Chapter 5

Dense Matrix Factorisation Operations

The following routines are described in the following pages:

To use these routines use the include statement

```
#include "matrix2.h"
```

and for the complex routines

```
#include "zmatrix2.h"
```

NAME

BKPfactor, BKPsolve – Bunch–Kaufman–Parlett symmetric indefinite factorise and solve

SYNOPSIS

```
#include "matrix2.h"
MAT     *BKPfactor(MAT *A, PERM *pivot, PERM *blocks)
VEC     *BKPsolve(MAT *A, PERM *pivot, PERM *blocks,
                  VEC *b, VEC *x)
```

DESCRIPTION

The routine **BKPfactor()** forms *in situ* a symmetric indefinite factorisation of the matrix **A** of the form

$$P^T AP = MDM^T$$

where P is a permutation matrix, M is lower triangular, and D is block diagonal, with 1×1 or 2×2 blocks. The matrix P is represented by the permutation **pivot** and D_{ii} is a 1×1 block if and only if **blocks->pe[i] == i**; otherwise **blocks->pe[i]** is the index of the other row/column in the 2×2 block. After the routine the D and M factors are stored in **A** in compact form. This avoids the requirement for additional vectors or matrices for storage.

Note that **pivot** and **blocks** must both be non–NULL and **pivot != blocks** for both **BKPfactor()** and **BKPsolve()**.

The routine **BKPsolve()** solves the equation $Ax = b$ for x. The solve routine **BKPsolve()** is designed specifically to work with **BKPfactor()** as they operate on the same compact storage scheme. Note that the factorisation may succeed when the matrix **A** passed is singular, and that the solve routine may then fail, raising an **E_SING** error. The solve routine may be used *in situ* with **b == x**. If **x** is NULL or too small to hold the result, then a new vector is created of the appropriate size for storing the result. In either case the resulting solution vector is returned.

This factorisation routine, and the accompanying solve routine are derived from "Decomposition of a Symmetric Matrix" by J. Bunch, L. Kaufman and B. Parlett, *Numerische Mathematik* **27**, 95–109 (1976).

Errors will be raised if **A** or **pivot** or **blocks** are NULL, or if **A** is not square, or if the sizes of **A**, **pivot** or **blocks** are not compatible.

EXAMPLE

```
MAT     *A;
PERM    *pivot, *blocks;
VEC     *x, *b;
  ......
A = m_input(MNULL);
```

```
b = v_input(VNULL);
pivot = px_get(A->m);
blocks = px_get(A->m);
/* assuming A symmetric */
BKPfactor(A,pivot,blocks);
x = BKPsolve(A,pivot,blocks,b,VNULL);
```

SEE ALSO

`CHfactor()` and `CHsolve()`

SOURCE FILE: `BKPfactor.c`

NAME

CHfactor, MCHfactor, CHsolve, LDLfactor, LDLsolve – Cholesky factor and solve

SYNOPSIS

```
#include "matrix2.h"
MAT     *CHfactor(MAT *A)
MAT     *MCHfactor(MAT *A, double tol)
VEC     *CHsolve(MAT *A, VEC *b, VEC *x)
MAT     *LDLfactor(MAT *A)
VEC     *LDLsolve(MAT *A, VEC *b, VEC *x)
```

DESCRIPTION

Both `CHfactor()` and `LDLfactor()` factor the matrix `A` *in situ* and returns the factored matrix (in compact form). The Cholesky factorisation routine and the LDL^T routines both use only the lower triangular part of `A`, but the Cholesky factorisation routine fills the upper triangular part of `A` also.

These routines require that `A` is square. The Cholesky factorisation, in particular, requires that `A` be sufficiently positive definite (e.g. lowest eigenvalue of `A` is at least machine epsilon away from zero). If non–positive definiteness is detected during factorisation, then an `E_POSDEF` error will be raised. If you wish to catch such an error, see information on the `catch()` macro. If your matrix is indefinite, then it would be best to use the `BKPfactor()` and `BKPsolve()` routines.

The routine `MCHfactor()` computes a *modified* Cholesky factorisation. This is not a true Cholesky factorisation, but rather the Cholesky factorisation of $A + D$ where D is a diagonal matrix with non-negative entries. Whether the A matrix is modified in this way is determined by the `tol` parameter; the diagonal entry of the Cholesky factorisation is ensured to be $\geq \sqrt{\texttt{tol}}$. The D matrix is guaranteed to be zero in exact arithmetic if $u^T Au \geq \texttt{tol}\, u^T u$ for all u.

EXAMPLE

```
MAT     *A, *LLT, *LDL;
VEC     *b, *x;
double tol;
  ......
A = m_input(MNULL);
b = v_input(VNULL);
input("Input tol for modified Cholesky: ", "%lf", &tol);
LLT = m_copy(A,MNULL);
/* If A positive definite... */
CHfactor(LLT);
x = CHsolve(LLT,b,VNULL);
```

```
/* ...otherwise, get approximate solution... */
LLT = m_copy(A,MNULL);
MCHfactor(LLT,tol);        /* LLT now has factors of A + D */
MCHsolve(LLT,b,x);
/* ...or use LDL factorisation */
LDL = m_copy(A,MNULL);
LDLfactor(LDL);
LDLsolve(LDL,b,x);
```

SEE ALSO

`catch()` and `BKPfactor()`

SOURCE FILE: `CHfactor.c`

NAME

band2mat, mat2band – Band matrix utility routines

SYNOPSIS

```
#include "matrix.h"
MAT  *band2mat(BAND *bdA, MAT *out)
BAND *mat2band(MAT *A, int lb, int ub, BAND *out)
```

DESCRIPTION

The routine **band2mat()** creates an ordinary dense matrix **out** (a Meschach **MAT** structure) that is represented by the band matrix structure **bdA** represents. The returned matrix is square.

The routine **mat2band()** extracts the banded part of **A** with lower bandwidth **lb** and upper bandwidth **ub** and stores the result in the **BAND** structure **out**. The input matrix **A** must be square; if not an **E_SQUARE** error is raised.

