Chapter 2

Data structures

2.1 General principles

In this chapter an overview of the data structures is given, as well as indicating how memory management is undertaken. For more information about how to use and develop data structures, you should see chapter 8 on designing data structures.

One of the main thrusts of Meschach is to use C's data structuring ability to "package" the objects so that they are self-contained and can be dealt with as single entities. This is combined with C's memory allocation and de-allocation techniques to make basic mathematical objects (vectors, matrices, permutations etc) work more like their mathematical counterparts. So, a vector structure contains not only the array of its components, but also the dimension of the vector, and the amount of allocated memory (which may be larger than the dimension). This vector can be used for ordinary vector operations, computing matrix–vector products, solving systems of linear equations, or just for storing data. If there is a mismatch in, say, the size of the vector and the vectors or matrices that it operates with, then an error is raised to indicate this. The vector can also be created when needed, and destroyed when not. It can be re-sized when desired to be larger or smaller.

The type of floating point number is `Real`, which is one of the floating point types. The default floating point type is `double`.

The integer vector and permutation data structures are very similar to the vector data structure, and contain not only the array of values, but also the current dimension or size of the integer vector or permutation and the amount of allocated memory in this array. Permutations are really restricted integer vectors; they are initialised differently (to the identity permutation, instead of all zeros) and the permutation routines preserve the property of being a permutation.

Matrices are represented by a more complex data structures, and are essentially a two-level data structure. To have variable size 2-dimensional arrays in C, pointer-to-pointer structures are needed, such as

```
Real **Aentries;
  ......
```

```
Aentries[3][4] = 2.0;
```

The matrix data structure therefore has a pointer-to-pointer entry which can be used just as the `Aentries` variable can. The data structure also has entries containing the number of rows and columns of the matrix, and also the allocated number of rows, columns etc.

Sparse matrices are the most complex data structures and are, in fact, a three level system of data structures. They are also the most dynamic, as when operations are performed on sparse matrices, the number of non-zero entries in a row changes. There are also a number of additional components of the data structures that are used to facilitate operations, and are not needed to specify the sparse matrix that is represented.

Iterative routines operate on a data structure that combines a number of items into a single package. These items include the defining data structures for the system to be solved, preconditioners, current (approximate) solution, desired accuracy, limits on the number of iterations, and functions implementing the stopping criterion and for providing information to the user. By packaging the information in this way, and providing suitable defaults on initialisation, it enables the user to use the iterative routines in either a simple way (just use the defaults), or in a very sophisticated way (by specifying limits, preconditioners, stopping criteria etc).

2.2 Vectors

The vector data structure is the `VEC` structure:

```
typedef unsigned int    u_int;
/* vector definition */
typedef struct  {
                u_int   dim, max_dim;
                Real    *ve;
                } VEC;
```

The type `u_int` is a short-hand for `unsigned int`. The field `dim` is the dimension of the vector, while `ve` is a pointer to the actual elements of the vector. The field `max_dim` is the actual length of the `ve` array. Clearly we require `dim` $\leq$ `max_dim`.

The normal method of obtaining a vector of a specified length is to call `v_get()`, which returns a pointer to `VEC`. To illustrate how this scheme operates, the code to obtain a vector of length n is shown below:

```
#include        "matrix.h"
        ...
VEC     *x;
int     n;
        ...
x = v_get(n);
        ...
```

To access the i^{th} element of `x` we have to go through the `ve` field:

```
x_i = x->ve[i];
```

Note that the array index `i` is understood to be "*zero relative*"; that is, the valid values of `i` are $0, 1, 2, \ldots, n-1$.

The call `v_resize(x,newdim)` "resizes" the vector `x` to have dimension `newdim`. In this call, it is first checked if `newdim` $\leq$ `x->max_dim`. If so, then all that happens is that `x->dim` is set to `newdim`. Otherwise, memory is `realloc()`'d for a vector of size `newdim`. Provided the `realloc()` is successful, both `x->dim` and `x->max_dim` are set to `newdim`. Note that under this "high-water mark" system, the physical size of the vector's allocated memory can never decrease. To regain the memory that has been allocated, the vector must be deallocated entirely using `V_FREE()` or `v_free()`. (The former is a safer macro that uses `v_free()`.)

