Proceedings of

THE CENTRE FOR MATHEMATICS AND ITS APPLICATIONS

THE AUSTRALIAN NATIONAL UNIVERSITY

Volume 32, 1994

Meschach:

Matrix Computations in C

by
David E. Stewart and Zbigniew Leyk

Proceedings of

THE CENTRE FOR MATHEMATICS
AND ITS APPLICATIONS

THE AUSTRALIAN NATIONAL UNIVERSITY

Volume 32, 1994

Meschach:

Matrix Computations in C

by
David E. Stewart and Zbigniew Leyk

First published in Australia 1994

AMS classification: 65–04, 65F.

National Library of Australia Cataloging–in–Publication.

Stewart, David E. (David Edward), 1961—.
Meschach: matrix computations in C

Includes index.
ISBN 0 7315 1900 0.

1. Meschach (Computer file). 2. Matrices — Data processing. 3. Computer algorithms. 4. C (Computer program language). I. Leyk, Zbigniew, 1955—. II. Australian National University. Centre for Mathematics and its Applications. III. Title. (Series: Proceedings of the Centre for Mathematics and its Applications, Australian National University; v. 32).

512.94340285

This book was typeset using $\mathcal{AMS}$-LaTeX and the Adobe PostScript fonts.

Meschach:

Matrix Computations in C

Version 1.2

$$PA = LU$$
$$solve\ Ax = b$$

```
LUfactor(A,pivot);
LUsolve(A,pivot,b,x);
```

David E. Stewart and Zbigniew Leyk

Unix is a trademark of AT&T
MATLAB is a trademark of The MathWorks Inc.
MATCALC is a trademark of the University of New South Wales
MS-DOS and Quick C are trademarks of MicroSoft Corp.
SUN and SPARC are trademarks of Sun Microsystems Inc.
Pyramid is a trademark of Pyramid Computers
IBM RT, IBM RS/6000 and IBM PC are trademarks of IBM
8086 and i860 are trademarks of Intel
68000 is a trademark of Motorola
Weitek is a trademark of Weitek Inc.

" . . . and they walked in the heart of the flames . . . "

Daniel 3

Preface

Most of numerical analysis relies on algorithms for performing calculations on matrices and vectors. The operations most needed are ones which solve systems of linear equations, solve least squares problems, and eigenvalue and eigenvector calculations. These operations form the basis of most algorithms for solving systems of nonlinear equations, numerically computing the maximum or minimum of a function, or solving differential equations.

The Meschach library contains routines to address all of the basic operations for dealing with matrices and vectors, and a number of other issues as well. I do not claim that it contains every useful algorithm in numerical linear algebra, but it does provide a basis on which to build more advanced algorithms. The library is intended for people who know something of the 'C' programming language, something of how to solve the numerical problem they are faced with (which involves matrices and/or vectors) but don't want to have the hassle of building all the necessary operations from the ground up. I hope that researchers, mathematicians, engineers and programmers will find this library makes the task of developing and producing code for their numerical problems easier, and easier to maintain than would otherwise be possible.

To this end the source code is available to be perused, used and passed on without cost, while ensuring that the quality of the software is not compromised. The software *is* copyrighted; however, the copyright agreement follows in the footsteps of the Free Software Foundation in preventing abuse that occurs with totally "public domain" software.

This is not the first or only library of numerical routines in C. However, there are still a number of niches which have not been filled. Some of the currently available libraries are essentially translations of Fortran routines into C. Those that attempt to make use of C's features usually address a relatively small class of problems. There is a commercial package of C++ routines (and classes) for performing matrix computations, and NAG and IMSL are producing C versions of their libraries. None of these is "public domain".

The Meschach library makes extensive use of C's special features (pointers, memory allocation/deallocation, structures/records, low level operations) to ease use and ensure good performance. In addition, Meschach addresses the need for both dense and sparse matrix operations within a single framework.

There is another issue which needs to be addressed by a matrix library like this. At one end, libraries that are essentially translations from Fortran will make little use of memory allocation. At the other end, interactive matrix "calculators" such as MATLAB and MATCALC use memory allocation and garbage collection as a matter of course and have to interpret your "program". This latter approach is very flexible, but resource hungry. These matrix calculator programs were not designed to deal with large problems.

This matrix library is intended to provide a "middle ground" between efficient but inflexible Fortran-style programs, and flexible but resource hungry calculator/interpreter programs. When and how memory is allocated in Meschach can be controlled by us-

ing the allocation/deallocation and resizing routines; result matrices and vectors can be created dynamically when needed, or allocated once, and then used as a static array. Unnecessary memory allocation is avoided where necessary. This means that prototyping can often be done on MATLAB or MATCALC, and final code can be written that is efficient and can be incorporated into other C programs and routines without having to re-write all the basic routines from scratch.

This documentation describes Meschach 1.2 which has a number of improvements over previous versions of Meschach. Amongst these improvements are:

- easier installation (at least on Unix machines).
- complex numbers, vectors and matrices, including complex matrix factorisation.
- band matrix structures, and band factorise and solve routines.
- better control of static workspace arrays.
- more iterative methods for large, sparse or structured matrices, and a comprehensive "iteration" data structure.
- more consistent naming schemes.
- matrix polynomials and exponentials.
- extensible error handling.

Finally, we would like to thank all those at the University of Queensland Mathematics Department, at Opcom, and at the Australian National University for their interest in and comments on this matrix library. In particular, we would like to thank Martin Sharry, Michael Forbes, Phil Kilby, John Holt, Phil Pollett and Tony Watts at the University of Queensland, and Mike Osborne, Teresa Leyk at the Australian National University and Karen George from the University of Canberra. Email has become significant part of work, and many people have pointed out bugs, inconsistencies and improvements to Meschach by email. These people include Ajay Shah of the University of Southern California, Dov Grobgeld of the Weizmann Institute, John Edstrom of the University of Calgary, Eric Grosse, one of the netlib organisers, Ole Saether of somewhere in Norway, Alfred Thiele and Pierre Asselin of Carnegie-Mellon Univeristy, Daniel Polani of the University of Mainz, Marian Slodicka of Slovakia, Kaifu Wu of Pomona, Hidetoshi Shimodaira of the University of Tokyo, Eng Siong of Edinburgh, Hirokawa Rui of the University of Tokyo, Marko Slyz of the University of Michigan, and Brook Milligan of the University of Texas. This list is only partial, and there are many others who have corresponded with me on details about Meschach and the like. Finally my thanks go to all those that have had to struggle with compilers and other things to get Meschach to work.

David E. Stewart & Zbigniew Leyk, Canberra, Australia, 1993

Meschach

Matrix Computations in C

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 "`foutpout`" 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

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

for $x(t_i)$, $i = 1, 2, 3, \ldots$ 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

$$\begin{aligned} v_1 &= f(t_i, x_i) \\ v_2 &= f(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hv_1) \\ v_3 &= f(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hv_2) \\ v_4 &= f(t_i + h, x_i + hv_3) \end{aligned}$$

(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 $v_1, \ldots, v_4$ 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 Unix^{TM}, if it is in a file `tutorial.c` is:

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

or, if you have an ANSI compiler,

```
cc -DANSI_C -o tutorial tutorial.c meschach.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 `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 `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 vaariants that take a variable number of arguments. These are the routines `..get_vars()`, `..._resize_vars()` and `..._free_vars()`. These correspond to the 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 combinartion. 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

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

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

$$\|Ax - b\|_2 = \|Rx - Q^T b\|_2 = \| \begin{bmatrix} R_1 \\ O \end{bmatrix} x - \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} b \|_2 \tag{1.2}$$

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 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 Unix^{TM} using

```
cc -o leastsq leastsq.c meschach.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

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

where $u_{0,j} = u_{i,0} = u_{N+1,j} = u_{i,N+1} = 0$ for $i, j = 1, \ldots, N$ 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 eiegenvalues */
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.

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.

Chapter 3

Numerical Linear Algebra

This chapter aims to provide a brief introduction to numerical linear algebra. People who are unfamiliar with how to go about (say) solving linear equations, or how to compute eigenvalues and eigenvectors might find this useful for selecting the best routine(s) to solve their particular problem, and to understand the rationale for the way the routines are set up in the way they are.

3.1 What numerical linear algebra is about

There are a number of core operations and tasks that make up numerical linear algebra. At the lowest level these include calculating linear combinations of vectors and inner products, and at the higher level consists of solving linear equations, solving least-squares problems and finding eigenvalues and eigenvectors.

The lower level operations are usually quite straightforward in terms of what they do and what the accuracy of the results are. However, with higher level operations more care must be taken with regard to both efficiency and the accuracy of the answers. The routines used to perform these higher level operations are more varied and allow a number of different ways of performing the same computation. The difference between them lies often in the speed (or lack of it) and the accuracy of the answers obtained.

There are further complications because of some intrinsic limits to the computations that a computer can do accurately, at least with floating point arithmetic. Floating point arithmetic cannot store numbers to an accuracy (relative to the number stored) better than what is called "*machine epsilon*", or "*unit roundoff*". This quantity is usually denoted by **u**, but is represented in the library by **MACHEPS**. It is also referred to in the ANSI C header file `<float.h>` as **DBL_EPSILON** for double precision and **FLT_EPSILON** for single precision. For most machines this quantity is about 2×10^{-16} for double precision, and 10^{-7} for single precision.

Practically all floating point calculations introduce errors of size of machine epsilon times the size of the quantities involved; for all intents and purposes, these errors are unavoidable. Perturbations in the *data* of a problem are essentially unavoidable. Algorithms that compute answers that would be exact for slightly perturbed data

are called *backward stable*; algorithms which give answers that are close to the exact answer are called *forward stable*. Sometimes the problems that are solved are inherently unstable, or "ill conditioned" (see below). In these circumstances, *no* algorithm can be expected to be *forward* stable. However, well designed algorithms are at least *backward* stable; the answers are exact for slightly perturbed data. The algorithms in Meschach are essentially all backward stable in this sense. Combining these algorithms in programs can sometimes lead to methods that are not stable in this sense. Careful analysis of the algorithm may need to be done to check this.

3.2 Complex conjugates and adjoints

Unlike real matrices, inner products of complex vectors have to involve complex conjugates:

$$\langle x, y \rangle = \sum_i \bar{x}_i y_i.$$

This cannot be written as $x^T y$, but is often written as $\bar{x}^T y$. The vector $\bar{x}^T$ not only is a row vector, but has the components replaced by their complex conjugates. (The complex number $z = u + iv$ has complex conjugate $\bar{z} = u - iv$ where u and v are real numbers.)

The vector $\bar{x}^T$ is called the *adjoint* of x and is denoted in this documentation as x^*. Some texts use this convention, others use related conventions.

There are also adjoints of matrices: $A^* = \bar{A}^T$. Generally, where one would use a transpose for real matrices, one should use an adjoint for complex matrices. Of course, if x is a real vector, and A is a real matrix, then $x^* = x^T$ and $A^* = A^T$.

While real orthogonal matrices satisfy $Q^T = Q^{-1}$, their complex cousins, the unitary matrices, satisfy $Q^* = Q^{-1}$.

3.3 Vector and matrix norms

While it is quite straightforward to talk about the magnitude of a number, it is less so with vectors and matrices as there are a number of different ways of defining it. These "magnitudes" or *norms* must have a number of basic properties in order to be of some use. These properties for vector norms are written out below; the norm itself is written as $\| \|$.

$$\begin{gathered} \|x\| \text{ is a non-negative real number} \\ \|x + y\| \leq \|x\| + \|y\| \\ \|\alpha x\| = |\alpha| \, \|x\| \qquad \text{where } \alpha \text{ is a real or complex number.} \end{gathered} \tag{3.1}$$

Matrix norms have not only these properties (with x and y replaced with matrices), but often have an additional one:

$$\|XY\| \leq \|X\| \, \|Y\|.$$

This inequality holds for all matrix norms implemented in Meschach.

Some standard vector norms are

$$\|x\|_1 = \sum_i |x_i|, \qquad \|x\|_\infty = \max_i |x_i|$$

$$\|x\|_2 = \left(\sum_i |x_i|^2 \right)^{1/2}.$$

(3.2)

The last norm ($\|\ \|_2$) is actually the standard or "Euclidean" norm and is the definition of "magnitude" used in geometry and mechanics etc. However, different problems often have natural ways of measuring vectors related to the specific problem. For example, if e is a vector of errors, then $\|e\|_\infty \leq .01$ means that no error is larger than .01.

These vector norms can be computed by the routines `v_norm1()`, `v_norm2()` and `v_norm_inf()`, for the $\|\ \|_1$ norm, the $\|\ \|_2$ norm and the $\|\ \|_\infty$ norm respectively.

Associated with these vector norms are matrix norms that are defined by

$$\|A\| = \max_{x \neq 0} \|Ax\| / \|x\|.$$

The associated matrix norms for the above vector norms are:

$$\|A\|_1 = \max_j \sum_i |a_{ij}|, \qquad \|A\|_\infty = \max_i \sum_j |a_{ij}|$$

$$\|A\|_2 = \left(\text{maximum eigenvalue of } A^T A\right)^{1/2}.$$

(3.3)

Some matrix norms are not associated with any particular vector norm, such as the *Frobenius* norm:

$$\|A\|_F = \left(\sum_{i,j} |a_{ij}|^2 \right)^{1/2}.$$

These matrix norms can be computed by the routines `m_norm1()` for the $\|\ \|_1$ norm, `m_norm_inf()` for the $\|\ \|_\infty$ norm, and `m_norm_frob()` for the Frobenius norm $\|\ \|_F$. The matrix 2-norm has not been implemented as it is a rather expensive operation. The matrix 2-norm is best computed using the SVD, which is discussed later.

3.4 "Ill conditioning" or intrinsically bad problems

Users of numerical routines sometimes find that the results they get are erratic or obviously wrong for some reason or other. Barring programming errors, there are some reasons why this can happen. Often it comes under the heading *ill conditioning*, which means that the problem is inherently difficult.

Whenever the computer does some calculation with real numbers (like 3.1415926 ...) it almost always adds some error to the result whose magnitude is about "machine epsilon" times the magnitude of the result. If such a change in the data can radically change the answer, then the problem or task is called "ill conditioned". This is a property of the problem, not of any algorithm to solve it.

As with most things in numerical analysis, it is a good idea to quantify "how badly conditioned". For the problem of solving linear systems of equations, the measure of conditioning for a particular norm $\| \ \|$ is

$$\kappa(A) = \|A\| \, \|A^{-1}\|$$

which is called the *condition number* of A. The condition numbers for the $\| \ \|_1$, $\| \ \|_2$ or $\| \ \|_\infty$ norms are usually denoted $\kappa_1(A)$, $\kappa_2(A)$ or $\kappa_\infty(A)$ respectively.

A justification of why this is used as a measure of the conditioning of a system of linear equations, is given in the following theorem:

Theorem 3.4.1 *If A is nonsingular and $\|A^{-1}\| \, \|E\| < 1$ and*

$$Ax = b, \quad \textit{and} \quad (A+E)(x+e) = b+f,$$

then

$$\frac{\|e\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \kappa(A)(\|E\|/\|A\|)} \left[\frac{\|E\|}{\|A\|} + \frac{\|f\|}{\|b\|} \right].$$

A proof of this may be found in a number of numerical analysis textbooks such as *Matrix Computations*, by Golub and van Loan, §2.7, pp. 79–80, 2nd Edition, (1989), or in *An Introduction to Numerical Analysis*, by K. Atkinson, Ch. 8, pp. 462–463, 1st Edition, (1979).

Do ill conditioned problems or tasks occur in practice? The answer is "All too often." One family of matrices that are notoriously ill-conditioned are the Hilbert matrices:

$$H_n = \begin{bmatrix} 1 & 1/2 & 1/3 & \dots & 1/n \\ 1/2 & 1/3 & 1/4 & \dots & 1/(n+1) \\ 1/3 & 1/4 & 1/5 & \dots & 1/(n+2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/n & 1/(n+1) & 1/(n+2) & \dots & 1/(2n-1) \end{bmatrix}.$$

These matrices arise quite naturally in finding best integral-least square error fits for functions in terms of 1, x, x^2, $\dots$, x^{n-1}. The condition number of H_n for $n = 5$ is already $\approx 4.8 \times 10^5$ and for $n = 10$ is $\approx 1.6 \times 10^{13}$. In fact the condition number of H_n for large n increases super-exponentially in n. Because they are so ill-conditioned, they are a favourite family of matrices to test linear equation solvers.

