Go to the first, previous, next, last section, table of contents.

Linear Algebra

This chapter describes functions for solving linear systems. The library provides simple linear algebra operations which operate directly on the `gsl_vector` and `gsl_matrix` objects. These are intended for use with "small" systems where simple algorithms are acceptable.

Anyone interested in large systems will want to use the sophisticated routines found in LAPACK. The Fortran version of LAPACK is recommended as the standard package for linear algebra. It supports blocked algorithms, specialized data representations and other optimizations.

The functions described in this chapter are declared in the header file `` `gsl_linalg.h' ``.

LU Decomposition

A general square matrix A has an LU decomposition into upper and lower triangular matrices,

```
P A = L U
```

where P is a permutation matrix, L is unit lower triangular matrix and U is upper triangular matrix. For square matrices this decomposition can be used to convert the linear system A x = b into a pair of triangular systems (L y = P b, U x = y), which can be solved by forward and back-substitution.

Function: int **gsl_linalg_LU_decomp** *(gsl_matrix * A, gsl_permutation * p, int *signum)*
Function: int **gsl_linalg_complex_LU_decomp** *(gsl_matrix_complex * A, gsl_permutation * p, int *signum)*

These functions factorize the square matrix *A* into the LU decomposition PA = LU. On output the diagonal and upper triangular part of the input matrix *A* contain the matrix U. The lower triangular part of the input matrix (excluding the diagonal) contains L. The diagonal elements of L are unity, and are not stored.

The permutation matrix P is encoded in the permutation *p*. The j-th column of the matrix P is given by the k-th column of the identity matrix, where k = p_j the j-th element of the permutation vector. The sign of the permutation is given by *signum*. It has the value (-1)^n, where n is the number of interchanges in the permutation.

The algorithm used in the decomposition is Gaussian Elimination with partial pivoting (Golub & Van Loan, *Matrix Computations*, Algorithm 3.4.1).

Function: int **gsl_linalg_LU_solve** *(const gsl_matrix * LU, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)*
Function: int **gsl_linalg_complex_LU_solve** *(const gsl_matrix_complex * LU, const gsl_permutation * p, const gsl_vector_complex * b, gsl_vector_complex * x)*

These functions solve the system A x = b using the LU decomposition of A into (*LU*, *p*) given by `gsl_linalg_LU_decomp` or `gsl_linalg_complex_LU_decomp`.

Function: int **gsl_linalg_LU_svx** *(const gsl_matrix * LU, const gsl_permutation * p, gsl_vector * x)*
Function: int **gsl_linalg_complex_LU_svx** *(const gsl_matrix_complex * LU, const gsl_permutation * p, gsl_vector_complex * x)*

These functions solve the system A x = b in-place using the LU decomposition of A into (*LU*,*p*). On input *x* should contain the right-hand side b, which is replaced by the solution on output.

Function: int **gsl_linalg_LU_refine** *(const gsl_matrix * A, const gsl_matrix * LU, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x, gsl_vector * residual)*
Function: int **gsl_linalg_complex_LU_refine** *(const gsl_matrix_complex * A, const gsl_matrix_complex * LU, const gsl_permutation * p, const gsl_vector_complex * b, gsl_vector_complex * x, gsl_vector_complex * residual)*

These functions apply an iterative improvement to *x*, the solution of A x = b, using the LU decomposition of A into (*LU*,*p*). The initial residual r = A x - b is also computed and stored in *residual*.

Function: int **gsl_linalg_LU_invert** *(const gsl_matrix * LU, const gsl_permutation * p, gsl_matrix * inverse)*
Function: int **gsl_complex_linalg_LU_invert** *(const gsl_matrix_complex * LU, const gsl_permutation * p, gsl_matrix_complex * inverse)*

These functions compute the inverse of a matrix A from its LU decomposition (*LU*,*p*), storing the result in the matrix *inverse*. The inverse is computed by solving the system A x = b for each column of the identity matrix. It is preferable to avoid direct computation of the inverse whenever possible.

Function: double **gsl_linalg_LU_det** *(gsl_matrix * LU, int signum)*
Function: gsl_complex **gsl_linalg_complex_LU_det** *(gsl_matrix_complex * LU, int signum)*

These functions compute the determinant of a matrix A from its LU decomposition, *LU*. The determinant is computed as the product of the diagonal elements of U and the sign of the row permutation *signum*.