For more information about band matrix data structures and storage patterns see the chapter on data structures.

Note that the conversion routines do *not* directly copy the **mat** field of the band structure. If you need efficient storage of band matrices, the routines **band2mat()** and **mat2band()** should probably be avoided.

SEE ALSO

bdLDLfactor() and **bdLUfactor()**.

SOURCE FILE: **bdfactor.c**

NAME

bdLDLfactor, bdLDLsolve – Band Cholesky factorise and solve

SYNOPSIS

```
#include "matrix2.h"
BAND  *bdLDLfactor(BAND *bdA)
VEC   *bdLDLsolve (BAND *bdA, VEC *b, VEC *x)
```

DESCRIPTION

These routines compute the LDL^T factorisation, and solve, a symmetric system of banded equations. These routines only use the lower band and the main diagonal of A.

After the call **bdLDLfactor(A)**, **A** is in factored form which compactly represents both the diagonal matrix D, but also the unit lower triangular matrix L.

If the matrix is exactly singular on factorisation, then an **E_SING** error is raised.

EXAMPLE

To extract a tridiagonal matrix from a dense matrix A, and to factorise and solve a system $Ax = b$:

```
MAT   *A;
VEC   *b, *x;
BAND *bdA;
  ......
/* Note: only need lower triangular part */
bdA = mat2band(A,1,0,(BAND *)NULL);
bdLDLfactor(bdA);
x = bdLDLsolve(bdA,b,VNULL);
```

BUGS

This method can be numerically unstable for matrices that are not positive definite.

The routine **bdLDLfactor()** does not test for symmetry.

SEE ALSO

bdLUfactor(), LDLfactor()

SOURCE FILE: **bdfactor.c**

NAME

LUfactor, LUsolve, LUTsolve, LUcondest, m_inverse, zLUfactor, zLUsolve, zLUAsolve, zLUcondest, zm_inverse – *LU* factorisation (Gaussian elimination) and solve

SYNOPSIS

```
#include "matrix2.h"
MAT      *LUfactor(MAT *A, PERM *pivot)
VEC      *LUsolve (MAT *A, PERM *pivot, VEC *b, VEC *x)
VEC      *LUTsolve(MAT *A, PERM *pivot, VEC *b, VEC *x)
double    LUcondest(MAT *LU, PERM *pivot)
MAT      *m_inverse(MAT *A, MAT *out)

#include "zmatrix2.h"
ZMAT     *zLUfactor(ZMAT *A, PERM *pivot)
ZVEC     *zLUsolve (ZMAT *A, PERM *pivot, ZVEC *b, ZVEC *x)
ZVEC     *zLUAsolve(ZMAT *A, PERM *pivot, ZVEC *b, ZVEC *x)
double    zLUcondest(ZMAT *LU, PERM *pivot)
ZMAT     *zm_inverse(ZMAT *A, ZMAT *out)
```

DESCRIPTION

The routines `LUfactor()` and `zLUfactor()` perform *LU* factorisation, which is otherwise known as Gaussian elimination with implicit scaled partial pivoting. The `zLUfactor()` performs the complex *LU* factorisation. The *LU* factors of `A` are stored in `A` in compact form. Once this is done, the routine `LUsolve()` can be used to solve equations of the form $Ax = b$ for x by forward and back substitution. For real matrices, the system $A^T x = b$ can be solved by using `LUTsolve()`, while for complex matrices $A^* x = b$ can be solved using `zLUAsolve()`. The code for a full factorisation and solving $Ax = b$ and $A^T y = b$ is:

```
/* set up A and b */
  ......
pivot = px_get(A->m);
x = v_get(A->n);
y = v_get(A->m);
LU = m_copy(A,MNULL);
LUfactor(LU,pivot);
x = LUsolve(LU,pivot,b,x);
y = LUTsolve(LU,pivot,b,y);
condition = LUcondest(LU,pivot);
```

A full description of Gaussian elimination with partial pivoting and its numerical behaviour can be found in a number of books, though we refer the reader specifically

to *Matrix Computations* by G.H. Golub and C. van Loan, North Oxford Academic, §§3.2–3.4, pp. 92–122, 2nd Edition (1989). The variant here is that scaling is used *implicitly*. That is, scaling is only used to decide which rows to swap during the partial pivoting process.

Note that the factorisation routine `LUfactor()` may succeed where the solve routine `LUsolve()` fails if, for example, `A` is singular. Also note that *LU* factorisation also succeeds when `A` is not even square, though this is a requirement for the success of `LUsolve()` or `zLUsolve()`. Errors are raised by `LUfactor()` or `zLUfactor()` if `A` or `pivot` is NULL, or if the size of `pivot` is less than the number of rows of `A`. Errors are raised by `LUsolve`, `LUTsolve()`, `zLUsolve()` or `zLUAsolve()` if these conditions occur, if `b` is NULL, or if `A` is not square. Then if `x` is NULL or too small to contain the result a new vector of the appropriate size is created. In either case the solution of $Ax = b$, `x`, is returned. The routines `LUsolve()`, `LUTsolve()`, `zLUsolve()` or `zLUAsolve()` may be used *in situ* (that is, with `b == x`) with version 1.2 or later.

The condition number (relative to the infinity norm) can be *estimated* using the routine `LUcondest()` or the routine `zLUcondest()`. This estimate is not guaranteed to under- or over-estimate the true condition number; however, it can usually be relied on to give an estimate correct to within an order of magnitude, which is usually all that is required.

The routines `m_inverse()` and `zm_inverse()` compute the inverse of `A` and returns the result in `out`. This is carried out using the LU factorisation routines. As is usually noted in numerical analysis texts, inverse matrices should rarely be computed. If a system of equations need to be solved, use the above code calling `LUfactor()` and `LUsolve()`, or `zLUfactor()` and `zLUsolve()` directly.

