Chapter 6

Sparse Matrix & Iterative Operations

The following routines are described in the following pages:

To use these routines use the include statement

```
#include "sparse.h"
```

for the basic sparse routines (nnote that this includes `matrix.h`); use

```
#include "sparse2.h"
```

for the sparse factorisation routines (this includes `sparse.h`); use

```
#include "iter.h"
```

for using the iterative routines (this includes `sparse.h`). Note that including `sparse.h` means that `matrix.h` is automatically included.

NAME

sp_get, sp_free, SP_FREE, sp_resize, sp_compact, sp_get_list, sp_free_list, sp_resize_list – allocate, free and resize sparse matrices

SYNOPSIS

```
#include "sparse.h"
SPMAT *sp_get(int m, int n, int maxlen)
void   sp_free(SPMAT *A)
void   SP_FREE(SPMAT *A)
SPMAT *sp_resize(SPMAT *A, int m, int n)
SPMAT *sp_compact(SPMAT *A, double tol)
int    sp_get_vars(int m, int n, int maxlen,
                   SPMAT **A1, SPMAT **A2, ..., NULL)
int    sp_free_vars(SPMAT **A1, SPMAT **A2, ..., NULL)
int    sp_resize_vars(int m, int n,
                   SPMAT **A1, SPMAT **A2, ..., NULL)
```

DESCRIPTION

The routine `sp_get()` allocates and initialises a `SPMAT` data structure. It is initialised so that the `SPMAT` 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()` deallocates all memory associated with the sparse matrix structure `A`. The macro `SP_FREE()` calls `sp_free()` to deallocate `A`, but also sets `A` to NULL, which makes this a safer way of freeing a sparse matrix.

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.

The routines `sp_get_vars()`, `sp_free_vars()` and `sp_resize_vars()` respectively allocate, free and resize NULL-terminated lists of sparse matrices. These operate in the same way as do the other `.._get_list()`, `.._free_list()` and `.._resize_list()` routines; note that `sp_free_vars()` sets `A1`, `A2`, etc. to NULL pointers.

EXAMPLE

```
SPMAT *A;
int    i, j, m, n;
  ......
/* get sparse matrix, with room for 5 entires per row */
A = sp_get(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(A)
```

SOURCE FILE: `sparse.c`

NAME

sp_copy, sp_copy2 – Spare matrix copy routines

SYNOPSIS

```
#include "sparse.h"
SPMAT   *sp_copy (SPMAT *A)
SPMAT   *sp_copy2(SPMAT *A, SPMAT *OUT)
```

DESCRIPTION

The routine sp_copy() 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_copy2() 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_copy2() 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_copy2() into the symbolically 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

```
SPMAT  *A, *B;
   ......
A = sp_get(100,100,4);
for ( i = 0; i < A->m; i++ )
    sp_set_val(A,i,i+1,...);
   ......
/* copy A matrix */
B = sp_copy(A);
   ......
for ( i = 0; i < B->m; i++ )
    sp_set_val(B,i,i+2,...);
sp_copy2(A,B);
/* now B and A represent same matrix,
   but B has allocated (i,i+2) entries */
```

SEE ALSO

sp_get() and sp_resize()

SOURCE FILE: `sparse.c`

NAME

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

SYNOPSIS

```
#include "sparse.h"
double   sp_get_val(SPMAT *A, int i, int j)
double   sp_set_val(SPMAT *A, int i, int 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 `sprow_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

```
SPMAT *A;
int    i, j;
double val;
  ......
A = sp_get(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        "sparse.h"
VEC      *sp_mv_mlt(SPMAT *A, VEC *x, VEC *out)
VEC      *sp_vm_mlt(SPMAT *A, VEC *x, VEC *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 and "saxpy" operations efficiently.