This condition number can be computed in $O(n^3)$ floating point operations essentially by calculating the inverse of the original matrix. Alternatively, it can be *estimated* relatively cheaply (in $O(n^2)$ operations) once the LU factors of the matrix are known. This can be done using the routine `LUcondest()`.

3.5 Least squares and pseudo-inverses

It is quite common, when analysing data, to perform a "least squares fit". For example, if there are three controlled quantities and one measured quantity in an experiment, it is common to fit a linear model:

$$y_i \approx \alpha_1 x_{i,1} + \alpha_2 x_{i,2} + \alpha_3 x_{i,3}$$

where each α_j is a parameter to be fitted, and y_i is the ith measured value, and $x_{i,j}$ is the ith value of the jth controlled quantity.

The "least squares fit" is the α vector that minimises

$$\sum_{i=1}^{m} (y_i - (\alpha_1 x_{i,1} + \alpha_2 x_{i,2} + \alpha_3 x_{i,3}))^2 .$$

This can be cast in terms of matrices and vectors by setting X to be the matrix of the $x_{i,j}$, and y to be the vector $[y_1, y_2, \ldots, y_m]^T$. Then the approximation is $y \approx X\alpha$, and more specifically, the least squares fit is obtained by minimising $\|y - X\alpha\|_2^2 = (y - X\alpha)^T(y - X\alpha)$. By taking partial derivatives with respect to the α_j's gives the system of linear equations known as the *normal equations*:

$$X^T X\alpha = X^T y.$$

If the columns of X are linearly independent, then the matrix $X^T X$ is positive definite and the Cholesky factorisation can be used to solve this system of equation once $X^T X$ is formed. The following piece of code does this:

```
MAT      *X, *XTX;
VEC      *y, *XTy, *alpha;
  ......
/* set up X and y */
  ......
XTX = mtrm_mlt(X,X,MNULL);
XTy = vm_mlt(X,y,VNULL);
CHfactor(XTX);
alpha = CHsolve(XTX,XTy,VNULL);
```

If the columns of X are *not* linearly independent, then there are redundant variables being set in the experiment: at least one of the variables being set is just a linear combination of the others. In the above piece of code, this may result in an error being raised to the effect that the matrix `XTX` is not positive definite. Whether this happens or not depends on the way that the rounding errors go.

In practice it may well be that some of the set quantities are *nearly*, but not exactly, redundant. The Cholesky factorisation may not be able to pick this up. However, there are other "factorisations" that can. These are the QR factorisation (with column pivoting) and the SVD. Later, we will return to the QR factorisation as another means of solving least squares problems.

3.5.1 Singular Value Decompositions

The SVD or Singular Value Decomposition is analogous in some ways to finding eigenvalues and eigenvectors. The SVD of a matrix X is a decomposition $X = U^T \Sigma V$ where U and V are orthogonal matrices, and Σ is a diagonal matrix. The values on the diagonal of Σ are unique, except for their sign. If the entries of Σ are all nonnegative and ordered so that they are nonincreasing going down the diagonal, then the diagonal entries are called *singular values*, and are denoted by σ_i. The columns of U and V are called *singular vectors*.

How well or ill conditioned a least squares problem is can be determined directly from the singular values. The usual condition number for least square problems is $\kappa_{LS}(X) = \sigma_1/\sigma_n$ where X is $m \times n$ and $m \geq n$. If $\sigma_n = 0$ then X has linearly dependent columns, and the problem cannot be solved to any degree of accuracy. Such a matrix is also referred to as being *rank deficient.*

3.5.2 Pseudo-inverses

Whether a matrix is square or rectangular, rank deficient or has full rank, it always has a *pseudo-inverse*. This is the matrix $X^+ = V^T \Sigma^+ U$ where the ith diagonal of Σ^+ is $1/\sigma_i$ is $\sigma_i \neq 0$ and zero otherwise. This has a number of useful properties such as the Moore–Penrose properties:

$$\begin{aligned} XX^+X &= X, & (XX^+)^T &= XX^+ \\ X^+XX^+ &= X^+, & (X^+X)^T &= X^+X. \end{aligned}$$

(3.4)

This means that XX^+ is an orthogonal projection onto range(X) and X^+X is an orthogonal projection onto range(X^T).

The least squares problem can, in general, be solved by setting $\alpha = X^+y$. This solution is, in fact, the *smallest* α that minimises the sum of errors squared. This approach appears quite simple for providing a way of solving least squares problems (and others) involving rank deficient matrices. However, there are a number of practical difficulties. The first of these is that small perturbations to rank deficient matrices usually result in full rank matrices; the σ_i's that were formerly zero before the perturbation, become nonzero, but small after the perturbation. This means that where Σ^+ had a zero on the diagonal before the perturbation, after the perturbation it has $1/\sigma_i$ which is quite large. In short, the pseudo-inverse is not a continuous function of the matrix entries; small perturbations can give very large changes in the results.

While the SVD can be computed numerically, roundoff error will ensure that almost always the computed σ_i's are all nonzero. In these cases it is important to *estimate* the rank by considering the size of the σ_i's. For such problems an error tolerance is needed to decide how small the σ_i's need to be before they are considered "too small". The choice of such an error tolerance should be based on the size of the errors in the matrix, and their source. If, for example, the values in the X matrix have a measurement error

of about 10^{-3}, then a tolerance of about 10 times this should detect near rank deficient matrices. If, on the other hand, the only errors are those from roundoff error, then a value of 100 times unit roundoff (`MACHEPS` in the library) should be adequate.

3.5.3 QR factorisations and least squares

An alternative approach to solving least squares problems for full rank matrices (i.e. those that are not rank deficient) is to use the QR factorisation. This method is also described in section 3 of the tutorial chapter. The QR factorisation of a matrix A is a factorisation $A = QR$ where Q is orthogonal and R is upper triangular.

This QR factorisation is computed by means of *Householder matrices*. These are discussed in more detail in the manual entry for the routines that implements these operations, `hhvec()`, `hhtrvec()`, `hhtrcols()` and `hhtrrows()`. The QR factorisation can also be computed by using *Givens' rotations* which are discussed in the manual entries for `givens()`, `rot_vec()`, `rot_cols()` and `rot_rows()`.

To use this factorisation to solve a linear least squares problem $X\alpha \approx y$ we compute, first, the QR factorisation of $X = QR$. For X $m \times n$ and $m > n$, as the R matrix is upper triangular,

$$R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}.$$

If X has full rank, then R_1 is a nonsingular $n \times n$ matrix. The matrix Q should be split in a consistent way: $Q = [Q_1, Q_2]$.

The residual vector's norm is then

$$\|X\alpha - y\|_2 = \|R\alpha - Q^T b\|_2 = \left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \alpha - \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} y \right\|_2.$$

This means that

$$\|X\alpha - y\|_2^2 = \|R_1\alpha - Q_1^T y\|_2^2 + \|Q_2^T y\|_2^2.$$

The minimum 2-norm of $X\alpha - y$ (with respect to α)is obtained by solving

$$R_1\alpha = Q_1^T y$$

and has the value $\|Q_2^T y\|_2$. The code in section 3 of the chapter 1 provides a complete program for solving least squares problems of this sort.

There are some advantages of this method over the "normal equations" approach, of which the main one is accuracy. In the normal equations approach, the system $X^T X\alpha = X^T y$ is solved for α. The error in the computed α in the 2-norm is of the order of $\mathbf{u}\kappa_2(X^T X)$. On the other hand, the error in the computed α for the QR factorisation method is of the order of $\mathbf{u}\kappa_{LS}(X)$. Now if $X = U^T \Sigma V$ is the SVD of X, then

$$X^T X = V^T \Sigma^T \Sigma V = V^T \operatorname{diag}(\sigma_1^2, \ldots, \sigma_n^2) V$$

and the eigenvalues of $X^T X$ are the squares of the singular values of X. So

$$\kappa_2(X^T X) = \|X^T X\|_2 \|(X^T X)^{-1}\|_2 = \sigma_1^2/\sigma_n^2 = \kappa_{LS}(X)^2$$

and forming $X^T X$ effectively squares the condition number of the problem. This is particularly important for badly conditioned problems with $\kappa_{LS}(X) \approx 1/\sqrt{\mathbf{u}}$; for such problems the QR factorisation method would work, but the normal equations approach would fail.

3.6 Eigenvalues and eigenvectors

There are two main classes of problems and algorithms for computing eigenvalues and eigenvectors. They are problems involving symmetric matrices, and problems involving nonsymmetric matrices. The case of symmetric matrices is easier both in theory and practice. It is also less vulnerable to the effects of roundoff errors.

Symmetric matrices all have real eigenvalues, and the corresponding eigenvectors are both real and orthogonal. Thus for any symmetric matrix A there is an orthogonal matrix Q such that $Q^T A Q = \Lambda$ where Λ is the diagonal matrix of eigenvalues. If the ith diagonal element of Λ is λ_i, and q_i is the ith column of Q, then $Aq_i = \lambda_i q_i$. Regarding stability of the eigenvalues to perturbations of the matrix A, the ith eigenvalue of $A + E$, denoted $\tilde{\lambda}_i$, satisfies $\lambda_i - \|E\|_2 \leq \tilde{\lambda}_i \leq \lambda_i + \|E\|_2$.

The eigenvectors are not so stable with respect to perturbations of A, especially if eigenvalues are close together. The extreme case is where there is a repeated eigenvalue, in which case the eigenvalues are not essentially unique (up to a scale factor). Instead, there is a two or three or higher dimensional subspace of eigenvectors. If all the eigenvalues are distinct, then for a matrix $A + E$, $\|E\|_2$ "small", the perturbation in the eigenvector q_i is of size roughly bounded by

$$\|E\|_2 \sqrt{\sum_{k \neq i} \frac{1}{(\lambda_k - \lambda_i)^2}}.$$

As for previous problems, the perturbations in A due to roundoff error is roughly $\|E\|_2 \approx \mathbf{u}\|A\|_2$. This means that the eigenvectors would not usually be reliably computed if its eigenvalue is no more than about $\mathbf{u}\|A\|_2$ from other eigenvalues.

The eigenvalues for a symmetric matrix can be computed using the `symmeig()` library routine, which will compute the Q matrix of eigenvectors as well as a vector containing the eigenvalues if desired.

For the nonsymmetric case, a rather different strategy has to be adopted for several reasons:

1. The matrix A may not be diagonalisable; the Jordan canonical form is not numerically stable.

2. The matrix of eigenvectors may not be well conditioned.

3. The eigenvalues may not be real.

The standard strategy used is to compute the *real Schur decomposition*. This is a variant of the complex Schur decomposition. The complex Schur decomposition is a

factorisation

$$Q^*AQ = T$$

where T is upper triangular, and Q is unitary; that is, $Q^*Q = I$ where Q^* is the adjoint of Q. The diagonal entries of T are the eigenvalues of A. The complex Schur decomposition can be computed for complex matrices by the routine `zschur()`.

For the real case,

$$Q^TAQ = T$$

where T is *block upper triangular* with 1×1 and 2×2 blocks on the diagonal and Q is orthogonal. The eigenvalues of the 1×1 and 2×2 diagonal blocks of T are the eigenvalues of A. This real Schur decomposition is computed by the `schur()` routine. If you wish to obtain the actual eigenvalues and eigenvectors, there are the auxiliary routines `schur_vals()` and `schur_vecs()`. The `schur_vals()` routine computes the (complex) eigenvalues and returns the real and imaginary parts of the eigenvalues. The `schur_vecs()` routine computes the eigenvectors of a matrix by means of its real Schur decomposition, by using one cycle of inverse iteration for each eigenvector. That is, the system

$$(T - \lambda I)\hat{x} = r$$

is solved for $\hat{x}$ where r is a random real vector.

Unfortunately, if there are repeated eigenvalues, this method cannot be expected to give good results: the matrix of eigenvectors would be ill-conditioned. Indeed, it is usually not possible to get a nonsingular matrix of eigenvectors if there are repeated eigenvalues. Consider the general 2×2 matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

This matrix has repeated eigenvalues if and only if $(a - d)^2 = -4bc$. The repeated eigenvalue is $(a + d)/2$. If X is the matrix of eigenvectors, and is nonsingular,then

$$X^{-1} \begin{bmatrix} a & b \\ c & d \end{bmatrix} X = (a + d)/2\, I$$

which implies that

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = (a + d)/2\, I$$

and $a = d$ and $b = c = 0$. Clearly, small perturbations of matrices with repeated eigenvalues usually result in matrices which do not have a nonsingular matrix of eigenvectors.

The proper way to handle the situation of repeated eigenvalues is either to use the Schur decomposition (real or complex), or to use the *Jordan Normal form*. The Jordan

Normal form of the matrix A has the form

$$X^{-1}AX = \begin{bmatrix} J_1 & 0 & 0 & \dots & 0 \\ 0 & J_2 & 0 & \dots & 0 \\ 0 & 0 & J_3 & \dots & 0 \\ 0 & 0 & 0 & \dots & J_l \end{bmatrix}$$

where each J_i (called a *Jordan block*) has the form

$$J_i = \begin{bmatrix} \lambda_i & 1 & 0 & \dots & 0 \\ 0 & \lambda_i & 1 & \dots & 0 \\ 0 & 0 & \lambda_i & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & & \lambda_i \end{bmatrix}.$$

Note that J_i may be as small as 1×1 or 2×2.

This form is not favoured by numerical analysts as it is difficult to compute when roundoff errors are present, and the criterion for deciding how big a Jordan block should be is a difficult task as it requires numerically estimating the rank of a number of matrices. Golub and van Loan's *Matrix Computations* discusses the difficulties of computing the Jordan Normal form pp. 390–392 (2nd Edition, 1989). Also, the Schur form can be used for almost all the same purposes as the Jordan Normal form, such as computing matrix exponentials.

3.7 Sparse matrix operations

Sparse matrices are simply matrices where most of the entries are zero. These are important as they can be stored in a more compact way by storing only the nonzero entries and their position in the matrix. The zero entries can usually be ignored for most computations. Thus far larger problems can be dealt with, and more quickly, than if array storage is used.

While the previous discussion holds for all matrices whether sparse or not, if sparse matrices are to be used effectively then their sparsity needs to be preserved. This quickly rules out a lot of algorithms which work well for matrices that are not sparse (i.e. *dense*). For example, the Schur decomposition and explicit matrix inverses usually result in intermediate and result matrices where most of the entries are nonzero.

Sparse matrices have a *structure* that dense matrices don't. This is essentially the set of (i, j) entries of a matrix that are nonzero, or at least that have memory allocated for a value. And it is often important to keep this structure and to prevent the number of nonzeros in intermediate matrices from increasing too quickly. The introduction of nonzero entries into sparse matrices is called *fill-in*. Not only does fill-in result in more space required to store the intermediate matrices and result indirectly in more floating point computations, but it also requires some sort of dynamic memory management. (This is easier in 'C' than in Fortran, but still has a cost in both time and memory

space.) The routines provided for manipulating sparse matrix data structures hides much of the complexity of the data structures and operations that need to be performed when there is fill-in.

Sparse matrices are also important as they are often more suitable for iterative rather than the direct methods that have been discussed so far. Often some mix of iterative and direct methods will provide the best performance for solving some large problems.

The direct routines implemented for sparse matrices include sparse Cholesky and sparse LU factorisation, with a number of variants which are provided for control the "structure" of the sparse factorisations. The iterative methods for solving systems of linear equations include pre-conditioned conjugate gradients for solving symmetric, positive definite systems, the CGS method of Sonneveld, the GMRES method of Saad and Schultz, the MGCR method of Leyk for solving systems of non-symmetric matrices, and the LSQR method of Paige and Saunders for non-square least squares problems. For eigenvalues, the Lanczos method is provided for symmetric matrices, and the Arnoldi method for nonsymmetric matrices.

