Chapter 6

Sparse Matrix Operations

The following routines are described in the following pages:

To use these routines use the include statements

```
#include "matrix.h"
#include "sparse.h"
#include "sparse2.h"
```

NAME

`sp_get_mat, sp_free_mat, sp_resize, sp_compact` – allocate, free and resize sparse matrices

SYNOPSIS

```
#include "matrix.h"
#include "sparse.h"
sp_mat   *sp_get_mat(m, n, maxlen)
int      m, n, maxlen;

void    sp_free_mat(A)
sp_mat  *A;

sp_mat  *sp_resize(A,m,n)
sp_mat  *A;
int     m, n;

sp_mat  *sp_compact(A,tol)
sp_mat  *A;
double  tol;
```

DESCRIPTION

The routine `sp_get_mat()` allocates and initialises a `sp_mat` data structure. It is initialised so that the `sp_mat` returned is $m \times n$, and that there are already `maxlen` elements allocated for each row. This is to avoid excessive memory allocation/de-allocation later on. Initially there are no elements in the matrix and so the `len` entry of every row will be zero just after calling this routine.

The routine `sp_free_mat()` de-allocates all memory associated with the sparse matrix structure `A`.

The routine `sp_resize()` re-sizes the matrix `A` to be size $m \times n$. Rows are expanded as necessary, and information is not lost unless the matrix is reduced in size.

It should be noted that the sparse matrix data structure requires a separate memory allocation for each row, unlike the dense matrix data structure. Thus more care must be taken with sparse matrix data structures to avoid excessive time spent in memory allocation and de-allocation.

An `E_MEM` error will be raised if the memory cannot be allocated.

Finally, the routine `sp_compact()` removes zero elements and elements with magnitude no more than `tol` from the sparse matrix `A`. It does this *in situ* and requires no additional storage. It may, however, raise an `E_RANGE` error if `tol` is negative.

EXAMPLE

```
sp_mat *A;
int    i, j, m, n;
  ......
/* get sparse matrix, with room for 5 entires per row */
A = sp_get_mat(m,n,5);
  ......
sp_set_val(A,i,j,3.1415926);
  ......
/* double size of A matrix */
sp_resize(A,2*m,2*n);
  ......
```

```
/* remove entries of size <= 10^{-7} */
sp_compact(A,1e-7);
  ......
/* destroy A matrix */
sp_free_mat(A)
```

SOURCE FILE: `sparse.c`

NAME

`sp_cp_mat, sp_cp_mat2` – Spare matrix copy routines

SYNOPSIS

```
#include "matrix.h"
#include "sparse.h"
sp_mat  *sp_cp_mat(A)
sp_mat  *A;

sp_mat  *sp_cp_mat2(A,OUT)
sp_mat  *A, *OUT;
```

DESCRIPTION

The routine `sp_cp_mat()` returns a copy of `A` so that the object returned can be freely modified without affecting `A`. (That is, it is a "deep" copy.) A new data structure is allocated and initialised in the process.

The routine `sp_cp_mat2()` copies `A` into `OUT`, using all allocated entries in `OUT` in doing so. In this way it avoids memory allocation and preserves the structure of the nonzeros of `OUT` as much as possible.

The routine `sp_cp_mat2()` is especially useful in conjunction with the symbolic and incomplete Cholesky factorisation routines. The idea is that the symbolic Cholesky factorisation allocates all the necessary nonzero entries; if a matrix with the original nonzero pattern is to be factored, it can be copied using `sp_cp_mat2()` into the symbolicly factored matrix, and the incomplete Cholesky factorisation routine can then be used to factor the copied matrix without fill-in or memory allocation. See the manual entries on `spICHfactor()` and `spCHsymb()` for more details.

EXAMPLE

```
sp_mat  *A, *B;
  ......
A = sp_get_mat(100,100,4);
for ( i = 0; i < A->m; i++ )
    sp_set_val(A,i,i+1,...);
  ......
/* copy A matrix */
B = sp_cp_mat(A);
  ......
for ( i = 0; i < B->m; i++ )
    sp_set_val(B,i,i+2,...);
sp_cp_mat2(A,B);
/* now B and A represent same matrix,
   but B has allocated (i,i+2) entries */
```

SEE ALSO

`sp_get_mat()` and `sp_resize()`

SOURCE FILE: `sparse.c`

NAME

`sp_get_val, sp_set_val` – Access to entries of a sparse matrix

SYNOPSIS

```
#include "matrix.h"
#include "sparse.h"
double  sp_get_val(A,i,j)
sp_mat  *A;
int     i, j;

double  sp_set_val(A,i,j,val)
sp_mat  *A;
int     i, j;
double  val;
```

DESCRIPTION

The routine `sp_get_val()` returns the value in the (i,j)'th entry of A. If the (i,j)'th entry has not been allocated, then zero is returned. The routine `sp_set_val()` sets the value of the (i,j)'th entry of A to `val`. If the (i,j)'th entry is not already allocated, then if there is sufficient allocated space for the new entry, other entries will be shifted as needed; if there is not sufficient space, then the row will be expanded by `row_xpd()`. Setting the value of an entry to zero does not "de-allocate" the entry.

