Matlab's Krylov Methods Library

For

Large Sparse

$Ax = b$ Problems

PCG   Preconditioned Conjugate Gradients Method.

```
X = PCG(A,B) attempts to solve the system of linear equations A*X=B for
X. The N-by-N coefficient matrix A must be symmetric and positive
definite and the right hand side column vector B must have length N.

X = PCG(AFUN,B) accepts a function handle AFUN instead of the matrix A.
AFUN(X) accepts a vector input X and returns the matrix-vector product
A*X. In all of the following syntaxes, you can replace A by AFUN.

X = PCG(A,B,TOL) specifies the tolerance of the method. If TOL is []
then PCG uses the default, 1e-6.

X = PCG(A,B,TOL,MAXIT) specifies the maximum number of iterations. If
MAXIT is [] then PCG uses the default, min(N,20).

X = PCG(A,B,TOL,MAXIT,M) and X = PCG(A,B,TOL,MAXIT,M1,M2) use symmetric
positive definite preconditioner M or M=M1*M2 and effectively solve the
system inv(M)*A*X = inv(M)*B for X. If M is [] then a preconditioner
is not applied. M may be a function handle MFUN returning M\X.

X = PCG(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess. If X0 is
[] then PCG uses the default, an all zero vector.

[X,FLAG] = PCG(A,B,...) also returns a convergence FLAG:
 0 PCG converged to the desired tolerance TOL within MAXIT iterations
 1 PCG iterated MAXIT times but did not converge.
 2 preconditioner M was ill-conditioned.
 3 PCG stagnated (two consecutive iterates were the same).
 4 one of the scalar quantities calculated during PCG became too
   small or too large to continue computing.

[X,FLAG,RELRES] = PCG(A,B,...) also returns the relative residual
NORM(B-A*X)/NORM(B). If FLAG is 0, then RELRES <= TOL.

[X,FLAG,RELRES,ITER] = PCG(A,B,...) also returns the iteration number
at which X was computed: 0 <= ITER <= MAXIT.

[X,FLAG,RELRES,ITER,RESVEC] = PCG(A,B,...) also returns a vector of the
residual norms at each iteration including NORM(B-A*X0).

Example:
   n1 = 21; A = gallery('moler',n1);  b1 = A*ones(n1,1);
   tol = 1e-6;  maxit = 15;  M = diag([10:-1:1 1 1:10]);
   [x1,flag1,rr1,iter1,rv1] = pcg(A,b1,tol,maxit,M);
Or use this parameterized matrix-vector product function:
   afun = @(x,n)gallery('moler',n)*x;
   n2 = 21; b2 = afun(ones(n2,1),n2);
   [x2,flag2,rr2,iter2,rv2] = pcg(@(x)afun(x,n2),b2,tol,maxit,M);

Class support for inputs A,B,M1,M2,X0 and the output of AFUN:
   float: double
```

