

Chapter 1

Tutorial

In this chapter, the basic data structures are introduced, and some of the more basic operations are illustrated. Then some examples of how to use the data structures and procedures to solve some simple problems are given. The first example program is a simple 4th order Runge–Kutta solver for Ordinary Differential Equations. The second is a general least squares equation solver for over–determined equations. The third example illustrates how to solve a problem involving sparse matrices. These examples illustrate the use of matrices, matrix factorisations and solving systems of linear equations. The examples described in this chapter are implemented in `tutorial.c`.

While the description of each aspect of the system is brief and far from comprehensive, the aim is to show the different aspects of how to set up programs and routines and how these work in practice, which includes I/O and error–handling issues.

1.1 The data structures and some basic operations

The three main data structures are those describing vectors, matrices and permutations. These have been used to create data structures for simplex tableaus for linear programming, and used with data structures for sparse matrices etc. To use the system reliably, you should always use pointers to these data structures and use library routines to do all the necessary initialisation. In fact, for the operations that involve memory management (creation, destruction and resizing), it is essential that you use the routines provided.

For example, to create a matrix A of size 3×4 , a vector x of dimension 10, and a permutation p of size 10, use the following code:

```
#include "matrix.h"
.....
main()
{
    MAT    *A;
    VEC    *x;
    PERM   *p;
```

```

.....
A = m_get(3,4);
x = v_get(10);
p = px_get(10);
.....
}

```

This initialises these data structures to have the given size. The matrix A and the vector x are initially all zero, while p is initially the identity permutation. They can be disposed of by calling `M_FREE(A)`, `V_FREE(x)` and `PX_FREE(p)` respectively if you need to re-use the memory for something else. The elements of each data structure can be accessed directly using the members (or fields) of the corresponding structures. For example the (i, j) component of A is accessed by `A->me[i][j]`, x_i by `x->ve[i]` and p_i by `p->pe[i]`.

Their sizes are also directly accessible: `A->m` and `A->n` are the number of rows and columns of A respectively, `x->dim` is the dimension of x , and size of p is `p->size`. Note that the indexes are *zero relative* just as they are in ordinary C, so that the index i in `x->ve[i]` can range from 0 to `x->dim - 1`. Thus the total number of entries of a vector is exactly `x->dim`.

While this alone is sufficient to allow a programmer to do any desired operation with vectors and matrices it is neither convenient for the programmer, nor efficient use of the CPU. A whole library has been implemented to reduce the burden on the programmer in implementing algorithms with vectors and matrices. For instance, to copy a vector from x to y it is sufficient to write `y = v_copy(x, VNULL)`. The `VNULL` is the NULL vector, and usually tells the routine called to create a vector for output. Thus, the `v_copy` function will create a vector which has the same size as x and all the components are equal to those of x . If y has already been created then you can write `y = v_copy(x, y)`; in general, writing “`v_copy(x, y)` ;” is not enough! If y is NULL, then it is created (to have the correct size, i.e. the same size as x), and if it is the wrong size, then it is resized to have the correct size (i.e. same size as x). Note that for all the following functions, the output value is returned, even if you have a non-NULL value as the output argument. This is the standard across the entire library.

Addition, subtraction and scalar multiples of vectors can be computed by calls to library routines: `v_add(x, y, out)`, `v_sub(x, y, out)`, `sv_mlt(s, x, out)` where x and y are input vectors (with data type `VEC *`), `out` is the output vector (same data type) and s is a double precision number (data type `double`). There is also a special combination routine, which computes $out = v_1 + s v_2$ in a single routine: `v_mltadd(v1, v2, s, out)`. This is not only extremely useful, it is also more efficient than using the scalar-vector multiply and vector addition routines separately.

Inner products can be computed directly: `in_prod(x, y)` returns the inner product of x and y . Note that extended precision evaluation is not guaranteed. The standard installation options uses double precision operations throughout the library.

Equivalent operations can be performed on matrices: `m_add(A, B, C)` which returns $C = A + B$, and `sm_mlt(s, A, C)` which returns $C = sA$. The data types

of **A**, **B** and **C** are all **MAT ***, while that of **s** is type **double** as before. The matrix **NULL** is called **MNULL**.

Multiplying matrices and vectors can be done by a single function call:

mv_mlt(A, x, out) returns $out = Ax$ while **vm_mlt(A, x, out)** returns $out = A^T x$, or equivalently, $out^T = x^T A$. Note that there is no distinction between row and column vectors unlike certain interactive environments such as **MATLAB** or **MATCALC**.

Permutations are also an essential part of the package. Vectors can be permuted by using **px_vec(p, x, p_x)**, rows and columns of matrices can be permuted by using **px_rows(p, A, p_A)**, **px_cols(p, A, A_p)**, and permutations can be multiplied using **px_mlt(p1, p2, p1_p2)** and inverted using **px_inv(p, p_inv)**. The **NULL** permutation is called **PXNULL**.