If `i` or `j` are negative or larger than or equal to `A->m` or `A->n` respectively, then an `E_BOUNDS` error will be raised.

EXAMPLE

```
sp_mat *A;
int    i, j;
double val;
  ......
A = sp_get_mat(100,100,4);
  ......
sp_set_val(A,i,j,(double)(i+j));
  ......
val = sp_get_val(A,i,j);
```

SEE ALSO

`row_set_val()`

BUGS

A more efficient approach would be to use a balanced tree structure.

SOURCE FILE: `sparse.c`

NAME

`sp_mv_mlt, sp_vm_mlt` – sparse matrix–vector multiplication routines

SYNOPSIS

```
#include        "matrix.h"
#include        "sparse.h"
VEC     *sp_mv_mlt(A,x,out)
sp_mat  *A;
VEC     *x, *out;

VEC     *sp_vm_mlt(A,x,out)
sp_mat  *A;
VEC     *x, *out;
```

DESCRIPTION

The routine `sp_mv_mlt()` sets `out` to be the matrix–vector product Ax, and `sp_vm_mlt()` sets `out` to be the vector–matrix product $x^T A$ (or equivalently, $A^T x$). The vector `out` is created or resized if necessary, in particular, if `out == VNULL`.

Both avoid thrashing on virtual memory machines. Unlike the dense matrix routines, there is no set of "core" routines for performing the underlying inner products etc efficiently.

EXAMPLE

```
sp_mat *A;
VEC    *x, *y;
  ......
A = sp_get_mat(100,100,4);
x = get_vec(A->m);
  ......
/* compute y <- A.x */
y = sp_mv_mlt(A,x,VNULL);
/* compute y^T <- x^T.A */
sp_vm_mlt(A,x,y);
```

SOURCE FILE: `sparse.c`

NAME

`sp_col_access, sp_diag_access` – set up access paths

SYNOPSIS

```
#include        "matrix.h"
#include        "sparse.h"
sp_mat  *sp_col_access(A)
sp_mat  *A;

sp_mat  *sp_diag_access(A)
sp_mat  *A;
```

DESCRIPTION

In order to achieve fast access down columns, extra access paths were added. However, operations such as setting values of (unallocated) entries upset these access paths. Rather than keep them up-to-date continuously, which is rather expensive in computational time, these access paths are only updated when requested.

There are flags in the sparse matrix data structure which indicate if these access paths are still valid: they are `A->flag_col` and `A->flag_diag` respectively. (Nonzero indicates they are valid.)

The fields of `A` that are set up by `sp_col_access()` are the `A->start_row[]` and `A->start_idx[]` fields. The values `A->start_row[col]` and `A->start_idx[col]` give the first row, and index into that row where the first allocated entry of column `col`. The other fields set up by `sp_col_access()` are the `nxt_row` and `nxt_idx` fields of each `row_elt` data structure in the sparse matrix `A`. For a more thorough description of how these may be used, see §3.2.

The `sp_diag_access()` function only sets the `diag` field of the `sp_row` data structure for each row in the sparse matrix `A`.

EXAMPLE

Using the column access fields to chase the entries in

```
sp_mat *A;
int    i, j, idx;
sp_row *r;
row_elt *e;
  ......
/* set up A matrix */
sp_set_val(A,i,j,3.1415926);
  ......
sp_col_access(A);
/* chase column j of A */
i = A->start_row[j];
idx = A->start_idx[j];
while ( i >= 0 )
{
    r = &(A->row[i]);
    e = &(r->elt[idx]);
    printf("Value A[%d][%d] = %g\n", i, j, e->val);
    i = e->nxt_row;
    idx = e->nxt_idx;
}
```

Getting diagonal values:

```
sp_mat *A;
int    i, idx;
double val;
  ......
sp_diag_access(A);
  ......
/* to get A[i][i] */
idx = A->row[i].diag;
if ( idx < 0.0 )
    val = 0.0;
else
    val = A->row[i].elt[idx].val;
```

BUGS

The flags are not guaranteed to remain correct if you modify the sparse matrix data structures directly, only if you use `sp_set_val()` etc.

SOURCE FILE: `sparse.c`

NAME

sp_zero_mat – Zeros sparse matrix

SYNOPSIS

```
#include "matrix.h"
#include "sparse.h"
sp_mat  *sp_zero_mat(A)
sp_mat  *A;
```

DESCRIPTION

Zeros the allocated entries of A. Does not change the "allocation" status of entries of A.