SYMMLQ   Symmetric LQ Method.

```
X = SYMMLQ(A,B) attempts to solve the system of linear equations A*X=B
for X. The N-by-N coefficient matrix A must be symmetric but need not
be positive definite. The right hand side column vector B must have
length N.

X = SYMMLQ(AFUN,B) accepts a function handle AFUN instead of the matrix
A. AFUN(X) accepts a vector input X and returns the matrix-vector
product A*X. In all of the following syntaxes, you can replace A by
AFUN.

X = SYMMLQ(A,B,TOL) specifies the tolerance of the method. If TOL is []
then SYMMLQ uses the default, 1e-6.

X = SYMMLQ(A,B,TOL,MAXIT) specifies the maximum number of iterations.
If MAXIT is [] then SYMMLQ uses the default, min(N,20).

X = SYMMLQ(A,B,TOL,MAXIT,M) and X = SYMMLQ(A,B,TOL,MAXIT,M1,M2) use the
symmetric positive definite preconditioner M or M=M1*M2 and effectively
solve the system inv(sqrt(M))*A*inv(sqrt(M))*Y = inv(sqrt(M))*B for Y
and then return X = inv(sqrt(M))*Y. If M is [] then a preconditioner is
not applied. M may be a function handle returning M\X.

X = SYMMLQ(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess. If X0
is [] then SYMMLQ uses the default, an all zero vector.

[X,FLAG] = SYMMLQ(A,B,...) also returns a convergence FLAG:
 0 SYMMLQ converged to the desired tolerance TOL within MAXIT iterations.
 1 SYMMLQ iterated MAXIT times but did not converge.
 2 preconditioner Mwas ill-conditioned.
 3 SYMMLQ stagnated (two consecutive iterates were the same).
 4 one of the scalar quantities calculated during SYMMLQ became
   too small or too large to continue computing.
 5 preconditioner M was not symmetric positive definite.

[X,FLAG,RELRES] = SYMMLQ(A,B,...) also returns the relative residual
NORM(B-A*X)/NORM(B). If FLAG is 0, then RELRES <= TOL.

[X,FLAG,RELRES,ITER] = SYMMLQ(A,B,...) also returns the iteration
number at which X was computed: 0 <= ITER <= MAXIT.

[X,FLAG,RELRES,ITER,RESVEC] = SYMMLQ(A,B,...) also returns a vector of
of estimates of the SYMMLQ residual norms at each iteration, including
NORM(B-A*X0).

[X,FLAG,RELRES,ITER,RESVEC,RESVECCG] = SYMMLQ(A,B,...) also returns a
vector of estimates of the Conjugate Gradients residual norms at each
iteration.

Example:
   n = 100; on = ones(n,1); A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
   b = sum(A,2); tol = 1e-10; maxit = 50; M = spdiags(4*on,0,n,n);
   x = symmlq(A,b,tol,maxit,M);
Or, use this matrix-vector product function
   %-----------------------------------%
   function y = afun(x,n)
   y = 4 * x;
   y(2:n) = y(2:n) - 2 * x(1:n-1);
   y(1:n-1) = y(1:n-1) - 2 * x(2:n);
   %-----------------------------------%
as input to SYMMLQ:
   x1 = symmlq(@(x)afun(x,n),b,tol,maxit,M);
```

MINRES   Minimum Residual Method.

```
X = MINRES(A,B) attempts to find a minimum norm residual solution X to
the system of linear equations A*X=B. The N-by-N coefficient matrix A
must be symmetric but need not be positive definite. The right hand
side column vector B must have length N.

X = MINRES(AFUN,B) accepts a function handle AFUN instead of the matrix
A. AFUN(X) accepts a vector input X and returns the matrix-vector
product A*X. In all of the following syntaxes, you can replace A by
AFUN.

X = MINRES(A,B,TOL) specifies the tolerance of the method. If TOL is []
then MINRES uses the default, 1e-6.

X = MINRES(A,B,TOL,MAXIT) specifies the maximum number of iterations.
If MAXIT is [] then MINRES uses the default, min(N,20).

X = MINRES(A,B,TOL,MAXIT,M) and X = MINRES(A,B,TOL,MAXIT,M1,M2) use
symmetric positive definite preconditioner M or M=M1*M2 and effectively
solve the system inv(sqrt(M))*A*inv(sqrt(M))*Y = inv(sqrt(M))*B for Y
and then return X = inv(sqrt(M))*Y. If M is [] then a preconditioner is
not applied.  M may be a function handle returning M\X.

X = MINRES(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess. If X0
is [] then MINRES uses the default, an all zero vector.

[X,FLAG] = MINRES(A,B,...) also returns a convergence FLAG:
 0 MINRES converged to the desired tolerance TOL within MAXIT iterations.
 1 MINRES iterated MAXIT times but did not converge.
 2 preconditioner M was ill-conditioned.
 3 MINRES stagnated (two consecutive iterates were the same).
 4 one of the scalar quantities calculated during MINRES became
   too small or too large to continue computing.
 5 preconditioner M was not symmetric positive definite.

[X,FLAG,RELRES] = MINRES(A,B,...) also returns the relative residual
NORM(B-A*X)/NORM(B). If FLAG is 0, then RELRES <= TOL.

[X,FLAG,RELRES,ITER] = MINRES(A,B,...) also returns the iteration
number at which X was computed: 0 <= ITER <= MAXIT.

[X,FLAG,RELRES,ITER,RESVEC] = MINRES(A,B,...) also returns a vector of
estimates of the MINRES residual norms at each iteration, including
NORM(B-A*X0).

[X,FLAG,RELRES,ITER,RESVEC,RESVECCG] = MINRES(A,B,...) also returns a
a vector of estimates of the Conjugate Gradients residual norms at each
iteration.

Example:
   n = 100; on = ones(n,1); A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
   b = sum(A,2); tol = 1e-10; maxit = 50; M = spdiags(4*on,0,n,n);
   x = minres(A,b,tol,maxit,M);
Or, use this matrix-vector product function
   %-----------------------------------%
   function y = afun(x,n)
   y = 4 * x;
   y(2:n) = y(2:n) - 2 * x(1:n-1);
   y(1:n-1) = y(1:n-1) - 2 * x(2:n);
   %-----------------------------------%
as input to MINRES:
   x1 = minres(@(x)afun(x,n),b,tol,maxit,M);
```