There are also utility routines to initialise or re-initialise these data structures:

v_zero(x), **m_zero(A)**, **m_ident(A)** (which sets $A = I$ of the correct size), **v_rand(x)**, **m_rand(A)** which sets the entries of **x** and **A** respectively to be randomly and uniformly selected between zero and one, and **px_ident(p)** which sets **p** to be an identity permutation.

Input and output are accomplished by library routines **v_input(x)**, **m_input(A)**, and **px_input(p)**. If a null object is passed to any of these input routines, all data will be obtained from the input file, which is **stdin**. If input is taken from a keyboard then the user will be prompted for all the data items needed; if input is taken from a file, then the input will have to be of the same format as that produced by the output routines, which are: **v_output(x)**, **m_output(A)** and **px_output(p)**. This output is both human and machine readable!

If you wish to send the data to a file other than the standard output device **stdout**, or receive input from a file or device other than the standard input device **stdin**, take the appropriate routine above, use the “**foutput**” suffix instead of just “**output**”, and add a file pointer as the first argument. For example, to send a matrix **A** to a file called “fred”, use the following:

```
#include "matrix.h"
.....
main()
{
    FILE *fp;
    MAT *A;
    .....
    fp = fopen("fred", "w");
    m_foutput(fp, A);
    .....
}
```

These input routines allow for the presence of comments in the data. A comment in the input starts with a “hash” character “#”, and continues to the end of the line. For example, the following is valid input for a 3-dimensional vector:

```
# The initial vector must not be zero
# x =
Vector: dim: 3
-7      0      3
```

For general input/output which conforms to this format, allowing comments in the input files, use the `input()` and `finput()` macros. These are used to print out a prompt message if `stdin` is a terminal (or “tty” in Unix jargon), and to skip over any comments if input is from a non-interactive device. An example of the usage of these macros is:

```
input("Input number of steps: ", "%d", &nsteps);
fp = stdin;
finput(fp, "Input number of steps: ", "%d", &nsteps);
fp = fopen("fred", "r");
finput(fp, "Input number of steps: ", "%d", &nsteps);
```

The “%d” strings are the format strings as used by `scanf()` and `fscanf()`; the last argument is the pointer to the variable (unless the variable is a string) just as for `scanf()` and `fscanf()`. The first two macro calls read input from `stdin`, the last from the file `fred`. If, in the first two calls, `stdin` is a keyboard (a “tty” in Unix jargon) then the prompt string “Input number of steps: ” is printed out on the terminal.

The second part of the library contains routines for various factorisation methods. To use it put

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

at the beginning of your program. It contains factorisation and solution routines for LU, Cholesky and QR-factorisation methods, as well as update routines for Cholesky and QR factorisations. Supporting these are a number of Householder transformation and Givens’ rotation routines. Also there is a routine for generating the Q matrix for a QR-factorisation, if it is needed explicitly, as it often is. There are routines for band factorisation and solution for LU and LDL^T factorisations.

For using complex numbers, vectors and matrices include

```
#include "zmatrix.h"
```

for using the basic routines, and

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

for the complex matrix factorisation routines. The `zmatrix2.h` file includes `matrix.h` and `zmatrix.h` so you don’t need these files included together.

For using the sparse matrix routines in the library you need to put

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

or, if you use any sparse factorisation routines

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

at the beginning of your file. The routines contained in the library include routines for creating, destroying, initialising and updating sparse matrices, and also routines for sparse matrix–dense vector multiplication, sparse LU factorisation and sparse Cholesky factorisation.

For using the iterative routines you need to use

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

This includes the `sparse.h` and `matrix.h` file. There are also routines for applying iterative methods such as pre-conditioned conjugate gradient methods to sparse matrices.

And if you use the standard maths library (`sin()`, `cos()`, `tan()`, `exp()`, `log()`, `sqrt()`, `acos()` etc.) don't forget to include the standard mathematics header:

```
#include <math.h>
```

This file is *not* automatically included by any of the Meschach header files.

1.2 How to manage memory

Unlike many other numerical libraries, Meschach allows you to allocate, deallocate and resize the vectors, matrices and permutations that you are using. To gain maximum benefit from this it is sometimes necessary to think a little about where memory is allocated and deallocated. There are two reasons for this.

1. Memory allocation, deallocation and resizing takes a significant amount of time compared with (say) vector operations, so it should not be done too frequently.
2. Allocating memory but not deallocating it means that it can't be used by any other data structure. Data structures that are no longer needed should be explicitly deallocated, or kept as static variables for later use. Unlike other interpreted systems (such as Lisp) there is no implicit "garbage collection" of no-longer-used memory.

There are three main strategies that are recommended for deciding how to allocate, deallocate and resize objects. These are "*no deallocation*" which is really only useful for demonstration programs, "*allocate and deallocate*" which minimises overall memory requirements at the expense of speed, and "*resize on demand*" which is useful for routines that are called repeatedly. A new technique for static workspace arrays is to "*register workspace variables*".