EXAMPLE

```
SPMAT *A;
VEC    *x, *y;
  ......
A = sp_get(100,100,4);
x = v_get(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 "sparse.h"
SPMAT  *sp_col_access (SPMAT *A)
SPMAT  *sp_diag_access(SPMAT *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 §2.6.

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

EXAMPLE

Using the column access fields to chase the entries in

```
SPMAT *A;
int    i, j, idx;
SPROW *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:

```
SPMAT *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. is it guaranteed.

SOURCE FILE: `sparse.c`

NAME

sp_zero, sp_add, sp_sub, sp_smlt, sp_mltadd – General sparse matrix operations

SYNOPSIS

```
#include "sparse.h"
SPMAT *sp_zero(SPMAT *A)
SPMAT *sp_add (SPMAT *A, SPMAT *B, SPMAT *out)
SPMAT *sp_sub (SPMAT *A, SPMAT *B, SPMAT *out)
SPMAT *sp_smlt(SPMAT *A, double alpha, SPMAT *out)
SPMAT *sp_mltadd(SPMAT *A, SPMAT *B, double alpha,
                  SPMAT *out)
```

DESCRIPTION

The routine `sp_zero()` zeros the allocated entries of `A`. Does not change the "allocation" status of entries of `A`.

The routine `sp_add()` adds the sparse matrices `A` and `B`, and puts the result in `out`. This routine may not be used *in situ* with either `A == out` or `B == out`.

The routine `sp_sub()` subtracts `B` from `A` and puts the result in `out`. This routine may not be used *in situ* with either `A == out` or `B == out`.

The routine `sp_smlt()` computes the scalar product of `alpha` and `A` and puts the result in `out`.

The routine `sp_mltadd()` computes $A + \alpha B$ and puts the result in `out`. This routine may not be used *in situ* with either `A == out` or `B == out`.

EXAMPLE

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

```
SPMAT *A;
  ......
sp_zero(A);       /* zeros entries */
sp_compact(A,0.0);   /* removes zero entries */
```

SOURCE FILE: `sparse.c`

NAME

sp_foutput, sp_output – Sparse matrix output

SYNOPSIS

```
#include <stdio.h>
#include "sparse.h"
void   sp_foutput(FILE *fp, SPMAT *A)
void   sp_output(SPMAT *A)
```

DESCRIPTION

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

The routine `sp_output()` is just a macro

```
#define sp_output(A)    sp_foutput(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

```
SPMAT *A;
int    i, j;
FILE   *fp;
  ......
sp_set_val(A,i,j,3.1415926);
  ......
sp_output(A);       /* prints to stdout */
```

```
if ( (fp=fopen("output.dat","w")) == NULL )
    error(E_EOF,"func_name");
sp_foutput(fp,A); /* prints to output.dat */
```

SEE ALSO

`sp_finput()`, `sp_input()`

SOURCE FILE: `sparseio.c`

NAME

sp_finput, sp_input – Input sparse matrix

SYNOPSIS

```
#include <stdio.h>
#include "sparse.h"
SPMAT   *sp_finput(FILE *fp)
SPMAT   *sp_input()
```

DESCRIPTION

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

```
#define sp_input()      sp_finput(stdin)
```

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

EXAMPLE

```
SPMAT *A;
FILE   *fp;
  ......
A = sp_input();     /* read matrix from stdin */
if ( (fp=fopen("input.dat","r")) == NULL )
    error(E_INPUT,"func_name");
A = sp_finput(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 per row.

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

SOURCE FILE: `sparseio.c`

NAME

`sprow_add, sprow_sub, sprow_smlt, sprow_foutput, sprow_get_idx, sprow_get, sprow_xpd, sprow_merge, sprow_mltadd, sprow_set_val` – Sparse row support routines

SYNOPSIS