LSQR   LSQR Method.

```
X = LSQR(A,B) attempts to solve the system of linear equations A*X=B
for X if A is consistent, otherwise it attempts to solve the least
squares solution X that minimizes norm(B-A*X). The M-by-N coefficient
matrix A need not be square but the right hand side column vector B
must have length M.

X = LSQR(AFUN,B) accepts a function handle AFUN instead of the matrix A.
AFUN(X,'notransp') accepts a vector input X and returns the
matrix-vector product A*X while AFUN(X,'transp') returns A'*X. In all
of the following syntaxes, you can replace A by AFUN.

X = LSQR(A,B,TOL) specifies the tolerance of the method. If TOL is []
then LSQR uses the default, 1e-6.

X = LSQR(A,B,TOL,MAXIT) specifies the maximum number of iterations. If
MAXIT is [] then LSQR uses the default, min([M,N,20]).

X = LSQR(A,B,TOL,MAXIT,M1) and LSQR(A,B,TOL,MAXIT,M1,M2) use N-by-N
preconditioner M or M = M1*M2 and effectively solve the system
A*inv(M)*Y = B for Y, where X = M*Y. If M is [] then a preconditioner
is not applied. M may be a function handle MFUN such that
MFUN(X,'notransp') returns M\X and MFUN(X,'transp') returns M'\X.

X = LSQR(A,B,TOL,MAXIT,M1,M2,X0) specifies the N-by-1 initial guess. If
X0 is [] then LSQR uses the default, an all zero vector.

[X,FLAG] = LSQR(A,B,...) also returns a convergence FLAG:
 0 LSQR converged to the desired tolerance TOL within MAXIT iterations.
 1 LSQR iterated MAXIT times but did not converge.
 2 preconditioner M was ill-conditioned.
 3 LSQR stagnated (two consecutive iterates were the same).
 4 one of the scalar quantities calculated during LSQR became too
   small or too large to continue computing.

[X,FLAG,RELRES] = LSQR(A,B,...) also returns estimates of the relative
residual NORM(B-A*X)/NORM(B). If RELRES <= TOL, then X is a
consistent solution to A*X=B. If FLAG is 0 but RELRES > TOL, then X is
the least squares solution which minimizes norm(B-A*X).

[X,FLAG,RELRES,ITER] = LSQR(A,B,...) also returns the iteration number
at which X was computed: 0 <= ITER <= MAXIT.

[X,FLAG,RELRES,ITER,RESVEC] = LSQR(A,B,...) also returns a vector of
estimates of the residual norm at each iteration including NORM(B-A*X0).

[X,FLAG,RELRES,ITER,RESVEC,LSVEC] = LSQR(A,B,...) also returns a vector
of least squares estimates at each iteration:
NORM((A*inv(M))'*(B-A*X))/NORM(A*inv(M),'fro'). Note the estimate of
NORM(A*inv(M),'fro') changes, and hopefully improves, at each iteration.

Example:
   n = 100; on = ones(n,1); A = spdiags([-2*on 4*on -on],-1:1,n,n);
   b = sum(A,2); tol = 1e-8; maxit = 15;
   M1 = spdiags([on/(-2) on],-1:0,n,n);  M2 = spdiags([4*on -on],0:1,n,n);
   x = lsqr(A,b,tol,maxit,M1,M2);
Or, use this matrix-vector product function
   %-----------------------------------%
   function y = afun(x,n,transp_flag)
   if strcmp(transp_flag,'transp')
      y = 4 * x; y(1:n-1) = y(1:n-1) - 2 * x(2:n); y(2:n) = y(2:n) - x(1:n-1);
   elseif strcmp(transp_flag,'notransp')
      y = 4 * x;  y(2:n) = y(2:n) - 2 * x(1:n-1); y(1:n-1) = y(1:n-1) - x(2:n);
   end
   %-----------------------------------%
as input to LSQR:
   x1 = lsqr(@(x,tflag)afun(x,n,tflag),b,tol,maxit,M1,M2);
```