1.2.1 No deallocation

This is the strategy of allocating but never deallocating data structures. This is only useful for demonstration programs run with small to medium size data structures. For example, there could be a line

```
QR = m_copy(A, MNULL);      /* allocate memory for QR */
```

to allocate the memory, but without the call `M_FREE(QR)`; in it. This can be acceptable if `QR = m_copy(A, MNULL)` is only executed once, and so the allocated memory never needs to be explicitly deallocated.

This would *not* be acceptable if `QR = m_copy(A, MNULL)` occurred inside a `for` loop. If this were so, then memory would be “lost” as far as the program is concerned until there was insufficient space for allocating the next matrix for `QR`. The next subsection shows how to avoid this.

1.2.2 Allocate and deallocate

This is the most straightforward way of ensuring that memory is not lost. With the example of allocating `QR` it would work like this:

```
for ( ... ; ... ; ... )
{
    QR = m_copy(A, MNULL); /* allocate memory for QR */
    /* could have been allocated by m_get() */
    /* use QR */
    .....
    .....
    /* deallocate QR so memory can be reused */
    M_FREE(QR);
}
```

The allocate and deallocate statements could also have come at the beginning and end of a function or procedure, so that when the function returns, all the memory that the function has allocated has been deallocated.

This is most suitable for functions or sections of code that are called repeatedly but involve fairly extensive calculations (at least a matrix–matrix multiply, or solving a system of equations).

1.2.3 Resize on demand

This technique reduces the time involved in memory allocation for code that is repeatedly called or used, especially where the same size matrix or vector is needed. For example, the vectors `v1`, `v2`, etc. in the Runge–Kutta routine `rk4()` are allocated according to this strategy:

```
rk4(...,x,...)
{
    static VEC *v1=VNULL, *v2=VNULL, *v3=VNULL,
              *v4=VNULL, *temp=VNULL;

    .....
    v1   = v_resize(v1,x->dim);
    v2   = v_resize(v2,x->dim);
    v3   = v_resize(v3,x->dim);
    v4   = v_resize(v4,x->dim);
    temp = v_resize(temp,x->dim);
    .....
}
```

The intention is that the `rk4()` routine is called repeatedly with the same size `x` vector. It then doesn't make as much sense to allocate `v1`, `v2` etc. whenever the function is called. Instead, `v_resize()` only performs memory allocation if the memory already allocated to `v1`, `v2` etc. is smaller than `x->dim`.

The vectors `v1`, `v2` etc. are declared to be `static` to ensure that their values are not lost between function calls. Variables that are declared `static` are set to `NULL` or zero by default. So the declaration of `v1`, `v2`, etc., could be

```
static VEC *v1, *v2, *v3, *v4, *temp;
```

This strategy of resizing static workspace variables is not so useful if the object being allocated is extremely large. The previous "allocate and deallocate" strategy is much more efficient for memory in those circumstances. However, the following section shows how to get the best of both worlds.

1.2.4 Registration of workspace

From version 1.2 onwards, workspace variables can be *registered* so that the memory they reference can be freed up on demand. To do this, the function containing the static workspace variables has to include calls to `MEM_STAT_REG(var, type)` where `var` is a pointer to a Meschach data type (such as `VEC` or `MAT`). This call should be placed *after* the call to the appropriate resize function. The `type` parameter should be a `TYPE_...` macro where the "..." is the name of a Meschach type such as `VEC` or `MAT`. For example,

```
rk4(...,x,...)
{
    static VEC *v1, *v2, *v3, *v4, *temp;
    .....
    v1   = v_resize(v1,x->dim);
    MEM_STAT_REG(v1,TYPE_VEC);
    v2   = v_resize(v2,x->dim);
    MEM_STAT_REG(v2,TYPE_VEC);
```

```

    .....
}

```

Normally, these registered workspace variables remain allocated. However, to implement the “deallocate on exit” approach, use the following code:

```

    .....
mem_stat_mark(1);
rk4(...,x,...)
mem_stat_free(1);
    .....

```

To keep the workspace vectors allocated for the duration of a loop, but then deallocated, use

```

    .....
mem_stat_mark(1);
for (i = 0; i < N; i++ )
    rk4(...,x,...);
mem_stat_free(1);
    .....

```

The number used in the `mem_stat_mark()` and `mem_stat_free()` calls is the *workspace group number*. The call `mem_stat_mark(1);` designates 1 as the current workspace group number; the call `mem_stat_free(1);` deallocates (and sets to NULL) all static workspace variables registered as belonging to workspace group 1.

1.3 Simple vector operations: An RK4 routine

The main purpose of this example is to show how to deal with vectors and to compute linear combinations.