EXAMPLE

One way to clear the sparsity structure of a matrix follows:

```
sp_mat *A;
  ......
sp_zero_mat(A);     /* zeros entries */
sp_compact(A,0.0);  /* removes zero entries */
```

SOURCE FILE: sparse.c

NAME

`sp_fout_mat, sp_out_mat` – Sparse matrix output

SYNOPSIS

```
#include <stdio.h>
#include "matrix.h"
#include "sparse.h"
sp_fout_mat(fp,A)
FILE    *fp;
sp_mat  *A;

sp_out_mat(A)
sp_mat  *A;
```

DESCRIPTION

The routine `sp_fout_mat()` produces a printed representation of the sparse matrix `A` on the file or stream `fp`. This representation can also be read in by `sp_fin_mat()`.

The routine `sp_out_mat()` is just a macro

```
#define sp_out_mat(A)   sp_fout_mat(stdout,(A))
```

which sends the output to `stdout`.

The form of the output consists of a header, a list of rows, each of which contains a sequence of entries. Each entry is made up of a column number, a colon, and the value for that entry. For example, the dense matrix

```
Matrix: 3 by 4
row 0:            0            1            0           -1
row 1:            1            2            0            0
row 2:            0            0            1            1
```

can be represented as the sparse matrix with printed representation

```
SparseMatrix: 3 by 4
row 0: 1:1                  3:-1
row 1: 0:1                  1:2
row 2: 2:1                  3:1
```

EXAMPLE

```
sp_mat *A;
int    i, j;
FILE   *fp;
  ......
sp_set_val(A,i,j,3.1415926);
  ......
sp_out_mat(A);     /* prints to stdout */
if ( (fp=fopen("output.dat","w")) == NULL )
    error(E_EOF,"func_name");
sp_fout_mat(fp,A); /* prints to output.dat */
```

SEE ALSO

`sp_fin_mat()`, `sp_in_mat()`

SOURCE FILE: `sparseio.c`

NAME

`sp_fin_mat, sp_in_mat` – Input sparse matrix

SYNOPSIS

```
#include <stdio.h>
#include "matrix.h"
#include "sparse.h"
sp_mat  *sp_fin_mat(fp)
FILE    *fp;

sp_mat  *sp_in_mat()
```

DESCRIPTION

The routine `sp_fin_mat()` allocates, initialises and inputs a sparse matrix of the size input from file/stream `fp`. The routine `sp_in_mat()` is just a macro

```
#define sp_in_mat()     sp_fin_mat(stdin)
```

If the input is not from a terminal, then the format must be the same as that produced by `sp_fout_mat()` or `sp_out_mat()`. If the input is from a terminal (`isatty(fileno(fp)) != 0`) then the user is prompted for the necessary values and information.

EXAMPLE

```
sp_mat *A;
FILE   *fp;
  ......
A = sp_in_mat();    /* read matrix from stdin */
if ( (fp=fopen("input.dat","r")) == NULL )
    error(E_INPUT,"func_name");
A = sp_fin_mat(fp); /* read matrix from input.dat */
```

Example of interactive input session:

```
SparseMatrix: input rows cols: 10 15
Row 0:
Enter <col> <val> or 'e' to end row
Entry 0: 2 -7.32
Entry 1: 3 1.5
Entry 2: 0 2.75       # Note: entry ignored
Entry 2: 4 1.3
Entry 3: e
Row 1:
Enter <col> <val> or 'e' to end row
Entry 0: e            # Note: empty row
Row 2:
Enter <col> <val> or 'e' to end row
Entry 0: ....
  ......
```

BUGS

Does not allow more than a hundred entries pre row.

The simple "editing" facilities of `fin_mat()` are not provided.

SOURCE FILE: `sparseio.c`

NAME

sp_get_idx, sp_get_row, row_xpd, sp_row_merge, _row_mltadd, row_set_val, fout_row – Sparse row support routines

SYNOPSIS

```
#include "matrix.h"
#include "sparse.h"
int     sp_get_idx(r,col)
sp_row  *r;
int     col;

sp_row  *sp_get_row(maxlen)
int     maxlen;

sp_row  *row_xpd(r,newlen)
sp_row  *r;
int     newlen;

sp_row  *sp_row_merge(r1,r2,r_out)
sp_row  *r1, *r2, *r_out;

sp_row  *_row_mltadd(r1,r2,alpha,j0,r_out)
sp_row  *r1, *r2, *r_out;
double  alpha;
int     j0;

double  row_set_val(r,j,val)
sp_row  *r;
int     j;
double  val;

void    fout_row(fp,r)
FILE    *fp;
sp_row  *r;
```

DESCRIPTION

The routine `sp_get_idx()` uses binary search to find the location of the element in row `r` whose column number is `col`, which is returned. If the row `r` contains an entry with column number `col`, then the index `idx` into `r->elt[idx]` (being the entry in that row) is given by `idx = sp_get_idx(r,col)`. If there is no element in row `r` whose column is `col`, then `idx = sp_get_idx(r,col)` is negative, but `-(idx+2)` is the index where an entry with column number `col` would be inserted. An internal error is flagged by returning -1.