```
#include "sparse.h"
int      sprow_get_idx(SPROW *r, int col)
SPROW   *sprow_get(int maxlen)
SPROW   *sprow_xpd(SPROW *r, int newlen, int type)
SPROW   *sprow_resize(SPROW *r, int newlen, int type)
SPROW   *sprow_merge(SPROW *r1, SPROW *r2,
                     SPROW *r_out, int type)
SPROW   *sprow_add(SPROW *r1, SPROW *r2, int j0,
                   SPROW *r_out, int type)
SPROW   *sprow_sub(SPROW *r1, SPROW *r2, int j0,
                   SPROW *r_out, int type)
SPROW   *sprow_smlt(SPROW *r, double alpha, int j0,
                    SPROW *r_out, int type)
SPROW   *sprow_mltadd(SPROW *r1, SPROW *r2, double alpha,
                      int j0, SPROW *r_out, int type)
double   sprow_set_val(SPROW *r, int j, double val)
void     sprow_foutput(FILE *fp, SPROW *r)
void     sprow_dump(FILE *fp, SPROW *r)
```

DESCRIPTION

The routine `sprow_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 = sprow_get_idx(r,col)`. If there is no element in row `r` whose column is `col`, then `idx = sprow_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 `sprow_get()` allocates and initialises a sparse row data structure (type `SPROW`) with memory for `maxlen` entries.

The routine `sprow_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. For this routine and the some of the following `sprow_..()` routines the `type` parameter is `TYPE_SPROW` for a stand-alone sparse row, and `TYPE_SPMAT` for a sparse row in a sparse matrix (`SPMAT`) data structure.

The routine `sprow_resize()` resizes the sparse row `r` to have length `newlen`; if `r` is NULL, then a sparse row is created and returned.

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

The routine sprow_add() adds r1 to r2 to compute r_out by a "merging" process. The applies only to columns with column numbers greater than or equal to j0.

The routine sprow_sub() subtracts r2 from r1 to compute r_out = r1 - r2 by a "merging" process. The applies only to columns with column numbers greater than or equal to j0.

The routine sprow_smlt() computes the scalar product r_out = alpha*r.

The routine sprow_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 sprow_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 sprow_foutput() 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:

```
SPMAT *A;
SPROW *r, r1, r2;
row_elt *e;
int    i, j, idx, idx1;
  ......
/* compute A[i][j] */
r = &(A->row[i]);
idx = sprow_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 = sprow_get(10);
for ( idx = 0; idx < r->len; idx++ )
{
    e = &(r->elt[idx]);
    old_col = e->col;
    new_col = ......;
    sprow_set_val(r1,new_col,e->val);
    /* r1 will be expanded if necessary */
}
```

Expanding a temporary row:

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

Printing out a row as a separate structure for debugging:

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

SOURCE FILE: `sparse.c`

NAME

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

SYNOPSIS

```
#include "sparse2.h"
SPMAT   *spCHfactor(SPMAT *A)
VEC     *spCHsolve(SPMAT *LLT, VEC *b, VEC *out)
SPMAT   *spICHfactor(SPMAT *A)
SPMAT   *spCHsymb(SPMAT *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(pattern);
     sp_copy2(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 `iter_cg()` and `iter_spcg()`.

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_copy2()`, `sp_zero()`, `iter_cg()`, `iter_spcg()`

SOURCE FILE: `spCHfactor.c`

NAME

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

SYNOPSIS

```
#include "sparse2.h"
SPMAT *spLUfactor (SPMAT *A, PERM *pivot, double alpha)
SPMAT *spILUfactor(SPMAT *A, double alpha)
VEC   *spLUsolve  (SPMAT *LU, PERM *pivot, VEC *b, VEC *x)
VEC   *spLUTsolve(SPMAT *LU, PERM *pivot, VEC *b, VEC *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 $A^Tx = b$. Both of these use the the matrix as factored by `spLUfactor()`. They can also be used *in situ* with `x == b`.

EXAMPLE

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

```
/* Set up A and b */
  ......
pivot = px_get(A->m);
x     = v_get(A->n);
y     = v_get(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 **`iter_cg()`**, **`iter_cgs()`** and **`iter_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

`spBKPfactor, spBKPsolve` – sparse Bunch–Kaufmann–Parlett factorisation

SYNOPSIS

```
#include "sparse2.h"
SPMAT   *spBKPfactor(SPMAT *A, PERM *pivot, PERM *blocks,
                     double alpha)
VEC     *spBKPsolve (SPMAT *A, PERM *pivot, PERM *blocks,
                     VEC *b, VEC *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.

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