Function: double **gsl_linalg_LU_lndet** *(gsl_matrix * LU)*
Function: double **gsl_linalg_complex_LU_lndet** *(gsl_matrix_complex * LU)*

These functions compute the logarithm of the absolute value of the determinant of a matrix A, \ln|det(A)|, from its LU decomposition, *LU*. This function may be useful if the direct computation of the determinant would overflow or underflow.

Function: int **gsl_linalg_LU_sgndet** *(gsl_matrix * LU, int signum)*
Function: gsl_complex **gsl_linalg_complex_LU_sgndet** *(gsl_matrix_complex * LU, int signum)*

These functions compute the sign or phase factor of the determinant of a matrix A, det(A)/|det(A)|, from its LU decomposition, *LU*.

QR Decomposition

A general rectangular M-by-N matrix A has a QR decomposition into the product of an orthogonal M-by-M square matrix Q (where Q^T Q = I) and an M-by-N right-triangular matrix R,

```
A = Q R
```

This decomposition can be used to convert the linear system A x = b into the triangular system R x = Q^T b, which can be solved by back-substitution. Another use of the QR decomposition is to compute an orthonormal basis for a set of vectors. The first N columns of Q form an orthonormal basis for the range of A, ran(A), when A has full column rank.

Function: int **gsl_linalg_QR_decomp** *(gsl_matrix * A, gsl_vector * tau)*

This function factorizes the M-by-N matrix *A* into the QR decomposition A = Q R. On output the diagonal and upper triangular part of the input matrix contain the matrix R. The vector *tau* and the columns of the lower triangular part of the matrix *A* contain the Householder coefficients and Householder vectors which encode the orthogonal matrix *Q*. The vector *tau* must be of length k=\min(M,N). The matrix Q is related to these components by, Q = Q_k ... Q_2 Q_1 where Q_i = I - \tau_i v_i v_i^T and v_i is the Householder vector v_i = (0,...,1,A(i+1,i),A(i+2,i),...,A(m,i)). This is the same storage scheme as used by LAPACK.

The algorithm used to perform the decomposition is Householder QR (Golub & Van Loan, *Matrix Computations*, Algorithm 5.2.1).

Function: int **gsl_linalg_QR_solve** *(const gsl_matrix * QR, const gsl_vector * tau, const gsl_vector * b, gsl_vector * x)*

This function solves the system A x = b using the QR decomposition of A into (*QR*, *tau*) given by `gsl_linalg_QR_decomp`.

Function: int **gsl_linalg_QR_svx** *(const gsl_matrix * QR, const gsl_vector * tau, gsl_vector * x)*

This function solves the system A x = b in-place using the QR decomposition of A into (*QR*,*tau*) given by `gsl_linalg_QR_decomp`. On input *x* should contain the right-hand side b, which is replaced by the solution on output.

Function: int **gsl_linalg_QR_lssolve** *(const gsl_matrix * QR, const gsl_vector * tau, const gsl_vector * b, gsl_vector * x, gsl_vector * residual)*

This function finds the least squares solution to the overdetermined system A x = b where the matrix *A* has more rows than columns. The least squares solution minimizes the Euclidean norm of the residual, ||Ax - b|| .The routine uses the QR decomposition of A into (*QR*, *tau*) given by `gsl_linalg_QR_decomp`. The solution is returned in *x*. The residual is computed as a by-product and stored in *residual*.

Function: int **gsl_linalg_QR_QTvec** *(const gsl_matrix * QR, const gsl_vector * tau, gsl_vector * v)*

This function applies the matrix Q^T encoded in the decomposition (*QR*,*tau*) to the vector *v*, storing the result Q^T v in *v*. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q^T.

Function: int **gsl_linalg_QR_Qvec** *(const gsl_matrix * QR, const gsl_vector * tau, gsl_vector * v)*

This function applies the matrix Q encoded in the decomposition (*QR*,*tau*) to the vector *v*, storing the result Q v in *v*. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q.