GMRES   Generalized Minimum Residual Method.

```
X = GMRES(A,B) attempts to solve the system of linear equations A*X = B
for X.  The N-by-N coefficient matrix A must be square and the right
hand side column vector B must have length N. This uses the unrestarted
method with MIN(N,10) total iterations.

X = GMRES(AFUN,B) accepts a function handle AFUN instead of the matrix
A. AFUN(X) accepts a vector input X and returns the matrix-vector
product A*X. In all of the following syntaxes, you can replace A by
AFUN.

X = GMRES(A,B,RESTART) restarts the method every RESTART iterations.
If RESTART is N or [] then GMRES uses the unrestarted method as above.

X = GMRES(A,B,RESTART,TOL) specifies the tolerance of the method.  If
TOL is [] then GMRES uses the default, 1e-6.

X = GMRES(A,B,RESTART,TOL,MAXIT) specifies the maximum number of outer
iterations. Note: the total number of iterations is RESTART*MAXIT. If
MAXIT is [] then GMRES uses the default, MIN(N/RESTART,10). If RESTART
is N or [] then the total number of iterations is MAXIT.

X = GMRES(A,B,RESTART,TOL,MAXIT,M) and
X = GMRES(A,B,RESTART,TOL,MAXIT,M1,M2) use preconditioner M or M=M1*M2
and effectively solve the system inv(M)*A*X = inv(M)*B for X. If M is
[] then a preconditioner is not applied.  M may be a function handle
returning M\X.

X = GMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0) specifies the first initial
guess. If X0 is [] then GMRES uses the default, an all zero vector.

[X,FLAG] = GMRES(A,B,...) also returns a convergence FLAG:
 0 GMRES converged to the desired tolerance TOL within MAXIT iterations.
 1 GMRES iterated MAXIT times but did not converge.
 2 preconditioner M was ill-conditioned.
 3 GMRES stagnated (two consecutive iterates were the same).

[X,FLAG,RELRES] = GMRES(A,B,...) also returns the relative residual
NORM(B-A*X)/NORM(B). If FLAG is 0, then RELRES <= TOL. Note with
preconditioners M1,M2, the residual is NORM(M2\(M1\(B-A*X))).

[X,FLAG,RELRES,ITER] = GMRES(A,B,...) also returns both the outer and
inner iteration numbers at which X was computed: 0 <= ITER(1) <= MAXIT
and 0 <= ITER(2) <= RESTART.

[X,FLAG,RELRES,ITER,RESVEC] = GMRES(A,B,...) also returns a vector of
the residual norms at each inner iteration, including NORM(B-A*X0).
Note with preconditioners M1,M2, the residual is NORM(M2\(M1\(B-A*X))).

Example:
   n = 21; A = gallery('wilk',n);  b = sum(A,2);
   tol = 1e-12;  maxit = 15; M = diag([10:-1:1 1 1:10]);
   x = gmres(A,b,10,tol,maxit,M);
Or, use this matrix-vector product function
   %-----------------------------------------------------------------%
   function y = afun(x,n)
   y = [0; x(1:n-1)] + [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x+[x(2:n); 0];
   %-----------------------------------------------------------------%
and this preconditioner backsolve function
   %------------------------------------------%
   function y = mfun(r,n)
   y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
   %------------------------------------------%
as inputs to GMRES:
   x1 = gmres(@(x)afun(x,n),b,10,tol,maxit,@(x)mfun(x,n));
```