```
SPMAT   *A, *BKP;
PERM     *pvt, *blks;
VEC      *b, *x;
  ......
/* set up A matrix */
  ......
pvt  = px_get(A->m);
blks = px_get(A->m);
BKP = sp_copy(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

`iter_get`, `iter_free`, `iter_resize`, `iter_copy`, `iter_copy2`, `iter_Ax`, `iter_ATx`, `iter_Bx`, `iter_dump` – Iteration data structure initialisation

SYNOPSIS

```
#include "iter.h"
ITER *iter_get(int m, int n)
int   iter_free(ITER *ip)
ITER *iter_resize(ITER *ip, int new_m, int new_n)
ITER *iter_copy (ITER *in, ITER *out)
ITER *iter_copy2(ITER *in, ITER *out)
int   iter_Ax (ITER *ip, Fun_Ax Ax,  void *Ax_par)
int   iter_ATx(ITER *ip, Fun_Ax ATx, void *ATx_par)
int   iter_Bx (ITER *ip, Fun_Ax Bx,  void *Bx_par)
void  iter_dump(FILE *fp, ITER *ip)
```

DESCRIPTION

These routines initialise the `ITER` data structure for use in applying iterative methods for large sparse or structured matrices. The routine `iter_get(m,n)` allocates and initialises an `ITER` data structure for an $m \times n$ linear system $Ax = b$. The `ITER` data structure can be deallocated by calling `iter_free(ip)`. The routine `iter_resize()` resizes the vectors in the `ITER` data structure appropriately for a `new_m` $\times$ `new_n` matrix.

The routine `iter_copy()` copies all of the values stored in `in` to `out`, and also copies the vectors `in->x` and `in->b` to `out->x` and `out->b` respectively. The routine `iter_copy2()` also copies all of the values stored in `in` to `out`, but the vectors `out->x` and `out->b` are unchanged.

For the iterative routines matrices are represented by functions. In particular, the matrix A is represented by a function `Ax` which computes $y = Ax$ given x by means of

```
VEC  *x, *y;
void *Ax_par;
  ......
y = (*Ax)(Ax_par, x, y);
```

Indeed the type `Fun_Ax` is defined by

```
typedef VEC *(*Fun_Ax)(void *Ax_par, VEC *x, VEC *out);
```

That is, an object of type `Fun_Ax` is a function (or equivalently a pointer to a function) which takes a (user-definable) parameter `Ax_par`, the vector x and the destination vector, and returns a vector. Strictly speaking the **Ax_par** parameter is not necessary as one can set a global variable with **Ax_par** and use it directly in the function **Ax**. However, this requires communication through global variables (which is not a good software engineering practice), and also requires the user to set and unset global variables whenever the matrix changes. By using an extra (user-definable) parameter, general routines can be written which can deal with a general class of problems.

While most of the values in the **ITER** structure must be set directly if you wish to override the default values, the **iter_Ax()**, **iter_ATx()** and **iter_Bx()** macros are provided to simplify setting the fields which define the matrix A, its transpose A^T, and the preconditioner B. For a list of the values stored in the **ITER** structure, and their default values, see §2.8.

The contents of an **ITER** data structure can be dumped to a file or stream **fp** using **iter_dump(fp, ip)**. This representation is just for debugging purposes and cannot be read back in.

As an example, here is how sparse matrix data structures can be represented in an **ITER** data structure:

```
SPMAT *A;
ITER  *ip;
  ......
ip = iter_get(A->m,A->n);
iter_Ax (ip, sp_mv_mlt, A);
iter_ATx(ip, sp_vm_mlt, A);
/* some extra parameters */
ip->limit = 10000;  /* limit to max number of steps */
ip->eps   = 1e-9;   /* error tolerance */
```

The routine is **sp_mv_mlt(A,x,out)**, which is the sparse matrix–vector product routine; the sparse matrix data structure **A** is the first parameter, and is the "user-definable" pointer. If the matrix A^T is to be used in an iterative routine, then the sparse matrix data structure does not have to be touched; instead the **sp_mv_mlt()** routine just needs to be replaced by **sp_vm_mlt()**, which computes $y = A^T x$.