Function: int **gsl_linalg_QR_Rsolve** *(const gsl_matrix * QR, const gsl_vector * b, gsl_vector * x)*

This function solves the triangular system R x = b for *x*. It may be useful if the product b' = Q^T b has already been computed using `gsl_linalg_QR_QTvec`.

Function: int **gsl_linalg_QR_Rsvx** *(const gsl_matrix * QR, gsl_vector * x)*

This function solves the triangular system R x = b for *x* in-place. On input *x* should contain the right-hand side b and is replaced by the solution on output. This function may be useful if the product b' = Q^T b has already been computed using `gsl_linalg_QR_QTvec`.

Function: int **gsl_linalg_QR_unpack** *(const gsl_matrix * QR, const gsl_vector * tau, gsl_matrix * Q, gsl_matrix * R)*

This function unpacks the encoded QR decomposition (*QR*,*tau*) into the matrices *Q* and *R*, where *Q* is M-by-M and *R* is M-by-N.

Function: int **gsl_linalg_QR_QRsolve** *(gsl_matrix * Q, gsl_matrix * R, const gsl_vector * b, gsl_vector * x)*

This function solves the system R x = Q^T b for *x*. It can be used when the QR decomposition of a matrix is available in unpacked form as (*Q*,*R*).

Function: int **gsl_linalg_QR_update** *(gsl_matrix * Q, gsl_matrix * R, gsl_vector * w, const gsl_vector * v)*

This function performs a rank-1 update w v^T of the QR decomposition (*Q*, *R*). The update is given by Q'R' = Q R + w v^T where the output matrices Q' and R' are also orthogonal and right triangular. Note that *w* is destroyed by the update.

Function: int **gsl_linalg_R_solve** *(const gsl_matrix * R, const gsl_vector * b, gsl_vector * x)*

This function solves the triangular system R x = b for the N-by-N matrix *R*.

Function: int **gsl_linalg_R_svx** *(const gsl_matrix * R, gsl_vector * x)*

This function solves the triangular system R x = b in-place. On input *x* should contain the right-hand side b, which is replaced by the solution on output.

QR Decomposition with Column Pivoting

The QR decomposition can be extended to the rank deficient case by introducing a column permutation P,

```
A P = Q R
```

The first r columns of this Q form an orthonormal basis for the range of A for a matrix with column rank r. This decomposition can also be used to convert the linear system A x = b into the triangular system R y = Q^T b, x = P y, which can be solved by back-substitution and permutation. We denote the QR decomposition with column pivoting by QRP^T since A = Q R P^T.

Function: int **gsl_linalg_QRPT_decomp** *(gsl_matrix * A, gsl_vector * tau, gsl_permutation * p, int *signum, gsl_vector * norm)*

This function factorizes the M-by-N matrix *A* into the QRP^T decomposition A = Q R P^T. On output the diagonal and upper triangular part of the input matrix contain the matrix R. The permutation matrix P is stored in the permutation *p*. The sign of the permutation is given by *signum*. It has the value (-1)^n, where n is the number of interchanges in the permutation. The vector *tau* and the columns of the lower triangular part of the matrix *A* contain the Householder coefficients and vectors which encode the orthogonal matrix *Q*. The vector *tau* must be of length k=\min(M,N). The matrix Q is related to these components by, Q = Q_k ... Q_2 Q_1 where Q_i = I - \tau_i v_i v_i^T and v_i is the Householder vector v_i = (0,...,1,A(i+1,i),A(i+2,i),...,A(m,i)). This is the same storage scheme as used by LAPACK. On output the norms of each column of *R* are stored in the vector *norm*.

The algorithm used to perform the decomposition is Householder QR with column pivoting (Golub & Van Loan, *Matrix Computations*, Algorithm 5.4.1).

Function: int **gsl_linalg_QRPT_decomp2** *(const gsl_matrix * A, gsl_matrix * q, gsl_matrix * r, gsl_vector * tau, gsl_permutation * p, int *signum, gsl_vector * norm)*

This function factorizes the matrix *A* into the decomposition A = Q R P^T without modifying *A* itself and storing the output in the separate matrices *q* and *r*.

Function: int **gsl_linalg_QRPT_solve** *(const gsl_matrix * QR, const gsl_vector * tau, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)*

This function solves the system A x = b using the QRP^T decomposition of A into (*QR*, *tau*, *p*) given by `gsl_linalg_QRPT_decomp`.