QMR   Quasi-Minimal Residual Method.

```
X = QMR(A,B) attempts to solve the system of linear equations A*X=B for
X. The N-by-N coefficient matrix A must be square and the right hand
side column vector B must have length N.

X = QMR(AFUN,B) accepts a function handle AFUN instead of the matrix A.
AFUN(X,'notransp') accepts a vector input X and returns the
matrix-vector product A*X while AFUN(X,'transp') returns A'*X. In all
of the following syntaxes, you can replace A by AFUN.

X = QMR(A,B,TOL) specifies the tolerance of the method. If TOL is []
then QMR uses the default, 1e-6.

X = QMR(A,B,TOL,MAXIT) specifies the maximum number of iterations. If
MAXIT is [] then QMR uses the default, min(N,20).

X = QMR(A,B,TOL,MAXIT,M) and X = QMR(A,B,TOL,MAXIT,M1,M2) use
preconditioners M or M=M1*M2 and effectively solve the system
inv(M)*A*X = inv(M)*B for X. If M is [] then a preconditioner is not
applied. M may be a function handle MFUN such that MFUN(X,'notransp')
returns M\X and MFUN(X,'transp') returns M'\X.

X = QMR(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess. If X0 is
[] then QMR uses the default, an all zero vector.

[X,FLAG] = QMR(A,B,...) also returns a convergence FLAG:
 0 QMR converged to the desired tolerance TOL within MAXIT iterations.
 1 QMR iterated MAXIT times but did not converge.
 2 preconditioner M was ill-conditioned.
 3 QMR stagnated (two consecutive iterates were the same).
 4 one of the scalar quantities calculated during QMR became too
   small or too large to continue computing.

[X,FLAG,RELRES] = QMR(A,B,...) also returns the relative residual
NORM(B-A*X)/NORM(B). If FLAG is 0, then RELRES <= TOL.

[X,FLAG,RELRES,ITER] = QMR(A,B,...) also returns the iteration number
at which X was computed: 0 <= ITER <= MAXIT.

[X,FLAG,RELRES,ITER,RESVEC] = QMR(A,B,...) also returns a vector of the
residual norms at each iteration, including NORM(B-A*X0).

Example:
   n = 100; on = ones(n,1); A = spdiags([-2*on 4*on -on],-1:1,n,n);
   b = sum(A,2); tol = 1e-8; maxit = 15;
   M1 = spdiags([on/(-2) on],-1:0,n,n);
   M2 = spdiags([4*on -on],0:1,n,n);
   x = qmr(A,b,tol,maxit,M1,M2,[]);
Or, use this matrix-vector product function
   %-----------------------------------%
   function y = afun(x,n,transp_flag)
   if strcmp(transp_flag,'transp')
      y = 4 * x;
      y(1:n-1) = y(1:n-1) - 2 * x(2:n);
      y(2:n) = y(2:n) - x(1:n-1);
   elseif strcmp(transp_flag,'notransp')
      y = 4 * x;
      y(2:n) = y(2:n) - 2 * x(1:n-1);
      y(1:n-1) = y(1:n-1) - x(2:n);
   end
   %-----------------------------------%
as input to QMR:
   x1 = qmr(@(x,tflag)afun(x,n,tflag),b,tol,maxit,M1,M2);
```

BICG   BiConjugate Gradients Method.

```
X = BICG(A,B) attempts to solve the system of linear equations A*X=B
for X.  The N-by-N coefficient matrix A must be square and the right
hand side column vector B must have length N.

X = BICG(AFUN,B) accepts a function handle AFUN instead of the matrix A.
AFUN(X,'notransp') accepts a vector input X and returns the
matrix-vector product A*X while AFUN(X,'transp') returns A'*X. In all
of the following syntaxes, you can replace A by AFUN.

X = BICG(A,B,TOL) specifies the tolerance of the method.  If TOL is []
then BICG uses the default, 1e-6.

X = BICG(A,B,TOL,MAXIT) specifies the maximum number of iterations.  If
MAXIT is [] then BICG uses the default, min(N,20).

X = BICG(A,B,TOL,MAXIT,M) and X = BICG(A,B,TOL,MAXIT,M1,M2) use the
preconditioner M or M=M1*M2 and effectively solve the system
inv(M)*A*X = inv(M)*B for X. If M is [] then a preconditioner is not
applied. M may be a function handle MFUN such that MFUN(X,'notransp')
returns M\X and MFUN(X,'transp') returns M'\X.

X = BICG(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess. If X0 is
[] then BICG uses the default, an all zero vector.

[X,FLAG] = BICG(A,B,...) also returns a convergence FLAG:
 0 BICG converged to the desired tolerance TOL within MAXIT iterations
 1 BICG iterated MAXIT times but did not converge.
 2 preconditioner M was ill-conditioned.
 3 BICG stagnated (two consecutive iterates were the same).
 4 one of the scalar quantities calculated during BICG became
   too small or too large to continue computing.

[X,FLAG,RELRES] = BICG(A,B,...) also returns the relative residual
NORM(B-A*X)/NORM(B). If FLAG is 0, then RELRES <= TOL.

[X,FLAG,RELRES,ITER] = BICG(A,B,...) also returns the iteration number
at which X was computed: 0 <= ITER <= MAXIT.

[X,FLAG,RELRES,ITER,RESVEC] = BICG(A,B,...) also returns a vector of
the residual norms at each iteration including NORM(B-A*X0).

Example:
   n = 100; on = ones(n,1); A = spdiags([-2*on 4*on -on],-1:1,n,n);
   b = sum(A,2); tol = 1e-8; maxit = 15;
   M1 = spdiags([on/(-2) on],-1:0,n,n);
   M2 = spdiags([4*on -on],0:1,n,n);
   x = bicg(A,b,tol,maxit,M1,M2);
Or, use this matrix-vector product function
   %-------------------------------------------%
   function y = afun(x,n,transp_flag)
   if strcmp(transp_flag,'transp')      % y = A'*x
      y = 4 * x;
      y(1:n-1) = y(1:n-1) - 2 * x(2:n);
      y(2:n) = y(2:n) - x(1:n-1);
   elseif strcmp(transp_flag,'notransp') % y = A*x
      y = 4 * x;
      y(2:n) = y(2:n) - 2 * x(1:n-1);
      y(1:n-1) = y(1:n-1) - x(2:n);
   end
   %-------------------------------------------%
as input to BICG:
   x1 = bicg(@(x,tflag)afun(x,n,tflag),b,tol,maxit,M1,M2);
```