SOURCE FILE: `lufactor.c, zlufctr.c`

NAME

bdLUfactor, bdLUsolve – Band LU factorise and solve

SYNOPSIS

```
#include "matrix2.h"
BAND  *bdLUfactor(BAND *bdA, PERM *pivot)
VEC   *bdLUsolve (BAND *bdA, PERM *pivot, VEC *b, VEC *x)
```

DESCRIPTION

The routine **bdLUfactor()** computes the LU factorisation of a band matrix A with partial pivoting. This routine performs essentially the same calculations as **LUfactor()**. This operation is done *in situ* in **bdA**. Because partial pivoting is used, the (upper) bandwidth of the matrix being factorised increases. Specifically, the final upper bandwidth is $lb + ub$ where lb is the original lower bandwidth and ub is the original upper bandwidth.

The routine **bdLUsolve()** computes the solution to the banded system $Ax = b$ using the band matrix **bdA** in factored form. Note that only square matrices can be represented as banded matrices. This can be done *in situ* (**x == b**).

These routines raise an **E_NULL** error if either **bdA** or **pivot** is NULL.

EXAMPLE

To factor and solve $Ax = b$:

```
BAND *bdA;
PERM *pivot;
VEC  *x, *b;
  ......
/* set up bdA */
  ......
/* get a random right-hand side */
b = v_rand(v_get(A->mat->n));
/* factor bdA ... */
pivot = px_get(A->mat->n);
bdLUfactor(bdA,pivot);
/* ...and solve system */
x = v_get(b->dim);
bdLUsolve(bdA,pivot,b,x);
```

BUGS

Unless **bdA** is resized to its original size (which can be done very efficiently by **bd_resize()**) repeated calls to **bdLUfactor(bdA,...)** will result in the upper bandwidth increasing until it is $n - 1$ where **bdA** represents an $n \times n$ matrix.

SEE ALSO

`LUfactor()`

SOURCE FILE: `bdfactor.c`

NAME

QRfactor, QRCPfactor, QRsolve, QRCPsolve, QRTsolve, QRcondest, zQRfactor, zQRCPfactor, zQRsolve, zQRCPsolve, zQRAsolve, zQRcondest – QR factorisation and solve

SYNOPSIS

```
#include "matrix2.h"
MAT     *QRfactor(MAT *A, VEC *diag)
MAT     *QRCPfactor(MAT *A, VEC *diag, PERM *pivot)
VEC     *QRsolve(MAT *A, VEC *diag, VEC *b, VEC *x)
VEC     *QRTsolve(MAT *A, VEC *diag, VEC *b, VEC *x)
VEC     *QRCPsolve(MAT *A, VEC *diag, PERM *pivot,
                   VEC *b, VEC *x)
double   QRcondest(MAT *QR)

#include "zmatrix2.h"
ZMAT    *zQRfactor(ZMAT *A, ZVEC *diag)
ZMAT    *zQRCPfactor(ZMAT *A, ZVEC *diag, PERM *pivot)
ZVEC    *zQRsolve (ZMAT *A, ZVEC *diag, ZVEC *b, ZVEC *x)
ZVEC    *zQRAsolve(ZMAT *A, ZVEC *diag, ZVEC *b, ZVEC *x)
ZVEC    *zQRCPsolve(ZMAT *A, ZVEC *diag, PERM *pivot,
                    ZVEC *b, ZVEC *x)
double   zQRcondest(ZMAT *QR)
```

DESCRIPTION

The routines **QRfactor()** and **zQRfactor()** perform straightforward QR factorisations of **A**. The routine **zQRfactor()** computes the complex QR factorisation. For those unfamiliar with the terminology, the QR factorisation of A is a factorisation of the form

$$A = Q\,R$$

where R is upper triangular, and Q is orthogonal in the real case and unitary in the complex case. That is $Q^{-1} = Q^T$ and $Q^TQ = I$ in the real case, and $Q^{-1} = Q^*$ and $Q^*Q = I$ in the complex case. This factorisation exists whether or not A is singular or even square. The QR factorisation is performed using Householder transformations. (These are orthogonal matrices of the form $P_i = I - \alpha_i v_i v_i^T$ (real case) or $P_i = I - \alpha_i v_i v_i^*$ (complex case) where $\alpha_i = 2/v_i^T v_i$ (real case) or $\alpha_i = 2/v_i^* v_i$ (complex case).)

The routines **QRCPfactor()** and **zQRCPfactor()** perform a QR factorisation with column pivoting, which is a factorisation of the form

$$A\,\Pi^T = Q\,R$$

where additionally, Π is a permutation matrix. The Π matrix is represented by **pivot**. This is done exactly as for **QRfactor()** and **zQRfactor()** except for the pivoting.

Both of these factorisations are performed *in situ*, and store the Q and R factors compactly in `A` and `diag`. This compact form is used consistently within this package, and is essentially that of Golub and van Loan's *Matrix Computations*, §5.2, p. 212, 2nd edition, (1989), except that the v's are not normalised in this package. The dimensions of both `diag` must be at least as large as the minimum of the number of rows and columns of `A`.

Once `A` and `diag` contain this compact representation of the QR factors of A, we can use `QRsolve()` to solve systems of linear equations, and indeed, find least square error solutions to overdetermined systems of equations. See *Matrix Computations*, §1.4, p. 11 for an example. Indeed, the code

```
MAT     *QR;
  ......
QR = m_copy(A,MNULL);
QRfactor(QR,diag);
QRsolve(QR,diag,b,x);
```

finds the least squares solution x to

$$A\,x \approx b.$$

Similarly, if `QRCPfactor()` is to be used to factor A, then `QRCPsolve()` can be used to solve the least squares problem $Ax \approx b$. The code to do this is:

```
QR = m_copy(A,MNULL);
QRCPfactor(QR,diag,pivot);
QRCPsolve(QR,diag,pivot,b,x);
```

The corresponding operations for complex matrices simply requires prefixing the functions by a "`z`" and replacing `MAT` by `ZMAT`.

Note that in the real case, `QRTsolve(QR,diag,b,x)` solves the *underdetermined* problem $Ax = b$; that is, it computes the minimum 2-norm x that satisfies $Ax = b$ for $m \leq n$. The corresponding complex routine is `zQRAsolve(QR,diag,b,x)`.