Function: int **gsl_linalg_QRPT_svx** *(const gsl_matrix * QR, const gsl_vector * tau, const gsl_permutation * p, gsl_vector * x)*

This function solves the system A x = b in-place using the QRP^T decomposition of A into (*QR*,*tau*,*p*). On input *x* should contain the right-hand side b, which is replaced by the solution on output.

Function: int **gsl_linalg_QRPT_QRsolve** *(const gsl_matrix * Q, const gsl_matrix * R, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)*

This function solves the system R P^T x = Q^T b for *x*. It can be used when the QR decomposition of a matrix is available in unpacked form as (*Q*,*R*).

Function: int **gsl_linalg_QRPT_update** *(gsl_matrix * Q, gsl_matrix * R, const gsl_permutation * p, gsl_vector * u, const gsl_vector * v)*

This function performs a rank-1 update w v^T of the QRP^T decomposition (*Q*, *R*,*p*). The update is given by Q'R' = Q R + w v^T where the output matrices Q' and R' are also orthogonal and right triangular.
Note that *w* is destroyed by the update. The permutation *p* is not changed.

Function: int **gsl_linalg_QRPT_Rsolve** *(const gsl_matrix * QR, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)*

This function solves the triangular system R P^T x = b for the N-by-N matrix R contained in *QR*.

Function: int **gsl_linalg_QRPT_Rsvx** *(const gsl_matrix * QR, const gsl_permutation * p, gsl_vector * x)*

This function solves the triangular system R P^T x = b in-place for the N-by-N matrix R contained in *QR*. On input *x* should contain the right-hand side b, which is replaced by the solution on output.

Singular Value Decomposition

A general rectangular M-by-N matrix A has a singular value decomposition (SVD) into the product of an M-by-N orthogonal matrix U, an N-by-N diagonal matrix of singular values S and the transpose of an N-by-N orthogonal square matrix V,

```
A = U S V^T
```

The singular values \sigma_i = S_{ii} are all non-negative and are generally chosen to form a non-increasing sequence \sigma_1 >= \sigma_2 >= ... >= \sigma_N >= 0.

The singular value decomposition of a matrix has many practical uses. The condition number of the matrix is given by the ratio of the largest singular value to the smallest singular value. The presence of a zero singular value indicates that the matrix is singular. The number of non-zero singular values indicates the rank of the matrix. In practice singular value decomposition of a rank-deficient matrix will not produce exact zeroes for singular values, due to finite numerical precision. Small singular values should be edited by choosing a suitable tolerance.

Function: int **gsl_linalg_SV_decomp** *(gsl_matrix * A, gsl_matrix * V, gsl_vector * S, gsl_vector * work)*

This function factorizes the M-by-N matrix *A* into the singular value decomposition A = U S V^T. On output the matrix *A* is replaced by U. The diagonal elements of the singular value matrix S are stored in the vector *S*. The singular values are non-negative and form a non-increasing sequence from S_1 to S_N. The matrix *V* contains the elements of V in untransposed form. To form the product U S V^T it is necessary to take the transpose of *V*. A workspace of length *N* is required in *work*.

This routine uses the Golub-Reinsch SVD algorithm.

Function: int **gsl_linalg_SV_decomp_mod** *(gsl_matrix * A, gsl_matrix * X, gsl_matrix * V, gsl_vector * S, gsl_vector * work)*

This function computes the SVD using the modified Golub-Reinsch algorithm, which is faster for M>>N. It requires the vector *work* and the N-by-N matrix *X* as additional working space.

Function: int **gsl_linalg_SV_decomp_jacobi** *(gsl_matrix * A, gsl_matrix * V, gsl_vector * S)*

This function computes the SVD using one-sided Jacobi orthogonalization (see references for details). The Jacobi method can compute singular values to higher relative accuracy than Golub-Reinsch algorithms.

Function: int **gsl_linalg_SV_solve** *(gsl_matrix * U, gsl_matrix * V, gsl_vector * S, const gsl_vector * b, gsl_vector * x)*

This function solves the system A x = b using the singular value decomposition (*U*, *S*, *V*) of A given by `gsl_linalg_SV_decomp`.