SEE ALSO

iter_cg, iter_cgs and the other iterative methods

SOURCE FILE: **iter0.c**

NAME

`iter_cg, iter_cgne, iter_cgs, iter_mgcr, iter_lsqr, iter_gmres, iter_spcg, iter_spcgne, iter_spcgs, iter_spmgcr, iter_splsqr` – Iterative methods for linear equations

SYNOPSIS

```
#include "iter.h"
VEC *iter_cg    (ITER *ip)
VEC *iter_cgne  (ITER *ip)
VEC *iter_cgs   (ITER *ip, VEC *r0)
VEC *iter_lsqr  (ITER *ip)
VEC *iter_gmres(ITER *ip)
VEC *iter_mgcr  (ITER *ip)
VEC *iter_spcg  (SPMAT *A, SPMAT *LLT, VEC *b, Real tol,
                VEC *x, int limit, int *steps)
VEC *iter_spcgne(SPMAT *A, SPMAT *B, VEC *b, Real tol,
                 VEC *x, int limit, int *steps)
VEC *iter_spcgs(SPMAT *A, SPMAT *B, VEC *b, VEC *r0,
                Real tol, VEC *x, int limit, int *steps)
VEC *iter_splsqr(SPMAT *A, VEC *b, Real tol, VEC *x,
                 int limit, int *steps)
VEC *iter_spgmres(SPMAT *A, SPMAT *B, VEC *b, Real tol,
                  VEC *x, int k, int limit, int *steps)
VEC *iter_spmgcr(SPMAT *A, SPMAT *B, VEC *b, Real tol,
                 VEC *x, int k, int limit, int *steps)
```

DESCRIPTION

These routines provide iterative methods for solving systems of linear equations, both symmetric and non-symmetric. The `ITER` data structure `ip` contains the information about the matrix along with preconditioners, error tolerances, limits on numbers of steps etc. The routines set some values in the `ip` data structure such as the solution and the number of steps of the iterative method actually taken. The solution vector `ip->x` is returned.

Of these routines, `iter_cg()` is the method of choice for positive definite symmetric matrices; `iter_lsqr()` is probably the most reliable; `iter_cgs()` probably the least stable, but relatively fast when it works; `iter_mgcr()` and `iter_gmres()`| probably provides the best compromises between speed and reliability for most nonsymmetric systems. The routine `iter_cg()` and `iter_lsqr()` require the least amount of memory.

The routine `iter_cg()` implements the conjugate gradient method. This is for symmetric positive definite matrices only, with symmetric positive definite preconditioners. This is a well-known method for solving such systems since the 1970's. The routine `iter_cg()` implements the standard (pre-conditioned) conjugate gradient method as presented in Golub and Van Loan's *Matrix Computations*, §10.3, 2nd Edition (1989).

The routine `iter_cgne()` implements the conjugate gradient method for the normal equations $A^T Ax = A^T b$. This requires the **`ATx`** and **`ATx_par`** fields of `ip` to be set. The preconditioner B (represented by **`Bx`** and **`Bx_par`**) must be symmetric and positive definite, and is interpreted as the preconditioner for $(A + A^T)/2$. In fact, this routine applies the conjugate gradient algorithm to $A^T BA$ using a modified inner product. One way to obtain a suitable preconditioner is to use imcplete Cholesky factorisation to get approximate factors of $(A + A^T)/2$. Note that an alternative to this routine for least squares and related problems is `iter_lsqr()`.

The routine `iter_cgs()` implements Sonneveld's CGS (Conjugate Gradients Squared) method as described in *CGS: A fast Lanczos-type solver for nonsymmetric lilnear systems*, SIAM J. Scientific and Statistical Comp., **10**, pp. 36–52 (1989). This is a somewhat unstable but fast algorithm for non-symmetric systems. The vector `r0` passed to `iter_cgs()` is an auxiliary vector. A simple strategy is to set `r0` to be a random vector on entry. It does not contain any useful information on exit. The solution vector is returned.