Those who are familiar with the standard "classical" iterative methods (Gauss–Jacobi, Gauss–Seidel and Successive Over-Relaxation etc.) may be disappointed that they are not implemented. There are three reasons for this. The first is that the iterative routines that have been implemented do not require an explicit representation of the matrix; all that is needed is a way of forming Ax for any vector x. That is, only a *functional representation* of the matrix (A) is needed. The second is the difficulty in obtaining good convergence with the classical methods. These classical methods require good estimates of convergence rates and the like, and are difficult to turn into general purpose routines when the "rate estimation code" is included. The third is that, for instance, conjugate gradients (without pre-conditioning) give the same order of convergence as that for SOR with the optimum over-relaxation parameter for standard test problems. It therefore appears that there is not a great deal of reason to implement SOR over conjugate gradient methods, although conjugate gradient methods can be modified to use an SSOR-based pre-conditioner M:

$$M = (D + \omega L)D^{-1}(D + \omega L)^T$$

where D is the diagonal part of A, and L is the strictly lower triangular part of A and ω is the (over)relaxation parameter. Solving $Mz = w$ for z can be done essentially by backward and forward substitution and can be easily programmed without explicitly forming M. The Gauss–Seidel pre-conditioner is obtained by setting $\omega = 1$.

The crucial point about iterative methods is that there is usually no natural limit to the number of iterations. A relative precision for the residual must usually be specified, and it needs to be significantly larger than $\mathbf{u}$ (or, as it is represented in the library `MACHEPS`). The number of iterations is also important for the speed with which a system of linear equations is solved. If the relative error tolerance is set to ϵ, then the number of iterations is roughly proportional to $\sqrt{\kappa_2(A)}\ln(1/\epsilon)$ for conjugate gradient methods. For LSQR, it is roughly proportional to $\kappa_{LS}(A)\ln(1/\epsilon)$. For finding eigenvalues of symmetric matrices, the Lanczos routine finds the bottom eigenvalue to an accuracy

of ϵ in time roughly proportional to $\sqrt{(\lambda_k - \lambda_2)/(\lambda_2 - \lambda_1)}\ln(n(\lambda_k - \lambda_1)/\epsilon)$ where $\lambda_1 < \lambda_2 < \cdots < \lambda_k$ are the distinct eigenvalues of A. (i.e. λ_k is the largest eigenvalue of A.)

The use of functional representation also opens up the possibility of pre-conditioning for the CGS and LSQR, and even the Lanczos methods. Here incomplete factorisations may be able to improve performance, such as the incomplete Cholesky factorisation or the incomplete/modified LU factorisation.

Chapter 4

Basic Dense Matrix Operations

The following routines are described in the following pages:

Continued ...

To use these routines use the include statement

```
#include "matrix.h"
```

To use the complex variants use the include statement

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

NAME

catch, catchall, catch_FPE, tracecatch – catch errors

SYNOPSIS

```
#include "matrix.h"
catch(int err_num, normal_code_to_execute,
        code_to_execute_if_error)
catchall(normal_code_to_execute,
        code_to_exectue_if_error)
tracecatch(normal_code_to_execute, char *fn_name)
catch_FPE()
```

DESCRIPTION

The **catch()** macro provides a way of interposing your own error-handling routines and code in the usual error-handling procedures. The **catch()** macro works like this: The global variable **restart** (of type **jmp_buf**) is saved. Then the code **normal_code_to_execute** is executed. If an error with error number **err_num** is raised, then **code_to_execute_if_error** is executed. If an error with another error number is raised, an error will be raised with the same error number as the original error, but will appear to have come from the **catch()** macro. If no error is raised then the macro will exit and **restart** is reset to its old values.

The **catchall()** macro works just like the **catch()** macro except that **code_to_execute_if_error** is executed if *any* error is raised.

The **tracecatch()** macro is really a specialised version of the **catchall()** macro that sets the error-handling flag to print out the underlying error when it is raised.

In every case the old error handling status will be restored on exiting the macro.

The routine **catch_FPE()** sets up a signal handler so that if a **SIGFPE** signal is raised, it is caught and **error()** is called as appropriate. The error raised by **error()** is an **E_SIGNAL** error.

EXAMPLE

```
main()
{
    MAT  *A;
    PERM *pivot;
    VEC  *x, *b;
      ......
    tracecatch(
        LUfactor(A,pivot);
        LUsolve(A,pivot,b,x);
        , "main");
```

```
......
```

would result in the error messages

```
"lufactor.c", line 28: NULL objects passed in function
            LUfactor()
"junk.c", line 20: NULL objects passed in function main()
Sorry, exiting program
```

being printed to `stdout` if one of `A` or `pivot` or `b` were `NULL`. These messages would also be printed out to `stderr` if `stdout` is not a terminal.

On the other hand,

```
catch(E_NULL,
            LUfactor(A,pi);
            LUsolve(A,pi,b,x);
   , printf("Ooops, found a NULL object\n"));
```

simply produces the message `Ooops, found a NULL object` in this case.

However, if another error occurs (say, `b` is the wrong size) then `LUsolve()` raises an `e_SIZES` error, and

```
"junk.c", line 22: sizes of objects don't match in
            function catch()
Sorry, exiting program
```

is printed out.

SEE ALSO

`signal()`, `error()`, `set_err_flag()`, `ERREXIT()` etc.

BUGS

If a different error to the one caught in `catch()` is raised, then the file and line numbers of the original error are lost.

In an if–then–else statement, `tracecatch()` needs to be enclosed by braces (`{...}`).

SOURCE FILE: `matrix.h`

NAME

error, set_err_flag, ev_err, err_list_attach, err_is_list_attached, err_list_free, warning – raise errors and warnings

SYNOPSIS

```
#include   "matrix.h"
int    error(int err_num, char *func_name)
int    ev_err(char *file, int err_num, int line_num,
              char *fn_name, int list_num)
int    set_err_flag(int new_flag)
int    err_list_attach(int list_num, int list_len,
                       char **err_ptr, int warn)
int    err_list_free(int list_num)
int    err_is_list_attached(int list_num)
int    warning(int warn_num, char *func_name)
```

DESCRIPTION

This is where errors are flagged in the system. The call **error(err_num,func_name)** is in fact a macro which expands to

```
ev_err(__FILE__,err_num,__LINE__,func_name,0)
```

This call does not return.

Warnings are raised by **warning(warn_num,func_name)** which are expands to

```
ev_err(__FILE__,warn_num,__LINE__,func_name,1)
```

This call returns zero.

The call to **ev_err()** prints out a message to **stderr** indicating that an error has occurred, and where in which function it occurred, and the list of error messages to use (0 is the default). For example, it could look like:

```
"test1.c", line 79: sizes of objects don't match in
                    function f()
```

which indicates that an error was flagged in file "**test1.c**" at line 79, function "**f**" where the sizes of two objects (vectors in this case) were incompatible.

Once this information is printed out, control is passed to the the address saved in the buffer called **restart** by the last associated call to **setjmp**. The most convenient way of setting up **restart** is to use a **...catch...()** macro or by an **ERREXIT()** or **ERRABORT()** macro. If **restart** has not been set then the program exits.

If you wish to do something particular if a certain error occurs, then you could include a code fragment into `main()` such as the following:

```
if ( (code=setjmp(restart)) != 0 )
{
    if ( code = E_MEM ) /* memory error, say */
        /* something particular */
        { .... }
    else
        exit(0);
}
else
    /* make sure that error handler does jump */
    set_err_flag(EF_JUMP);
```

The list of standard error numbers is given below:

```
E_UNKNOWN       0 /* unknown error (unused) */
E_SIZES         1 /* incompatible sizes */
E_BOUNDS        2 /* index out of bounds */
E_MEM           3 /* memory (de)allocation error */
E_SING          4 /* singular matrix */
E_POSDEF        5 /* matrix not positive definite */
E_FORMAT        6 /* incorrect format input */
E_INPUT         7 /* bad input file/device */
E_NULL          8 /* NULL object passed */
E_SQUARE        9 /* matrix not square */
E_RANGE        10 /* object out of range */
E_INSITU2      11 /* only in-situ for square matrices */
E_INSITU       12 /* can't do operation in-situ */
E_ITER         13 /* too many iterations */
E_CONV         14 /* convergence criterion failed */
E_START        15 /* bad starting value */
E_SIGNAL       16 /* floating exception */
E_INTERN       17 /* some internal error */
E_EOF          18 /* unexpected end-of-file */
E_SHARED_VECS  19 /* cannot release shared vectors */
E_NEG          20 /* negative argument */
E_OVERWRITE    21 /* cannot overwrite object */
```

The `set_err_flag()` routine sets a flag which controls the behaviour of the error handling routine. The old value of this flag is returned, so that it can be restored if necessary.

The list of values of this flag are given below:

```
EF_EXIT   0  /* exit on error  -- default */
EF_ABORT  1  /* abort on error -- dump core */
EF_JUMP   2  /* do longjmp()   -- see above code */
EF_SILENT 3  /* do not report error, but do longjmp() */
```

If there is a just a warning, then the default behaviour is to print out a message to `stdout`, and possibly `stderr`; the only value of the flag which has any effect is `EF_SILENT`. This suppresses the printing.

The set of error messages, and the set of errors, can be expanded on demand by the user by means of `err_list_attach(list_num, list_len, err_ptr, warn)`. The list number `list_num` should be greater than one (as numbers zero and one are taken by the standard system). The parameter `list_len` is the number of errors and error messages. The parameter `err_ptr` is an array of `list_len` strings. The parameter `warn` is `TRUE` or `FALSE` depending on whether this class of "errors" should be regarded as being just warnings, or whether they are (potentially) fatal. Then when an "error" should be raised, call

```
ev_err(__FILE__,err_num,__LINE__,func_name,list_num);
```

It may well be worthwhile to write a macro such as:

```
#define my_error(my_err_num,func_name) \
     ev_err(__FILE__,err_num,__LINE__,func_name,list_num)
```

If when originally set, the `warn` parameter was `TRUE`, then these calls behave similarly to `warning()`, and if it was `FALSE`, then these calls behave similarly to `error()`. These errors and exceptions are controlled using `catch()`, `catchall()` and `tracecatch()` (if `warn` was `FALSE`), just as for `error()` calls.

The call `err_list_free(list_num)` unattaches the error list numbered `list_num`, and allows it to be re-used.

The call `err_is_list_attached(list_num)` returns `TRUE` if error list `list_num` is attached, and `FALSE` otherwise. This can be used to find the next available free list.

EXAMPLE

Use of `error()` and `warning()`:

```
if ( ! A )                  error(E_NULL,   "my_function");
if ( A->m != A->n )         error(E_SQUARE,"my_function");
if ( i < 0 || i >= A->m ) error(E_BOUNDS,"my_function");
/* this should never happen */
if ( panic && something_really_bad )
    error(E_INTERN,"my_function");
/* issue a warning -- can still continue */
warning(WARN_UNKNOWN,"my_function");
```

Use of `err_list_attach()`:

```
char *my_list[] = { "short circuit", "open circuit" };
int   my_list_num = 0;

main()
{
    for ( my_list_num = 0; ; my_list_num++ )
        if ( ! err_is_list_attached(my_list_num) )
            break;
    err_list_attach(my_list_num,2,my_list,FALSE);
      ......
    tracecatch(circuit_simulator(....),"main");
      ......
    err_list_free(my_list_num);
}

void circuit_simulator(....)
{
      ......
    /* open circuit error */
    ev_err(__FILE__,1,__LINE__,
           "circuit_simulator",my_list_num);
      .....
}
```

SEE ALSO

`ERREXIT()`, `ERRABORT()`, `setjmp()` and `longjmp()`.

BUGS

Not many routines use `tracecatch()`, so that the trace is far from complete. Debuggers are needed in this case, if only to obtain a backtrace.

SOURCE FILE: `err.c`

NAME

ERREXIT, ERRABORT, ON_ERROR – what to do on error

SYNOPSIS

```
#include    "matrix.h"
ERREXIT();
ERRABORT();
ON_ERROR();
```

DESCRIPTION

If **ERREXIT()** is called, then the program exits once the error occurs, and the error message is printed. **This is the default.**

If **ERRABORT()** is called, then the program aborts once the error occurs, and the error message is printed. Aborting in Unix systems means that a **core** file is dumped and can be analysed, for example, by (symbolic) debuggers. Behaviour on non-Unix systems is undefined.

If **ON_ERROR()** is called, the current place is set as the default return point if an error is raised, though this can be modified by the **catch()** macro. The **ON_ERROR()** call can be put at the beginning of a main program so that control always returns to the start. One way of using it is as follows:

```
main()
{
    ......
    ON_ERROR();
    printf("At start of program; restarts on error\n");
    /* initialisation stuff here */
    ......
    /* real work here */
    ......
}
```

This is a slightly dangerous way of doing things, but may be useful for implementing matrix calculator type programs.

Other, more sophisticated, things can be done with error handlers and error handling, though the topic is too advanced to be treated in detail here.

SEE ALSO

error() and **ev_err()**.

BUGS

With all of these routines, care must be taken not to use them inside called functions, unless the calling function immediately re-sets the **restart** buffer after the called

function returns. Otherwise the `restart` buffer will reference a point on the stack which will be overwritten by subsequent calculations and function calls. This is a problem inherent in the use of `setjmp()` and `longjmp()`. The only way around this problem is through the implementation of co-routines.

With `ON_ERROR()`, infinite loops can occur very easily.

SOURCE FILE: `matrix.h`

NAME

bd_copy, iv_copy, px_copy, m_copy, v_copy, zm_copy, zv_copy, m_move, v_move, zm_move, zv_move – copy objects

SYNOPSIS

```
#include  "matrix.h"
BAND   *bd_copy(BAND *in, BAND *out)
IVEC   *iv_copy(IVEC *in, IVEC *out)
MAT    *m_copy (MAT *in,  MAT *out)
MAT    *_m_copy(MAT *in,  MAT *out, int i0, int j0)
PERM   *px_copy(PERM *in, PERM *out)
VEC    *v_copy (VEC *in,  VEC *out)
VEC    *_v_copy(VEC *in,  VEC *out, int i0)
MAT    *m_move (MAT *in,  int i0, int j0, int m0, int n0,
                MAT *out, int i1, int j1)
VEC    *v_move (VEC *in,  int i0, int dim0,
                VEC *out, int i1)
VEC    *mv_move(MAT *in,  int i0, int j0, int m0, int n0,
                VEC *out, int i1)
MAT    *vm_move(VEC *in,  int i0,
                MAT *out, int i1, int j1, int m1, int n1)

#include  "zmatrix.h"
ZMAT   *zm_copy(ZMAT *in, ZMAT *out)
ZMAT   *_zm_copy(ZMAT *in, ZMAT *out, int i0, int j0)
ZVEC   *zv_copy(ZVEC *in, ZVEC *out)
ZVEC   *_zv_copy(ZVEC *in, ZVEC *out)
ZMAT   *zm_move (ZMAT *in,  int i0, int j0, int m0, int n0,
                 ZMAT *out, int i1, int j1)
ZVEC   *zv_move (ZVEC *in,  int i0, int dim0,
                 ZVEC *out, int i1)
ZVEC   *zmv_move(ZMAT *in,  int i0, int j0, int m0, int n0,
                 ZVEC *out, int i1)
ZMAT   *zvm_move(ZVEC *in,  int i0,
                 ZMAT *out, int i1, int j1, int m1, int n1)
```

DESCRIPTION

All the routines `bd_copy()`, `iv_copy()`, `m_copy()`, `px_copy()`, `v_copy()`, `zm_copy()` and `zv_copy()` copy all of the data from one data structure to another, creating a new object if necessary (i.e. a NULL object is passed or `out` is not sufficiently big), by means of a call to `bd_get()`, `iv_get()`, `m_get()`, `px_get()` or `v_get()` etc. as appropriate.

For `m_copy()`, `v_copy()`, `bd_copy()`, `iv_copy()`, `zm_copy()`, and `zv_copy()` if `in` is smaller than the object `out`, then it is copied into a region in `out` of the same size. If the sizes of the permutations differ in `px_copy()` then a new permutation is created and returned.