Only non-zero singular values are used in computing the solution. The parts of the solution corresponding to singular values of zero are ignored. Other singular values can be edited out by setting them to zero before calling this function.

In the over-determined case where *A* has more rows than columns the system is solved in the least squares sense, returning the solution *x* which minimizes ||A x - b||_2.

Cholesky Decomposition

A symmetric, positive definite square matrix A has a Cholesky decomposition into a product of a lower triangular matrix L and its transpose L^T,

```
A = L L^T
```

This is sometimes referred to as taking the square-root of a matrix. The Cholesky decomposition can only be carried out when all the eigenvalues of the matrix are positive. This decomposition can be used to convert the linear system A x = b into a pair of triangular systems (L y = b, L^T x = y), which can be solved by forward and back-substitution.

Function: int **gsl_linalg_cholesky_decomp** *(gsl_matrix * A)*

This function factorizes the positive-definite square matrix *A* into the Cholesky decomposition A = L L^T. On output the diagonal and lower triangular part of the input matrix *A* contain the matrix L. The upper triangular part of the input matrix contains L^T, the diagonal terms being identical for both L and L^T. If the matrix is not positive-definite then the decomposition will fail, returning the error code `GSL_EDOM`.

Function: int **gsl_linalg_cholesky_solve** *(const gsl_matrix * cholesky, const gsl_vector * b, gsl_vector * x)*

This function solves the system A x = b using the Cholesky decomposition of A into the matrix *cholesky*

given by `gsl_linalg_cholesky_decomp`.

Function: int **gsl_linalg_cholesky_svx** *(const gsl_matrix * cholesky, gsl_vector * x)*

This function solves the system A x = b in-place using the Cholesky decomposition of A into the matrix *cholesky* given by `gsl_linalg_cholesky_decomp`. On input *x* should contain the right-hand side b, which is replaced by the solution on output.

Tridiagonal Decomposition of Real Symmetric Matrices

A symmetric matrix A can be factorized by similarity transformations into the form,

```
A = Q T Q^T
```

where Q is an orthogonal matrix and T is a symmetric tridiagonal matrix.

Function: int **gsl_linalg_symmtd_decomp** *(gsl_matrix * A, gsl_vector * tau)*

This function factorizes the symmetric square matrix *A* into the symmetric tridiagonal decomposition Q T Q^ T. On output the diagonal and subdiagonal part of the input matrix *A* contain the tridiagonal matrix T. The remaining lower triangular part of the input matrix contains the Householder vectors which, together with the Householder coefficients *tau*, encode the orthogonal matrix Q. This storage scheme is the same as used by LAPACK. The upper triangular part of *A* is not referenced.

Function: int **gsl_linalg_symmtd_unpack** *(const gsl_matrix * A, const gsl_vector * tau, gsl_matrix * Q, gsl_vector * diag, gsl_vector * subdiag)*

This function unpacks the encoded symmetric tridiagonal decomposition (*A*, *tau*) obtained from `gsl_linalg_symmtd_decomp` into the orthogonal matrix *Q*, the vector of diagonal elements *diag* and the vector of subdiagonal elements *subdiag*.

Function: int **gsl_linalg_symmtd_unpack_T** *(const gsl_matrix * A, gsl_vector * diag, gsl_vector * subdiag)*

This function unpacks the diagonal and subdiagonal of the encoded symmetric tridiagonal decomposition (*A*, *tau*) obtained from `gsl_linalg_symmtd_decomp` into the vectors *diag* and *subdiag*.

Tridiagonal Decomposition of Hermitian Matrices

A hermitian matrix A can be factorized by similarity transformations into the form,

```
A = U T U^T
```

where U is an unitary matrix and T is a real symmetric tridiagonal matrix.

Function: int **gsl_linalg_hermtd_decomp** *(gsl_matrix_complex * A, gsl_vector_complex * tau)*

This function factorizes the hermitian matrix *A* into the symmetric tridiagonal decomposition U T U^T. On output the real parts of the diagonal and subdiagonal part of the input matrix *A* contain the tridiagonal matrix T. The remaining lower triangular part of the input matrix contains the Householder vectors which, together with the Householder coefficients *tau*, encode the orthogonal matrix Q. This storage scheme is the same as used by LAPACK. The upper triangular part of *A* and imaginary parts of the diagonal are not referenced.

