-means clustering of the rows of a REAL matrix using an algorithm programmed by Douglas Hawkins. After *k* initial clusters have been determined, cases are reallocated among clusters in an attempt to minimize the sum of the within-cluster sums of squares. Initial clusters may be pre-specified, selected randomly, selected so as to optimally cluster the first variable, or selected to be nearest to the rows of a matrix as putative mean vectors to be used as "seeds."

You can specify a range $k_{\min} \le k \le k_{\max}$ for k. In this case, `kmeans()` firsts finds $k_{\max}$ clusters; it then merges the two closest clusters to get $k_{\max} - 1$ new starting clusters and does a new clustering, and so on. The output is a structure with components `criterion`, a vector containing the minimized value of the within-cluster sums of squares for each number of clusters, and `classes`, a class membership matrix of integers whose columns successively define cluster membership for $k_{\max}$, $k_{\max} - 1$, ..., $k_{\min}$ cluster solutions. By default, the data are standardized before clustering.

Unless you suppress it by keyword phrase `quiet:T`, `kmeans()` prints a brief history of the merging process, including the values of the criterion being minimized.

The simplest usage is `kmeans(y,kmax:k1,kmin:k2)`, where `y` is a REAL matrix and `k1` ≥ `k2` ≥ 1 are positive integers. This selects initial clusters randomly using the same pseudo random number generator as is used by `runi()` and `rnorm()` (see Sec 2.13.1).

You use keyword `start` with permissible values `"random"`, `"optimal"`, `"means"` and `"classes"`, to specify alternative ways of determining the initial clusters.

`kmeans(y,kmax:k1,kmin:k2,start:"random")` is identical to `kmeans(y,`

kmax:k1,kmin:k2).

kmeans(y,kmax:k1,kmin:k2,start:"optimal") attempts to select the initial clusters so as to minimize the within-cluster sums of squares for column 1 of y.

kmeans(y,means,kmin:k2,start:"means"), where means is a REAL matrix with ncols(y) columns each row of which is a putative cluster "seed." kmeans() selects as initial cluster *j* those rows of y that are closer (using Euclidean distance) to row *j* of means than to any other row of means. The initial number k1 =k_{max} of clusters is the number of distinct rows of means. The rows of means should be in the units of y, not standardized units.

kmeans(y,classes,kmin:k2,start:"classes"), uses classes, a vector of nrows(y) positive integers ≤ 255, to specify initial clusters. The initial number k1 = k_{max} of clusters is the number of distinct integers in classes. If there are empty classes (not all integers between 1 and max(classes) are present), the empty classes are "squeezed out."

In all these usages, kmin:k2 is omitted, k2 = k_{min} is assumed the same as k1 = k_{max}.

There are three additional keywords phrases.

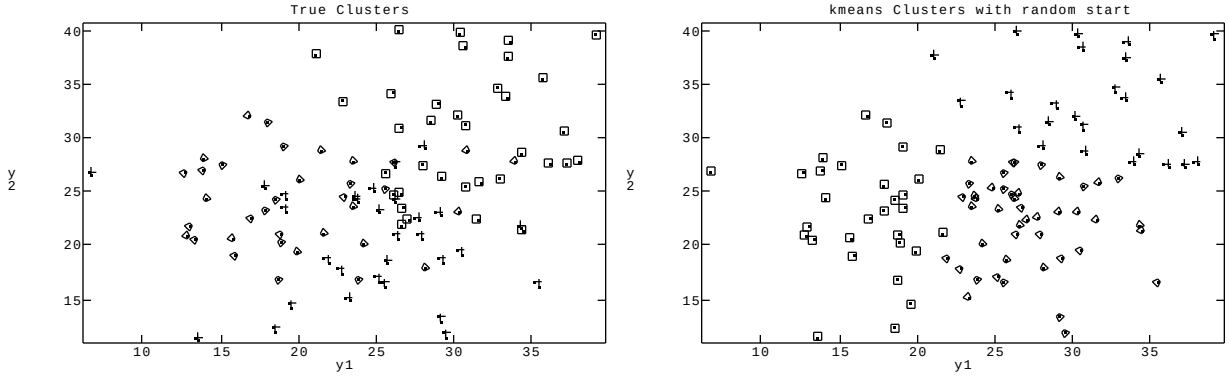
Keyword Phrase	Meaning
standard:F	Do not standardize before clustering
weights:wts	Use weighted means and sums of squares where wts is a REAL vector of length nrows(y) with w[i] > 0
quiet:T	Suppress printing of clustering history