The "raw" copy routines are `_m_copy(in,out,i0,j0)` and `_v_copy(in,out,i0)`. Here `(i0,j0)` is the position where the $(0,0)$ element of the `in` matrix is copied to; `in` is copied into a block of `out`. Similarly, for `_v_copy()`, `i0` is the position of `out` where the zero element of `in` is copied to; `in` is copied to a block of components of `out`.

The `.._copy()` routines all work *in situ* with `in == out`, however, the `_.._copy()` routines will only work *in situ* if `i0` (and also `j0` if this is also passed) is (are) zero.

The complex routines `zm_copy(in,out)`, `zv_copy(in,out)`, and their "raw" versions `_zm_copy(in,out,i0,j0)` and `_zv_copy(int,out,i0)` operate entirely analogously to their real counterparts.

The routines `.._move()` move blocks between matrices and vectors. A source block in a matrix is identified by the matrix structure (`in`), the co-ordinates (`(i0,j0)`) of the top left corner of the block and the number of rows (`m0`) and columns (`n0`) of the block. The target block of a matrix is identified by `out` and the co-ordinates of the top left corner of the block (`(i1,j1)`), except in the case of moving a block from a vector to a matrix (`vm_move()`). In that case the number of rows and columns of the target need to be specified.

The source block of a vector is identified by the source vector (`in`), the starting index of the block (`i0`) and the dimension of the block (`dim0`). The target block of a vector is identified by the target vector `out` and the starting index (`i1`).

The routine `m_move()` moves blocks between matrices, `v_move()` moves blocks between vectors, `mv_move()` moves blocks from matrices to vectors (copying by rows), and `vm_move()` moves blocks from vectors to matrices (again copying by rows). The routine `zm_move()` moves blocks between complex matrices, `zv_move()` moves blocks between complex vectors, `zmv_move()` moves blocks from complex matrices to complex vectors (copying by rows), and `zvm_move()` moves blocks from complex vectors to complex matrices (again copying by rows).

EXAMPLE

```
/* copy x to y */
v_copy(x,y);
/* create a new vector z = x */
z = v_copy(x,VNULL);
/* copy A to the block in B with top-left corner (3,5) */
_m_copy(A,B,3,5);
/* an equivalent operation with m_move() */
m_move(A,0,0,A->m,A->n, B,3,5);
```

```
/* copy a matrix into a block in a vector ... */
mv_move(A,0,0,A->m,A->n, y,3);
/* ... and restore the matrix */
vm_move(y,3,A->m*A->n, A,0,0,A->m,A->n);
/* construct a block diagonal matrix C = diag(A,B) */
C = m_get(A->m+B->m,A->n+B->n);
m_move(A,0,0,A->m,A->n, C,0,   0);
m_move(B,0,0,B->m,B->n, C,A->m,A->n);
```

SEE ALSO

`..._get()` routines

SOURCE FILE: `copy.h, ivecop.c, zcopy.c, bdfactor.c`

NAME

iv_finput, m_finput, px_finput, v_finput, z_finput, zm_finput, zv_finput – input object from a file

SYNOPSIS

```
#include   <stdio.h>
#include   "matrix.h"
IVEC    *iv_finput(FILE *fp, IVEC *iv)
iv = iv_finput(fp,VNULL);

MAT     *m_finput(FILE *fp, MAT *A)
A = m_finput(fp,MNULL);

PERM    *px_finput(FILE *fp, PERM *pi)
pi = px_finput(fp,PXNULL);

VEC     *v_finput(FILE *fp, VEC *v)
v = v_finput(fp,VNULL);

complex  z_finput(FILE *)
z = z_finput(fp);

ZMAT    *zm_finput(FILE *fp, ZMAT *A)
A = zm_finput(fp,ZMNULL);

ZVEC    *zv_finput(FILE *fp, ZVEC *v)
v = zv_finput(fp,ZVNULL);
```

DESCRIPTION

These functions read in objects from the specified file. These functions first determine if `fp` is a file pointer for a "tty" (i.e. keyboard/terminal). There are also the macros `m_input(A)`, `px_input(pi)`, `v_input(x)`, `zm_input(A)`, `zv_input(x)`, and which are equivalent to `m_finput(stdin,A)`, `px_finput(stdin,pi)`, `v_finput(stdin,x)`, `zm_finput(stdin,A)`, and `zv_finput(stdin,x)` respectively. If so, then an interactive version of the input functions is called; if not, then a "file" version of the input functions is called.

The interactive input prompts the user for input for the various entries of an object; the file input simply reads input from the file (or pipe, or device etc.) and parses it as necessary. For complex numbers, the format is different between interactive and file input: interactive input has the format "x y" or just "x" for zero real part. File input of complex numbers uses `(x,y)`. For example, $-3.2 + 5.1i$ is entered as `-3.2 +5.1` in interactive mode, and as `(-3.2,5.1)` in file mode.

Note that the format for file input is essentially the same as the output produced by the `..._foutput()` and `..._output()` functions. This means that if the output is sent to a file or to a pipe, then it can be read in again without modification. Note also that for file input, that lines before the start of the data that begin with a "#" are treated as comments and ignored. For example, this might be the contents of a file **`my.dat`**:

```
# this is an example
# of a matrix input
Matrix: 3 by 4
row 0: 0   1   -2   -1
row 1:-2   0   1.5   2
row 2: 5  -4   0.5   0

# this is an example
# a vector input
Vector: dim: 4
 2     7     -1.372  3.4

# this is an example
# of a permutation input
Permutation: size: 4
 0->1 1->3 2->0 3->2

# this is a complex number
(3.765, -1.465324)
# this is a complex matrix
ComplexMatrix: 3 by 4
row 0:  (1,0)  (-2,0)  (3,0)  (-1,0)
row 1:  (5,3)  (-2,-3) (1,-4) (2,1)
row 2:  (1,0)  (2.5, 0)  ( 2.5, -3.56)  (2.5,0)
# and this is a complex vector...
ComplexVector: dim: 3
 (     -1.342235,          -1.342)  (2.3,-5)
 (             1,                1)
```

Interactive input is read line by line. This means that only one data item can be entered at a time. A user can also go backwards and forwards through a matrix or vector by entering "`b`" or "`f`" instead of entering data. Entering invalid data (such as hitting the return key) is not accepted; you must enter valid data before going on to the next entry. When permutations are entered, the value given is checked to see if lies within the acceptable range, and if that value had been given previously.

If the input routines are passed a NULL object, they create a new object of the size determined by the input. Otherwise, for interactive input, the size of the object passed must have the same size as the object being read, and the data is entered into the object

passed to the input routine. For file input, if the object passed to the input routine has a different size to that read in, a new object is created and data entered in it, which is then returned.

EXAMPLE

The above input file can be read in from **stdin** using:

```
complex z;
MAT  *A;
VEC  *b;
PERM *pi;
ZMAT *zA;
ZVEC *zv;
  ......
A  =  m_input(MNULL);
b  =  v_input(VNULL);
pi = px_input(PXNULL);
z  =  z_input();
zA = zm_input(ZMNULL);
zv = zv_input(ZVNULL);
```

If you know that a vector must have dimension m for interactive input, use:

```
b = v_get(m);
v_input(b);  /* use b's allocated memory */
```

SEE ALSO

.._output() entries, **.._input()** entries

BUGS

Memory can be lost forever; objects should be resize'd.

On end-of-file, an "unexpected end-of-file" error (**E_EOF**) is raised.

Note that the test for whether the input is an interactive device is made by **isatty(fileno(fp))**. This may not be portable to some systems.

Interactive complex input does not allow (x, y) format; nor does it allow entry of the imaginary part without the real part.

SOURCE FILE: **matrixio.c, zmatio.c**

NAME

iv_foutput, m_foutput, px_foutput, v_foutput, z_foutput, zm_foutput, zv_foutput, iv_dump, m_dump, px_dump, v_dump, zm_dump, zv_dump – output to a file or stream

SYNOPSIS

```
#include "matrix.h"
void     iv_foutput(FILE *fp, IVEC *v)
void      m_foutput(FILE *fp, MAT  *A)
void     px_foutput(FILE *fp, PERM *pi)
void      v_foutput(FILE *fp, VEC  *v)

#include "zmatrix.h"
void      z_foutput(FILE *fp, complex z)
void     zm_foutput(FILE *fp, ZMAT *A)
void     zv_foutput(FILE *fp, ZVEC *v)
```

DESCRIPTION

These output is a representation of the respective objects to the file (or device, or pipe etc.) designated by the file pointer `fp`. The format in which data is printed out is meant to be both human and machine readable; that is, there is sufficient information for people to understand what is printed out, and furthermore, the format can be read in by the `.._finput()` and `.._input()` routines.

An example of the format for matrices is given in the entry for the `.._finput()` routines.

There are also the routines `m_output(A)`, `px_output(pi)` and `v_output(x)` which are equivalent to `m_foutput(stdout,A)`, `px_foutput(stdout,pi)` and `v_foutput(stdout,x)` respectively.

Note that the `.._output()` routines are in fact just macros which translate into calls of these `.._foutput()` routines with "`fp = stdin`".

In addition there are a number of routines for dumping the data structures in their entirety for debugging purposes. These routines are `m_dump(fp,A)`, `px_dump(fp,px)`, `v_dump(fp,x)`, `zm_dump(fp,zA)` and `zv_dump(fp,zv)` where `fp` is a `FILE *`, `A` is a `MAT *`, `px` is a `PERM *` and `x` is a `VEC *`, `zA` is a `ZMAT *`, and `zv` is a `ZVEC *`. These print out pointers (as hex numbers), the maximum values of various quantities (such as `max_dim` for a vector), as well as all the quantities normally printed out. The output from these routines is not machine readable, and can be quite verbose.

EXAMPLE

```
/* output A to stdout */
```

```
m_output(A);
/* ...or to file junk.out */
if ( (fp = fopen("junk.out","w")) == NULL )
    error(E_EOF,"my_function");
m_foutput(fp,A);
/* ...but for debugging, you may need... */
m_dump(stdout,A);
```

SEE ALSO

`..._finput()`, `..._input()`

SOURCE FILE: `matrixio.c`, `zmatio.c`

NAME

finput, input, fprompter, prompter – general input/output routines

SYNOPSIS

```
#include <stdio.h>
#include "matrix.h"
int    finput(FILE *fp, char *prompt, char *fmt, void *var)
int    input(char *prompt, char *fmt, void *var)
int    fprompter(FILE *fp, char *prompt)
int    prompter(char *prompt)
```

DESCRIPTION

The macros **finput()** and **input()** are for general input, allowing for comments as accepted by the **.._finput()** routines. That is, if input is from a file, then comments (text following a '#' until the end of the line) are skipped, and if input is from a terminal, then the string **prompt** is printed to **stderr**. The input is read for the file/stream **fp** by **finput()** and by **stdin** by **input()**. The **fmt** argument is a string containing the **scanf()** format, and **var** is the argument expected by **scanf()** according to the format string **fmt**.

For example, to read in a file name of no more than 30 characters from **stdin**, use

```
char  fname[31];
  ......
input("Input file name: ","%30s",fname);
```

The macros **fprompter()** and **prompter()** send the **prompt** string to **stderr** if the input file/stream (**fp** in the case of **fprompter()**, **stdin** for **prompter()**) is a terminal; otherwise any comments are skipped over.

SEE ALSO

scanf(), .._finput()

SOURCE FILE: **matrix.h**

NAME

IV_FREE, M_FREE, PX_FREE, V_FREE, ZM_FREE, ZV_FREE, iv_free_vars, m_free_vars, px_free_vars, v_free_vars, zm_free_vars, zv_free_vars – destroy objects and free up memory

SYNOPSIS

```
#include "matrix.h"
void IV_FREE(IVEC *iv)
void M_FREE (MAT *A)
void PX_FREE(PERM *pi)
void V_FREE (VEC *v)
int  iv_free_vars(IVEC **iv1, IVEC **iv2, ..., NULL)
int   m_free_vars(MAT  **A1,  MAT  **A2,  ..., NULL)
int  px_free_vars(PERM **pi1, PERM **pi2, ..., NULL)
int   v_free_vars(VEC  **v1,  VEC  **v2,  ..., NULL)

#include "zmatrix.h"
void ZM_FREE(ZMAT *A)
void ZV_FREE(ZVEC *v)
int  zm_free_vars(ZMAT  **A1, ZMAT  **A2, ..., NULL)
int  zv_free_vars(ZVEC  **v1, ZVEC  **v2, ..., NULL)
```

DESCRIPTION

The `.._FREE()` routines are in fact all macros which result in calls to thje corresponding `.._free()` function, so that `IV_FREE(iv)` calls `iv_free(iv)`. The effect of calling `.._free()` is to release all the memory associated with the object passed. The effect of the macros `.._FREE(object)` is to firstly release all the memory associated with the object passed, and to then set `object` to have the value NULL. The reason for using macros is to avoid the "dangling pointer" problem.

The problems of dangling pointers cannot be entirely overcome within a conventional language, such as 'C', as the following code illustrates:

```
VEC     *x, *y;
  ....
x = v_get(10);
y = x;          /* y and x now point to the same place */
V_FREE(x);      /* x is now VNULL */
/* y now "dangles" -- using y can be dangerous */
y->ve[9] = 1.0; /* overwriting malloc area! */
V_FREE(y);      /* program will probably crash here! */
```

The `.._free_vars()` functions free a NULL-terminated list of pointers to variables all of the same type. Calling

```
..._free_vars(&x1,&x2,...,&xN,NULL)
```

is equivalent to

```
..._free(x1);   x1 = NULL;
..._free(x2);   x2 = NULL;
  ......
..._free(xN);   xN = NULL;
```

The returned value of the `..._free_vars()` routines is the number of objects freed.

SEE ALSO

`..._get()` routines

BUGS

Dangling pointer problem is neither entirely fixed, nor is it fixable.

SOURCE FILE: `memory.c, zmemory.c`

NAME

bd_get, iv_get, m_get, px_get, v_get, zm_get, zv_get, iv_get_vars, m_get_vars, px_get_vars, v_get_vars, zm_get_vars, zv_get_vars – create and initialise objects

SYNOPSIS

```
#include "matrix.h"
BAND    *bd_get(int lb, int ub, int n)
IVEC    *iv_get(unsigned dim)
MAT     * m_get(unsigned m, unsigned n)
PERM    *px_get(unsigned size)
VEC     * v_get(unsigned dim)
int     *iv_get_vars(unsigned dim,
                     IVEC **x1, IVEC **x2, ..., NULL)
int     * m_get_vars(unsigned m, unsigned n,
                     MAT **A1, MAT **A2, ..., NULL)
int     *px_get_vars(unsigned size,
                     PERM **px1, PERM **px2, ..., NULL)
int     * v_get_vars(unsigned dim,
                     VEC **x1, VEC **x2, ..., NULL)

#include "zmatrix.h"
ZMAT    *zm_get(unsigned m, unsigned n)
ZVEC    *zv_get(unsigned dim)
int     *zm_get_vars(unsigned m, unsigned n,
                     ZMAT **A1, ZMAT **A2, ..., NULL)
int     *zv_get_vars(unsigned dim,
                     ZVEC **x1, ZVEC **x2, ..., NULL)
```

DESCRIPTION

All these routines create and initialise data structures for the associated type of objects. Any extra memory needed is obtained from `malloc()` and its related routines.

Also note that *zero relative* indexing is used; that is, the vector `x` returned by `x = v_get(10)` can have indexes `x->ve[i]` for `i` equal to 0, 1, 2, ... , 9, *not* 1, 2, ... , 9, 10. This also applies for both the rows and columns of a matrix.

The `bd_get(lb,ub,n)` routine creates a band matrix of size $n \times n$ with a lower bandwidth of `lb` and an upper banwidth of `ub`. The `iv_get(dim)` routine creates an integer vector of dimension `dim`. Its entries are initialised to be zero. The `m_get(m, n)` routine creates a matrix of size `m` $\times$ `n`. That is, it has `m` rows and `n` columns. The matrix elements are all initialised to being zero. The `px_get(size)` routine creates and returns a permutation of size `size`. Its entries are initialised to being those of an identity permutation. Consistent with C's array index conventions, a permutation of the given `size` is a permutation on the set {0,1, ... ,`size`-1}. The

`v_get(dim)` routine creates and returns a vector of dimension `dim`. Its entries are all initialised to zero.