The problem here is to implement the standard 4th order Runge–Kutta method for the ODE

$$x' = f(t, x), \quad x(t_0) = x_0$$

for $x(t_i)$, $i = 1, 2, 3, \dots$ where $t_i = t_0 + i h$ and h is the step size. The formulae for the 4th order Runge–Kutta method are:

$$x_{i+1} = x_i + \frac{h}{6} \{v_1 + 2v_2 + 2v_3 + v_4\}$$

where

$$v_1 = f(t_i, x_i)$$

$$v_2 = f\left(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hv_1\right)$$

$$v_3 = f\left(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hv_2\right)$$

$$v_4 = f(t_i + h, x_i + hv_3)$$

(1.1)

where the v_i are vectors.

The procedure for implementing this method (`rk4()`) will be passed (a pointer to) the function f ; the implementation of f could, in this system, create a vector to hold the return value each time it is called. However, such a scheme is memory intensive and the calls to the memory allocation functions could easily dominate the time performed doing numerical computations. So, the implementation of f will also be passed an already allocated vector to be filled in with the appropriate values.

The procedure `rk4()` will also be passed the current time t , the step size h , and the current value for x . The time after the step will be returned by `rk4()`.

The code that does this follows.

```
#include "matrix.h"

/* rk4 -- 4th order Runge--Kutta method */
double rk4(f,t,x,h)
double t, h;
VEC      *(*f)(), *x;
{
    static VEC *v1=VNULL, *v2=VNULL, *v3=VNULL, *v4=VNULL;
    static VEC *temp=VNULL;

    /* do not work with NULL initial vector */
    if ( x == VNULL )
        error(E_NULL,"rk4");

    /* ensure that v1, v2, etc. are of the correct size */
    v1  = v_resize(v1,x->dim);
    v2  = v_resize(v2,x->dim);
    v3  = v_resize(v3,x->dim);
    v4  = v_resize(v4,x->dim);
    temp = v_resize(temp,x->dim);
    /* register workspace variables */
    MEM_STAT_REG(v1,TYPE_VEC);
    MEM_STAT_REG(v2,TYPE_VEC);
    MEM_STAT_REG(v3,TYPE_VEC);
    MEM_STAT_REG(v4,TYPE_VEC);
    MEM_STAT_REG(temp,TYPE_VEC);
    /* end of memory allocation */
    (*f)(t,x,v1); /* most compilers allow: "f(t,x,v1);" */
    v_mltadd(x,v1,0.5*h,temp); /* temp = x+.5*h*v1 */
    (*f)(t+0.5*h,temp,v2);
    v_mltadd(x,v2,0.5*h,temp); /* temp = x+.5*h*v2 */
    (*f)(t+0.5*h,temp,v3);
    v_mltadd(x,v3,h,temp); /* temp = x+h*v3 */
    (*f)(t+h,temp,v4);
```

```

/* now add: v1+2*v2+2*v3+v4 */
v_copy(v1,temp);           /* temp = v1 */
v_mltadd(temp,v2,2.0,temp); /* temp = v1+2*v2 */
v_mltadd(temp,v3,2.0,temp); /* temp = v1+2*v2+2*v3 */
v_add(temp,v4,temp);       /* temp = v1+2*v2+2*v3+v4 */

/* adjust x */
v_mltadd(x,temp,h/6.0,x);  /* x = x+(h/6)*temp */
return t+h;                /* return the new time */
}

```

Note that the last parameter of $f()$ is where the *output* is placed. Often this can be `NULL` in which case the appropriate data structure is allocated and initialised. Note also that this routine can be used for problems of arbitrary size, and the dimension of the problem is determined directly from the data given. The vectors v_1, \dots, v_4 are created to have the correct size in the lines

```

v1 = v_resize(v1,x->dim);
v2 = v_resize(v2,x->dim);
....

```

Here `v_resize(v,dim)` resizes the `VEC` structure `v` to hold a vector of length `dim`. If `v` is initially `NULL`, then this creates a new vector of dimension `dim`, just as `v_get(dim)` would do. For the above piece of code to work correctly, `v1`, `v2` etc., must be initialised to be `NULL` vectors. This is done by the declaration

```
static VEC *v1=VNULL, *v2=VNULL, *v3=VNULL, *v4=VNULL;
```

or

```
static VEC *v1, *v2, *v3, *v4;
```

The operations of vector addition and scalar addition are really the only *vector* operations that need to be performed in `rk4`. Vector addition is done by `v_add(v1,v2,out)`, where `out=v1+v2`, and scalar multiplication by `sv_mlt(scale,v,out)`, where `out=scale*v`.

These can be combined into a single operation `v_mltadd(v1,v2,scale,out)`, where `out=v1+scale*v2`. As many operations in numerical mathematics involve accumulating scalar multiples, this is an extremely useful operation, as we can see above. For example:

```
v_mltadd(x,v1,0.5*h,temp); /* temp = x+.5*h*v1 */
```

We also need a number of “utility” operations. For example `v_copy(in,out)` copies the vector `in` to `out`. There is also `v_zero(v)` to zero a vector `v`.