Usually, no objects of type `VEC` are declared within a program, routine or function. Rather, *pointers* to `VEC` structures are declared within a program, routine or function. Pointers are returned by `v_get()`, `v_copy()` and `v_input()` which also take care of any initialisation that is needed. Pointers (as returned by these functions) can also be freed up. You should not declare objects to be of type `VEC` (as opposed to objects of type `VEC *`) unless you know what you are doing. For example,

```
VEC x;
   ......
V_FREE(&x);
```

will result in a compile-time error. Using `v_free()` instead of `V_FREE()` would most likely result in a program crash!

2.2.1 Integer vectors

There are also *integer vectors* which are pointers to type `IVEC`. These are implemented an a way that is essentially equivalent to the `VEC` data structures. There is the allocation and initialisation routine `iv_get()`, resizing routine `iv_resize()`, and `iv_free()` to destroy an integer vector.

The dimension (i.e. number of entries) of an integer vector `iv` is `iv->dim`. The `i`th entry of an integer vector `iv` is `iv->ive[i]`, and indexing is zero relative so `i` must be in the range $0, 1, \ldots,$ `iv->dim-1`.

These are useful for constructing index lists as well as other, general data structures.

2.2.2 Complex vectors

Complex vectors and matrices have been included in Meschach version 1.2. The basic complex data type in Meschach is a standard pair of floating point numbers:

```
typedef struct {   Real re, im;   } complex;
```

There are a number of routines for dealing with complex numbers. The most basic is `z = zmake(real,imag);` which returns a complex number with real part `real` and imaginary part `imag`. There are also routines to add complex numbers `zadd(z1,z2)`, to subtract `zsub(z1,z2)`, multiply `zmlt(z1,z2)`, divide `zdiv(z1,z2)`, negate `zneg(z)`, conjugate `zconj(z)`, and compute square roots, exponentials and logarithms `zsqrt(z)`, `zexp(z)`, `zlog(z)`. There is also the magnitude function which returns a floating point number: `zmag = zabs(z);`.

Complex vectors are vectors of these `complex` data structures, and have the type `ZVEC`. The structure of these vectors is otherwise equivalent to that of ordinary floating point vectors. For example, the `i`'th entry of a complex vector `zv` is `zv->ve[i]`; to extract its real part use `zv->ve[i].re`, and for its imaginary part use `zv->ve[i].im`.

The operations on complex vectors are also very similar to that for ordinary vectors: `zv = zv_get(10);` to get a complex vector of length 10;
`zv3=zv_add(zv1,zv2,ZVNULL);` to add two complex vectors ($z_3 = z_1 + z_2$).

2.3 Matrices

Matrices are very important throughout numerical mathematics, so it is natural that we have a separate data structure for them:

```
typedef unsigned int    u_int;
/* matrix definition */
typedef struct  {
                u_int   m, n;
                u_int   max_m, max_n, max_size;
                Real    **me, *base;
                /* base is base of alloc'd mem */
                } MAT;
```

Here `m` is the number of rows of the matrix, `n` is the number of columns of the matrix (i.e. it is `m` × `n`). The `me` field gives the actual means of accessing the elements of the matrix. For example, to access the (i, j) element of the matrix `A` we use:

```
MAT   *A;
Real   A_ij;
  ....
A_ij = A->me[i][j];
```

The `base` field is the pointer to the beginning of the memory allocated for the entries of the matrix. The `max_size` field is the size of this area in terms of `Real` numbers.

It should be noted that `me` is actually an array with elements of type `Real *`. The actual size of this array is given by the field `max_m`. This is a (usually small) memory overhead which speeds up the accessing of elements: only two additions are needed to locate `me[i][j]`, while a multiply and an addition are needed to locate

`base[m*i+j]`. The rows in a matrix are allocated contiguously, as long as this is reasonable, so that no problems arise from memory overhead or cache misses. Even if a matrix is resized, the rows are copied so that the rows of the resized matrix are contiguous.

As with vectors, only pointers to matrices are used, and this allows memory allocation and deallocation to be done conveniently. Also note that matrices are resized using a "high-water mark" approach so that the total amount of physical memory for row pointers and for entries of a matrix does not decrease unless the matrix is completely deallocated by `M_FREE()` (which is a safe macro) or `m_free()`.