The `.._get_vars()` routines allocate and initialise a NULL-terminated list of pointers to variables, all of the same type. All of the variables are initialised to objects of the same size. Calling

```
.._get_vars([m,]n,&x1,&x2,...,&xN,NULL)
```

is equivalent to

```
x1 = .._get([m,]n);
x2 = .._get([m,]n);
  ......
xN = .._get([m,]n);
```

(Note that "`[m,]`" indicates that "`m,`" might or might not be present, depending on whether the data structure involved is a matrix or not.) The returned value of the `.._get_vars()` routines is the number of objects created.

EXAMPLE

```
MAT  *A;
  ......
/* allocate 10 x 15 matrix */
A = m_get(10,15);
```

SEE ALSO

`.._free()`, `.._FREE()`, and `.._resize()`.

BUGS

As dynamic memory allocation is used, and it is not possible to build garbage collection into C, memory can be lost. It is the programmer's responsibility to free allocated memory when it is no longer needed.

SOURCE FILE: `memory.c, zmemory.c, bdfactor.c`

NAME

get_col, get_row, zget_col, zget_row – extract columns or rows from matrices

SYNOPSIS

```
#include "matrix.h"
VEC     *get_col(MAT *A, int col_num, VEC *v)
VEC     *get_row(MAT *A, int row_num, VEC *v)

#include "zmatrix.h"
ZVEC    *zget_col(ZMAT *A, int col_num, ZVEC *v)
ZVEC    *zget_row(ZMAT *A, int row_num, ZVEC *v)
```

DESCRIPTION

These put the designated column or row of the matrix **A** and puts it into the vector **v**. If **v** is NULL or too small, then a new vector object is created and returned by **get_col()** and **get_row()**. Otherwise, **v** is filled with the necessary data and is then returned. If **v** is larger than necessary, then the additional entries of **v** are unchanged.

The complex routines operate exactly analogously to the real routines.

EXAMPLE

```
MAT  *A;
VEC  *row, *col;
int  row_num, col_num;
  ......
row = v_get(A->n);
col = v_get(A->m);
get_row(A, row_num, row);
get_col(A, col_num, col);
```

SEE ALSO

set_col(), **set_row()**, and **zset_col()**, **zset_row()**.

SOURCE FILE: **matop.c, zmatop.c**

NAME

m_ident, m_ones, v_ones, m_rand, v_rand, m_zero, v_zero, zm_rand, zv_rand, zm_zero, zv_zero, mrand, smrand, mrandlist – initialisation routines

SYNOPSIS

```
#include "matrix.h"
MAT     *m_ident(MAT *A)
MAT     *m_ones(MAT *A)
VEC     *v_ones(VEC *x)
MAT     *m_rand(MAT *A)
VEC     *v_rand(VEC *x)
MAT     *m_zero(MAT *A)
VEC     *v_zero(VEC *x)
Real    mrand()
void    smrand(int seed)
void    mrandlist(Real a[], int len)

#include "zmatrix.h"
ZMAT    *zm_rand(ZMAT *A)
ZVEC    *zv_rand(ZVEC *x)
ZMAT    *zm_zero(ZMAT *A)
ZVEC    *zv_zero(ZVEC *x)
```

DESCRIPTION

The routine **m_ident()** sets the matrix **A** to be the identity matrix. That is, the diagonal entries are set to 1, and the off-diagonal entries to 0.

The routines **m_ones()**, **v_ones()** fill **A** and **x** with ones.

The routines **v_rand()**, **m_rand()** and **zv_rand()**, **zm_rand()** fill **A** and **x** with random entries. For real vectors or matrices the entries are between zero and one as determined by the **mrand()** function. For complex vectors or matrices, the entries have both real and imaginary parts between zero and one as determined by the **mrand()** function.

The routines **m_zero()**, **v_zero()** and **zm_zero()**, **zv_zero()** fill **A** and **x** with zeros.

These routines will raise an **E_NULL** error if **A** is NULL.

The routine **mrand()** returns a pseudo-random number in the range $[0, 1)$ using an algorithm based on Knuth's lagged Fibonacci method in *Seminumerical Algorithms: The Art of Computer Programming*, vol. 2 §§3.2–3.3. The implementation is based on that in *Numerical Recipes in C*, pp. 212–213, §7.1. Note that the seeds for **mrand()** are initialised using **smrand()** with a fixed **seed**. Thus **mrand()** will produce the

same pseudo-random sequence (unless **smrand()** is called) in different runs, different programs, and but for differences in floating point systems, on different machines.

The routine **smrand()** allows the user to re-set the seed values based on a user-specified **seed**. Thus **mrand()** can produce a wide variety of reproducible pseudo-random numbers.

The routine **mrandlist()** fills an array with pseudo-random numbers using the same algorithm as **mrand()**, but is somewhat faster for reasonably long vectors.

EXAMPLE

Let $e = [1, 1, \ldots, 1]^T$.

```
MAT  *A;
ZMAT *zA;
VEC  *x;
ZVEC *zx;
PERM *pi;
  ......
m_zero(A);   /* A == zero matrix */
m_ident(A);     /* A == identity matrix */
m_ones(A);   /* A == e.e^T */
m_rand(A);   /* A[i][j] is random in interval [0,1) */
zm_rand(zA);/* zA[i][j] is random in [0,1) x [0,1) */
v_zero(x);   /* x == zero vector */
v_ones(x);   /* x == e */
v_rand(x);   /* x[i] is random in interval [0,1) */
zv_rand(zx);/* zx[i] is random in [0,1) x [0,1) */
```

BUGS

The routine **m_ident()** "works" even if **A** is not square.

There is also the observation of von Neumann, *Various techniques used in connection with random digits*, National Bureau of Standards (1951), p. 36:

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."

SOURCE FILE: **init.c, matop.c, zmatop.c, zmemory.c, zvecop.c**

NAME

in_prod, zin_prod – inner product

SYNOPSIS

```
#include "matrix.h"
double  in_prod(VEC *x, VEC *y)

#include "zmatrix.h"
complex zin_prod(ZVEC *x, ZVEC *y)
```

DESCRIPTION

The inner product $x^T y = \sum_i x_i y_i$ of **x** and **y** is returned by **in_prod()**. The complex inner product $\bar{x}^T y = \sum_i \bar{x}_i y_i$ of **x** and **y** is returned by **zin_prod()**. This will fail if **x** or **y** is NULL.

These are built on the "raw" inner product routines:

```
double  _in_prod (VEC *x,  VEC *y,  int i0)
complex _zin_prod(ZVEC *x, ZVEC *y, int i0, int conj)
```

which compute the inner products ignoring the first **i0** entries. For the routine **_zin_prod()** if the flag **conj** is **Z_CONJ** (or **TRUE**) then the entries in the **x** vector are conjugated and $\sum_{i \geq i0} \bar{x}_i y_i$ is returned; otherwise if **conj** is **Z_NOCONJ** (or **FALSE**) then $\sum_{i \geq i0} x_i y_i$ is returned.

EXAMPLE

```
VEC  *x, *y;
ZVEC *zx, *zy;
Real x_dot_y;
complex zx_do_zy;
  ......
x_dot_y = in_prod(x,y);
zx_dot_zy = zin_prod(zx,zy);
```

SEE ALSO

__ip__(), **__zip__()** and the core routines.

BUGS

The accumulation is not guaranteed to be done in a higher precision than **Real**, although the return type is **double**. To guarantee more than this, we would either need an explicit extended precision **long double** type or force the accumulation to be done in a single register. While this is in principle possible on IEEE standard hardware, the routines to ensure this are not standard, even for IEEE arithmetic.

SOURCE FILE: **vecop.c, zvecop.c**

NAME

iv_add, iv_sub – Integer vector operations

SYNOPSIS

```
#include "matrix.h"
IVEC      *iv_add(IVEC *iv1,IVEC *iv2, IVEC *out)
IVEC      *iv_sub(IVEC *iv1, IVEC *iv2, IVEC *out)
```

DESCRIPTION

The two arithmetic operations implemented for integer vectors are addition (**iv_add()**) and subtraction (**iv_sub()**). In each of these routines, **out** is resized to be of the correct size if it does not have the same dimension as **iv1** and **iv2**.

This dearth of operations is because it is envisaged that the main purpose for using integer vectors is to hold indexes or to represent combinatorial objects.

EXAMPLE

```
IVEC *x, *y, *z;
  ......
x = ...;
y = ...;
/* z = x+y, allocate z */
z = iv_add(x,y,IVNULL);
/* z = x-y, z already allocated */
iv_sub(x,y,z);
```

SEE ALSO

Vector operations **v_...()** and **iv_resize()**.

SOURCE FILE: **ivecop.c**

NAME

bd_resize, iv_resize, m_resize, px_resize, v_resize, zm_resize, zv_resize, iv_resize_vars, m_resize_vars, px_resize_vars, v_resize_vars, zm_resize_vars, zv_resize_vars – Resizing data structures

SYNOPSIS

```
#include "matrix.h"
BAND   *bd_resize(BAND *A,
                  int new_lb, int new_ub, int new_n)
IVEC   *iv_resize(IVEC *iv, int new_dim)
MAT    *m_resize (MAT *A, int new_m, int new_n)
PERM   *px_resize(PERM *px, int new_size)
VEC    *v_resize (VEC *x, int new_dim)
int    *iv_resize_vars(unsigned new_dim,
                       IVEC **x1, IVEC **x2, ..., NULL)
int    *m_resize_vars (unsigned new_m, unsigned new_n,
                       MAT **A1, MAT **A2, ..., NULL)
int    *px_resize_vars(unsigned new_size,
                       PERM **px1, PERM **px2, ..., NULL)
int    *v_resize_vars (unsigned new_dim,
                       VEC **x1, VEC **x2, ..., NULL)

#include "zmatrix.h"
ZMAT   *zm_resize(ZMAT *A, int new_m, int new_n)
ZVEC   *zv_resize(ZVEC *x, int new_dim)
int    *zm_resize_vars(unsigned new_m, unsigned new_n,
                       ZMAT **A1, ZMAT **A2, ..., NULL)
int    *zv_resize_vars(unsigned new_dim,
                       ZVEC **x1, ZVEC **x2, ..., NULL)
```

DESCRIPTION

Each of these routines sets the (apparent) size of data structure to be identical to that obtained by using `..._get(new_...)`. Thus the `VEC *` returned by `v_resize(x,new_dim)` has `x->dim` equal to `new_dim`. The `MAT *` returned by `m_resize(A,new\_m,new\_n)` is a `new_m` × `new_n` matrix.

The following rules hold for all of the above functions except for `px_resize()`. Whenever there is overlap between the object passed and the re-sized data structure, the entries of the new data structure are identical, and elsewhere the entries are zero. So if `A` is a 5 × 2 matrix and `new_A = m_resize(A,2,5)`, then `new_A->me[1][0]` is identical to the old `A->me[1][0]`. However, `new_A->me[1][3]` is zero.

For `px_resize()` the rules are somewhat different because permutations do not remain permutations under such arbitrary operations. Instead, if the `size` is *reduced*,

then the returned permutation is an identity permutation. If `size` is *increased*, then `new_px->pe[i] == i` for `i` greater than or equal to the old `size`.

Allocation or reallocation and copying of data structure entries is avoided if possible (except, to some extent, in `m_resize()`). There is a "high-water mark" field contained within each data structure; for the `VEC` and `IVEC` data structures it is `max_dim`, which contains the actual amount of memory that has been allocated (at some time) for this data structure. Thus **resizing does not deallocate memory**! To actually free up memory, use one of the `.._free()` routines or the `.._FREE()` macros.

You should not rely on the values of entries outside the apparent size of the data structures but inside the maximum allocated area. These areas may be zeroed or overwritten, especially by the `m_resize()` routine.

The `.._resize_vars()` routines resize a NULL-terminated list of pointers to variables, all of the same type. The new sizes of the all variables in the list are the same. Calling

```
.._resize_vars([m,]n,&x1,&x2,...,&xN,NULL)
```

is equivalent to

```
x1 = .._resize(x1,[m,]n);
x2 = .._resize(x2,[m,]n);
   ......
xN = .._resize(xN,[m,]n);
```

(Note that "`[m,]`" indicates that "`m,`" might or might not be present, depending on whether the data structure involved is a matrix or not.) The returned value of the `.._resize_vars()` routines is the number of objects resized.

EXAMPLE

```
/* an alternative to workspace arrays */
... my_function(...)
{
    static VEC *x = VNULL;
      ......
    x = v_resize(x,new_size);
    MEM_STAT_REG(x,TYPE_VEC);
      ......
    v_copy(..., x);
      ......
}
```

BUGS

Note the above comment: **resizing does not deallocate memory**! To free up the actual memory allocated you will need to use the `.._FREE()` macros or the `.._free()` function calls.

SEE ALSO

`.._get()` routines; `MEM_STAT_REG()`.

SOURCE FILE: `memory.c`, `zmemory.c`, `bdfactor.c` and `ivecop.c`

NAME

MACHEPS – machine epsilon

SYNOPSIS

```
#include "matrix.h"
Real  macheps = MACHEPS;
```

DESCRIPTION

The quantity **MACHEPS** is a **#define**'d quantity which is the "machine epsilon" or "unit roundoff" for a given machine. For more information on this concept, see, e.g., Introduction to Numerical Analysis by K. Atkinson, or Matrix Computations by G. Golub and C. Van Loan. The value given is for the standard floating point type **Real** only. Normally the standard floating point type is **double**, but in the installation this can be changed to be **float** or **long double**. (See the chapter on installation.)

For ANSI C implementations, this is set to the value of the **DBL_EPSILON** or **FLT_EPSILON** macro defined in **<float.h>**.

EXAMPLE

```
while ( residual > 100*MACHEPS )
{   /* iterate */   }
```

BUGS

The value of **MACHEPS** has to be modified in the source whenever moving to another machine if the floating point processing is different.

SOURCE FILE: **machine.h**

NAME

m_add, m_mlt, m_sub, sm_mlt, zm_add, zm_mlt, zm_sub, zsm_mlt – matrix addition and multiplication

SYNOPSIS

```
#include "matrix.h"
MAT     *m_add(MAT *A, MAT *B, MAT *C)
MAT     *m_mlt(MAT *A, MAT *B, MAT *C)
MAT     *m_sub(MAT *A, MAT *B, MAT *C)
MAT     *sm_mlt(double s, MAT *A, MAT *OUT)

#include "zmatrix.h"
ZMAT    *zm_add(ZMAT *A, ZMAT *B, ZMAT *C)
ZMAT    *zm_mlt(ZMAT *A, ZMAT *B, ZMAT *C)
ZMAT    *zm_sub(ZMAT *A, ZMAT *B, ZMAT *C)
ZMAT    *zsm_mlt(complex s, ZMAT *A, ZMAT *OUT)
```

DESCRIPTION

The functions **m_add()**, **zm_add()** adds the matrices **A** and **B** and puts the result in **C**. If **C** is NULL, or is too small to contain the sum of **A** and **B**, then the matrix is resized to the correct size, which is then returned. Otherwise the matrix **C** is returned.

The functions, **m_sub()**, **zm_sub()** subtracts the matrix **B** from **A** and puts the result in **C**. If **C** is NULL, or is too small to contain the sum of **A** and **B**, then the matrix is resized to the correct size, which is then returned. Otherwise the matrix **C** is returned. Similarly, **m_mlt()** multiplies the matrices **A** and **B** and puts the result in **C**. Again, if **C** is NULL or too small, then a matrix of the correct size is created which is returned.

The routines **sm_mlt()**, **zsm_mlt()** above puts the results of multiplying the matrix **A** by the scalar **s** in the matrix **OUT**. If, on entry, **OUT** is NULL, or is too small to contain the results of this operation, then **OUT** is resized to have the correct size. The result of the operation is returned. This operation may be performed *in situ*. That is, you may use **A == OUT**.