Here is an implementation of the function f for simple harmonic motion:

```

/* f -- right-hand side of ODE solver */
VEC *f(t,x,out)
VEC *x, *out;
double t;
{
    if ( x == VNULL || out == VNULL )
        error(E_NULL,"f");
    if ( x->dim != 2 || out->dim != 2 )
        error(E_SIZES,"f");

    out->ve[0] = x->ve[1];
    out->ve[1] = - x->ve[0];

    return out;
}

```

As can be seen, most of this code is error checking code, which, of course, makes the routine safer but a little slower. For a procedure like $f()$ it is probably not necessary, although then the main program would have to perform checking to ensure that the vectors involved have the correct size etc. The i th component of a vector x is $x->ve[i]$, and indexing is zero-relative (i.e., the “first” component is component 0). The ODE described above is for simple harmonic motion: $x_0' = x_1$, $x_1' = -x_0$, or equivalently, $x_0'' + x_0 = 0$.

Here is the main program:

```

#include <stdio.h>
#include "matrix.h"

main()
{
    VEC      *x;
    VEC      *f();
    double   h, t, t_fin;
    double   rk4();

    input("Input initial time: ", "%lf",&t);
    input("Input final time: ", "%lf",&t_fin);
    x = v_get(2); /* this is the size needed by f() */
    prompter("Input initial state:\n"); x = v_input(VNULL);
    input("Input step size: ", "%lf",&h);

    printf("# At time %g, the state is\n",t);
    v_output(x);
}

```

```

while ( t < t_fin )
{
    t = rk4(f,t,x,min(h,t_fin-t));/* new t is returned */
    printf("# At time %g, the state is\n",t);
    v_output(x);
    t += h;
}
}

```

Here the initial values are entered as a vector by `v_input()`. If `v_input()` is passed a vector, then this vector will be used to store the input, and this vector has the size that `x` had on entry to `v_input()`. The original values of `x` are also used as a prompt on input from a tty. If a `NULL` is passed to `v_input()` then `v_input()` will return a vector of whatever size the user inputs. So, to ensure that only a two-dimensional vector is used for the initial conditions (which is what `f()` is expecting) we use

```
x = v_get(2);      x = v_input(x);
```

To compile the program under UnixTM, if it is in a file `tutorial.c` is:

```
cc -o tutorial tutorial.c meshach.a
```

or, if you have an ANSI compiler,

```
cc -DANSI_C -o tutorial tutorial.c meshach.a
```

Here is a sample session with the above program:

```

% tutorial
.....
Input initial time: 0
Input final time: 1
Input initial state:
Vector: dim: 2
entry 0: -1
entry 1: b
entry 0: old      -1 new: 1
entry 1: old      0 new: 0
Input step size: 0.1
At time 0, the state is
Vector: dim: 2
          1          0
At time 0.1, the state is
Vector: dim: 2
    0.995004167  -0.0998333333
.....

```

At time 1, the state is

Vector: dim: 2

0.540302967 -0.841470478

By way of comparison, the state at $t = 1$ for the true solution is $x_0(1) = 0.5403023058$, $x_1(1) = -0.8414709848$. The “b” that is typed in entering the \mathbf{x} vector allows the user to alter previously entered components; in this case once this is done, the user is prompted with the old values when entering the new values. The user can also type in “f” for skipping over the vector’s components, which are then unchanged. If an incorrectly sized initial value vector \mathbf{x} is given, the error handler comes into action:

```
% tutorial
```

```
.....
```

```
Input initial time: 0
```

```
Input final time: 1
```

```
Input initial state:
```

```
Vector: dim: 3
```

```
entry 0: 3
```

```
entry 1: 2
```

```
entry 2: -1
```

```
Input step size: 0.1
```

```
At time 0, the state is
```

```
Vector: dim: 3
```

```
3
```

```
2
```

```
-1
```

```
"tutorial.c", line 79: sizes of objects don't match in
function f()
```

```
Sorry, aborting program
```

```
%
```

The error handler prints out the error message giving the source code file and line number as well as the function name where the error was raised. The relevant section of $f()$ in file `test1.c` is:

```
if ( x->dim != 2 || out->dim != 2 )
```

```
error(E_SIZES, "f");
```

```
/* line 79 */
```

The standard routines in this system perform error checking of this type, and also checking for undefined results such as division by zero in the routines for solving systems of linear equations. There are also error messages for incorrectly formatted input and end-of-file conditions.