The routines `iter_lsqr()` implements the LSQR method of Paige and Saunders as described in *LSQR: an algorithm for sparse linear equations and sparse least squares*, ACM Transactions on Mathematical Software, **8**, pp. 43–71 (1982). This computes solutions to the least squares problem: achieving $\min_x \|Ax - b\|_2$. For this routine, the functional parameter **`ATx`** for computing $y = A^T x$ must also be set in the `ip` data structure as well as the **`Ax`** parameter. The matrix A represented may be non-square.

The routine `iter_gmres()` implements the Generalised Minimal RESidual method (GMRES) of Saad and Schultz as presented in *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Scientific and Statistical Comp., 7, pp. 856–869 (1986). A single step of GMRES involves building up an approximation to `A` on a Krylov subspace span$\{r,\ Ar,\ A^2 r,\ \ldots,\ A^{k-1} r\}$ where k is the dimension of the Krylov subspace and r is the current residual. The entry `ip->k` of `ip` contains the value of k used by `iter_gmres()`.

The routine `iter_mgcr()` implements a fast Modified Generalized Conjugate Residual algorithm of Leyk as presented in *Modified generalized conjugate residuals method for nonsymmetric systems of linear equations*, Technical Report CMA-MR33-93 of the School of Mathematical Sciences, Australian National University (1993).

There are also versions `iter_sp...()` which work with the sparse matrix data structures. Here `A` is the sparse matrix and `b` is the right-hand side vector for the linear system $Ax = b$; `tol` is the residual tolerance; `limit` is the maximum number of steps of the iterative method; `steps` is set to the actual number of steps of the iterative method actually used. If the last argument (for `steps`) is NULL, then it is not used.

In `iter_spcg()`, **`LLT`** is the sparse matrix structure containing an approximate Cholesky factorisation of A; If **`LLT`** is NULL then no preconditioning is used. In iter_spcgs(), r0 is the auxiliary vector. In iter_spcgne(), iter_spcgs(), iter_spgmres() and iter_spmgcr(), B is the (explicit) preconditioner. If B is NULL then no preconditioning is used. In iter_splsqr() there is no preconditioning. In iter_spgmres() and iter_spmgcr(), k is the dimension of the Krylov subspace used.

EXAMPLE

To implement Incomplete Cholesky/Conjugate Gradients (ICCG) for a sparse symmetric positive definite matrix A:

```
  ......
LLT = sp_copy(A);
spICHfactor(LLT);
x = iter_spcg(A,LLT,b,1e-6,VNULL,1000,&steps)
```

An example of using incomplete LU preconditioners for a nonsymmetric system is:

```
VEC *myILUsolve(SPMAT *LU, VEC *x, VEC *y)
{
    return spLUsolve(LU,PXNULL,x,y);
}

main()
{
ITER *ip;
  ......
LU = sp_copy(A);
spILUfactor(LU,alpha);
ip = iter_get(A->m,A->n);
iter_Ax(ip,sp_mv_mlt, A);
iter_Bx(ip,myILUsolve,LU);
r0 = v_rand(v_get(A->m));
iter_cgs(ip,r0);     /* using CGS... */
ip->k = 20;          /* for GMRES */
iter_gmres(ip);      /* using GMRES... */
iter_mgcr(ip);       /* using MGCR... */
iter_ATx(ip, sp_vm_mlt, A);
iter_lsqr(ip);       /* using LSQR... */
/* extract solution */
printf("Solution is:\n");  v_output(ip->x);
printf("Used %d steps\n", ip->steps);
}
```

SEE ALSO

`iter_get()` and related routines; `spICHfactor(), spILUfactor()`

SOURCE FILE: `itersym.c, iternsym.c`

NAME

`iter_lanczos, iter_lanczos2, iter_arnoldi, iter_arnoldi_iref, iter_splanczos, iter_splanczos2, iter_sparnoldi, iter_sparnoldi_iref` – Krylov subspace algorithms