BICGSTAB   BiConjugate Gradients Stabilized Method.

```
X = BICGSTAB(A,B) attempts to solve the system of linear equations
A*X=B for X. The N-by-N coefficient matrix A must be square and the
right hand side column vector B must have length N.%

X = BICGSTAB(AFUN,B) accepts a function handle AFUN instead of the
matrix A. AFUN(X) accepts a vector input X and returns the
matrix-vector product A*X. In all of the following syntaxes, you can
replace A by AFUN.

X = BICGSTAB(A,B,TOL) specifies the tolerance of the method. If TOL is
[] then BICGSTAB uses the default, 1e-6.

X = BICGSTAB(A,B,TOL,MAXIT) specifies the maximum number of iterations.
If MAXIT is [] then BICGSTAB uses the default, min(N,20).

X = BICGSTAB(A,B,TOL,MAXIT,M) and X = BICGSTAB(A,B,TOL,MAXIT,M1,M2) use
preconditioner M or M=M1*M2 and effectively solve the system
inv(M)*A*X = inv(M)*B for X. If M is [] then a preconditioner is not
applied. M may be a function handle returning M\X.

X = BICGSTAB(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess.  If
X0 is [] then BICGSTAB uses the default, an all zero vector.

[X,FLAG] = BICGSTAB(A,B,...) also returns a convergence FLAG:
 0 BICGSTAB converged to the desired tolerance TOL within MAXIT iterations.
 1 BICGSTAB iterated MAXIT times but did not converge.
 2 preconditioner M was ill-conditioned.
 3 BICGSTAB stagnated (two consecutive iterates were the same).
 4 one of the scalar quantities calculated during BICGSTAB became
   too small or too large to continue computing.

[X,FLAG,RELRES] = BICGSTAB(A,B,...) also returns the relative residual
NORM(B-A*X)/NORM(B). If FLAG is 0, then RELRES <= TOL.

[X,FLAG,RELRES,ITER] = BICGSTAB(A,B,...) also returns the iteration
number at which X was computed: 0 <= ITER <= MAXIT. ITER may be an
integer + 0.5, indicating convergence half way through an iteration.

[X,FLAG,RELRES,ITER,RESVEC] = BICGSTAB(A,B,...) also returns a vector
of the residual norms at each half iteration, including NORM(B-A*X0).

Example:
   n = 21; A = gallery('wilk',n);  b = sum(A,2);
   tol = 1e-12;  maxit = 15; M = diag([10:-1:1 1 1:10]);
   x = bicgstab(A,b,tol,maxit,M);
Or, use this matrix-vector product function
   %-----------------------------------------------------------------%
   function y = afun(x,n)
   y = [0; x(1:n-1)] + [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x+[x(2:n); 0];
   %-----------------------------------------------------------------%
and this preconditioner backsolve function
   %------------------------------------------%
   function y = mfun(r,n)
   y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
   %------------------------------------------%
as inputs to BICGSTAB:
   x1 = bicgstab(@(x)afun(x,n),b,tol,maxit,@(x)mfun(x,n));
```