Function: int **gsl_linalg_hermtd_unpack** *(const gsl_matrix_complex * A, const gsl_vector_complex * tau, gsl_matrix_complex * Q, gsl_vector * diag, gsl_vector * subdiag)*

This function unpacks the encoded tridiagonal decomposition (*A*, *tau*) obtained from `gsl_linalg_hermtd_decomp` into the unitary matrix *U*, the real vector of diagonal elements *diag* and the real vector of subdiagonal elements *subdiag*.

Function: int **gsl_linalg_hermtd_unpack_T** *(const gsl_matrix_complex * A, gsl_vector * diag, gsl_vector * subdiag)*

This function unpacks the diagonal and subdiagonal of the encoded tridiagonal decomposition (*A*, *tau*) obtained from `gsl_linalg_hermtd_decomp` into the real vectors *diag* and *subdiag*.

Bidiagonalization

A general matrix A can be factorized by similarity transformations into the form,

```
A = U B V^T
```

where U and V are orthogonal matrices and B is a N-by-N bidiagonal matrix with non-zero entries only on the diagonal and superdiagonal. The size of *U* is M-by-N and the size of *V* is N-by-N.

Function: int **gsl_linalg_bidiag_decomp** *(gsl_matrix * A, gsl_vector * tau_U, gsl_vector * tau_V)*

This function factorizes the M-by-N matrix *A* into bidiagonal form U B V^T. The diagonal and superdiagonal of the matrix B are stored in the diagonal and superdiagonal of *A*. The orthogonal matrices U and *V* are stored as compressed Householder vectors in the remaining elements of *A*. The Householder coefficients are stored in the vectors *tau_U* and *tau_V*. The length of *tau_U* must equal the number of elements in the diagonal of *A* and the length of *tau_V* should be one element shorter.

Function: int **gsl_linalg_bidiag_unpack** *(const gsl_matrix * A, const gsl_vector * tau_U, gsl_matrix * U, const gsl_vector * tau_V, gsl_matrix * V, gsl_vector * diag, gsl_vector * superdiag)*

This function unpacks the bidiagonal decomposition of *A* given by `gsl_linalg_bidiag_decomp`, (*A*, *tau_U*, *tau_V*) into the separate orthogonal matrices *U*, *V* and the diagonal vector *diag* and superdiagonal *superdiag*.

Function: int **gsl_linalg_bidiag_unpack2** *(gsl_matrix * A, gsl_vector * tau_U, gsl_vector * tau_V, gsl_matrix * V)*

This function unpacks the bidiagonal decomposition of *A* given by `gsl_linalg_bidiag_decomp`, (*A*, *tau_U*, *tau_V*) into the separate orthogonal matrices *U*, *V* and the diagonal vector *diag* and superdiagonal *superdiag*. The matrix *U* is stored in-place in *A*.

Function: int **gsl_linalg_bidiag_unpack_B** *(const gsl_matrix * A, gsl_vector * diag, gsl_vector * superdiag)*

This function unpacks the diagonal and superdiagonal of the bidiagonal decomposition of *A* given by `gsl_linalg_bidiag_decomp`, into the diagonal vector *diag* and superdiagonal vector *superdiag*.

Householder solver for linear systems

Function: int **gsl_linalg_HH_solve** *(gsl_matrix * A, const gsl_vector * b, gsl_vector * x)*

This function solves the system A x = b directly using Householder transformations. On output the solution is stored in *x* and *b* is not modified. The matrix *A* is destroyed by the Householder transformations.

Function: int **gsl_linalg_HH_svx** *(gsl_matrix * A, gsl_vector * x)*

This function solves the system A x = b in-place using Householder transformations. On input *x* should contain the right-hand side b, which is replaced by the solution on output. The matrix *A* is destroyed by the Householder transformations.