The condition number of a matrix factored using either `QRfactor()` or `QRCPfactor()` can be estimated using `QRcondest()`:

```
printf("2-norm condition no. approx. = %g\n", QRcondest(QR));
```

The corresponding complex function is `zQRcondest()`. The function `QRcondest()` returns a *lower bound* for the least squares condition number of the factored matrix A

$$\kappa_{LS}(A) = \|A\|_2 \|A^+\|_2$$

provided A has full rank. If A is square, then this is exactly equal to the 2-norm condition number

$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2.$$

If the QR factors are exactly singular, then **QRcondest()** will return **HUGE** (**HUGE_VAL** for ANSI C).

The estimate is obtained by obtaining estimates for $\|R\|_2$ and $\|R^{-1}\|_2$. Note that Q and Π do not affect the 2-norm or least squares condition numbers. The estimate of $\|R^{-1}\|_2$ is found using the techniques of **LUcondest()** to obtain a vector y with unit ∞-norm such that $\|R^{-1}y\|_\infty$ is quite small. This is described in Golub and van Loan, 2nd Edition pp. 128–130, (1989). Then the power method is applied to the matrix $(R^TR)^{-1}$ (real case) or $(R^*R)^{-1}$ (complex case) a total of three times with initial vector y. The corresponding estimate of $\|R\|_2$ is obtained by a related method of finding a vector y with unit ∞-norm and $\|Ry\|_\infty$ quite large. The power method is applied to the matrix R^TR (real case) or R^*R (complex case). Taking square root of the estimated eigenvalues gives a lower bound to the 2-norm condition number of R.

A simple, and usually reliable, estimate of the rank of a matrix is to factor the matrix **A** using **QRCPfactor()** (real case) or **zQRCPfactor()** (complex case), and then to count the number of diagonal entries of **A** greater than a certain tolerance in magnitude. A more reliable approach is to use the Singular Value Decomposition. See **svd()**.

SEE ALSO

Householder routines **hhvec()**, **hhtrvec()**, **hhtrrows()** and **hhtrcols()**, **zhhvec()**, **zhhtrvec()**, **zhhtrrows()** and **zhhtrcols()**; **svd()**.

SOURCE FILE: **qrfactor.c**, **zqrfctr.c**

NAME

makeQ, makeR, zmakeQ, zmakeR – explicitly form Q and R factors

SYNOPSIS

```
#include "matrix2.h"
MAT     *makeQ(MAT *QR, VEC *diag, MAT *Qout)
MAT     *makeR(MAT *QR, MAT *Rout)

#include "zmatrix2.h"
ZMAT    *zmakeQ(ZMAT *QR, ZVEC *diag, ZMAT *Qout)
ZMAT    *zmakeR(ZMAT *QR, ZMAT *Rout)
```

DESCRIPTION

The routines **makeQ()** and **zmakeQ()** explicitly forms the real orthogonal Q or complex unitary Q of the QR factorisation from the compact representation in **QR** and **diag**. The result is stored in **Qout**. This routine may not be used to form **Qout** *in situ*.

The routines **makeR()** and **makeR()** explicitly forms the upper triangular R matrix of the QR factorisation. The result is stored in **Rout**. These two routines may be used *in situ*; that is, with **QR == Rout**. (Actually the routine just zeros the strictly lower triangular half of **QR**.)

If **Qout** or **Rout** is NULL or too small to contain the result then a new matrix is created and returned.

EXAMPLE

```
MAT     *A, *QR, *Q, *R;
VEC     *diag;
  ......
diag = v_get(A->m);
QR = m_copy(A,MNULL);
QRfactor(QR,diag);
Q = makeQ(QR,diag,MNULL);
R = makeR(QR,MNULL);
/* makeR(QR,QR); replaces QR with the R matrix */
```

SOURCE FILE: qrfactor.c, zqrfctr.c

NAME

givens, rot_cols, rot_rows, rot_vec, zgivens, zrot_cols, zrot_rows, rot_zvec – Givens' rotations routines

SYNOPSIS

```
#include "matrix2.h"
void     givens(double x, double y, Real &c, Real &s)
MAT     *rot_cols(MAT *A, int i, int k,
                  double c, double s, MAT *out)
MAT     *rot_rows(MAT *A, int i, int k,
                  Real c, Real s, MAT *out)
VEC     *rot_vec (VEC *x, int i, int k,
                  double c, double s, VEC *out)
```

```
#include "zmatrix2.h"
void     zgivens(complex x, complex y, Real &c, complex &s)
ZMAT    *zrot_cols(ZMAT *A, int i, int k,
                   double c, complex s, ZMAT *out)
ZMAT    *zrot_rows(ZMAT *A, int i, int k,
                   double c, complex s, ZMAT *out)
ZVEC    *rot_zvec (ZVEC *x, int i, int k,
                   double c, complex s, ZVEC *out)
```

DESCRIPTION

The routine **givens()** computes a pair (c, s) such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix} \tag{5.1}$$

where $c^2 + s^2 = 1$. The routine **zgivens()** computes a pair (c, s), c real and s complex where

$$\begin{bmatrix} c & -s \\ s^* & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix} \tag{5.2}$$

The matrix formed from the (c, s) pair is a real orthogonal or a complex unitary matrix, and is often referred to as a Givens' rotation. The other routines apply such an orthogonal matrix to vectors and matrices. The actual orthogonal matrix (from

`givens()`)that is applied to vectors and matrices is the matrix

$$J_{ik}(c,s) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \dots & s & & \\ & & \vdots & \ddots & \vdots & & \\ & & -s & \dots & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{5.3}$$

for the real case, and

$$J_{ik}(c,s) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \dots & -s & & \\ & & \vdots & \ddots & \vdots & & \\ & & s^* & \dots & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{5.4}$$

in the complex case.

The routine `rot_cols()` forms $AJ_{ik}(c,s)^T$ and stores the result in `out`. The routine `zrot_cols()` forms $AJ_{ik}(c,s)^*$ and stores the result in `out`.