LUINC Sparse Incomplete LU factorization.

```
LUINC produces two different kinds of incomplete LU factorizations -- the
drop tolerance and the 0 level of fill-in factorizations.  These factors
may be useful as preconditioners for a system of linear equations being
solved by an iterative method such as BICG (BiConjugate Gradients).

LUINC(X,DROPTOL) performs the incomplete LU factorization of
X with drop tolerance DROPTOL.

LUINC(X,OPTS) allows additional options to the incomplete LU
factorization.  OPTS is a structure with up to four fields:
    droptol --- the drop tolerance of incomplete LU
    milu    --- modified incomplete LU
    udiag   --- replace zeros on the diagonal of U
    thresh  --- the pivot threshold (see also LU)

Only the fields of interest need to be set.

droptol is a non-negative scalar used as the drop
tolerance for the incomplete LU factorization.  This factorization
is computed in the same (column-oriented) manner as the
LU factorization except after each column of L and U has
been calculated, all entries in that column which are smaller
in magnitude than the local drop tolerance, which is
droptol * NORM of the column of X, are "dropped" from L or U.
The only exception to this dropping rule is the diagonal of the
upper triangular factor U which remains even if it is too small.
Note that entries of the lower triangular factor L are tested
before being scaled by the pivot.  Setting droptol = 0
produces the complete LU factorization, which is the default.

milu stands for modified incomplete LU factorization.  Its
value is either 0 (unmodified, the default) or 1 (modified).
Instead of discarding those entries from the newly-formed
column of the factors, they are subtracted from the diagonal
of the upper triangular factor, U.

udiag is either 0 or 1.  If it is 1, any zero diagonal entries
of the upper triangular factor U are replaced by the local drop
tolerance in an attempt to avoid a singular factor.  The default
is 0.

thresh is a pivot threshold in [0,1].  Pivoting occurs
when the diagonal entry in a column has magnitude less
than thresh times the magnitude of any sub-diagonal entry in
that column.  thresh = 0 forces diagonal pivoting.  thresh = 1 is
the default.

Example:

   load west0479
   A = west0479;
   nnz(A)
   nnz(lu(A))
   nnz(luinc(A,1e-6))

   This shows that A has 1887 nonzeros, its complete LU factorization
   has 16777 nonzeros, and its incomplete LU factorization with a
   drop tolerance of 1e-6 has 10311 nonzeros.


[L,U,P] = LUINC(X,'0') produces the incomplete LU factors of a sparse
matrix with 0 level of fill-in (i.e. no fill-in).  L is unit lower
trianglar, U is upper triangular and P is a permutation matrix.  U has the
same sparsity pattern as triu(P*X).  L has the same sparsity pattern as
tril(P*X), except for 1's on the diagonal of L where P*X may be zero.  Both
L and U may have a zero because of cancellation where P*X is nonzero.  L*U
differs from P*X only outside of the sparsity pattern of P*X.

[L,U] = LUINC(X,'0') produces upper triangular U and L is a permutation of
```

unit lower triangular matrix. Thus no comparison can be made between the sparsity patterns of L,U and X, although nnz(L) + nnz(U) = nnz(X) + n. L*U differs from X only outside of its sparsity pattern.

LU = LUINC(X,'0') returns "L+U-I", where L is unit lower triangular, U is upper triangular and the permutation information is lost.

Example:

```
   load west0479
   A = west0479;
   [L,U,P] = luinc(A,'0');
   isequal(spones(U),spones(triu(P*A)))
   spones(L) ~= spones(tril(P*A))
   D = (L*U) .* spones(P*A) - P*A
```

spones(L) differs from spones(tril(P*A)) at some positions on the diagonal and at one position in L where cancellation zeroed out a nonzero element of P*A. The entries of D are of the order of eps.

LUINC works only for sparse matrices.