To round off the discussion of this program, note that we have seen interactive input of vectors. If the input file or stream is not a tty (e.g., a file, a pipeline or a device) then it expects the input to *have the same form as the output for each of the data structures*. Each of the input routines (`v_input()`, `m_input()`, `px_input()`)

skips over “comments” in the input data, as do the macros `input()` and `finput()`. Anything from a ‘#’ to the end of the line (or EOF) is considered to be a comment. For example, the initial value problem could be set up in a file `ivp.dat` as:

```
# Initial time
0
# Final time
1
# Solution is  $x(t) = (\cos(t), -\sin(t))$ 
#  $x(0) =$ 
Vector: dim: 2
1      0
# Step size
0.1
```

The output of the above program with the above input (from a file) gives essentially the same output as shown above on p. 12, except that no prompts are sent to the screen.

1.4 Using routines for lists of arguments

Some of the most common routines have variants that take a variable number of arguments. These are the routines `..get_vars()`, `..resize_vars()` and `..free_vars()`. These correspond to the basic routines `..get()`, `..resize()` and `..free()` respectively. Also there is the `mem_stat_reg_vars()` routine which registers a list of static workspace variables; this corresponds to `mem_stat_reg_list()` for a single variable. Here is an example of how to use these functions. This example, also uses the routine `v_linlist()` to compute a linear combination. Note that the code is much more compact, but don’t forget that these “`.._vars()`” routines usually need the address-of operator “&” and NULL termination of the arguments for these to work correctly.

```
#include "matrix.h"

/* rk4 -- 4th order Runge--Kutta method */
double rk4(f,t,x,h)
double t, h;
VEC    *(*f)(), *x;
{
    static VEC *v1, *v2, *v3, *v4, *temp;

    /* do not work with NULL initial vector */
    if ( x == VNULL )        error(E_NULL,"rk4");

    /* ensure that v1, v2 etc. are of the correct size */
    v_resize_vars(x->dim,&v1,&v2,&v3,&v4,&temp,NULL);
```

```

/* register workspace variables */
mem_stat_reg_vars(0,TYPE_VEC,&v1,&v2,&v3,&v4,&temp,NULL);
/* end of memory allocation */
(*f)(t,x,v1);          v_mltadd(x,v1,0.5*h,temp);
(*f)(t+0.5*h,temp,v2); v_mltadd(x,v2,0.5*h,temp);
(*f)(t+0.5*h,temp,v3); v_mltadd(x,v3,h,temp);
(*f)(t+h,temp,v4);

/* now add: temp = v1+2*v2+2*v3+v4 */
v_linlist(temp,v1,1.0,v2,2.0,v3,2.0,v4,1.0,VNULL)
/* adjust x */
v_mltadd(x,temp,h/6.0,x);      /* x = x+(h/6)*temp */

return t+h;                    /* return the new time */
}

```

1.5 A least squares problem

Here we need to use matrices and matrix factorisations (in particular, a QR factorisation) in order to find the best linear least squares solution to some data. Thus in order to solve the (approximate) equations

$$Ax \approx b \quad \text{for } x$$

where A is an $m \times n$ matrix ($m > n$) we really need to solve the optimisation problem

$$\min_x \|Ax - b\|_2^2.$$

If we write $A = QR$ where Q is an orthogonal $m \times m$ matrix and R is an upper triangular $m \times n$ matrix then

$$(1.2) \quad \|Ax - b\|_2 = \|Rx - Q^T b\|_2 = \left\| \begin{bmatrix} R_1 \\ O \end{bmatrix} x - \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} b \right\|_2$$

where R_1 is an $n \times n$ upper triangular matrix. If A has full rank then R_1 will be an invertible matrix, and the best least squares solution of $Ax \approx b$ is $x = R_1^{-1} Q_1^T b$.

These calculations can be done quite easily as there is a `QRfactor()` function available with the system. `QRfactor()` is declared to have the prototype

```
MAT      *QRfactor(MAT *A, VEC *diag);
```

The matrix **A** is overwritten with the factorisation of **A** “in compact form”; that is, while the upper triangular part of **A** is indeed the R matrix described above, the Q matrix is stored as a collection of Householder vectors in the strictly lower triangular part of **A** and in the **diag** vector. The `QRsolve()` function knows and uses this compact form and solves $QRx \approx b$ with the call `QRsolve(A,diag,b,x)`, which also returns **x**.

Here is the code to obtain the matrix A , perform the QR factorisation, obtain the data vector b , solve for x , and determine what the norm of the errors ($\|Ax - b\|_2$) is.

```

#include "matrix2.h"

main()
{
    MAT *A, *QR;
    VEC *b, *x, *diag;

    /* read in A matrix */
    printf("Input A matrix:\n");

    A = m_input(MNULL); /* A has whatever size is input */

    if ( A->m < A->n )
    {
        printf("Need m >= n to obtain least squares fit\n");
        exit(0);
    }
    printf("# A =\n");          m_output(A);
    diag = v_get(A->m);
    /* QR is to be the QR factorisation of A */
    QR = m_copy(A,MNULL);
    QRfactor(QR,diag);
    /* read in b vector */
    printf("Input b vector:\n");
    b = v_get(A->m);
    b = v_input(b);
    printf("# b =\n");          v_output(b);

    /* solve for x */
    x = QRsolve(QR,diag,b,VNULL);
    printf("Vector of best fit parameters is\n");
    v_output(x);
    /* ... and work out norm of errors... */
    printf("||A*x-b|| = %g\n",
           v_norm2(v_sub(mv_mlt(A,x,VNULL),b,VNULL)));
}