The routines `rot_rows()` and `zrot_rows()` form $J_{ik}(c,s)A$ and stores the result in `out`.

The routines `rot_vec()` and `rot_vec()` form $J_{ik}(c,s)x$ and stores the result in `out`.

All of the `..rot_...()` routines may be used *in situ* and create a new vector or matrix if the `out` parameter is NULL or is too small to contain the result. The result of the application of the Givens' rotation is returned by each of the `..rot_...()` routines.

Note that $J_{ik}(c,s)^T = J_{ik}(c,-s)$ in the real case, and $J_{ik}(c,s)^* = J_{ik}(c,-s)$ in the complex case. This makes pre- and post-multiplying by transposes of $J_{ik}(c,s)$ easy.

EXAMPLE

```
int     i, k;
VEC     *x;
MAT     *A;
Real    c, s;
  ......
```

```
/* get Givens transformation */
givens(x->ve[i],x->ve[k],&c,&s);
/* apply to x */
rot_vec(x,i,k,c,s);
/* apply symmetrically to A */
rot_cols(A,i,k,c,s);
rot_rows(A,i,k,c,s);
```

BUGS

The `givens()` routine may result in overflow if the `x` and/or `y` parameters are of size greater than $\sqrt{\texttt{HUGE}}$.

SOURCE FILE: `givens.c, zgivens.c`

NAME

hhvec, hhtrcols, hhtrrows, hhtrvec, zhhvec, zhhtrcols, zhhtrrows, zhhtrvec – Householder transformation operations

SYNOPSIS

```
#include "matrix2.h"
VEC  *hhvec(VEC *x, unsigned i0, Real *beta,
            VEC *out, Real *newval)
MAT  *hhtrcols(MAT *A, int i0, int j0, VEC *hh, double beta)
MAT  *hhtrrows(MAT *A, int i0, int j0, VEC *hh, double beta)
VEC  *hhtrvec(VEC *hh, double beta, int i0, VEC *x, VEC *out)

#include "zmatrix2.h"
ZVEC *zhhvec(ZVEC *x, unsigned i0, Real *beta,
             ZVEC *out, complex *newval)
ZMAT *zhhtrcols(ZMAT *A, int i0, int j0, ZVEC *hh,
                double beta)
ZMAT *zhhtrrows(ZMAT *A, int i0, int j0, ZVEC *hh,
                double beta)
ZVEC *zhhtrvec(ZVEC *hh, double beta, int i0, ZVEC *x,
               ZVEC *out)
```

DESCRIPTION

The routines **hhvec()** and **zhhvec()** compute the parameters for a Householder transformation. In particular, given a vector x, a vector v (**== out**) and a real numbers β (**== beta**) and a (possibly complex) number *newval* are computed where the Householder transformation $P = I - \beta vv^*$ satisfies

$$Px = \begin{bmatrix} newval \\ 0 \end{bmatrix}. \tag{5.5}$$

Note that in the case of **x** a real vector, **newval** is real. Note also that **zhhvec()** computes the parameters for a complex vector.

The **x** parameter is not modified. The formulae used are taken from *Matrix Computations* by G. Golub and C. van Loan, p. 40, 1st Edition, (1983), §5.1, pp. 196–196, 2nd Edition, (1989).

If **out** is NULL or too small to hold the v vector, then a new vector is created to store the result. In either case, the result is returned. An error is raised if the **x** vector is NULL.

The routine **hhtrcols()** forms the product AP^T where P is the Householder transformation defined by hh and β (**== beta**). (That is, $P = I - \beta hh\, hh^T$.) The routine **zhhtrcols()** forms the product AP^* where P is the Householder transformation defined by hh and β (**== beta**). (That is, $P = I - \beta hh\, hh^*$.) All rows i with

$i <$ `i0` and columns j with $j <$ `j0` are ignored. The operations are performed *in situ* in `A`.

The routines `hhtrrows()` and `zhhtrrows()` form the product PA where P is the Householder transformation defined by hh and β. Again, all rows i with $i <$ `i0` and columns j with $j <$ `j0` are ignored. The operations is performed *in situ* in `A`.

Finally, the routines `hhtrvec()` and `zhhtrvec()` forms the vector Px where P is the Householder transformation defined by hh and β. The result is stored in `out`. If `out` is NULL or too small to hold the results of the operation, then a new vector is created of the appropriate size. In either case the result is returned.

SOURCE FILE: `hsehldr.c`

NAME

Dsolve, Lsolve, LTsolve, Usolve, UTsolve, zDsolve, zLsolve, zLAsolve, zUsolve, zUAsolve – Basic solve routines

SYNOPSIS

```
#include "matrix2.h"
VEC      *Dsolve (MAT *A, VEC *b, VEC *x)
VEC      *Lsolve (MAT *A, VEC *b, VEC *x, double diag)
VEC      *LTsolve(MAT *A, VEC *b, VEC *x, double diag)
VEC      *Usolve (MAT *A, VEC *b, VEC *x, double diag)
VEC      *UTsolve(MAT *A, VEC *b, VEC *x, double diag)

#include "zmatrix2.h"
ZVEC     *zDsolve (ZMAT *A, ZVEC *b, ZVEC *x)
ZVEC     *zLsolve (ZMAT *A, ZVEC *b, ZVEC *x, double diag)
ZVEC     *zLAsolve(ZMAT *A, ZVEC *b, ZVEC *x, double diag)
ZVEC     *zUsolve (ZMAT *A, ZVEC *b, ZVEC *x, double diag)
ZVEC     *zUAsolve(ZMAT *A, ZVEC *b, ZVEC *x, double diag)
```

DESCRIPTION

The routines `Dsolve()` and `zDsolve()` find and return the solution x of $Dx = b$ where D is the diagonal part of the matrix A (`== A`).

The routines `Lsolve()` and `zLsolve()` find and return the solution x of $Lx = b$ where L is the lower triangular part of A if `diag` is zero; L is the *strictly* lower triangular part of A with `diag` on the diagonal if `diag` is not zero. These routines use forward substitution.