CHOLINC Sparse Incomplete Cholesky and Cholesky-Infinity factorizations.

```
CHOLINC produces two different kinds of incomplete Cholesky factorizations
-- the drop tolerance and the 0 level of fill-in factorizations.  These
factors may be useful as preconditioners for a symmetric positive definite
system of linear equations being solved by an iterative method such as PCG
(Preconditioned Conjugate Gradients).

R = CHOLINC(X,DROPTOL) performs the incomplete Cholesky factorization of
X, with drop tolerance DROPTOL.

R = CHOLINC(X,OPTS) allows additional options to the incomplete Cholesky
factorization.  OPTS is a structure with up to three fields:
    DROPTOL --- the drop tolerance of the incomplete factorization
    MICHOL  --- modified incomplete Cholesky
    RDIAG   --- replace zeros on the diagonal of R

Only the fields of interest need to be set.

DROPTOL is a non-negative scalar used as the drop tolerance for the
incomplete Cholesky factorization.  This factorization is computed by
performing the incomplete LU factorization with the pivot threshold option
set to 0 (which forces diagonal pivoting) and then scaling the rows of the
incomplete upper triangular factor, U, by the square root of the
diagonal entries in that column.  Since the nonzero entries U(I,J) are
bounded below by DROPTOL*NORM(X(:,J)) (see LUINC), the nonzero
entries R(I,J) are bounded below by DROPTOL*NORM(X(:,J))/R(I,I).
Setting DROPTOL = 0 produces the complete Cholesky factorization,
which is the default.

MICHOL stands for modified incomplete Cholesky factorization.  Its
value is either 0 (unmodified, the default) or 1 (modified).  This
performs the modified incomplete LU factorization of X and then scales
the returned upper triangular factor as described above.

RDIAG is either 0 or 1.  If it is 1, any zero diagonal entries
of the upper triangular factor R are replaced by the square root of
the local drop tolerance in an attempt to avoid a singular factor.  The
default is 0.

Example:

   A = delsq(numgrid('C',25));
   nnz(A)
   nnz(chol(A))
   nnz(cholinc(A,1e-3))

   This shows that A has 2063 nonzeros, its complete Cholesky factorization
   has 8513 nonzeros, and its incomplete Cholesky factorization with a
   drop tolerance of 1e-3 has 4835 nonzeros.


R = CHOLINC(X,'0') produces the incomplete Cholesky factor of a real
symmetric positive definite sparse matrix with 0 level of fill-in (i.e. no
fill-in).  The upper triangular R has the same sparsity pattern as
triu(X), although R may be zero in some positions where X is nonzero due
to cancellation.  The lower triangle of X is assumed to be the transpose
of the upper.  Note that the positive definiteness of X does not guarantee
the existence of a factor with the required sparsity.  An error message
results if the factorization is not possible.  R'*R agrees with X over its
sparsity pattern.

[R,p] = CHOLINC(X,'0'), with two output arguments, never produces an
error message.  If R exists, then p is 0.   But if the incomplete
factor does not exist, then p is a positive integer and R is an upper
triangular matrix of size q-by-n where q = p-1 so that the sparsity
pattern of R is that of the q-by-n upper triangle of X.  R'*R agrees with
X over the sparsity pattern of its first q rows and first q columns.
```

```
Example:

   A = delsq(numgrid('N',10));
   R = cholinc(A,'0');
   isequal(spones(R), spones(triu(A)))

   A(8,8) = 0;
   [R2,p] = cholinc(A,'0');
   isequal(spones(R2), spones(triu(A(1:p-1,:))))

   D = (R'*R) .* spones(A) - A;

   D has entries of the order of eps.


R = CHOLINC(X,'inf') produces the Cholesky-Infinity factorization.  This
factorization is based on the Cholesky factorization, and handles real
positive semi-definite matrices as well.  It may be useful for finding
some sort of solution to systems which arise in primal-dual interior-point
method problems.  When a zero pivot is encountered in the ordinary
Cholesky factorization, the diagonal of the Cholesky-Infinity factor is set
to Inf and the rest of that row is set to 0.  This is designed to force a 0
in the corresponding entry of the solution vector in the associated system
of linear equations.  In practice, X is assumed to be positive
semi-definite so even negative pivots are replaced by Inf.

Example: This symmetric sparse matrix is singular, so the Cholesky
    factorization fails at the zero pivot in the third row.  But cholinc
    succeeds in computing all rows of the Cholesky-Infinity factorization.

    S = sparse([ 1     0     3     0;
                 0    25     0    30;
                 3     0     9     0;
                 0    30     0   661 ]);
    [R,p] = chol(S);
    Rinf = cholinc(S,'inf');

CHOLINC works only for sparse matrices.
```