The routines **m_add()**, **m_sub()** and **sm_mlt()** routines and their complex counterparts can work *in situ*; that is, **C** need not be different to either **A** or **B**. However, **m_mlt()** and **zm_mlt()** will raise an **E_INSITU** error if **A == C** or **B == C**.

These routines avoid thrashing on virtual memory machines.

EXAMPLE

```
MAT     *A, *B, *C;
Real    alpha;
   ......
C = m_add(A,B,MNULL);   /* C = A+B */
```

```
m_sub(A,B,C);           /* C = A-B */
sm_mlt(alpha,A,C);      /* C = alpha.A */
m_mlt(A,B,C);           /* C = A.B */
```

SEE ALSO

v_add(), mv_mlt(), sv_mlt(), zv_add(), zmv_mlt(), zv_mlt().

SOURCE FILE: matop.c, zmatop.c

NAME

mem_info, mem_info_on, mem_info_is_on, mem_info_bytes, mem_info_numvar, mem_info_file, mem_attach_list, mem_free_list, mem_bytes_list, mem_numvar_list, mem_dump_list, mem_is_list_attached – Meschach dynamic memory information

SYNOPSIS

```
#include "matrix.h"
void mem_info()
int  mem_info_on(int true_or_false)
int  mem_info_is_on(void)
void mem_info_file(FILE *fp, int list_num)
void mem_dump_list(FILE *fp, int list_num)
long mem_info_bytes (int type_num, int list_num)
int  mem_info_numvar(int type_num, int list_num)
int  mem_attach_list(int list_num, int ntypes, char *names[],
                     int (*frees[])(), MEM_ARRAY info_sum[])
int  mem_free_list(int list_num)
int  mem_is_list_attached(int list_num)
void mem_bytes(int type_num, int old_size, int new_size)
void mem_bytes_list(int type_num, int old_size, int new_size,
                    int list_num)
void mem_numvar(int type_num, int diff_numvar)
void mem_numvar_list(int type_num, int diff_numvar,
                     int list_num)
```

DESCRIPTION

These routines allow the user to obtain information about the amount of memory allocated for the Meschach data structures (**VEC**, **BAND**, **MAT**, **PERM**, **IVEC**, **ITER**, **SPMAT**, **SPROW**, **ZVEC** and **ZMAT**). The call **mem_info_on(TRUE);** sets a flag which directs the allocation and deallocation and resizing routines to store information about the memory that is (de)allocated and resized. The call **mem_info_on(FALSE);** turns the flag off.

The routine **mem_info_is_on()** returns the status of the memory information flag.

To get a general picture of the state of the memory allocated by Meschach data structures call **mem_info_file(fp,list_num)** which prints a summary of the amount of memory used for the different types of data structures to the file or stream **fp**. The **list_num** parameter indicates which list of types to use; use zero for the list of standard Meschach data types. The printout for **mem_info_file(stdout,0)**, or the equivalent macro **mem_info()** looks like this for one real and one complex vector of dimension 10 allocated (with the full system installed on an RS/6000):

```
MEMORY INFORMATION (standard types):
type MAT            0 alloc. bytes      0 alloc. variables
type BAND           0 alloc. bytes      0 alloc. variables
type PERM           0 alloc. bytes      0 alloc. variables
type VEC           92 alloc. bytes      1 alloc. variable
type IVEC           0 alloc. bytes      0 alloc. variables
type ITER           0 alloc. bytes      0 alloc. variables
type SPROW          0 alloc. bytes      0 alloc. variables
type SPMAT          0 alloc. bytes      0 alloc. variables
type ZVEC         204 alloc. bytes      1 alloc. variable
type ZMAT           0 alloc. bytes      0 alloc. variables
total:            296 alloc. bytes      2 alloc. variables
```

(Note that this is for the system built with all of Meschach, including the sparse part: `ITER`, `SPMAT`; and the complex part: `ZVEC`, `ZMAT`. The `mem_info_...()` routines also work for partial installations of Meschach.) There is also the routine `mem_dump_list()` which provides a more complete printout, which is suitable for debugging purposes.

To obtain information about the amount of memory allocated for objects of a particular type, use `mem_info_bytes()` (with `list_num` equal to zero for a standard Meschach structures). To find out the amount of memory allocated for ordinary vectors, use

```
printf("Bytes in VEC's = %ld = %ld\n",
       mem_info_bytes(TYPE_VEC,0));
```

The routine `mem_info_numvar()` returns the number of data structures that are allocated for each type. Use `list_num` equal to zero for standard Meschach structures.

Each Meschach type has an associated type macro `TYPE_...` which is a small integer. The "`...`" is the ordinary name of the type, such as `VEC`, `MAT` etc. This is the complete list of `TYPE_...` macros:

```
TYPE_MAT        0   /* real dense matrix */
TYPE_BAND       1   /* real band matrix */
TYPE_PERM       2   /* permutation */
TYPE_VEC        3   /* real vector */
TYPE_IVEC       4   /* integer vector */
TYPE_ITER       5   /* iteration structure */
TYPE_SPROW      6   /* real sparse matrix row */
TYPE_SPMAT      7   /* real sparse matrix */
TYPE_ZVEC       8   /* complex vector */
TYPE_ZMAT       9   /* complex dense matrix */
```

This is how different types are distinguished within the `mem_info_...` system.

Note that SPROW is an auxiliary type; when an SPROW (sparse row) is allocated as part of a SPMAT (sparse matrix), then the memory allocation is entered under SPMAT; only "stand-alone" SPROW's have their memory allocation entered under the typer SPROW.

The routine mem_attach_list() can be used to add new lists of types to the Meschach system for both tracking memory usage, and also for registering static workspace arrays with MEM_STAT_REG(). The routine is passed a collection of arrays: names is an array of strings being the names of the different types, frees is an array of the .._free() routines which deallocate and destroy objects of the corresponding types, info_sum is an array in which the memory allocation information is stored. This array has the component type MEM_ARRAY which is defined as

```
typedef struct {
   long bytes;  /* # allocated bytes for each type */
   int  numvar; /* # allocated variables for each type */
} MEM_ARRAY;
```

This is defined in matrix.h.

The parameter ntypes is the number of types, which should also be the common length of the arrays. The parameter list_num is the list number used to identify which list of types should be used. The routine mem_attach_list() returns the zero on successful completion, and (-1) if there is an invalid parameter. An E_OVERWRITE error will be raised if the specified list_num has already been used.

To track memory usage for any new types, the allocation, deallocation and resizing routines for these types you should use mem_bytes_list() and mem_numvar_list() to inform the mem_info_...() system of the change in the number of bytes allocated, and number of structures allocated, respectively, of an object of a particular type (as specified by the type_num and list_num parameters). In mem_bytes_list(), the parameter old_size should contain the old size in bytes, and new_size should contain the new size in bytes. In mem_numvar_list(), the parameter diff_numvar is the change in the number of allocated structures: $+1$ for allocating a new structure, and -1 for destroying a structure.

The routines mem_bytes() and mem_numvar() are just macros that call mem_bytes_list() and mem_numvar() respectively, with list_num zero for the standard Meschach structures.

The routine mem_attach_list() should be used once at the beginning of a program using these additional types.

Here is an example of how this might be used to extend Meschach with three types for nodes, edges and graphs:

```
/* Example with three new types: NODE, EDGE and GRAPH */
#define MY_LIST 1
#define TYPE_NODE  0
```

```
#define TYPE_EDGE  1
#define TYPE_GRAPH 2
static char *my_names[] = {  "NODE", "EDGE", "GRAPH"  };
static int  (*my_frees[]) = { n_free, e_free, gr_free };
static MEM_ARRAY my_tnums[3];  /* initialised to zeros */

main(...)
{
      ......  /* declarations */
    mem_attach_list(MY_LIST,3,my_names,my_frees,my_tnums);
      ......  /* actual work  */
    mem_info_file(stdout,MY_LIST);  /* list memory used */
}

/* n_get -- get a node data structure;
            NODE has type number 0 */
NODE *n_get(...)
{
    NODE  *n;

    n = NEW(NODE);
    if ( n == NULL )
      error(E_MEM,"n_get"); /* can't allocate memory */
    mem_bytes_list(TYPE_NODE,0,sizeof(NODE),MY_LIST);
    mem_numvar_list(TYPE_NODE,1,MY_LIST);
      ......
}

/* n_free -- deallocate node data structure */
int   n_free(NODE *n)
{
    if ( n != NULL )
    {
        free(n);
        mem_res_elem_list(TYPE_NODE,sizeof(NODE),0,MY_LIST);
        mem_numvar_list(TYPE_NODE,-1,MY_LIST);
    }
    return 0;
}
```

For more information see chapter 8.

BUGS

Memory used by the underlying memory (de)allocation system (`malloc()`,

`calloc()`, `realloc()`, `sbrk()` etc.) for headers are not included in the amounts of allocated memory.

The numbers of vectors, matrices etc. currently allocated cannot be found by this system.

SEE ALSO

`..._get()`, `..._free()`, `..._resize()` routines; `MEM_STAT_REG()` and the `mem_stat_...()` routines.

SOURCE FILE: `meminfo.c, meminfo.h`

NAME

MEM_STAT_REG, mem_stat_reg_list, mem_stat_reg_vars, mem_stat_mark, mem_stat_free, mem_stat_dump, mem_stat_show_mark – Static workspace control routines

SYNOPSIS

```
#include "matrix.h"
int  MEM_STAT_REG(void *var, int type)
int  mem_stat_reg_list(void **var, int type, int list_num)
int  mem_stat_reg_vars(int list_num, int type,
                       void **var1, void **var2, ..., NULL)
int  mem_stat_mark(int mark)
int  mem_stat_free(int mark)
void mem_stat_dump(FILE *fp)
int  mem_stat_show_mark()
```

DESCRIPTION

Older versions of Meschach (v.1.1b and previous) had a limitation in that it was essentially impossible to control the use of static workspace arrays used within Meschach functions. This can lead to problems where too much memory is taken up by these workspace arrays for memory intensive problems. The obvious alternative approach is to deallocate workspace at the end of every function, which can be quite expensive because of the time taken to deallocate and the reallocate the memory on every usage.

These functions provide a way of avoiding these problems, by giving users control over the (selective) destruction of workspace vectors, matrices, etc.

The simplest way to use this to deallocate workspace arrays in a routine **hairy1(...)** is as follows:

```
  ......
mem_stat_mark(1);  /* ''group 1'' of workspace arrays */
for ( i = 0; i < n; i++ )
    hairy1(...);   /* workspace registered as ''group 1'' */
mem_stat_free(1);  /* deallocate ''group 1'' workspace */
```

The call **mem_stat_mark(num)** sets the current workspace group number. This number must be a positive integer. Provided the appropriate workspace registration routines are used in **hairy1(...)** (see later), then the workspace arrays are registered as being in the current workspace group as determined by **mem_stat_mark()**. If **mem_stat_mark()** has not been called, then there is no current group number and the variables are not registered. The call **mem_stat_free(num)** deallocates all static workspace arrays allocated in workspace group **num**, and also unsets the current workspace group. So, to continue registering static workspace variables, **mem_stat_mark(num)**, or
mem_stat_mark(new_num) should follow.

Keeping two groups of registered static workspace variables (one for `hairy1()` and another for `hairy2()`) can be done as follows:

```
  ......
for ( i = 0; i < n; i++ )
{
    mem_stat_mark(1);
    hairy1(...);
    mem_stat_mark(2);
    hairy2(...);
}
mem_stat_free(2);      /* don't want hairy2()'s workspace */
hairy1(...);           /* keep hairy1()'s workspace */
```

For the person writing routines to use workspace arrays, there are a number of rules that must be followed if these routines are to be used.

- the workspace variables must be `static` pointers to Meschach data structures.
- they must be initialised to be NULL vectors in the type declaration.
- they are allocated using a `..._resize()` routine.
- they are allocated before registering.
- the pointer variable is passed to `MEM_STAT_REG()`, but `mem_stat_reg_vars()` and `mem_stat_reg_vars()` require the *address* of the pointer to be passed.

The `type` parameter of `MEM_STAT_REG()` should be a macro of the form `TYPE_...` where the "`...`" is the name of the type used. An example of its use follows:

```
VEC *hairy1(x, y, out)
VEC *x, *y, *out;
{
    static VEC *wkspace = VNULL;
    int    new_dim;
      ......
    wkspace = v_resize(wkspace,new_dim);
    MEM_STAT_REG(wkspace,TYPE_VEC);
      ......
    mv_mlt(....,wkspace);  /* use of wkspace */
      ......
    /* no need to deallocate wkspace */
    return out;
}
```

`MEM_STAT_REG()` is a macro which calls `mem_stat_reg_list()` with `list_num` set to zero.

The call `mem_stat_dump(fp)` prints out a representation of the registered workspace variables onto the file or stream `fp` suitable for debugging purposes. It is not expected that this would be needed by most users of Meschach.

The routine `mem_stat_show_mark()` returns the current workspace group, and zero if no group is active.

A NULL terminated list of variables can be registered at once using `mem_stat_reg_vars()`. The call

```
mem_stat_reg_vars(list_num,type_num,&x1,&x2,...,&xN,NULL);
```

is equivalent to

```
mem_stat_reg_list(&x1,type_num,list_num);
mem_stat_reg_list(&x2,type_num,list_num);
  ......
mem_stat_reg_list(&xN,type_num,list_num);
```

Note that `x1`, `x2`, ... , `xN` must be of the same type.

For non-Meschach data structures, you can use `mem_stat_reg_list()` in conjunction with `mem_attach_list()`. For more information on the use of this function see chapter 8.

SEE ALSO

`mem_info_...()` routines.

BUGS

There is a static registration area for workspace variables, so there is a limit on the number of variables that can be registered. The default limit is 509. If it is too small, an appropriate message will appear and information on how to change the limit will follow.

Attempts to register a workspace array that is neither `static` or global will most likely result in a crash when `mem_stat_free()` is called for the workspace group containing that variable.

SOURCE FILE: `memstat.c`

NAME

m_load, m_save, v_save, d_save, zm_load, z_save, zm_save, zv_save – MATLAB save/load to file

SYNOPSIS

```
#include "matlab.h"
MAT     *m_load(FILE *fp, char **name)
MAT     *m_save(FILE *fp, MAT *A, char **name)
VEC     *v_save(FILE *fp, VEC *x, char **name)
double   d_save(FILE *fp, double d, char **name)

#include "matlab.h"
ZMAT    *zm_load(FILE *fp, char **name)
ZMAT    *zm_save(FILE *fp, ZMAT *A, char **name)
ZVEC    *zv_save(FILE *fp, ZVEC *x, char **name)
complex  z_save (FILE *fp, complex z, char **name)
```

DESCRIPTION

These routines read and write MATLAB™ load/save files. This enables results to be transported between MATLAB and Meschach. The routine **m_load()** loads in a matrix from file **fp** in MATLAB save format. The matrix read from the file is returned, and **name** is set to point to the saved MATLAB variable name of the matrix. Both the matrix returned and **name** have allocated memory as needed. An example of the use of the routine to load a matrix **A** and a vector **x** is

```
MAT *A, *Xmat;
VEC *x;
FILE *fp;
char *name1, *name2;
  ......
if ( (fp=fopen("fred.mat","r")) != NULL )
{
    A    = m_load(fp,&name1);
    Xmat = m_load(fp,&name2);
    if ( Xmat->n != 1 )
    {  printf("Incorrect size matrix read in\n");
       exit(0);  }
    x = v_get(Xmat->m);
    x = mv_move(Xmat,0,0,Xmat->m,1,x,0);
}
```

The **m_save()** routine saves the matrix **A** to the file/stream **fp** in MATLAB save format. The MATLAB variable name is **name**.

The `v_save()` routine saves the vector `x` to the file/stream `fp` as an `x->dim` $\times$ 1 matrix (i.e. as a column vector) in MATLAB save format. The MATLAB variable name is `name`.

The `d_save()` routine saves the double precision number `d` to the file/stream `fp` in MATLAB save format. The MATLAB variable name is `name`.