Here is an example of the use of kmeans() with two variables artificial data set of size n = 100. All rows are bivariate normal with variance matrix $\begin{bmatrix} 25 & 0 \\ 0 & 25 \end{bmatrix}$. Rows 1 - 35 have mean [20, 25], rows 36-65 have mean [25,20], and rows 65 - 100 have mean [30,30].

```
Cmd> truegroups <- vector(rep(1,35),rep(2,30),rep(3,35))
Cmd> mu1 <- vector(20,25);mu2 <- vector(25,20);mu3 <- vector(30,30)
Cmd> mu <- vconcat(rep(1,35)*mu1',rep(1,30)*mu2',rep(1,35)*mu3')
Cmd> setseeds(1009295761,91594389)# so you can reproduce the results
Cmd> y <- mu + matrix(5*rnorm(200),100)
```

```
Cmd> results <- kmeans(y,kmax:5,kmin:3)
Cluster analysis by reallocation of objects using Trace W criterion
Variables are standardized before clustering
Initial allocation is random
  k      Initial        Final Reallocations
  5       187.44       63.617      76
  5       63.617       50.694      18
  5       50.694       49.371       7
  5       49.371       48.466       4
  5       48.466       48.376       2
  5       48.376       48.172       2
  5       48.172       48.172       0
Merging clusters 1 and 3; criterion = 67.543
  k      Initial        Final Reallocations
  4       67.543       58.955      11
  4       58.955       58.335       4
  4       58.335       57.867       3
  4       57.867       57.867       0
Merging clusters 3 and 4; criterion = 84.621
  k      Initial        Final Reallocations
  3       84.621       79.947       7
  3       79.947       78.709       6
  3       78.709       74.861       8
  3       74.861       74.755       1
  3       74.755       74.755       0

Cmd> compnames(results) # names of components of output
(1) "classes"
(2) "criterion"

Cmd> dim(results$classes) # results$classes is 100 by 3
(1)          100            3

Cmd> max(results$classes)#columns give 5, 4, and 3 cluster solutions
(1,1)            5            4            3

Cmd> results1 <- kmeans(y,hconcat(mu1,mu2,mu3)',start:"means")
Cluster analysis by reallocation of objects using Trace W criterion
Variables are standardized before clustering
Initial allocation is by nearest candidate mean
  k      Initial        Final Reallocations
  3       78.809       76.568       7
  3       76.568        75.92       6
  3        75.92       75.215       6
  3       75.215       74.755       3
  3       74.755       74.755       0

Cmd> print(format:"1.0f",truegroups,width:70,labels:F)
truegroups: "True" cluster membership
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3
 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
Cmd> print(format:"1.0f",vector(results$classes[,3]),width:70,\
labels:F)
VECTOR: 3-cluster solution from random choice of 5 clusters
 3 3 3 1 1 3 3 3 3 1 3 3 1 2 3 3 3 1 1 3 3 1 3 3 3 3 3 3 1 3 3 1 1 3 2
 1 3 3 1 1 2 1 1 1 3 3 1 1 1 1 1 3 1 1 1 1 1 1 1 1 3 3 1 1 1 1 2 2 2 2
 1 2 1 1 2 2 1 2 2 2 2 1 1 1 2 1 2 2 2 1 2 2 1 2 1 2 1 2 2 2

Cmd> print(format:"1.0f",vector(results1$classes[,1]),width:70,\
labels:F)
VECTOR: 3-cluster solution starting with true means
 1 1 1 2 2 1 1 1 1 2 1 1 2 3 1 1 1 2 2 1 1 2 1 1 1 1 1 1 2 1 1 2 2 1 3
 2 1 1 2 2 3 2 2 2 1 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2 1 1 2 2 2 2 3 3 3 3
 2 3 2 2 3 3 2 3 3 3 3 2 2 2 3 2 3 3 3 2 3 3 2 3 2 3 2 3 3 3

Cmd> # Note that random starting and "seeding" with mean vectors led

Cmd> # to same clusters but with numbered differently

Cmd> chars <- vector("\1","\2","\3") # plotting chars for clusters

Cmd> chplot(y[,1],y[,2],chars[truegroups],\
    xlab:"y1",ylab:"y2",title:"True Clusters")

Cmd> chplot(y[,1],y[,2],chars[results$classes[,3]],\
    xlab:"y1",ylab:"y2",title:"kmeans Clusters with random start")
```



See Sec. 2.15.3 for the use of `chplot()`.

6.8 **Factor rotation – rotation()** Several statistical procedures, including factor analysis and principal components analysis (PCA), involve determining a matrix p by m matrix L with $p > m$ whose columns are orthonormal, that is $L'L = I_m$. In factor analysis the elements in row i of L are the "loadings" of variable i on each of the m factors. That is, variable x_i is assumed to be representable as $x_i = \mu_i + \sum_{j=1}^{m} l_{ij} F_j + e_i$, where F_j is an unobserved random variable – a factor – and e_i is uncorrelated with F_1, ..., F_m. In addition, e_1, e_2, ..., e_p are themselves assumed uncorrelated. In matrix notation, the factor analysis representation can be written $x = \mu + LF + e$, where $F = [F_1, ..., F_m]'$.