2.3.1 Complex matrices

Complex matrices are also available and have the type `ZMAT`. These have the same structure as the ordinary `MAT` data type except that the entries are not of type `Real`, but of type `complex`. The operations that can be done to complex matrices are similar to those that can be performed on ordinary matrices. For example, here is some code to set an entry and to print out the value:

```
ZMAT *A;
complex z;
  ......
A = zm_get(10,10);
A->me[2][3] = z;
printf("Real part = %g, imaginary part = %g\n",
       A->me[2][3].re, A->me[2][3].im);
ZM_FREE(A);
```

2.3.2 Band matrices

Band matrices are a special class of sparse matrices where the nonzero entries all lie in a narrow band around the diagonal. Unlike general sparse matrices, these matrices can be factorised with well controlled fill-in. They can also be easily represented by listing the nonzero entries by their distance from the diagonal, and whether they lie above or below (or on) the diagonal.

There are two factorisation routines for band matrices: an LDL^T variant of the Cholesky factorisation, and an LU factorisation with partial pivoting. Rather than develop a complete new data structure for these two routines, the `BAND` data structure used is actually just a `MAT` structure together with the lower and upper bandwidths `lb` and `ub` respectively. This is the actual data structure:

```
/* band matrix definition */
typedef struct {
                MAT    *mat;     /* matrix */
                int    lb,ub;    /* lower & upper bandwidth */
                } BAND;
```

The actual entries of A are stored as matrix entries in `mat`, which has the following layout. Let A be the $n \times n$ band matrix that is represented by this data structure. Then n is the number of columns of `mat`. Also, lb is the lower bandwidth of A (this is the number of sub-diagonals in A), and ub is the upper bandwidth of A (this is the number of super-diagonals in A). Note that for a general diagonal matrix, $lb = ub = 0$, while for a tridigonal matrix, $lb = ub = 1$. For $0 \leq i < lb$, row $lb - i$ of `mat` is the ith sub-diagonal of A; row lb of `mat` is the diagonal of A; and for $lb < i \leq lb + ub$, row i of `mat` is the $(i - lb)$th super-diagonal of A. The (i, j) entry of A (provided $-lb \leq j - i \leq ub$) is the $(lb + j - i, j)$ entry of `mat`. This means that there are some wasted entries in `mat`, as is shown by this layout for $lb = 3$, $ub = 2$ and $n = 10$. A '.' denotes an unused entry of `mat`:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ \text{row} \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} a_{30} & a_{41} & a_{52} & a_{63} & a_{74} & a_{85} & a_{96} & . & . & . \\ a_{20} & a_{31} & a_{42} & a_{53} & a_{64} & a_{75} & a_{86} & a_{97} & . & . \\ a_{10} & a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & a_{76} & a_{87} & a_{98} & . \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} & a_{77} & a_{88} & a_{99} \\ . & a_{01} & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} & a_{67} & a_{78} & a_{89} \\ . & . & a_{02} & a_{13} & a_{24} & a_{35} & a_{46} & a_{57} & a_{68} & a_{79} \end{bmatrix} \begin{array}{l} \\ \text{(lower part)} \\ \\ \\ \text{(main diagonal)} \\ \\ \text{(upper part)} \end{array}$$

For creating a band matrix `A`, use `A = bd_get(lb,ub,n)`, for resizing use `bd_resize(A,lb,ub,n)` (where `lb` etc. are the new values), for freeing use `bd_free(A)`, and for transposing use `bd_transp(A,B)`.

2.4 Permutations

Permutations are immensely useful in a number of matrix factorisation techniques, as well as for the representation of sets and so on. It was therefore decided that, as well as being important *mathematical* objects in their own right, they should be implemented as a concrete data structure in their own right. Here is the definition of the data structure used:

```
typedef unsigned int    u_int;
/* permutation definition */
typedef struct  {
                u_int   size, max_size, *pe;
                } PERM;
```

The field `size` is the size of the permutation. The field `pe` is the means by which the elements of the permutation are accessed: to access $\pi(i)$ for a permutation π use

```
PERM  *pi;
  ...
pi_i = pi->pe[i];
```

The actual size of the `pe` array is given by the field `max_size`.