The routine `sp_get_row()` allocates and initialises a sparse row data structure (type `sp_row`) with memory for `maxlen` entries.

The routine `row_xpd()` reallocates the row `r` to allocate room for at least `newlen` entries. If the current length (`r->len`) is already at least size `newlen`, then the row's allocated memory is approximately double in size.

The routine `sp_row_merge()` merges two sparse rows, with values in `r1` taking precedence over values in `r2` if they have the same column number.

The routine `_row_mltadd()` sets `r_out` to be `r1+alpha.r2`, by a "merging" process. The applies only to columns with column numbers greater than or equal to `j0`.

The routine `row_set_val()` sets the `j`'th element of row `r` to be `val`. Memory allocation and shifting of entries is done as needed.

The routine `fout_row()` prints a representation of the sparse row `r` onto file/stream `fp`. This representation is not intended to be read back in.

EXAMPLE

Extracting a sparse matrix entry:

```
sp_mat *A;
sp_row *r, r1, r2;
row_elt *e;
int    i, j, idx, idx1;
  ......
/* compute A[i][j] */
r = &(A->row[i]);
idx = sp_get_idx(r,j);
if ( idx < 0 )
    /* -(idx+2) is where an entry in
       column j would go if there were one */
    val = 0.0;
else
    val = r->elt[idx].val;
```

Shuffling a row:

```
/* build temporary sparse row r1
     containing shuffled entries of r */
r1 = sp_get_row(10);
for ( idx = 0; idx < r->len; idx++ )
{
    e = &(r->elt[idx]);
    old_col = e->col;
    new_col = ......;
    row_set_val(r1,new_col,e->val);
    /* r1 will be expanded if necessary */
}
```

Expanding a temporary row:

```
r1 = row_xpd(r1,2*r1->len + 1);
```

Printing out a row as a separate structure for debugging:

```
printf("Temporary row r1:\n");
fout_row(stdout,r1);
```

SOURCE FILE: `sparse.c`

NAME

`spCHfactor, spCHsolve, spICHfactor, spCHsymb` – Sparse Cholesky factorisation and solve

SYNOPSIS

```
#include "matrix.h"
#include "sparse2.h"
sp_mat  *spCHfactor(A)
sp_mat  *A;

VEC     *spCHsolve(LLT,b,out)
sp_mat  *LLT;
VEC     *b, *out;

sp_mat  *spICHfactor(A)
sp_mat  *A;

sp_mat  *spCHsymb(A)
sp_mat  *A;
```

DESCRIPTION

The main routine of these is `spCHfactor()` which performs a sparse Cholesky factorisation of the matrix `A`, which is performed *in situ*. The resulting system can be solved by `spCHsolve()` which returns `out` which is set to be the solution of `A.out = b` where `LLT` is the result of applying `spCHfactor()` to `A`. To illustrate, the following code solves the system `A.x = b` for `x`:

```
/* Initialise A and b */
  ......
spCHfactor(A);
/* A is now the Cholesky factorisation of original A,
     stored in compact form */
spCHsolve(A,b,x);
```

The other routines provide alternatives to `spCHfactor()`. The routine `spCHfactor()` allocates memory for fill-in as needed. As noted above regarding `sp_col_access()` etc, this destroys the column access data structure's validity, and so results in more time spent searching for elements within rows. This can be avoided if there is no fill-in.

The routine `spICHfactor()` performs Cholesky factorisation **assuming no fill-in**. It does not even check that fill-in would occur in a correct Cholesky factorisation. This routine is considerably faster than using `spCHfactor()`, but if the actual factorisation results in fill-in, the computed "Cholesky" factor used in `spCHsolve()` will not give correct solutions.

The routine `spCHsymb()` performs a "symbolic" factorisation of `A`. That is, no numerical calculations are performed. Instead, the `A` matrix after `spCHsymb()` has executed, contains allocated all entries where fill-in would occur. This means that `spCHfactor()` is effectively equivalent to `spCHsymb()` followed by `spICHfactor()`. The advantage with having two separate routines is that the fill-in can be computed once for a given pattern of nonzeros, and used for a number of sparse matrices with just that pattern of nonzeros with `spICHfactor()`. The code to do this would look something like this:

```
/* Initialise pattern matrix */
  ......
```

```
spCHsymb(pattern);
for ( i = 0; i < num_matrices; i++ )
{   /* set up A matrix -- same nonzero pattern */
      ......
    sp_zero_mat(pattern);
    sp_cp_mat2(A,pattern);
    spICHfactor(pattern);
    /* set up b vector */
      ......
    spCHsolve(pattern,b,x);
      ......
}
```

The `spICHfactor()` routine can also be used to provide a good pre-conditioner for the pre-conditioned conjugate gradient routines `pccg()` and `sp_pccg()`.