```

Note that as well as the usual memory allocation functions like `m_get()`, the I/O functions like `m_input()` and `m_output()`, and the factorise-and-solve functions `QRfactor()` and `QRsolve()`, there are also functions for matrix-vector multiplication: `mv_mlt(MAT *A, VEC *x, VEC *out)`. and also vector-matrix multiplication (with the vector on the left): `vm_mlt(MAT *A, VEC *x, VEC *out)`, with $out = x^T A$. There are also functions to perform matrix arithmetic — matrix addition `m_add()`, matrix-scalar multiplication `sm_mlt()`, matrix-matrix multiplication `m_mlt()`.

Several different sorts of matrix factorisation are supported: LU factorisation (also known as Gaussian elimination) with partial pivoting, by `LUfactor()` and `LUsolve()`. Other factorisation methods include Cholesky factorisation `CHfactor()` and `CHsolve()`, and QR factorisation with column pivoting `QRCPfactor()`.

Pivoting involve *permutations* which have their own `PERM` data structure. Permutations can be created by `px_get()`, read and written by `px_input()` and `px_output()`, multiplied by `px_mlt()`, inverted by `px_inv()` and applied to vectors by `px_vec()`.

The above program can be put into a file `leastsq.c` and compiled under UnixTM using

```
cc -o leastsq leastsq.c meshach.a -lm
```

A sample session using `leastsq` follows:

```
% leastsq
Input A matrix:
Matrix: rows cols:5 3
row 0:
entry (0,0): 3
entry (0,1): -1
entry (0,2): 2
Continue:
row 1:
entry (1,0): 2
entry (1,1): -1
entry (1,2): 1
Continue: n
row 1:
entry (1,0): old          2 new: 2
entry (1,1): old          -1 new: -1
entry (1,2): old          1 new: 1.2
Continue:
row 2:
entry (2,0): old          0 new: 2.5
....
....          (Data entry)
....
# A =
Matrix: 5 by 3
row 0:          3          -1          2
row 1:          2          -1          1.2
row 2:          2.5         1          -1.5
row 3:          3          1          1
row 4:          -1         1          -2.2
```

```

Input b vector:
entry 0: old          0 new: 5
entry 1: old          0 new: 3
entry 2: old          0 new: 2
entry 3: old          0 new: 4
entry 4: old          0 new: 6
# b =
Vector: dim: 5
           5           3           2           4           6
Vector of best fit parameters is
Vector: dim: 3
      1.47241555      -0.402817858      -1.14411815
||A*x-b|| = 6.78938

```

The Q matrix can be obtained explicitly by the routine `makeQ()`. The Q matrix can then be used to obtain an orthogonal basis for the range of A . An orthogonal basis for the null space of A can be obtained by finding the QR-factorisation of A^T .

1.6 A sparse matrix example

To illustrate the sparse matrix routines, consider the problem of solving Poisson's equation on a square using finite differences, and incomplete Cholesky factorisation. The actual equations to solve are

$$u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{ij} = h^2 f(x_i, y_j), \quad \text{for } i, j = 1, \dots, N$$

where $u_{0,j} = u_{i,0} = u_{N+1,j} = u_{i,N+1} = 0$ for $i, j = 1, \dots, N$ and h is the common distance between grid points.

The first task is to set up the matrix describing this system of linear equations. The next is to set up the right-hand side. The third is to form the incomplete Cholesky factorisation of this matrix, and finally to use the sparse matrix conjugate gradient routine with the incomplete Cholesky factorisation as preconditioner.

Setting up the matrix and right-hand side can be done by the following code:

```

#define N 100
#define index(i,j) (N*((i)-1)+(j)-1)
.....
A = sp_get(N*N,N*N,5);
b = v_get(N*N);
h = 1.0/(N+1);      /* for a unit square */
.....

for ( i = 1; i <= N; i++ )
    for ( j = 1; j <= N; j++ )
        {

```

```

    if ( i < N )
        sp_set_val(A,index(i,j),index(i+1,j),-1.0);
    if ( i > 1 )
        sp_set_val(A,index(i,j),index(i-1,j),-1.0);
    if ( j < N )
        sp_set_val(A,index(i,j),index(i,j+1),-1.0);
    if ( j > 1 )
        sp_set_val(A,index(i,j),index(i,j-1),-1.0);
    sp_set_val(A,index(i,j),index(i,j),4.0);
    b->ve[index(i,j)] = -h*h*f(h*i,h*j);
}

```

Once the matrix and right-hand side are set up, the next task is to compute the sparse incomplete Cholesky factorisation of **A**. This must be done in a different matrix, so **A** must be copied.

```

LLT = sp_copy(A);
spICHfactor(LLT);

```

Now when that is done, the remainder is easy:

```

out = v_get(A->m);
.....
iter_spcg(A,LLT,b,1e-6,out,1000,&num_steps);
printf("Number of iterations = %d\n",num_steps);
.....