As with vectors and matrices, only pointers to permutation data structures are used. Permutations may be resized and deallocated. A "high-water mark" method is used when resizing permutations, so that the physical memory used for storing entries does not decrease in size.

Whether or not the elements of an array of integers forms a permutation clearly depends on the entries of that array. This, to some extent is up to the programmer. However, there are a number of routines that try to help this aspect: `px_get()` initialises the permutation to be the identity permutation; if the argument to `px_resize()` is a true permutation, the result will be a true permutation, though if a reduction of size is requested, *all the old data will be overwritten.* Also there is `px_transp()` which transposes two entries in a permutation; it is expected that this would be the most common means of modifying a permutation. Finally, the input routines check that what is input is indeed a permutation.

2.5 Basic sparse operations and structures

Sparse matrix data structures are somewhat more complex than dense matrix data structures. The form chosen here is a row oriented sparse matrix data structure. The matrix consists of an array of rows, and each row is an array of row elements. A row element contains a value, a column number and some other numbers to help access elements in the same column. (These latter data items are intended to improve access speed for column oriented operations.)

To use these sparse matrix data structures you need to have the following at the beginning of your program:

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

Sparse matrices are declared as pointers, as is done with other data structures in the system:

```
SPMAT       *A;
```

Initialising a sparse matrix requires calling the `sp_get()` function:

```
A = sp_get(m, n, maxlen);
```

Here m is the number of rows in A, n is the number of columns, and $maxlen$ is the number of *non-zero* elements expected in each row. If you add more than $maxlen$ elements to a row, then more memory has to be allocated to that row, which can be time consuming if it is done very frequently. Also note that the NULL sparse matrix is called `SMNULL`.

Unlike dense matrices, sparse matrices have a *structure* which can be understood as the pattern of nonzero entries. More accurately, it is the set of (i, j) where memory for the a_{ij} entry is allocated. All entries outside this set are understood to have the value zero. The structure can be altered by processes such as *fill-in* during matrix factorisations or updates. However, all such alterations have a cost in terms of additional

time needed to update the data structures (as well as the values), overheads for memory reallocation, and in terms of the total amount of memory needed. Fill-in should be kept to a reasonable minimum. This can be done by using iterative methods, often in conjunction with *incomplete factorisations*, as are described later in this chapter.

Setting values of A can be done using the `sp_set_val()` function: To set the value of a_{ij} to v, you should call `sp_set_val(A,i,j,v)`. The value of a_{ij} is returned from the function call `sp_get_val(A,i,j)`.

Copying sparse matrices can be done easily too: `B = sp_copy(A)` returns a copy of the sparse matrix A, while `B = sp_copy2(A,B)` stores a copy of A in B, *while preserving the structure of* B. Preserving this structure can be extremely important in keeping the speed of factorisation algorithms high.

Input/output is generally done by two pairs of routines: `A = sp_input()` and `sp_output(A)` for input and output respectively from `stdin` and to `stdout`. For sending the output to a different file, use `sp_foutput(fp,A)`, and for reading from a different file use `A = sp_finput(fp)` where `fp` is the corresponding file pointer. As for dense matrices and vectors, the printed output can be read back in from a file. If you are typing input from a keyboard, you will be prompted for all the relevant input. However, for both means of input there is a limit of 100 entries for each row.

If worst comes to worst, and pointers are being mangled somewhere in the sparse matrix data structure, a sparse matrix can always be "dumped" out to a file by calling `sp_dump(fp,A)` which will list all the pointer locations and column access numbers etc. as well as what is usually printed out by `sp_foutput()` and `sp_output()`.

There are routines for multiplying sparse matrices by (dense) vectors, both from the right and from the left: `sp_mv_mlt(A,x,out)` forms Ax and stores the result in *out*, while `sp_vm_mlt(A,x,out)` forms $A^T x$, which is stored in *out*. Here the data types for x and *out* are both `VEC *`, while A has type `SPMAT *`. However, there is currently no routine for multiplying sparse matrices together as there is always the danger that this will lead to dense matrices. (For example, if a row of A is all ones, and a column of B is all ones, then, unless cancellation occurs, AB will have every entry nonzero.)