Tridiagonal Systems

Function: int **gsl_linalg_solve_symm_tridiag** *(const gsl_vector * diag, const gsl_vector * e, const gsl_vector * b, gsl_vector * x)*

This function solves the general N-by-N system A x = b where *A* is symmetric tridiagonal. The form of *A* for the 4-by-4 case is shown below,

```
A = ( d_0 e_0          )
    ( e_0 d_1 e_1      )
    (     e_1 d_2 e_2  )
    (         e_2 d_3  )
```

Function: int **gsl_linalg_solve_symm_cyc_tridiag** *(const gsl_vector * diag, const gsl_vector * e, const gsl_vector * b, gsl_vector * x)*

This function solves the general N-by-N system A x = b where *A* is symmetric cyclic tridiagonal. The form of *A* for the 4-by-4 case is shown below,

```
A = ( d_0 e_0     e_3 )
    ( e_0 d_1 e_1     )
    (     e_1 d_2 e_2 )
    ( e_3     e_2 d_3 )
```

Examples

The following program solves the linear system A x = b. The system to be solved is,

```
[ 0.18 0.60 0.57 0.96 ] [x0]   [1.0]
[ 0.41 0.24 0.99 0.58 ] [x1] = [2.0]
[ 0.14 0.30 0.97 0.66 ] [x2]   [3.0]
[ 0.51 0.13 0.19 0.85 ] [x3]   [4.0]
```

and the solution is found using LU decomposition of the matrix A.

```
#include <stdio.h>
#include <gsl/gsl_linalg.h>

int
main (void)
{
  double a_data[] = { 0.18, 0.60, 0.57, 0.96,
                      0.41, 0.24, 0.99, 0.58,
                      0.14, 0.30, 0.97, 0.66,
                      0.51, 0.13, 0.19, 0.85 };

  double b_data[] = { 1.0, 2.0, 3.0, 4.0 };

  gsl_matrix_view m
    = gsl_matrix_view_array(a_data, 4, 4);
```

```
  gsl_vector_view b
    = gsl_vector_view_array(b_data, 4);

  gsl_vector *x = gsl_vector_alloc (4);

  int s;

  gsl_permutation * p = gsl_permutation_alloc (4);

  gsl_linalg_LU_decomp (&m.matrix, p, &s);

  gsl_linalg_LU_solve (&m.matrix, p, &b.vector, x);

  printf ("x = \n");
  gsl_vector_fprintf(stdout, x, "%g");

  gsl_permutation_free (p);
  return 0;
}
```

Here is the output from the program,

```
x = -4.05205
-12.6056
1.66091
8.69377
```

This can be verified by multiplying the solution x by the original matrix A using GNU OCTAVE,

```
octave> A = [ 0.18, 0.60, 0.57, 0.96;
              0.41, 0.24, 0.99, 0.58;
              0.14, 0.30, 0.97, 0.66;
              0.51, 0.13, 0.19, 0.85 ];

octave> x = [ -4.05205; -12.6056; 1.66091; 8.69377];

octave> A * x
ans =

  1.0000
  2.0000
  3.0000
  4.0000
```

This reproduces the original right-hand side vector, b, in accordance with the equation A x = b.

References and Further Reading

Further information on the algorithms described in this section can be found in the following book,

- G. H. Golub, C. F. Van Loan, *Matrix Computations* (3rd Ed, 1996), Johns Hopkins University Press, ISBN 0-8018-5414-8.

The LAPACK library is described in,

- *LAPACK Users' Guide* (Third Edition, 1999), Published by SIAM, ISBN 0-89871-447-8. http://www.netlib.org/lapack

The LAPACK source code can be found at the website above, along with an online copy of the users guide.

The Modified Golub-Reinsch algorithm is described in the following paper,

- T.F. Chan, "An Improved Algorithm for Computing the Singular Value Decomposition", *ACM Transactions on Mathematical Software*, 8 (1982), pp 72--83.

The Jacobi algorithm for singular value decomposition is described in the following papers,

- J.C.Nash, "A one-sided transformation method for the singular value decomposition and algebraic eigenproblem", *Computer Journal*, Volume 18, Number 1 (1973), p 74--76
- James Demmel, Kresimir Veselic, "Jacobi's Method is more accurate than QR", *Lapack Working Note 15* (LAWN-15), October 1989. Available from netlib, `http://www.netlib.org/lapack/` in the `lawns` or `lawnspdf` directories.

Go to the first, previous, next, last section, table of contents.