BUGS

An `E_POSDEF` error may be raised by `spICHfactor()` even if the `A` matrix is positive definite.

An `E_POSDEF` error will be raised by `spCHsymb()` if a diagonal entry is missing.

SEE ALSO

`sp_cp_mat2`, `sp_zero_mat`, `pccg`, `sp_pccg`

SOURCE FILE: `spCHfactor.c`

NAME

`spLUfactor, spLUsolve, spLUTsolve` – sparse LU factorisation (Gaussian elimination)

SYNOPSIS

```
#include "matrix.h"
#include "sparse2.h"
sp_mat  *spLUfactor(A,pivot,alpha)
sp_mat  *A;
PERM    *pivot;
double  alpha;

sp_mat  *spILUfactor(A,alpha)
sp_mat  *A;
double  alpha;

VEC     *spLUsolve(LU,pivot,b,x)
sp_mat  *LU;
PERM    *pivot;
VEC     *b, *x;

VEC     *spLUTsolve(LU,pivot,b,x)
sp_mat  *LU;
PERM    *pivot;
VEC     *b, *x;
```

DESCRIPTION

The routine `spLUfactor()` performs Gaussian elimination with partial pivoting on `A` with a Markowitz type modification to avoid excessive fill-in. The `alpha` parameter determines the trade-off between fill-in and numerical stability; the row that is swapped with the pivot row is the one with the smallest number of nonzero entries after the pivot column which has magnitude at least `alpha` times the largest magnitude entry in the pivot column. This parameter must therefore be between zero and one inclusive. If it is set to zero then `alpha` is effectively set to machine epsilon, `MACHEPS`.

Note that `A` is over-written during the factorisation, and that `pivot` must be set before being passed to `spLUfactor()`.

The routine `spILUfactor()` computes a modified incomplete LU factorisation without pivoting. Thus no fill-in is generated and all pivot (i.e. diagonal entries) are guaranteed to have magnitude $\geq \alpha$ by adding to the diagonal entries. Thus in exact arithmetic it computes $LU = A + D$ for some diagonal matrix D. Since it is not a factorisation of A, it cannot be used directly to solve systems of equations.

The routine `LUsolve()` solves the system $Ax = b$. The routine `LUTsolve()` solves the system $A^T x = b$. Both of these use the the matrix as factored by `spLUfactor()`. Neither of these can be used *in situ* with `x == b`.

EXAMPLE

Code for solving the sparse systems of equations $Ax = b$ and $A^T y = b$ is given below:

```
/* Set up A and b */
  ......
pivot = get_perm(A->m);
x     = get_vec(A->n);
```

```
y     = get_vec(A->m);
spLUfactor(A,pivot,0.1);
x = spLUsolve(A,pivot,b,x);
y = spLUTsolve(A,pivot,b,y);
```

An example of the use of `spILUfactor()` will be given under the entry for `pccg()`, `cgs()` and `lsqr()`.

BUGS

There may be problems with `spLUsolve()` and `spLUTsolve()` if `A` is not square.

The routine `spLUfactor()` does not implement a full Markowitz strategy.

SEE ALSO

`spCHfactor()`, `MACHEPS`, `LUfactor()`

SOURCE FILE: `spLUfctr.c`

NAME

`spLUfactor, spLUsolve, spLUTsolve` – sparse Bunch–Kaufmann–Parlett factorisation

SYNOPSIS

```
#include "matrix.h"
#include "sparse2.h"
sp_mat  *spBKPfactor(A,pivot,blocks,alpha)
sp_mat  *A;
PERM    *pivot, *blocks;
double  alpha

VEC     *spBKPsolve(A,pivot,blocks,b,x)
sp_mat  *A;
PERM    *pivot, *blocks;
VEC     *b, *x;
```

DESCRIPTION

The routine `spBKPfactor()` performs the symmetric indefinite factorisation methods of Bunch, Kaufmann and Parlett as described for `BKPfactor()`. However, this routine uses a Markowitz type strategy to determine what pivoting to do; the `alpha` argument is a lower limit on the relative size of the pivot block. The pivot which satisfies this lower limit and which has the smallest number of entires in the pivot row(s) is used. The value of `alpha` must be greater than zero but less or equal to one. The value of one gives essentially the pivoting as occurs in `BKPfactor()` for the same matrix. This approach follows that of ... in .... .

The actual factored matrix is stored in the upper triangular part of `A`; the strictly lower triangular part of `A` is left unchanged.

The routine `spBKPsolve()` is really just a translation of `BKPsolve()` to the sparse case, using just the upper triangular part of `A`.

EXAMPLE

A simple example of the use of these routines is

```
sp_mat  *A, *BKP;
PERM    *pvt, *blks;
VEC     *b, *x;
  ......
/* set up A matrix */
  ......
pvt  = get_perm(A->m);
blks = get_perm(A->m);
BKP = sp_cp_mat(A);
spBKPfactor(BKP,pvt,blks,0.1);
/* set up b vector */
  ......
x = spBKPsolve(BKP,pvt,blks,b,VNULL);
```

SEE ALSO

`BKPfactor()`, `BKPsolve()`, `spLUfactor()`, `spLUsolve()`.

