Chapter 2

Data structures

In this chapter an overview of the data structures is given, as well as indicating how memory management is undertaken.

2.1 Vectors

The vector data structure is the VEC structure:

```
typedef unsigned int    u_int;
/* vector definition */
typedef struct  {
                u_int   dim, max_dim;
                double  *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 get_vec(), which returns a pointer to a 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 = get_vec(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, ... , $n-1$.

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, then 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 freevec() 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 get_vec(), cp_vec() and in_vec() which also take care of any initialisation that is needed. Pointers (as returned by these functions) can also be freed up; this is *not* true of objects declared as type VEC. They are either allocated statically, or on a stack, and no memory is actually allocated for storing the vector's entries. Any attempt to use freevec() on such a "vector" will result in a compile-time error at best, and a program crash at worst. (Using freevec() will result in a compile-time error.)

2.1.1 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 get_ivec(), 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 ith entry of an integer vector iv is iv->ive[i], and indexing is zero relative so i must be in the range 0, 1, ... , iv->dim-1.

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

2.2 Matrices

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

```
typedef unsigned int    u_int;
/* matrix definition */
typedef struct  {
                u_int   m, n;
                u_int   max_m, max_size;
                double  **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;
double 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 double precision numbers.

It should be noted that me is actually an array with elements of type double *. 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. As noted with vectors, 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 freemat() or m_free().

2.3 Permutations

While permutations are not intrinsically part of numerical algorithms, they 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: get_perm() 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 is 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.

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

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

```
sp_mat     *A;
```

Initialising a sparse matrix requires calling the sp_get_mat() function:

```
A = sp_get_mat(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 is 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)`. Then 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_cp_mat(A)` returns a copy of the sparse matrix A, while `B = sp_cp_mat2(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_in_mat()` and `sp_out_mat(A)` for input and output respectively from `stdin` and to `stdout`. For sending the output to a different file, use `sp_fout_mat(fp,A)`, and for reading from a different file use `A = sp_fin_mat(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_mat(fp,A)` which will list all the pointer locations and column access numbers etc. as well as what is usually printed out by `sp_fout_mat()` and `sp_out_mat()`.

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 `sp_mat *`. 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.)

2.5 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;
    double     val;
        } row_elt;

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

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

The sparse matrix data structure is the `sp_mat` data structure; this in turn is built on the sparse row `sp_row` 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 is based on 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;
sp_row  *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`). 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 `A->start_idx[j]`.

Each entry (which has type `row_elt`) has its column number and also 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;
  ....
/* 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.

2.6 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 $LL^Tx = b$ for x once the (sparse) Cholesky factorisation $A = LL^T$ is found for A. For the sparse LU factorisation is `spLUsolve()` which solves $P^TLUx = b$ 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:

```
sp_mat   *pattern, *A;
  .....
/* get original A matrix */
  .....
pattern = sp_cp_mat(A);
spCHsymb(pattern);          /* determine fill-in pattern */
  .....
```

```
sp_cp_mat2(A,pattern);      /* preserve fill-in */
spICHfactor(pattern);       /* no additional fill-in */
  .....
/* get new A matrix */
  .....
/* assume same pattern of non-zeros in A */
sp_cp_mat2(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.

2.7 Iterative techniques

A number of iterative techniques have been implemented, such as conjugate gradient and Lanczos methods. These, however, do not directly use the sparse matrix data structure, but rather functions and `void *`'s, to define the matrices. For these routines, the matrix `A` would be represented by a function `VEC *A_fn(params,x,y)` where `params` is a `void *` containing pointing to a user-defined data structure (if any), `x` is the input vector and `y = A.x` is the result of multiplying the matrix `A` by `x` at the end of execution of `A_fn()`. `MTX_FN` has been `typedef`'d to be the type of `A_fn`.

As an example of its use, consider the pre-conditioned, conjugate gradient function, `pccg()`:

```
VEC     *pccg(A_fn,A_params,M_inv,M_params,b,eps,x)
MTX_FN  A_fn, M_inv;
VEC     *b, *x;
double  eps;
void    *A_params, *M_params;
```

To use this with the above sparse matrix data structures, there is also a function `sp_pccg()` which sets the parameters `A_params` and `M_params` to be pointers to the sparse matrix data structures for `A` and for the Cholesky factorisation `LLT` of `M`, as produced by `spCHfactor()`. Then the function `A_fn` is just `sp_mv_mlt`, and the function `M_inv` is just `spCHsolve()`. When `A_fn` is then called, the function call actually used is `(*A_fn)(A_params,x,y) == sp_mv_mlt(A,x,y)` which gives the desired result. When `M_inv` is then called the function call actually used is `(*M_inv)(M_params,x,y) == spCHsolve(LLT,x,y)` which returns with `y` equal to the solution of `M.y = x`.

The actual declaration of this extra function is simply

```
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); }
```

Then, to solve $Ax = b$, A symmetric, positive definite and sparse using a preconditioner you can use the following code:

```
/* Initialise A and b */
  ......
M = sp_get_mat(A->m,A->n,10);
/* set up preconditioner */
  ......
```

```
sp_set_val(M,i,j,.....);
  ......
spCHfactor(M);
sp_pccg(A,M,b,1e-6,x);
```

This can be used in concert with the incomplete Cholesky factorisation routine:

```
/* Initialise A and b */
 ......
/* set up preconditioner */
M = sp_cp_mat(A); /* copy A ... */
spICHfactor(M); /* ... and use an incomplete factorisation */
sp_pccg(A,M,b,1e-6,x);
```

In addition to pccg() there is also cgs() for solving non-symmetric systems of equations, lsqr() for large least-squares problems, and the Lanczos routines lanczos() and lanczos2(). All of these have sparse matrix versions which are named by pre-pending "sp_" to the above names.

2.8 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;
            double  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]}.

For implementing your own data structures, see chapter 9 on designing libraries in C.

Contents

2 Data structures **16**

2.1 Vectors 16

2.1.1 Integer vectors 17

2.2 Matrices 17

2.3 Permutations 18

2.4 Basic sparse operations and structures 18

2.5 The sparse data structures 19

2.6 Sparse matrix factorisation 21

2.7 Iterative techniques 22

2.8 Other data structures 23