The MATLAB save format can depend in subtle ways on the type of machine used, so you may need to set the machine type in `machine.h`. This should usually just mean adding a line to `machine.h` to be one of

```
#define MACH_ID INTEL              /* 80x87 format */
#define MACH_ID MOTOROLA           /* 6888x format */
#define MACH_ID VAX_D              /* VAX D format */
#define MACH_ID VAX_G              /* VAX G format */
```

to be the appropriate machine. The machine dependence involves both whether IEEE or non IEEE format floating point numbers are used, but also whether or not the machine is a "little-endian" or a "big-endian" machine.

BUGS

The `m_load()` routine will only read in the real part of a complex matrix.

The routines are machine-dependent as described above.

SOURCE FILE: `matlab.c, zmatlab.c`

NAME

bd_transp, m_transp, mmtr_mlt, mtrm_mlt, zm_adjoint, zmma_mlt, zmam_mlt – matrix transposes, adjoints and multiplication

SYNOPSIS

```
#include "matrix.h"
BAND   *bd_transp(BAND *A, BAND *OUT)
MAT    *m_transp(MAT *A, MAT *OUT)
MAT    *mmtr_mlt(MAT *A, MAT *B, MAT *OUT)
MAT    *mtrm_mlt(MAT *A, MAT *B, MAT *OUT)

#include "zmatrix.h"
ZMAT   *zm_adjoint(ZMAT *A, ZMAT *OUT)
ZMAT   *zmma_mlt(ZMAT *A, ZMAT *B, ZMAT *OUT)
ZMAT   *zmam_mlt(ZMAT *A, ZMAT *B, ZMAT *OUT)
```

DESCRIPTION

The routine **bd_transp()** computes the transpose of the banded matrix **A** and puts the result in **OUT**. Both are **BAND** structures.

The routine **m_transp()** transposes the matrix **A** and stores the result in **OUT**. The routine **m_adjoint()** takes the complex conjugate transpose (or complex adjoint) of **A** and stores the result in **OUT**. These routines may be *in situ* (i.e. **A == OUT**) only if **A** is square. (Note that **BAND** matrices are always square.) The complex adjoint of A is denoted A^*.

The routine **mmtr_mlt()** forms the product AB^T, which is stored in **OUT**. The routine **mma_mlt()** forms the product AB^*, which is stored in **OUT**. The routine **mtrm_mlt()** forms the product A^TB, which is stored in **OUT**. The routine **mam_mlt()** forms the product A^*B, which is stored in **OUT**. Neither of these routines can form the product *in situ*. This means that they must be used with **A != OUT** and **B != OUT**. However, you can still use **A == B**.

For all the above routines, if **OUT** is NULL or too small to contain the result, then it is resized to the correct size, and is then returned.

EXAMPLE

```
MAT    *A, *B, *C;
  ......
C = m_transp(A,MNULL);     /* C = A^T */
mmtr_mlt(A,B,C);           /* C = A.B^T */
mtrm_mlt(A,B,C);           /* C = A^T.B */
```

SOURCE FILE: matop.c, zmatop.c

NAME

m_norm1, m_norm_inf, m_norm_frob, zm_norm1, zm_norm_inf, zm_norm_frob – matrix norms

SYNOPSIS

```
#include "matrix.h"
Real    m_norm1(MAT *A)
Real    m_norm_inf(MAT *A)
Real    m_norm_frob(MAT *A)

#include "zmatrix.h"
Real    zm_norm1(ZMAT *A)
Real    zm_norm_inf(ZMAT *A)
Real    zm_norm_frob(ZMAT *A)
```

DESCRIPTION

These routines compute matrix norms. The routines **m_norm1()** and **zm_norm1()** compute the matrix norm of **A** in the matrix 1–norm; **m_norm_inf()** and **zm_norm_inf()** compute the matrix norm of **A** in the matrix ∞–norm; **m_norm_frob()** and **zm_norm_frob()** compute the Frobenius norm of **A**. All of these routines are unscaled; that is, there is no scaling vector for weighting the elements of **A**.

These norms are defined through the following formulae:

$$\|A\|_1 = \max_j \sum_i |a_{ij}|, \qquad \|A\|_\infty = \max_i \sum_j |a_{ij}|, \tag{4.1}$$

$$\|A\|_F = \sqrt{\sum_{ij} |a_{ij}|^2}. \tag{4.2}$$

The matrix 2–norm is not included as it requires the calculation of eigenvalues or singular values.

EXAMPLE

```
MAT    *A;
  ......
printf("||A||_1 = %g\n", m_norm1(A));
printf("||A||_inf = %g\n", m_norm_inf(A));
printf("||A||_F = %g\n", m_norm_frob(A));
```

SEE ALSO

v_norm1(), v_norm_inf(), zv_norm1(), zv_norm_inf().

BUGS

The Frobenius norm calculations may overflow if the elements of **A** are of order $\sqrt{\texttt{HUGE}}$.

SOURCE FILE: `norm.c, znorm.c`

NAME

mv_mlt, vm_mlt, mv_mltadd, vm_mltadd, zmv_mlt, zvm_mlt, zmv_mltadd, zvm_mltadd – matrix–vector multiplication

SYNOPSIS

```
#include "matrix.h"
VEC      *mv_mlt(MAT *A, VEC *x, VEC *out)
VEC      *vm_mlt(MAT *A, VEC *x, VEC *out)
VEC      *mv_mltadd(VEC *v1, VEC *v2, MAT *A,
                    double s, VEC *out)
VEC      *vm_mltadd(VEC *v1, VEC *v2, MAT *A,
                    double s, VEC *out)

#include "zmatrix.h"
ZVEC     *zmv_mlt(ZMAT *A, ZVEC *x, ZVEC *out)
ZVEC     *zvm_mlt(ZMAT *A, ZVEC *x, ZVEC *out)
ZVEC     *zmv_mltadd(ZVEC *v1, ZVEC *v2, ZMAT *A,
                     complex s, ZVEC *out)
ZVEC     *zvm_mltadd(ZVEC *v1, ZVEC *v2, ZMAT *A,
                     complex s, ZVEC *out)
```

DESCRIPTION

The routines **mv_mlt()** and **vm_mlt()** form Ax and $A^Tx = (x^TA)^T$ respectively and store the result in **out**. The routines **zmv_mlt()** and **zvm_mlt()** form Ax and $A^*x = (x^*A)^*$ respectively and store the result in **out**. The routines **mv_mltadd()** and **vm_mltadd()** form $v_1 + sAv_2$ and $v_1 + sA^Tv_2$ respectively, and stores the result in **out**. The routines **zmv_mltadd()** and **zvm_mltadd()** form $v_1 + sAv_2$ and $v_1 + sA^*v_2$ respectively, and stores the result in **out**. If **out** is NULL or too small to contain the product, then it is resized to the correct size.

These routines do not work *in situ*; that is, **out** must be different to **x** for **mv_mlt()** and **vm_mlt()**, and in the case of **mv_mltadd()** and **vm_mltadd()**, **out** must be different to **v2**.

These routines avoid thrashing virtual memory machines.

EXAMPLE

```
MAT     *A;
VEC     *x, *y, *out;
Real    alpha;
  ......
out = mv_mlt(A,x,VNULL);     /* out = A.x */
vm_mlt(A,x,out);             /* out = A^T.x */
mv_mltadd(x,y,A,out);        /* out = x + A.y */
vm_mltadd(x,y,A,out);        /* out = x + A^T.y */
```

SOURCE FILE: `matop.c, zmatop.c`

NAME

px_ident, px_inv, px_mlt, px_transp, px_sign – permutation identity, inverse and multiplication

SYNOPSIS

```
#include "matrix.h"
PERM    *px_ident(PERM *pi)
PERM    *px_mlt(PERM *pi1, PERM *pi2, PERM *out)
PERM    *px_inv(PERM *pi, PERM *out)
PERM    *px_transp(PERM *pi, int i, int j)
int      px_sign(PERM *pi)
```

DESCRIPTION

The routine `px_ident()` initialises `pi` to be the identity permutation of the size of `pi->size` on entry. The permutation `pi` is returned. If `pi` is NULL then an error is generated.

The routine `px_mlt()` multiplies `pi1` by `pi2` to give `out`. If `out` is NULL or too small, then `out` is resized to be a permutation of the correct size. This cannot be done *in situ*.

The routine `px_inv()` computes the inverse of the permutation `pi`. The result is stored in `out`. If `out` is NULL or is too small, a permutation of the correct size is created, which is returned. This can be done *in situ* if `pi == out`.

The routine `px_transp()` swaps `pi->pe[i]` and `pi->pe[j]`; it is a multiplication by the transposition $i \leftrightarrow j$.

The routine `px_sign(pi)` computes the sign of the permutation `pi`. This sign is $(-1)^p$ where `pi` can be written as the product of p permutations. This is done by sorting the entries of `pi` using quicksort, and counting the number of transpositions used. This is also the determinant of the permutation matrix represented by `pi`.

EXAMPLE

```
PERM  *pi1, pi2, pi3;
  ......
pi1 = px_get(10);
px_ident(pi1);          /* sets pi1 to identity */
px_transp(pi1,3,5);     /* pi1 is now a transposition */
px_inv(pi1,pi1);        /* invert pi1 -- in situ */
px_mlt(pi1,pi2,pi3);    /* pi3 = pi1.pi2 */
printf("sign(pi3) = %d = %d\n",
       px_sign(pi1)*px_sign(pi2), px_sign(pi3));
```

SOURCE FILE: `pxop.c`

NAME

px_cols, px_rows, px_vec, pxinv_vec, px_zvec, pxinv_zvec – permute rows or columns of a matrix, or permute a vector

SYNOPSIS

```
#include "matrix.h"
MAT     *px_rows(PERM *pi, MAT *A, MAT *OUT)
MAT     *px_cols(PERM *pi, MAT *A, MAT *OUT)
VEC     *px_vec (PERM *pi, VEC *x, VEC *out)
VEC     *pxinv_vec(PERM *pi, VEC *x, VEC *out)

#include "zmatrix.h"
ZVEC    *px_zvec    (PERM *pi, ZVEC *x, ZVEC *out)
ZVEC    *pxinv_zvec(PERM *pi, ZVEC *x, ZVEC *out)
```

DESCRIPTION

The routines **px_rows()** and **px_cols()** are for permuting matrices, permuting respectively the rows and columns of the matrix **A**. In particular, for **px_rows()** the **i**-th row of **OUT** is the **pi->pe[i]**-th row of **A**. Thus $OUT = PA$ where P is the permutation matrix described by **pi**. The routine **px_cols()** computes $OUT = AP$.

The result is stored in **OUT** provide it has sufficient space for the result. If **OUT** is NULL or too small to contain the result then it is replaced by a matrix of the appropriate size. In either case the result is returned.

Similarly, **px_vec()** and **px_zvec()** permute the entries of the vector **x** into the vector **out** by the rule that the **i**-th entry of **out** is the **pi->pe[i]**-th entry of **x**. Conversely, **pxinv_vec()** and **pxinv_zvec()** permute **x** into **out** by the rule that the **pi->pe[i]**-th entry of **out** is the **i**-th entry of **x**. This is equivalent to inverting the permutation **pi** and then applying **px_vec()**, respectively, **px_zvec()** for real, resp., complex vectors.

If **out** is NULL or too small to contain the result, then a new vector is created and the result stored in it. In either case the result is returned.

EXAMPLE

```
PERM  *pi;
VEC   *x, *tmp;
ZVEC  *z, *ztmp;
MAT   *A, *B;
  ......
/* permute x to give tmp */
tmp = px_vec(pi,x,tmp);
ztmp = px_zvec(pi,z,ZVNULL);
/* restore x & z */
```

```
x = pxinv_vec(pi,tmp,x);
pxinv_zvec(pi,ztmp,z);
/* symmetric permutation */
B = px_rows(pi,A,MNULL);
A = px_cols(pi,B,A);
```

SEE ALSO

The `px_...()` operations; in particular `px_inv()`

SOURCE FILE: `pxop.c, zvecop.c`

NAME

set_col, set_row, zset_col, zset_row – set rows and columns of matrices

SYNOPSIS

```
#include "matrix.h"
MAT    *set_col(MAT *A, int k, VEC *out)
MAT    *set_row(MAT *A, int k, VEC *out)

#include "zmatrix.h"
ZMAT   *zset_col(ZMAT *A, int k, ZVEC *out)
ZMAT   *zset_row(ZMAT *A, int k, ZVEC *out)
```

DESCRIPTION

The routines **set_col()** and **zset_col()** above sets the value of the *k*th column of **A** to be the values of **out**. The **A** matrix so modified is returned.

The routine **set_row()** above sets the value of the *k*th row of **A** to be the values of **out**. The **A** matrix so modified is returned.

If **out** is NULL, then an **E_NULL** error is raised. If **k** is negative or greater than or equal to the number of columns or rows respectively, an **E_BOUNDS** error is raised.

As the **MAT** and **ZMAT** data structures are row-oriented data structures, the **set_row()** routine is faster than the **set_col()** routine.

EXAMPLE

```
MAT    *A;
VEC    *tmp;
  ......
/* scale row 3 of A by 2.0 */
tmp = get_row(A,3,VNULL);
sv_mlt(2.0,tmp,tmp);
set_row(A,3,tmp);
```

SEE ALSO

get_col() and **get_row()**

SOURCE FILE: matop.c, zmatop.c

NAME

sv_mlt, v_add, v_mltadd, v_sub, zv_mlt, zv_add, zv_mltadd, zv_sub – scalar–vector multiplication and addition

SYNOPSIS

```
#include "matrix.h"
VEC     *sv_mlt(double s, VEC *x, VEC *out)
VEC     *v_add(VEC *v1, VEC *v2, VEC *out)
VEC     *v_mltadd(VEC *v1, VEC *v2, double s, VEC *out)
VEC     *v_sub(VEC *v1, VEC *v2, VEC *out)

#include "zmatrix.h"
ZVEC    *zv_mlt(complex s, ZVEC *x, ZVEC *out)
ZVEC    *zv_add(ZVEC *v1, ZVEC *v2, ZVEC *out)
ZVEC    *zv_mltadd(ZVEC *v1, ZVEC *v2, complex s, ZVEC *out)
ZVEC    *zv_sub(ZVEC *v1, ZVEC *v2, ZVEC *out)
```

DESCRIPTION

The routines **sv_mlt()** and **zv_mlt()** perform the scalar multiplication of the scalar **s** and the vector **x** and the results are placed in **out**.

The routines **v_add()** and **zv_add()** adds the vectors **v1** and **v2**, and the result is returned in **out**.

The routines **v_mltadd()** and **zv_mltadd()** set **out** to be the linear combination **v1+s.v2**.

The routines **v_sub()** and **zv_sub()** subtract **v2** from **v1**, and the result is returned in **out**.

For all of the above routines, if **out** is NULL, then a new vector of the appropriate size is created. For all routines the result (whether newly allocated or not) is returned. All these operations may be performed *in situ*. Errors are raised if **v1** or **v2** are NULL, or if **v1** and **v2** have different dimensions.