In PCA, the columns of L are the first m eigen vectors of the covariance matrix and the elements in column j are the coefficients multiplying each variable in computing the j^{th} principal component. In PCA, too the elements can be considered loadings in that variable x_i can be represented $x_i = \mu_i + \sum_{j=1}^{m} l_{ij} z_j + \tilde{e}_i$, where z_j is the j^{th} principal component and $\tilde{e}_i = \sum_{j=m+1}^{p} l_{ij} z_j$ is a linear combination of the last $p - m$ principal components and is uncorrelated with $z_1, ..., z_m$. This representation differs from the factor analytic representation in that $\tilde{e}_1, \tilde{e}_2, ..., \tilde{e}_p$ cannot be mutually uncorrelated since they are linear combinations of only $p - m$ random variables. In matrix notation, we have $x = \mu + LZ + \tilde{e}$, where $Z = [z_1, z_2, ..., z_m]'$.

A common feature of both these problems is that often what is important about L is not the individual elements but the m dimensional subspace the columns of L span. From the point of view of representing x, any p by m matrix $\tilde{L}$ satisfying $\tilde{L}'\tilde{L} = I_m$ and whose columns span the same subspace will do just as well, since $x = \mu + \tilde{L}\tilde{F} + e$ or $x = \mu + \tilde{L}\tilde{Z} + \tilde{e}$, where $\tilde{F} = R'F$ and $\tilde{Z} = R'Z$, with $R = L'\tilde{L}$ orthogonal ($R'R = I_m$) satisfying $LR = \tilde{L}$. The problem of *rotation* is to determine an orthogonal matrix R so that $\tilde{L} = LR$ so that the elements of $\tilde{F}$ or $\tilde{Z}$ are likely to be "interpretable." There are many methods of factor rotation used, among them *varimax* and *quartimax* (Morrison 1990).

`rotation(l,method:methodName)` computes the rotated version of `l` using the rotation method `methodName` which must be a quoted string or `CHARACTER` scalar. That is, an orthogonal matrix R is found so that `l %*% R` maximizes a criterion. The value returned is `l %*% R`. `l` must be a `REAL` matrix with `nrows(l)` ≥ `ncols(l)`. The result is a `REAL` matrix with the same dimensions as `l`. Matrix R can be recovered as `l' %*% rotation(l)`. If keyword `method` is not used, the default is `"method:varimax"`. (At present the only legal value for `methodName` is `"varimax"`.)

`rotation(l,verbose:T)` prints the value of the criterion before and after rotation.

`rotation(Loadings,reorder:T)` enables post-processing that multiplies each column of the result to make its sum positive and reorders columns in decreasing order of the column sums of squares.

We illustrate it by rotating the first three vectors of loadings derived from the first three eigenvectors of the correlation matrix of `hald`.

```
Cmd> eigs <- eigen(cor(hald))
Cmd> l <- eigs$vectors[,run(3,1)] * sqrt(eigs$values[run(3,1)])';l
             (3)          (2)          (1)
x1       0.25911      0.64338       0.7189
x2       -0.1739     -0.51422      0.83919
x3       0.30094     -0.76334      -0.5715
x4     -0.046327      0.56129     -0.82488
y       0.094213   -0.0042761      0.99173
```

```
Cmd> l1 <- rotation(l,method:"varimax",verbose:T); l1
Varimax starting criterion = 0.42614, final criterion = 1.773
6 iterations and 18 rotations
            (1)          (2)          (3)
x1      0.47841      0.85818       0.1804
x2    -0.066239      0.09318      0.99289
x3     0.079765     -0.99587    -0.041619
x4     -0.13875    0.0010823     -0.98913
y        0.2901      0.52743      0.79378

Cmd> r <- solve(l' %*% l, l' %*% l1);r # rotation matrix
             (1)          (2)          (3)
(3)      0.97122     -0.22065    -0.089641
(2)      0.12807      0.80119     -0.58455
(1)       0.2008      0.55625      0.80639

Cmd> l %*% r # Same as l1; l1 %*% r' is same as l
            (1)          (2)          (3)
x1      0.47841      0.85818       0.1804
x2    -0.066239      0.09318      0.99289
x3     0.079765     -0.99587    -0.041619
x4     -0.13875    0.0010823     -0.98913
y        0.2901      0.52743      0.79378

Cmd> rotation(l,reorder:T)
            (1)          (2)          (3)
x1       0.1804      0.85818      0.47841
x2      0.99289      0.09318    -0.066239
x3    -0.041619     -0.99587     0.079765
x4     -0.98913    0.0010823     -0.13875
y       0.79378      0.52743       0.2901
```

Caution: Do not confuse `rotation()` with `rotate()` (Sec. 5.2.5) which shifts the rows of its first argument up or down, wrapping around the end.