SOURCE FILE: `spbkp.c`

NAME

`pccg, cgs, lsqr` – Conjugate gradient like methods

SYNOPSIS

```
#include "matrix.h"
#include "sparse2.h"
int     cg_numiters;

VEC     *pccg(A,A_params,M_inv,M_params,b,tol,x)
MTX_FN  A, M_inv;
VEC     *b, *x;
double  tol;
void    *A_params, *M_params;

VEC     *sp_pccg(A,LLT,b,eps,x)
sp_mat  *A, *LLT;
VEC     *b, *x;
double  eps;
{       return pccg(sp_mv_mlt,A,spCHsolve,LLT,b,eps,x);    }

VEC     *cgs(A,A_params,b,r0,tol,x)
MTX_FN  A;
VEC     *x, *b;
VEC     *r0;
double  tol;
void    *A_params;

VEC     *sp_cgs(A,b,r0,tol,x)
sp_mat  *A;
VEC     *b, *r0, *x;
double  tol;
{       return cgs(sp_mv_mlt,A,b,r0,tol,x);    }

VEC     *lsqr(A,AT,A_params,b,tol,x)
MTX_FN  A, AT;
VEC     *x, *b;
double  tol;
void    *A_params;

VEC     *sp_lsqr(A,b,tol,x)
sp_mat  *A;
VEC     *b, *x;
double  tol;
{       return lsqr(sp_mv_mlt,sp_vm_mlt,A,b,tol,x);    }

int     cg_set_maxiter(maxiters)
int     maxiters;
```

DESCRIPTION

In the routines `pccg()`, `cgs()` and `lsqr()`, the type `MTX_FN` is just given by (in ANSI C)

```
typedef VEC  *(*MTX_FN)(void *params, VEC *x, VEC *out);
```

That is, the matrices are defined *implicitly* by functions; functions that take a vector `x` and computes (and returns) `out = A.x`. This is the standard form of *functional representation* used here. The `params` parameter is for user-defined data structures, for additional flexibility.

Each of `pccg()`, `cgs()` and `lsqr()` has an associated `sp_...` counterpart that has a slightly simpler interface and uses `sp_mat` data structures.

Also common to the different iterative routines is the routine `cg_set_maxiter()` which sets the maximum number of iterations to `maxiters` (and returns the old value of maxiters). The default limit to the number of iterations is 10 000. The actual number of iterations used in the last call of `pccg()`, `cgs()`, etc is stored in `cg_num_iters`.

The routine `pccg()` is a general pre-conditioned conjugate gradient routine. It returns its estimate for the solution `x` of `A.x = b` to within the tolerance `tol`. (That is, the 2-norm of the residual is no more than `tol` times the 2-norm of `b`.) The function `M_inv` represents the *inverse* of the preconditioner; that is the computed `out` vector is the solution of `M.out = in`. The pointers `A_params` and `M_params` are for user-defined data structures for describing `A` and `M_inv` respectively.

The matrices represented must be symmetric and positive definite for this routine to work correctly. Symmetry cannot be tested for, although `pccg` will raise a `E_POSDEF` error if it detects a violation of positive definiteness.

The routine `sp_pccg()` provides a simpler interface to `pccg()` that uses sparse matrices directly. The matrix `A` is the sparse matrix for which the solution `x` of `A.x = b` is wanted. The sparse matrix `LLT` contains the Cholesky factor(s) of `M`, the pre-conditioner, as produced by `spCHfactor()` or `spICHfactor()`.

For example, the "Incomplete Cholesky/Conjugate Gradients" method can be implemented simply as

```
/* Set up A matrix */
  ......
M = sp_cp_mat(A);
spICHfactor(M);
sp_pccg(A,M,b,1e-6,x);
  ......
```

for obtaining answers with a residual of $\leq 10^{-6}\|b\|_2$.

The routine `cgs` is an implementation of the "Conjugate Gradient Squared" algorithm of P. Sonneveld, *CGS, A fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. & Stat. Comp., **10**, pp. 36–52, (1989). This can be used for solving square nonsymmetric systems of equations `A.x = b`. On a successful return, the residual has 2-norm no more than `tol`. The vector `r0` should be the same size as `x` and it is suggested that it should be initialised to be random.

The routine `lsqr` is an implementation of the LSQR algorithm of Paige and Saunders *LSQR: an algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Soft., **8**, pp. 43–71, (1982). The matrix `A` may be nonsquare. The routine must also be passed a functional representation `AT` of the transpose of `A`. The computed `x` is (close to) a minimiser of $\|Ax - b\|_2$.