```

and the output can be used in whatever way desired.

For graphical output of the results, the solution vector can be copied into a square matrix, which is then saved in MATLAB™ format using `m_save()`, and graphical output can be produced by MATLAB™.

1.7 How do I?

For the convenience of the user, here a number of common tasks that people need to perform frequently, and how to perform the computations using Meschach.

1.7.1 solve a system of linear equations

If you wish to solve $Ax = b$ for x given A and b (without destroying A), then the following code will do this:

```
VEC      *x, *b;
MAT *A, *LU;
PERM *pivot;
.....
LU = m_get(A->m,A->n);
LU = m_copy(A,LU);
pivot = px_get(A->m);
LUfactor(LU,pivot);
/* set values of b here */
x = LUsolve(LU,pivot,b,VNULL);
```

1.7.2 solve a least-squares problem

To minimise $\|Ax - b\|_2^2 = \sum_i ((Ax)_i - b_i)^2$, the most reliable method is based on the QR-factorisation. The following code performs this calculation assuming that A is $m \times n$ with $m \geq n$:

```
MAT *A, *QR;
VEC *diag, *b, *x;
.....
QR = m_get(A->m,A->n);
QR = m_copy(A,QR);
diag = v_get(A->n);
QRfactor(QR,diag);
/* set values of b here */
x = QRsolve(QR,diag,b,x);
```

1.7.3 find all the eigenvalues (and eigenvectors) of a general matrix

The best method is based on the *Schur decomposition*. For symmetric matrices, the eigenvalues and eigenvectors can be computed by a single call to `symmeig()`. For non-symmetric matrices, the situation is more complex and the problem of finding eigenvalues and eigenvectors can become quite ill-conditioned. Provided the problem is not too ill-conditioned, the following code should give accurate results:

```

/* A is the matrix whose e-vals and e-vecs are sought */
MAT *A, *T, *Q, *X_re, *X_im;
VEC *evals_re, *evals_im;
.....
Q = m_get(A->m,A->n);
T = m_copy(A,MNULL);
/* compute Schur form: A = Q.T.Q^T */
schur(T,Q);
/* extract eigenvalues */
evals_re = v_get(A->m);
evals_im = v_get(A->m);
schur_evals(T,evals_re,evals_im);
/* Q not needed for eigenvalues */
X_re = m_get(A->m,A->n);
X_im = m_get(A->m,A->n);
schur_vecs(T,Q,X_re,X_im);
/* k'th eigenvector is k'th column of (X_re + i*X_im) */

```

1.7.4 solve a large, sparse, positive definite system of equations

An example of a large, sparse, positive definite matrix is the matrix obtained from a finite-difference approximation of the Laplacian operator. If an explicit representation of such a matrix is available, then the following code is suggested as a reasonable way of computing solutions:

```

/* A.x == b is the system to be solved */
sp_mat *A, *LLT;
VEC *x, *b;
int      num_steps;
.....
/* set up A and b */
.....
x = m_get(A->m);
LLT = sp_copy(A);
/* preconditioning using incomplete Cholesky */
spICHfactor(LLT);
/* now use pre-conditioned conjugate gradients */
x = iter_spcg(A,LLT,b,1e-7,x,1000,&num_steps);
/* solution computed with relative residual < 10^{-7} */

```

If explicitly storing such a matrix takes up too much memory, then if you can write a routine to perform the calculation of Ax for any given x , the following code may be more suitable (if slower):

```

VEC *mult_routine(user_def,x,out)

```

```

void   *user_def;
VEC *x, *out;
{
    /* compute out = A*x */
    .....
    return out;
}

main()
{
    ITER *ip;
    VEC *x, *b;
    .....
    b = v_get(BIG_DIM); /* right-hand side */
    x = v_get(BIG_DIM); /* solution */

    /* set up b */
    .....
    ip = iter_get(b->dim, x->dim);
    ip->rhs = v_copy(b, ip->rhs);
    ip->info = NULL; /* if you don't want information
                    about solution process */
    v_zero(ip->x); /* initial guess is zero */
    iter_Ax(ip, mult_routine, user_def);
    iter_cg(ip);
    printf("# Solution is:\n"); v_output(ip->x);
    .....
    ITER_FREE(ip); /* destroy ip */
}

```

The `user_def` argument is for a pointer to a user-defined structure (possibly NULL, if you don't need this) so that you can write a common function for handling a large number of different circumstances.