2.6 The sparse data structures

The data structures used for representing sparse matrices is given below:

```
typedef struct row_elt    {
    int         col, nxt_row, nxt_idx;
    Real        val;
        } row_elt;

typedef struct sp_row {
    int         len, maxlen, diag;
    row_elt     *elt;           /* elt[maxlen] */
        } SPROW;
```

```
typedef struct sp_mat {
    int         m, n, max_m, max_n;
    char        flag_col, flag_diag;
    SPROW       *row;           /* row[max_m] */
    int         *start_row;     /* start_row[max_n] */
    int         *start_idx;     /* start_idx[max_n] */
        } SPMAT;
```

The sparse matrix data structure is the **SPMAT** data structure; this in turn is built on the sparse row **SPROW** data structure, and the row element **row_elt** data structure. Thus, the sparse matrix data structure used here is a *row oriented* data structure. (By contrast, see George and Liu's book "*Computer Solution of Large, Sparse Positive Definite Systems*", Prentice Hall (1981), which uses a *column oriented* data structure.)

To scan the elements of a particular row a simple loop is all that is required:

```
int     i, j_idx, len;
  ....
len = A->row[i].len;
for ( j_idx = 0; j_idx < len; j_idx++ )
    printf("A[%d][%d] = %g\n", i, A->row[i].elt[j_idx].col,
                                  A->row[i].elt[j_idx].val);
```

Alternatively, using intermediate variables:

```
int     i, j_idx, len;
SPROW   *r;
row_elt *elt;
  ....
r = &(A->row[i]);
len = r->len;
elt = r->elt;
for ( j_idx = 0; j_idx < len; j_idx++, elt++ )
    printf("A[%d][%d] = %g\n", i, elt->col, elt->val);
```

To alleviate potential problems due to this row-oriented approach, some additional access paths were included to ease column-based access. These take the form of the **start_row** and **start_idx** arrays, and the **nxt_row** and **nxt_idx** fields of the **row_elt** data structure. These work as follows.

Suppose that **A** is a sparse matrix where this access path has been set up (i.e. **A->flag_col** is **TRUE**). To set the access paths, call **sp_col_access(A)**. The first row that a non-zero entry appears in column j is **i = A->start_row[j]**, and the index into the **A->row[i].elt** array which gives this entry is **k=A->start_idx[j]** (i.e., **A->row[i].elt[k].col == j**).

Each entry (which has type **row_elt**) has its column number, and the row number **nxt_row** and the index number **nxt_idx** of the next non-zero entry in that column. If there is no remaining non-zero entry in that column, **nxt_row** has the value -1. Listing all the entries of a particular column can then be written as a loop:

```
int     i, i_tmp, j, j_idx;
  ......
sp_col_access(A);
  ......
/* j is column number */
i     = A->start_row[j];
j_idx = A->start_idx[j];
while ( i >= 0 )
{
    printf("A[%d][%d] = %g\n", i, A->row[i].elt[j_idx].col,
                                  A->row[i].elt[j_idx].val);
    i_tmp = A->row[i].elt[j_idx].nxt_row;
    j_idx = A->row[i].elt[j_idx].nxt_idx;
    i = i_tmp;
}
```

Of course, the efficiency of this program fragment could be improved by doing the `A->row[i].elt[j_idx]` calculation only once:

```
int     i, i_tmp, j, j_idx;
row_elt *elt;
  ....
/* j is column number */
i     = A->start_row[j];
j_idx = A->start_idx[j];
while ( i >= 0 )
{
    elt = &(A->row[i].elt[j_idx]);
    printf("%g\n", elt->val);
    i_tmp = elt->nxt_row;
    j_idx = elt->nxt_idx;
    i = i_tmp;
}
```

What is assumed about this data structure is that the column indices (the `col` field of the `row_elt` data structure) are in order along the rows. This allows the use of binary searching to locate items. Adding new non-zero entries thus usually results in copying blocks of memory. The theoretically better techniques, such as B-trees and 2-3 trees, are considered too difficult to implement to be worthwhile in this context. Rather, we aim to avoid fill-in.