Each of `pccg`, `cgs` and `lsqr` may raise an `E_ITER` error if there are too many iterations are required to achieve the desired accuracy.

EXAMPLE

An example of using pre-conditioned conjugate gradients (`pccg()`) is given above.

Using CGS with ILU (incomplete *LU* factorisation) preconditioning:

```
struct LUobj {
```

```
    sp_mat  *A, *LU;
    PERM    *pivot;
             };

/* ilu_pc -- returns out <- (LU)^{-1}A.x */
static VEC *ilu_pc(luo, x, out)
LUobj  *luo;
VEC    *x, *out;
{
    static VEC *tmp = VNULL;

    tmp = v_resize(tmp,luo->A->m);
    sp_mv_mlt(luo->A,x,tmp);
    return spLUsolve(luo->LU,luo->pivot,tmp,out);
}

/* cgs_ilu -- uses CGS with ILU preconditioning */
VEC    *cgs_ilu(A,b,tol,out)
sp_mat *A;
double tol;
VEC    *b, *out;
{
    LUobj  luo;
    sp_mat *LU;
    PERM   *pivot;
    VEC    *r0;

    tracecatch(
        LU = sp_cp_mat(A);
        spILUfactor(LU,1e-2);
        r0 = get_vec(A->m);
        rand_vec(r0);
        pivot = get_perm(A->m);
        luo.A = A;
        luo.LU = LU;
        luo.pivot = pivot;
        out = cgs(ilu_pc,&luo,b,r0,tol,out);
    , "cgs_ilu");
    freeperm(pivot);
    freevec(r0);
    sp_free_mat(LU);

    return out;
}
```

SEE ALSO

spCHfactor(), spICHfactor()

SOURCE FILE: conjgrad.c

NAME

`lanczos, lanczos2` – Lanczos eigenvalue routines

SYNOPSIS

```
#include "matrix.h"
#include "sparse2.h"
void    lanczos(A_fn,A_params,m,x0,a,b,beta2,Q)
MTX_FN  A_fn;
void    *A_params;
int     m;
VEC     *x0, *a, *b;
double  *beta2;
MAT     *Q;

void    sp_lanczos(A,m,x0,a,b,beta2,Q)
sp_mat  *A;
int     m;
VEC     *x0, *a, *b;
double  *beta2;
MAT     *Q;
{       lanczos(sp_mv_mlt,A,m,x0,a,b,beta2,Q);    }

VEC     *lanczos2(A_fn,A_params,m,x0,evals,err_est)
VEC     *(*A_fn)();
void    *A_params;
int     m;
VEC     *x0;
VEC     *evals;
VEC     *err_est;

VEC     *sp_lanczos2(A,m,x0,evals,err_est)
sp_mat  *A;
int     m;
VEC     *x0;
VEC     *evals;
VEC     *err_est;
{       return lanczos2(sp_mv_mlt,A,m,x0,evals,err_est);    }
```

DESCRIPTION

The Lanczos algorithm is a method for finding eigenvalues of large symmetric matrices.

The routine `lanczos()` is a basic "raw" Lanczos routine that sets vectors `a` and `b` to be the entries of a tridiagonal matrix

$$T = \begin{bmatrix} a_0 & b_0 & & \\ b_0 & a_1 & b_1 & \\ & b_1 & a_2 & \ddots \\ & & \ddots & \ddots \end{bmatrix}$$

which is $Q^T AQ$, Q a matrix of (near-)orthonormal vectors generated by the Lanczos algorithm. This Q matrix will be accumulated if `Q` is not NULL; otherwise `Q` is ignored. The `m` parameter is the limit on the

number of iterations of the basic Lanczos algorithm, although it may terminate sooner if it detects an exact zero in one of the b's. This early termination is considered very unlikely. Also set by `lanczos()` is the `beta2` parameter; this is the value of b_m, were the algorithm to continue. This parameter is important for the error estimates developed by `lanczos2()`.

Note that A is represented by a function as for the conjugate gradient routines. The routine `sp_lanczos()` provides an alternative that directly uses a sparse matrix data structure.

A more complete code for finding the eigenvalues is:

```
sp_mat   *A;
VEC      *a, *b, *x0;
double   dummy;
/* Set up A matrix */
  ......
x0 = get_vec(A->m);
rand_vec(x0);
a = get_vec(A->m);
b = get_vec(A->m-1);
sp_lanczos(A,num_iter,x0,a,b,&dummy,MNULL)
trieig(a,b,MNULL);
/* Eigenvalues now stored in a */
  ......
```

Some possible problems with a standard Lanczos method should be noted. The basic idea of the Lanczos method is once the vectors q_1, q_2, ... , q_j have been computed, q_{j+1} is computed by first computing Aq_j and then orthogonalising this vector against q_1, q_2, ... , q_{j-1} and q_j. Since A is symmetric, it can be shown that Aq_j is orthogonal (in exact arithmetic) to all but q_{j-1} and q_j. Orthogonalising Aq_j against these two vectors and then normalising gives q_{j+1}.