SYNOPSIS

```
#include "iter.h"
void  iter_lanczos (ITER *ip, VEC *a, VEC *b, Real *beta2,
                   MAT *Q)
VEC  *iter_lanczos2(ITER *ip, VEC *evals, VEC *err_est)
MAT  *iter_arnoldi (ITER *ip, Real *h_rem, MAT *Q, MAT *H)
MAT  *iter_arnoldi_iref(ITER *ip, Real *h_rem,
                       MAT *Q, MAT *H)
void  iter_splanczos(SPMAT *A, int k, VEC *x0,
                    VEC *a, VEC *b, Real *beta2, MAT *Q)
VEC  *iter_splanczos2(SPMAT *A, int k, VEC *x0,
                     VEC *evals, VEC *err_est)
MAT  *iter_sparnoldi(SPMAT *A, VEC *x0, int k,
                    Real *h_rem, MAT *Q, MAT *H)
MAT  *iter_sparnoldi_iref(SPMAT *A, VEC *x0, int k,
                         Real *h_rem, MAT *Q, MAT *H)
```

DESCRIPTION

These routines implement the Lanczos and Arnoldi methods of extracting information about large matrices by computing Krylov subspaces, and the effect of the matrices on these subspaces. One of the main uses for these algorithms is to compute approximate eigenvalues. Of these, the Lanczos method is for symmetric matrices, and the Arnoldi method is for general matrices. For a matrix A and a start vector r, the Krylov subspace of dimension k generated is

$$K(A, r, k) = \text{span}\{\, r\, Ar,\, \ldots,\, A^{k-1}r \,\}.$$

Both the Lanczos and Arnoldi methods construct orthonormal bases (at least in exact arithmetic) of the Krylov subspace $K(A, r, k)$. The orthonormal bases form the *rows* of `Q`. The approximation to A on the Krylov subspace generated is taken to be QAQ^T. Note that the results of the Lanczos and Arnoldi methods are the same (in exact arithmetic) for symmetric matrices.

If A is symmetric then $T = QAQ^T$ is tridiagonal and is represented by the vectors `a` and `b` computed by the Lanczos algorithm:

$$T = \begin{bmatrix} a_0 & b_0 & & & \\ b_0 & a_1 & b1 & & \\ & b1 & a2 & \ddots & \\ & & \ddots & \ddots & b_{k-2} \\ & & & b_{k-2} & a_{k-1} \end{bmatrix}.$$

If the purpose is to compute approximate eigenvalues, but not eigenvectors, then `Q` can be NULL on entry to **`iter_lanczos()`**. Then `Q` is not accumulated and only `a` and `b` are computed. The eigenvalues of A can be approximated by eigenvalues of T.

For general matrices $H = QAQ^T$ is *upper Hessenberg* is computed by the Arnoldi algorithm. The matrix H is returned by **`iter_arnoldi()`**. That is, $h_{ij} = 0$ whenever $i > j + 1$; or alternatively, all entries below the first sub-diagonal of H are zero. The eigenvalues of A can be approximated by the eigenvalues of H. Unlike **`iter_lanczos()`**, the routine **`iter_arnoldi()`** requires `Q` to be non-NULL and of the correct size: $k \times n$ where A is $n \times n$.

In **`iter_lanczos()`**, **`beta2`** is set to the value b_{k-1} which is the value of the *next* off-diagonal entry should the process go one step further. If $Q^T = [q_0, q_1, \ldots, q_{k-1}]$ and q_k would be the next basis vector computed, then

$$AQ^T = Q^T T + b_{k-1} q_k e_k^T.$$

Thus, b_{k-1} can be used to estimate errors in the eigenvalues and eigenvectors estimated by the Lanczos method.

Similarly, in **`iter_arnoldi()`**, **`h_rem`** is the value of the *next* sub-diagonal entry that would occur if k was increased by one. Again, the formula

$$AQ^T = Q^T H + b_{k-1} q_k e_k^T$$

can be used to estimate errors in the eigenvalues and eigenvectors estimated by the Lanczos method.