Whenever fill-in takes place, the column access path is rendered incorrect, as is the `diag` entry for that row. The column access path for `A` can be reset by calling `sp_col_access(A)`. Note, however, that calling `sp_col_access(A)` takes $O(m + N)$ time where m is the number of rows of `A`, and N is the number of non-zero entries in `A`. The `diag` entries for the entire matrix can be reset by calling

`sp_diag_access()`. However, in some matrix factorisations (especially Cholesky factorisation) it is more efficient to update these extra fields `nxt_row` and `nxt_idx` as fill-in occurs.

2.7 Sparse matrix factorisation

Two kinds of factorisations has been implemented, which are the sparse Cholesky and LU factorisations. The main routines are `spCHfactor()` and `spLUfactor()`. Both of these routines perform the full factorisation and create the fill-in as necessary. Supporting the sparse Cholesky factorisation is `spCHsolve()` which solves $LL^T x = b$ for x once the (sparse) Cholesky factorisation $A = LL^T$ is found for A. For the sparse LU factorisation is `spLUsolve()` which solves $P^T LUx = b$ where P is the permutation defining the row pivots. Note that the sparse LU factorisation uses partial pivoting modified to avoid too much fill-in if this is possible.

Two other variants of the sparse Cholesky factorisation are included. They are `spICHfactor()` which forms an *incomplete* factorisation of A — that is, it is *assumed* that no fill-in will take place during the Cholesky factorisation of A. There is also `spCHsymb()` which does not do any floating point arithmetic, by rather does a *symbolic* factorisation of A. The routines `spICHfactor()` and `spCHsymb()` can work together: If a number of matrices have the same pattern of zeros and non-zeros, then the pattern of zeros and non-zeros can be worked out using `spCHsymb()`, and the matrices can be copied into the resulting matrix before using `spICHfactor()` applied to the copied matrix. The code for this follows:

```
SPMAT   *pattern, *A;
  .....
/* get original A matrix */
  .....
pattern = sp_copy(A);
spCHsymb(pattern);          /* determine fill-in pattern */
  .....
sp_copy2(A,pattern);        /* preserve fill-in */
spICHfactor(pattern);       /* no additional fill-in */
  .....
/* get new A matrix */
  .....
/* assume same pattern of non-zeros in A */
sp_copy2(A,pattern);
spICHfactor(pattern);
  .....
```

There is also an incomplete LU factorisation routine `spILUfactor()`. This is actually a *modified* incomplete factorisation which modifies the diagonal entries to ensure they do not become less than a certain user-specified amount in magnitude; if this amount is set to zero then the method is just a standard incomplete factorisation.

2.8 Iterative techniques

Dealing with large, sparse matrices often requires the use of iterative methods. However, writing iterative routines that only operate on sparse matrices is unlikely to be very flexible. To this end a general data structure **ITER** is used for a wide class of iterative methods, which can be used for a wide class of problems.

One of the basic types used in the **ITER** data structure is called **Fun_Ax**: this implements a "functional representation" of a matrix. An object **Afn** of type **Fun_Ax** is a function pointer where **(*Afn)(Aparams, x, y)** computes $y = Ax$ given x. The parameter **Aparams** is a pointer which can point to any user-defined data structure (or NULL if the function ignores it). Thus the user is completely freed from the trouble of having to deal with the built in sparse matrix data structures. If, for example, the matrix is defined in terms of networks, then the data structure describing the network can be passed as **Aparams**, and the matrix-vector multiply routine modified to work directly with the network data structure. Dealing with different networks doesn't require writing new functions: only the **Aparams** parameter needs to be changed. On the other hand, use of the standard sparse data structures isn't restricted: **Afn** is **sp_mv_mlt**, the sparse matrix-vector product routine, and **Aparams** is the actual sparse matrix data structure.