The routines `LTsolve()` and `zLAsolve()` find and return the solutions x of $L^T x = b$ and $L^* x = b$ respectively where L is the lower triangular part of A if `diag` is zero; L is the *strictly* upper triangular part of A with `diag` on the diagonal if `diag` is not zero.

The routines `Usolve()` and `zUsolve()` find and return the solution x of $Ux = b$ where U is the upper triangular part of A if `diag` is zero; U is the *strictly* upper triangular part of A with `diag` on the diagonal if `diag` is not zero. These routines use back substitution.

The routines `UTsolve()` and `zUAsolve()` find and return the solution x of $U^T x = b$ and $U^* x = b$ respectively where U is the upper triangular part of A if `diag` is zero; U is the *strictly* upper triangular part of A with `diag` on the diagonal if `diag` is not zero. These routines use back substitution.

All of these routines may be used *in situ*; that is, they can be used with `b == x`.

If `x` is too small to contain the result then a new vector is created of the appropriate dimension. In either case the solution of the equations is returned.

The rationale behind the use of the **diag** parameter is that often, as in LU factorisation or LDL^T factorisation, the diagonal entry for L is implicit (usually one). The **diag** parameter enables these routines to be used generally, including for the results of QR factorisation, for example.

EXAMPLE

For solving $Ax = b$ using Cholesky factorisation, with only L:

```
MAT     *L;
VEC     *b, *x;
  ......
Lsolve(L,b,x,0.0);    /* use L's diagonal entries */
LTsolve(L,x,x,0.0);
```

For solving $Ax = b$ using LU factorisation with L unit lower triangular and no pivoting:

```
MAT     *L, *U;
VEC     *b, *x;
  ......
Lsolve(L,b,x,1.0);    /* L unit lower triangular */
Usolve(U,b,x,0.0);
```

SEE ALSO

LUsolve(), **zLUsolve()**, **CHsolve()**, **LDLsolve()**, **QRsolve()**, **zQRsolve()**

SOURCE FILE: **solve.c**, **zsolve.c**

NAME

LDLupdate, QRupdate – factorisation update routines

SYNOPSIS

```
#include "matrix2.h"
MAT     *LDLupdate(MAT *LDL, VEC *w, double alpha)
MAT     *QRupdate (MAT *Q, MAT *R, VEC *u, VEC *v)
```

DESCRIPTION

The routine **LDLupdate()** modifies the matrix **LDL** which is assumed to contain (in compact form) the LDL^T factorisation of a matrix A. The L matrix is the strictly lower triangular part of **LDL**, except with ones on the diagonal, while D is the diagonal of **LDL**, so that $A = LDL^T$. The matrix **LDL** is modified *in situ* so that if L_+ and D_+ denote the factors described by **LDL** *after* the routine, then

$$L_+ D_+ L_+^T = A + \alpha w w^T$$

where α is the value of **alpha** and w is **w**. The modified **LDL** matrix is returned.

The method used for updating the factorisation is given in "Methods for modifying matrix factorisations" by P. Gill, G. Golub, W. Murray and M. Saunders, *Mathematics of Computations*, **28**, pp. 505–535 (1974). The particular algorithm used is the algorithm C1 of their paper.

This routine may fail if $A + \alpha w w^T$ is not sufficiently positive definite; if this failure occurs, then an **E_POSDEF** error is raised.

The routine **QRupdate()** updates the QR factorisation of a matrix $A = QR$. Unlike the previous routine, this routine requires the explicit factors Q and R of A. These can be obtained from the compact form by means of the routines **makeQ()** and **makeR()**. If the matrices **Q** and **R** after the routine are denoted Q_+ and R_+ respectively, then

$$Q_+ R_+ = Q(R + uv^T) = A + (Qu)v^T.$$

Setting $u = Q^T w$ gives $Q_+ R_+ = A + wv^T$.

If **Q** is NULL, then only the **R** matrix is modified. The **R** matrix is returned.

The routine is based on one given in *Matrix Computations* by G. Golub and C. van Loan, pp. 437–443, 1st Edition (1983), pp. 593–594, 2nd Edition (1989).

EXAMPLE

Updating LDL^T factorisation:

```
MAT     *A, *LDL;
VEC     *u;
double alpha;
```

```
  ......
LDL = m_copy(A,MNULL);
LDLfactor(LDL);
  ......
/* A <- A + alpha.u.u^T */
LDLupdate(LDL,u,alpha);
```

Updating QR factorisation:

```
MAT     *A, *QR, *Q, *R;
VEC     *diag, *beta, *u, *v, *w;
  ......
QR = m_copy(A,MNULL);
QRfactor(QR,diag,beta);
Q = makeQ(QR,diag,beta,MNULL);
R = makeR(QR,MNULL);
  ......
/* A <- A + w.v^T */
u = v_get(Q->m);
u = vm_mlt(Q,w,u);
QRupdate(Q,R,u,v);
```

SOURCE FILE: `update.c`

NAME

schur, symmeig, trieig, zschur – Eigenvalue routines

SYNOPSIS

```
#include "matrix2.h"
MAT     *schur(MAT *A, MAT *Q)
VEC     *symmeig(MAT *A, MAT *Q, VEC *out)
VEC     *trieig(VEC *a, VEC *b, MAT *Q)

#include "zmatrix2.h"
ZMAT    *zschur(MAT *A, MAT *Q)
```

DESCRIPTION

The routine **schur()** computes the Real Schur decomposition of the matrix **A**. That is, it computes a block upper triangular matrix T and an orthogonal matrix Q such that

$$Q^T A Q = T.$$

The matrix T has diagonal blocks of sizes 1×1 and 2×2. The eigenvalues of these diagonal blocks are the eigenvalues of the original A matrix. The algorithm used to find the eigenvalues of A is the Francis QR algorithm. This algorithm is described in *Matrix Computations* by G. Golub and C. van Loan, pp. 231–236, 1st Edition (1983), pp. 377–381, 2nd Edition (1989).

The matrix **A** is overwritten with T, and if **Q** is not NULL and the correct size, then the Q matrix is stored in it.