Note that for both the Lanczos and Arnoldi methods, the eigenvalues (and eigenvectors) that are first estimated with greatest accuracy are the most extreme one. For the symmetric case, since the eigenvalues are real, the most positive and the most negative eigenvalues can be quickly computed to reasonable accuracy. Interior eigenvalues take considerably longer to obtain reasonable accuracy if at all. To compute approximate eigenvectors: Let v be an eigenvector for T (in the Lanczos case) or H (in the Arnoldi case). Then an approximate eigenvector for A is given by $Q^T v$. Note, however, then eigenvalues converge faster than eigenvectors.

The Lanczos method is more efficient than the Arnoldi method. However, because of this it suffers from some numerical instabilities. The reason for both comes down to the fact that the Q matrix does not need to be stored for the Lanczos method. As a result, the computed $\widehat{Q}$ need not contain even nearly orthonormal rows; nearby rows are nearly orthonormal, but widely separated rows of Q are not necessarily nearly orthonormal. For the Arnoldi method, however, since Q is stored in its entirety, orthogonality of each can be (and is) enforced against all other rows. In the context of the Lanczos algorithm, this would be called *complete reorthogonalisation*, but is not usually done because of its expense. The lack of orthonormality of Q's rows results in some surprising behaviour: occasional spurious eigenvalues, and repeated eigenvalues with multiplicities higher than in A.

Spurious eigenvalues can be detected by the Cullum and Willoughby algorithm implemented by `iter_lanczos2()`. This routine is based on the algorithm in *Lanczos and the computation in specified intervals of the spectrum of large, sparse real symmetric matrices*, in "Sparse Matrix Proceedings 1978" pp. 220–255 (1979). This routine produces error estimates for the eigenvalues based on the `a`, `b` and `beta2` values produced from `iter_lanczos()`. The error estimate of the approximate eigenvalue $\widehat{\lambda}_i$ = `eval->ve[i]` is given by η_i = `err_est->ve[i]`. If the error interval $[\lambda_i - \eta_i, \lambda_i + \eta_i]$ contains another interval $[\widehat{\lambda}_j - \eta_j, \widehat{\lambda}_j + \eta_j]$, then the eigenvalue is spurious.

Complete reorthogonalisation avoids both spurious eigenvalues and repeated eigenvalues. This can be achieved by using `iter_arnoldi()` and then extracting just the tridiagonal part of H.

The basic Arnoldi routine `iter_arnoldi()` has a slight numerical instability in that it uses unmodified Gram–Schmidt orthogonalisation.

The routine `iter_arnoldi_iref()` uses a relatively cheap iterative refinement extension which prevents problems with the Gram–Schmidt orthogonalisation.

For more information about the Lanczos and Arnoldi methods see Golub and Van Loan's *Matrix Computations*, chapter 9, 2nd edition (1989).

There are versions `iter_sp...()` which work with matrix data structures.

EXAMPLE

To get a good approximation to the smallest eigenvalue of a positive definite symmetric matrix A:

```
SPMAT *A;
ITER  *ip;
VEC   *a, *b;
Real  dummy;
  ......
ip = iter_get(A->m,A->n);
iter_Ax(ip,sp_mv_mlt,A);
ip->k = krylov_dim;
v_rand(ip->x);
iter_lanczos(ip,a,b,&dummy,MNULL);
trieig(a,b,MNULL);  /* eigenvalues left in a */
printf("Min. e-val = %g\n", v_min(a));
```

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

```
H = m_get(k,k);
S = m_get(k,k);
Q = m_get(A->m,k);
```

```
Q2 = m_get(k,k);
evals_re = v_get(k);
evals_im = v_get(k);
  ......
ip = iter_get(A->m,A->n);
iter_Ax(ip,sp_mv_mlt,A);
ip->k = krylov_dim;
v_rand(ip->x);
iter_arnoldi_iref(ip,&dummy,Q,H);
S = m_copy(H,S);
schur(S,Q2);
schur_evals(S,evals_re,evals_im);
```

To go on to compute approximate eigenvectors:

```
X2_re = m_get(k,k)
X2_im = m_get(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);
```

SEE ALSO

`iter_get, ..., iter_gmres`

SOURCE FILE: `itersym.c iternsym.c`