This is the **ITER** data structure:

```
typedef struct Iter_data {
   int       shared_b, shared_x;
   /* TRUE if b, x aliased by other pointers */

   unsigned k;      /* no. of direction vectors; 0 = none */
   int      limit; /* upper bound on the no. of iter. steps */
   int      steps; /* no. of iter. steps done */
   Real     eps;   /* accuracy required */

   VEC      *x;    /* input: initial guess;
                      output: approx. solution */
   VEC      *b;    /* right hand side of A*x = b */

   Fun_Ax   Ax;    /* function computing y = A*x */
   void    *A_par; /* parameters for Ax */

   Fun_Ax   ATx;   /* function computing y = A^T*x */
   void    *AT_par;/* parameters for ATx */
   /* B = preconditioner */
   Fun_Ax   Bx;    /* function computing y = B*x */
   void    *B_par; /* parameters for Bx */

   /* for the following two functions: res = residual;
      nres = norm of residual res; pcres = B*res; */
```

Field	Value
`shared_b`	FALSE
`shared_x`	FALSE
`limit`	ITER_LIMIT_DEF = 1000
`k, steps`	0
`eps`	ITER_EPS_DEF = 10^{-6}
`x, b`	allocated
`Ax, Ax_par`	NULL
`ATx, ATx_par`	NULL
`Bx, Bx_par`	NULL
`info`	`iter_std_info()`
`stop_crit`	`iter_std_stop_crit()`

Table 2.1: Default values for the `ITER` structure

```
    /* function giving some information for a user */
    void (*info)(struct Iter_data *ip, double nres,
                 VEC *res, VEC *pcres);
    /* stopping criterion: stop if TRUE returned; */
    int (*stop_crit)(struct Iter_data *ip, double nres,
                 VEC *res, VEC *pcres);

    Real init_res; /* the norm of the initial residual */
}   ITER;
```

The main routine for setting up an `ITER` data structure is `ip = iter_get(b_dim, x_dim)` which creates an `ITER` data structure with NULL functions, default values for the other components of the data structure, and with two vectors `x` and `b` created (of lengths `x_dim` and `b_dim` respectively). The other memory operations involved are `iter_resize(ip, new_b_dim, new_x_dim)` to resize `ip`, and `iter_free(ip)` (function) and `ITER_FREE(ip)` (macro) to free `ip`. The default values of the various entries of the `ITER` structure are given in Table 2.1:

Setting the values in the data structure requires setting the fields of the `ITER` structure directly. The function `iter_dump(fp, ip)` prints out information about the the `ITER` data structure `ip` to stream/file `fp`. The routine `iter_copy(ip1,ip2)` copies the `ITER` structure and the `x` and `b` structures. (This is a *deep copy*.) The routine `iter_copy2(ip1,ip2)` copies all of the `ITER` structure's values but leaves `ip2->x` and `ip2->b` unchanged.

These `ITER` data structures are used in the main iterative routines, such as
`iter_cg(ip)` which implements (pre-conditioned) conjugate gradients;
`iter_lanczos(ip, ....)` which implements the basic Lanczos algorithm;
`iter_cgs(ip, r0)` which implements Sonneveld's CGS algorithm;
`iter_gmres(ip)` which implements Saad and Schultz's GMRES algorithm.

There are some additional routines which provide a simplified interface for applying iterative methods to sparse matrix data structures. These routines are named `iter_sp...(...)`, such as `iter_spcg(A,LLT,b,eps,x,limit,steps)` for (pre-conditioned) conjugate gradients. The `iter_sp...(...)` routines work by setting up an `ITER` data structure and calling the appropriate main routine.

The use of more than one level of interface means that simplicity is not sacrificed for the sake of more sophisticated users.

2.9 Other data structures

The above data structures can be used as parts of other data structures. For example, here is an data structure for holding simplex tableaus for linear programmes:

```
typedef struct lp {
            MAT     *tab;
            VEC     *rhs, *cost;
            Real    val;
            PERM    *basis, *invbase, *allow;
            int     card;
        } LP;
```

Routines for creating and destroying, inputting and outputting, and using this data structure have been written, based on the corresponding routines for the component data structures. It may be of interest that `basis` is a permutation, and that during operations on the simplex tableau, `in_base` is maintained as the inverse permutation to `basis`. Finally, the permutation `allow` together with `card` act as a set which consists of the elements

```
{allow->pe[0],allow->pe[1], allow->pe[2],
                ... ,allow->pe[card-1]}.
```

Meschach 1.2 allows you to incorporate your own data structures into various aspects of the library, such as tracking memory usage and deallocating static workspace when desired. For suggestions for implementing your own data structures and using Meschach routines in your applications, see chapter 8 on designing libraries in C.