The routine **zschur()** computes the complex Schur factorisation of **A**. That is, it computes an upper triangular matrix T and a unitary matrix Q such that

$$Q^* A Q = T.$$

The eigenvalues of A are the diagonal entries of T. The algorithm is a complex version of the Francis QR algorithm, and is, in fact, somewhat simplified in the complex case.

The routine **symmeig()** computes the eigenvalues of a *symmetric* matrix. It also computes an orthogonal matrix Q such that

$$Q^T A Q = \Lambda$$

where Λ is the diagonal matrix of eigenvalues. The algorithm used to find the eigenvalues of A consists of conversion to symmetric Hessenberg (symmetric tridiagonal) form, and then applying **trieig()** to obtain the eigenvalues of the tridiagonal matrix.

The eigenvalues are stored in **out** provided it is not NULL and is sufficiently large to contain all the eigenvalues. The vector containing the eigenvalues is returned. The matrix **A** is *not* overwritten.

The routine `trieig()` computes the eigenvalues of the symmetric tridiagonal matrix

$$T = \begin{bmatrix} a_0 & b_0 & & & \\ b_0 & a_1 & b_1 & & \\ & b_1 & a_2 & \ddots & \\ & & \ddots & \ddots & b_{n-2} \\ & & & b_{n-2} & a_{n-1} \end{bmatrix}. \tag{5.6}$$

The algorithm used is a "chasing" technique described in *Matrix Computations*, pp. 278–281, 1st Edition, pp. 421–424, 2nd Edition. It also accumulates the matrix Q such that Q^TTQ is diagonal. To compute the correct Q matrix, `Q` should be initialised to the identity matrix on entry to `trieig()`. (See `m_ident()`.)

The values in the `a` and `b` vectors are overwritten. At the end of the routine, `a` contains the eigenvalues, and the `b` vector is zero.

In all of the above routines, if the matrix `Q` is NULL on entry, then no calculation of the Q matrices is performed. This should speed up the routines somewhat if only the eigenvalues are needed.

EXAMPLE

Computing real Schur decomposition of (possibly) nonsymmetric A:

```
MAT    *A, *S, *Q, *X_re, *X_im;
VEC    *evals_re, *evals_im;
  ......
S = m_copy(A,MNULL);
Q = m_get(A->m,A->m);
schur(S,Q);
/* get eigenvalues (real, imaginary parts) */
evals_re = v_get(A->m);
evals_im = v_get(A->m);
schur_evals(S,evals_re,evals_im);
/* get eigenvectors (real, imaginary parts) */
X_re = m_get(A->m,A->m);
X_im = m_get(A->m,A->m);
schur_evecs(S,Q,X_re,X_im);
```

Computing eigenvalues and eigenvectors of a real symmetric matrix:

```
MAT    *A, *Q;
VEC    *evals;
  ......
evals = v_get(A->m);
evals = symmeig(A,Q,evals);
```

The Q matrix contains the eigenvectors.

Computing the eigenvalues and eigenvectors of a symmetric tridiagonal matrix defined by the vectors a (the diagonal entries) and b (the off-diagonal entries):

```
MAT     *Q;
VEC     *a, *b;
  ......
Q = m_get(a->dim,a->dim);
m_ident(Q);     /* must initialise Q */
trieig(a,b,Q);
/* a is now the vector of eigenvalues */
```

SEE ALSO

The Hessenberg routines in **hessen.c** and **zhessen.c**.

BUGS

It is up to the caller of **symmeig()** to ensure that the **A** matrix is symmetric. Symmetry of **A** is neither checked nor enforced in **symmeig()**.

SOURCE FILE: **symmeig.c, schur.c, zschur.c**

NAME

schur_evals, schur_vecs – Extracting eigenvalues and eigenvectors from the Schur form

SYNOPSIS

```
#include "matrix2.h"
void schur_evals(MAT *T, VEC *re_evals, VEC *im_evals)
MAT *schur_vecs(MAT *T, MAT *Q, MAT *X_re, MAT *X_im)
```

DESCRIPTION

Both of these routines assume that T is the matrix computed by the **schur()** routine; Q is the orthogonal matrix computed by **schur()**.

The routine **schur_evals()** compute the eigenvalues of a matrix T in Schur form (block diagonal with 1×1 or 2×2 blocks). The kth eigenvalue of $A = QTQ^T$ is **re_evals->ve[k]** $+i$**im_evals->ve[k]**. At worst this requires solving a series of quadratics; however, it does simplify the task of computing eigenvalues. Complex eigenvalues come in complex conjugate pairs.

The routine **schur_vecs()** computes the matrix $X =$ **X_re** $+ i$ **X_im** such that $X^{-1}AX$ is the diagonal matrix of eigenvalues where $T = Q^TAQ$ as computed by the **schur()** routine. The columns of X are computed by means of one step of inverse iteration using the eigenvalues as computed from the Schur form. This method is usually accurate provided the eigenvalues are not too close together. The computed kth column of X is real if the computed kth eigenvalue is real. The ordering of the columns is consistent with the ordering of the eigenvalues generated by **schur_evals()**.

EXAMPLE

See example for **schur()** above.

BUGS

It is a bit difficult to check that the computed X is correct if it is complex.

SEE ALSO

schur()

SOURCE FILE: **schur.c**

NAME

svd, bisvd – Singular Value Decomposition routines

SYNOPSIS

```
#include "matrix2.h"
VEC       *svd(MAT *A, MAT *U, MAT *V, VEC *out)
VEC       *bisvd(VEC *d, VEC *f, MAT *U, MAT *V)
```

DESCRIPTION

The routine **svd()** performs a complete Singular Value Decomposition (SVD) on the matrix A. That is, it computes orthogonal matrices U and V such that UAV^T is diagonal and the diagonal entries are called the *singular values* of the matrix A. The first $\min(m, n)$ singular values are stored in the **out** vector which is also returned. Note that the SVD is defined for nonsquare as well as square matrices.

If NULLs are passed for either or both **U** and **V**, then that orthogonal matrix will not be accumulated. This saves both time and space, if just the singular values are desired and not the U or V matrices. If **out** is NULL on entry to **svd()**, then a vector of the appropriate size is created to store the singular values, which is returned.