However, because of inexact arithmetic the q's are not in general orthogonal. Things are worse than this. Because of the three term recurrence, there is no guarantee that q_i and q_j are even *nearly* orthogonal if if i and j are far apart. This shows up in the results of the Lanczos algorithm most obviously as eigenvalues appearing with spurious multiplicities. There are also occasional spurious interior eigenvalues computed by the Lanczos algorithm due to this loss of orthogonality.

Loss of orthogonality can be avoided — by storing all the generated q's and orthogonalising Aq_j against *all* previous q's. This is done in the Arnoldi algorithm.

If you only want extreme eigenvalues, then there is no need for this reorthogonalisation. A more complete discussion of reorthogonalisation and alternative strategies, see Golub and Van Loan's *Matrix Computations*, 2nd Ed'n, pp. 484–489.

If you wish to compute just eigenvalues without reorthogonalisation there is the routine `lanczos2()`. It uses the methods of Cullum and Willoughby, as given in *Sparse Matrix Proc.*, pp. 220–255 (1978), Ed. I.S. Duff and G.W. Stewart, SIAM Publications. It returns a sorted vector of the eigenvalues `evals` together with the error estimates `err_est`. Denote the i'th eigenvalue by λ_i and the i'th error estimate by e_i. Then there is an eigenvalue of A in the range $[\lambda_i - e_i, \lambda_i + e_i]$. If one of these intervals completely contains another, then the latter eigenvalue may be ignored.

For computing eigenvectors, the following code can be used:

```
/* Setup A matrix */
  ......
Q_lan = get_mat(x0->dim,num_steps);
lanczos(mlt_fn,num_steps,x0,a,b,&tmp,Q_lan);
Q_eig = get_mat(num_steps,num_steps);
```

```
id_mat(Q_eig);
/* continued over... */
cp_vec(a,e_vals);
trieig(e_vals,b,Q_eig);
/* select which eigenvalue of T to use */
i = ..... /* by looking at e_vals array */
q = get_col(Q_eig,i,VNULL);
e_vec = mv_mlt(Q_lan,q,VNULL);
```

BUGS

As noted above, `lanczos()` does not return eigenvalues, only the `a` and `b` vectors.

No re-orthogonalisation is done by either `lanczos()` or `lanczos2()`.

SEE ALSO

`trieig()`, `symmeig()`

SOURCE FILE: `lanczos.c`

NAME

`arnoldi, sp_arnoldi` – Arnoldi routines

SYNOPSIS

```
#include "matrix.h"
#include "sparse.h"
#include "sparse2.h"
MAT     *arnoldi(A,A_param,x0,k,h_rem,Q,H)
VEC     *(*A)();     /* functional representation of A */
void    *A_param;
VEC     *x0;
int     k;
double  *h_rem;
MAT     *Q, *H;

MAT     *sp_arnoldi(A,x0,k,h_rem,Q,H)
sp_mat  *A;
VEC     *x0;
int     k;
double  *h_rem;
MAT     *Q, *H;
```

DESCRIPTION

Both of these routines compute an $k \times k$ matrix $H = Q^T AQ$ whose eigenvalues should approximate those of A; in exact arithmetic the columns of Q would be orthogonal. This matrix H is then returned. The matrix H represents the action of A on the Krylov subspace spanned by $\{x_0, Ax_0, \ldots, A^{k-1}x_0\}$, and the columns of the Q matrix form a basis for this subspace. Details can be found in, for example, *Matrix Computations*, §9.3, pp. 501–502, 2nd edition, (1989).

The eigenvalues of A (represented by a `sp_mat` data structure can be approximately computed by

```
H = get_mat(k,k);
S = get_mat(k,k);
Q = get_mat(A->m,k);
Q2 = get_mat(k,k);
evals_re = get_vec(k);
evals_im = get_vec(k);
  ......
sp_arnoldi(A,x0,k,&h_val,Q,H);
S = cp_mat(H,S);
schur(S,Q2);
schur_evals(S,evals_re,evals_im);
```

To go on to compute approximate eigenvectors:

```
X2_re = get_mat(k,k)
X2_im = get_mat(k,k);
schur_vecs(S,Q2,X2_re,X2_im);
X_re = mv_mlt(Q,X2_re,MNULL);
X_im = mv_mlt(Q,X2_im,MNULL);
```

Note that both the H and Q matrices must be created before calling `arnoldi()` or `sp_arnoldi()`. The `h_rem` parameter is the value $h_{k+1,k}$ would have if the H matrix was $(k+1)\times(k+1)$. If a complete invariant subspace had been found, then (in exact arithmetic) this quantity would be zero.

SEE ALSO

`schur()`, `schur_evals()` and `schur_vecs()`

BUGS

Neither routine uses re-orthogonalisation techniques.

SOURCE FILE: `arnoldi.c`

Contents