The SVD is computed by first transforming the matrix into a bidiagonal matrix (c.f. **schur()** where a matrix is transformed into Hessenberg form for eigenvalue calculations) and then applying **bisvd()**. If a matrix is already in bidiagonal form, then **bisvd()** can be called directly. The vector **d** contains the diagonal entries and **f** contains the super-diagonal entries. As for **svd()**, if NULLs are passed for either or both **U** and **V**, then that (or both) orthogonal matrix will not be accumulated. For correct results using **bisvd()**, you should initialise **U** and **V** to be identity matrices using **m_ident()** before calling **bisvd()**.

The rank of a matrix can be estimated by counting the number of singular values whose magnitude exceeds a specified tolerance. This tolerance for accurately computed matrices should probably be about 100 times **MACHEPS**; otherwise it should about an order of magnitude larger than the errors in the matrix.

The algorithm used follows *Matrix Computations* by Golub and van Loan, pp. 430–435, 2nd Edition (1989).

EXAMPLE

For computing the SVD of A:

```
MAT     *A, *U, *V;
VEC     *svdvals;
  ......
U = m_get(A->m,A->m);
V = m_get(A->n,A->n);
svdvals = svd(A,U,V,VNULL);
```

For computing the SVD of the bidiagonal matrix defined by d (the diagonal entries) and f (the super-diagonal entries):

```
MAT     *U, *V;
VEC     *d, *f;
  ......
U = m_get(d->dim,d->dim);
V = m_get(d->dim,d->dim);
m_ident(U);      /* must initialise U and V */
m_ident(V);
bisvd(d,f,U,V)
/* d now contains the singular values */
```

SOURCE FILE: `svd.c`

NAME

m_exp, m_poly, m_pow – Matrix exponentials, polynomials and powers

SYNOPSIS

```
#include "matrix2.h"
MAT * m_pow(MAT *A, int p, MAT *out)
MAT *_m_pow(MAT *A, int p, MAT *tmp, MAT *out)
MAT * m_exp(MAT *A, double eps, MAT *out)
MAT *_m_exp(MAT *A, double eps, MAT *out,
            int *qout, int *jout)
MAT *m_poly(MAT *A, VEC *a, MAT *out)
```

DESCRIPTION

The routine **m_pow** sets a matrix $A \in R^{n \times n}$ to the power p, where p can be any non-negative integer. (Use **m_inverse()** for negative p.) The result is placed in the matrix **out** = A^p. The routine is based on the binary powering algorithm (see Golub and Van Loan, *Matrix computations*, John Hopkins University Press, Baltimore, 2nd edition,1989). The algorithm requires at most $2\lfloor log_2(p) \rfloor n^3$ flops where n is the dimension of the matrix.
_m_pow it is a variant of the routine **m_pow** which uses **tmp** as a workspace matrix.

The routine **m_exp** computes an approximation of

$$e^A \approx I + A + A^2/2! + \ldots + A^q/q! + \ldots$$

using the Padé approximation

$$\exp(A) \approx R_{qq}(A) = D_q(A)^{-1} N_q(A)$$

where

$$N_q(A) = \sum_{k=0}^{q} c_k A^k, \quad D_q(A) = \sum_{k=0}^{q} c_k (-A)^k,$$

and

$$c_k = \frac{(2q-k)!q!}{(2q)!k!(q-k)!}.$$

The computed exponential is placed in **out**. The degree q is determined from an error tolerance **eps** given by the user. Padé approximation is good for A with a small norm, therefore this condition can be ensured by applying repeated squaring $(R_{qq}(A/2^j))^{2^j}$, where j is chosen so that $\|A/2^j\| \leq 1/2$. The Padé approximate can be more efficient by using special Horner regrouping techniques to evaluate matrix polynomial. The relative error of Padé approximate for a matrix with $\|A\| \leq 0.5$ can be estimated by

$$\frac{\|e^A - (R_{qq}(A/2^j))^{2^j}\|_\infty}{\|e^A\|_\infty} \leq \epsilon(q,q)\|A\|_\infty e^{\epsilon(q,q)\|A\|_\infty},$$

and $\epsilon(q,q) = 2^{3-(2q)}(q!)^2/((2q)!(2q+1)!)$.
In `_m_exp` the degree q is returned in `qout`, and j is returned in `jout`. The routines `m_exp` and `_m_exp` are based on the paper: *"Nineteen Dubious Ways to Compute The Exponential of the Matrix"*, SIAM Rev. 20(4), p.801–836, 1987 by C. Moler and C. Van Loan and the book G.H. Golub, C. Van Loan *"Matrix Computations"*, Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

`m_poly` evaluates the polynomial of a matrix A

$$p(A) = a_0 I + a_1 A + \ldots + a_q A^q,$$

where $a_0, a_1, a_2, \ldots, a_q$ are given by the vector `a` with $q =$ `a->dim-1`. The result is placed in `out`. The algorithm used to compute the matrix polynomials in the Padé approximation and in `m_poly` is based on the paper *"A note on the Evaluation of Matrix Polynomials"*, IEEE Transactions on Automatic Control 24 (1979), p. 209–228 by C.Van Loan. The paper describes a method that is faster and more memory efficient than the standard Horner's method.

SOURCE FILE: `mfunc.c`

NAME

`fft, ifft` – Fast Fourier Transform and inverse

SYNOPSIS

```
#include "matrix2.h"
void      fft(VEC *x_re, VEC *x_im)
void      ifft(VEC *x_re, VEC *x_im)
```

DESCRIPTION

The routine `fft()` performs a fast Fourier transform on the vector $x =$ `x_re` $+$ i`x_im`. The transform is computed *in situ*. It does require that the dimension of x is a power of two.

The routine `ifft()` performs the inverse fast Fourier transform of $x =$ `x_re` $+$ i`x_im`. As with `fft()` it is computed *in situ*, and the dimension of x must be a power of two.

SOURCE FILE: `fft.c`