EXAMPLE

```
VEC     *x, *y, *z, *tmp;
ZVEC    *v, *w;
Real    alpha;
complex beta;
  ......
tmp = v_get(x->dim);
z = v_get(x->dim);
printf("# 2-Norm of x - y = %g\n",
       v_norm2(v_sub(x,y,tmp)));
```

```
/* z = x + alpha.y */
v_mltadd(x,y,alpha,z);
/* ...or equivalently */
sv_mlt(alpha,y,z);
v_add(x,z,z);
zv_mltadd(v,w,beta,v);
```

SOURCE FILE: `vecop.c, zvecop.c`

NAME

v_conv, v_map, v_max, v_min, v_pconv, v_star, v_slash, v_sort, v_sum, zv_map, zv_star, zv_slash, zv_sum – Componentwise operations

SYNOPSIS

```
#include  "matrix.h"
VEC      *v_conv (VEC *x, VEC *y, VEC *out)
VEC      *v_pconv(VEC *x, VEC *y, VEC *out)
VEC      *v_map  (double (*fn)(double), VEC *x, VEC *out)
double    v_max  (VEC *x, int *index)
double    v_min  (VEC *x, int *index)
VEC      *v_star (VEC *x, VEC *y, VEC *out)
VEC      *v_slash(VEC *x, VEC *y, VEC *out)
VEC      *v_sort (VEC *x, PERM *order)
double    v_sum  (VEC *x)

#include  "mzatrix.h"
ZVEC     *zv_map(complex (*fn)(complex), ZVEC *x, ZVEC *out)
ZVEC     *zv_star(ZVEC *x, ZVEC *y, ZVEC *out)
ZVEC     *zv_slash(ZVEC *x, ZVEC *y, ZVEC *out)
complex   zv_sum(ZVEC *x)
```

DESCRIPTION

The routines `v_conv()` and `v_pconv()` compute convolution-type products of vectors. The routine `v_conv()` computes the vector z where $z_i = \sum_{0 \le j \le i} x_j y_{i-j}$. The routine `v_pconv()` computes a periodic convolution with period `y->dim`. The routine `v_conv()` can be used to compute the product of two polynomials, with the polynomial $x(t) = \sum_{i=0}^{\deg x} x_i t^i$ and $y(t) = \sum_{i=0}^{\deg y} y_i t^i$.

The routines `v_map()` and `zv_map()` apply the function `(*fn)()` to the components of `x` to give the vector `out`. That is, `out->ve[i] = (*fn)(x->ve[i])`. There are also versions

```
VEC     *_v_map(double (*fn)(void *,double),
                void *fn_params, VEC *x, VEC *out)
ZVEC    *_zv_map(complex (*fn)(void *,complex),
                 void *fn_params, ZVEC *x, ZVEC *out)
```

where `out->ve[i] = (*fn)(fn_params,x->ve[i])`. This enables more flexible use of this function. Both of these functions may be used *in situ* with `x == out`.

The routine `v_max()` returns the maximum entry of the vector `x`, and sets `index` to be the index of this maximum value in `x`. Note that `index` is the in-

dex for the *first* entry with this value. Thus `max_x = v_max(x, &i)` means that `x->ve[i] == max_x`.

The routine `v_min()` returns the minimum entry of the vector `x`, and sets `index` to be the index of this minimum value similarly to `v_max()`. Both `v_min()` and `v_max()` raise an `E_SIZES` error if they are passed zero dimensional vectors.

The routines `v_star()` and `zv_star()` compute the componentwise, or Hadamard, product of `x` and `y`. That is, `out->ve[i] = x->ve[i]*y->ve[i]` for all `i`. Note that `v_star()` is equivalent to multiplying `y` by a diagonal matrix whose diagonal entries are given by the entries of `x`. This routine may be used *in situ* with `x == out`.

The routines `v_slash()` and `zv_slash()` compute the componentwise ratio of entries of `y` and `x`. (Note the order!) That is, `out->ve[i] = y->ve[i]/x->ve[i]` for all `i`. Note that this is equivalent to multiplying `y` by the inverse of the diagonal matrix described in the previous paragraph. This could be useful for preconditioning, for example. This routine may be used *in situ* with `x == out` and/or `y == out`. The routine `v_slash()` raises an `E_SING` error if `x` has a zero entry (the rationale being that it is really solving the system of equations $Xz = y$ where z is `out`).

The routine `v_sort()` sorts the entries of the vector `x` *in situ*, and sets `order` to be the permutation that achieves this. Note that the old ordering of `x` can be obtained by using `pxinv_vec()` as illustrated in the example below. The algorithm used is a version of quicksort based on that given in *Algorithms in C*, by R. Sedgewick, pp. 116–124 (1990).

The routines `v_sum()` and `zv_sum()` return the sum of the entries of `x`.

Note that there are no complex "min", "max" or "sorting" routines, as there is no suitable ordering on the complex numbers.

EXAMPLE

An alternative way of computing $\|x\|_\infty$ (but slower):

```
VEC     *x, *y, *z;
PERM    *order;
Real    norm;
int     i;
  ......
y = v_map(fabs,x,VNULL);
norm = v_max(y,&i);
```

Sorting a vector:

```
v_sort(x,order);
/* x now sorted */
y = pxinv_vec(order,x,VNULL);
/* y is now the original x */
```

Using the Hadamard product for setting $y_i = w_i x_i$:

```
VEC     *weights;
  ......
for ( i = 0; i < weights->dim; i++ )
    weights->ve[i] = ...;
  ......
v_star(weights,x,y);
```

SEE ALSO

Other componentwise operations: `v_add()`, `v_sub()`, `sv_mlt()`.

Iterative routines benefiting from diagonal preconditioning: `iter_cg()`, `iter_cgs()`, and `iter_lsqr()`.

SOURCE FILE: `vecop.c, zvecop.c`

NAME

v_lincomb, v_linlist, zv_lincomb, zv_linlist – linear combinations

SYNOPSIS

```
#include "matrix.h"
VEC  *v_lincomb(int n, VEC *v_list[], double a_list[],
                VEC *out)
VEC  *v_linlist(VEC *out, VEC *v1, double a1,
                          VEC *v2, double a2, ..., VNULL)

#include "zmatrix.h"
ZVEC *zv_lincomb(int n, ZVEC *v_list[], complex a_list[],
                 ZVEC *out)
ZVEC *zv_linlist(ZVEC *out, ZVEC *v1, complex a1,
                           ZVEC *v2, complex a2, ..., ZVNULL)
```

DESCRIPTION

The routines `v_lincomb()` and `zv_lincomb()` compute the linear combination $\sum_{i=0}^{n-1} a_i v_i$ where v_i is identified with `v_list[i]` and a_i is identified with `a_list[i]`. The result is stored in `out`, which is created or resized as necessary. Note that `n` is the *length* of the lists.

An `E_INSITU` error will be raised if `out == v_list[i]` for any `i` other than `i == 0`.

The routines `v_linlist()` and `zv_linlist()` are variants of the above which do not require setting up an array before hand. This returns $\sum_i a_i v_i$ where the sum is over $i = 1, 2, \ldots$ until a `VNULL` is reached, which should take the place of one of the `vk`'s.

An `E_INSITU` error will be raised if `out == v2, v3, v4,...`.

EXAMPLE

```
VEC     *x[10], *v1, *v2, *v3, *v4, *out;
Real     a[10], h;
  ......
for ( i = 0; i < 10; i++ )
{   x[i] = ...;  a[i] = ...;   }
out = v_lincomb(10,x,a,VNULL)
/* for Runge--Kutta code:
      out = h/6*(v1+2*v2+2*v3+v4) */
v_zero(out);
out = v_linlist(out, v1, h/6.0, v2, h/3.0,
                     v3, h/3.0, v4, h/6.0,
                     VNULL);
```

SEE ALSO

`sv_mlt()`, `v_mltadd()`, `zv_mlt()`, `zv_mltadd()`

BUGS

SOURCE FILE: `vecop.c`, `zvecop.c`

NAME

v_norm1, v_norm2, v_norm_inf, zv_norm1, zv_norm2, zv_norm_inf – vector norms

SYNOPSIS

```
#include "matrix.h"
double  v_norm1(VEC *x)
double  v_norm2(VEC *x)
double  v_norm_inf(VEC *x)
double _v_norm1(VEC *x, VEC *scale)
double _v_norm2(VEC *x, VEC *scale)
double _v_norm_inf(VEC *x, VEC *scale)

#include "zmatrix.h"
double  zv_norm1(ZVEC *x)
double  zv_norm2(ZVEC *x)
double  zv_norm_inf(ZVEC *x)
double _zv_norm1(ZVEC *x, VEC *scale)
double _zv_norm2(ZVEC *x, VEC *scale)
double _zv_norm_inf(ZVEC *x, VEC *scale)
```

DESCRIPTION

These functions compute vector norms. In particular, **v_norm1()** and **zv_norm1()** give the 1–norm, **v_norm2()** and **zv_norm2()** give the 2–norm or Euclidean norm, and **v_norm_inf()** and **zv_norm_inf()** compute the ∞–norm. These are defined by the following formulae:

$$\|x\|_1 = \sum_i |x_i| \tag{4.3}$$

$$\|x\|_\infty = \max_i |x_i| \tag{4.4}$$

$$\|x\|_2 = \sqrt{\sum_i |x_i|^2}. \tag{4.5}$$

There are also *scaled* versions of these vector norms: **_v_norm1()**, **_v_norm2()** and **_v_norm_inf()**, and **_zv_norm1()**, **_zv_norm2()** and **_zv_norm_inf()**. These take a vector **x** whose norm is to be computed, and a scaling vector. Each component of the **x** vector is divided by the corresponding component of the **scale** vector, and the norm is computed for the "scaled" version of **x**. Note that the **scale** vector is a (real) **VEC** since only the magnitudes are important. If the corresponding component of **scale** is zero for that component of **x**, or if **scale** is NULL, then no scaling is done. (In fact, **v_norm1(x)** is a macro that expands to **_v_norm1(x,VNULL)**.)

For example, **_v_norm1(x,scale)** returns

$$\sum_i |x_i/scale_i|$$

provided **scale** is not NULL, and no element of **scale** is zero. The behaviour of **_v_norm2()** and **_v_norm_inf()** is similar.

EXAMPLE

```
VEC     *x, *scale;
  ......
printf("# 2-Norm of x = %g\n", v_norm2(x));
printf("# Scaled 2-norm of x = %g\n",
       _v_norm2(x,scale));
```

SEE ALSO

m_norm1(), **m_norm_inf()**, **zm_norm1()**, **zm_norm_inf()**.

BUGS

There is the possibility that **v_norm2()** may overflow if **x** has components with size of order $\sqrt{\texttt{HUGE}}$.

SOURCE FILE: **norm.c**

NAME

zmake, zconj, zneg, zabs, zadd, zsub, zmlt, zinv, zdiv, zsqrt, zexp, zlog – Operations on complex numbers

SYNOPSIS

```
#include "zmatrix.h"
complex zmake(double real, double imag)
complex zconj(complex z)
complex zneg(complex z)
double  zabs(complex z)
complex zadd(complex z1, complex z2)
complex zsub(complex z1, complex z2)
complex zmlt(complex z1, complex z2)
complex zinv(complex z)
complex zdiv(complex z1, complex z2)
complex zsqrt(complex z)
complex zexp(complex z)
complex zlog(complex z)
```

DESCRIPTION

These routines provide the basic operations on complex numbers.

Complex numbers are represented by the **complex** data structure which is defined as

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

and the real part of **complex z;** is **z.re** and its imaginary part is **z.im**. Let $z = x + iy$.

The routine **zmake(real,imag)** returns the complex number with real part **real** and imaginary part **imag**.

The routine **zconj(z)** returns $\bar{z} = x - iy$

The routine **zneg(z)** returns $-z$.

The routine **zabs(z)** returns $|z| = \sqrt{x^2 + y^2}$. Note that it is done safely to avoid overflow if $|x|$ or $|y|$ is close to floating point limits.

The routine **zadd(z1,z2)** returns $z_1 + z_2$.

The routine **zsub(z1,z2)** returns $z_1 - z_2$.

The routine **zmlt(z1,z2)** returns $z_1 z_2$.

The routine **zinv(z)** returns $1/z$. An **E_SING** error is raised if $z = 0$.

The routine **zdiv(z1,z2)** returns z_1/z_2. An **E_SING** error is raised if $z_2 = 0$.

The routine `zsqrt(z)` returns $\sqrt{z}$. The principle branch is used for a branch cut along the negative real axis, so the real part of $\sqrt{z}$ as computed is not negative.

The routine `zexp(z)` returns $\exp(z) = e^z = e^x(\cos y + i \sin y)$.

The routine `zlog(z)` returns $\log(z)$. The principle branch is used for a branch cut along the negative real axis, so the imaginary part of $\log(z)$ lies between or on $\pm\pi$.

EXAMPLE

To compute $\log(z + e^w)/\sqrt{1 + z^2}$:

```
complex w, z, result;
  ......
result = zdiv(zlog(zadd(z,zexp(w))),
              zsqrt(zadd(ONE,zmlt(z,z))));
```

where `ONE` is $1 + 0i$; `ONE = zmake(1.0,0.0);`.

SOURCE FILE: `zfunc.c`

NAME

__add__, __ip__ , __mltadd__, __smlt__, __sub__, __zero__, __zadd__, __zconj__, __zip__ , __zmltadd__, __zmlt__, __zsub__, __zzero__ – core routines

SYNOPSIS

```
#include "machine.h"
/* or #include "matrix.h" */
void    __add__ (Real dp1[], Real dp2[], Real out[], int len)
double __ip__  (Real dp1[], Real dp2[], int len)
void    __mltadd__(Real dp1[], Real dp2[], double s, int len)
void    __smlt__(Real dp[],  double s,   Real out[], int len)
void    __sub__ (Real dp1[], Real dp2[], Real out[], int len)
void    __zero__(Real dp[],  int len)

#include "zmatrix.h"
void     __zadd__ (complex z1[], complex z2[],
                   complex out[], int len);
void     __zconj__(complex z[],   int len);
complex __zip__  (complex z1[], complex z2[],
                                 int len, int conj);
void     __zmlt__ (complex z1[], complex s, complex z2[],
                                 int len);
void     __zmltadd__(complex z1[], complex z2[], complex s,
                                 int len, int conj);
void     __zsub__ (complex z1[], complex z2[], complex out[],
                                 int len);
void     __zzero__(complex z[],   int len);
```

DESCRIPTION

These routines are the underlying routines for almost all dense matrix routines. Unlike the other routines in this library they do not take pointers to structures as arguments. Instead they work directly with arrays of `Real`'s. It is intended that these routines should be *fast.* **If you wish to take full advantage of a particular architecture, it is suggested that you modify these routines.**

The current implementation does not use any special techniques for boosting speed, such as loop unrolling or assembly code, in the interests of simplicity and portability.

Note that `zconj(z)`, referred to below, returns the complex conjugate of `z`.

The routine `__add__()` sets `out[i] = dp1[i]+dp2[i]` for `i` ranging from zero to `len-1`. The routine `__zadd__()` sets `out[i] = z1[i]+z2[i]` for `i` ranging from zero to `len-1`.

The routine `__ip__()` returns the sum of `dp1[i]*dp2[i]` for `i` ranging from zero to `len-1`. The routine `__zip__()` returns the sum of `z1[i]*z2[i]` for

`i` ranging from zero to `len-1` if `conj` is `Z_NOCONJ`, and returns the sum of `zconj(z1[i])*z2[i]` for `i` ranging from zero to `len-1` if `conj` is `Z_CONJ`.

The routine `__mltadd__()` sets `dp1[i] = dp1[i]+s*dp2[i]` for `i` ranging from zero to `len-1`. The routine `__zmltadd__()` sets `z1[i] = z1[i]+s*z2[i]` for `i` ranging from zero to `len-1` if `conj` is `Z_NOCONJ`, and sets `dp1[i] = z1[i]+s*zconj(z2[i])` for `i` ranging from zero to `len-1` if `conj` is `Z_CONJ`.

The routine `__smlt__()` sets `out[i] = s*dp[i]` for `i` ranging from zero to `len-1`. The routine `__zmlt__()` sets `out[i] = s*z[i]` for `i` ranging from zero to `len-1`.

The routine `__sub__()` sets `out[i] = dp1[i]-dp2[i]` for `i` ranging from zero to `len-1`. The routine `__zsub__()` sets `out[i] = z1[i]-z2[i]` for `i` ranging from zero to `len-1`.

The routines `__zero__()` and `__zzero__()` set `out[i] = 0.0` for `i` ranging from zero to `len-1`. These routines should be used instead of the macro `MEM_ZERO()` or the ANSI C routine `memset()` for portability, in case the floating point zero is not represented by a bit string of zeros.

EXAMPLE

```
MAT     *A, *B;
ZVEC    *x, *y;
Real     alpha;
  ......
/* set A = A + alpha.B */
for ( i = 0; i < m; i++ )
    __mltadd__(A->me[i],B->me[i],alpha,A->n);
/* zero row 3 of A */
__zero__(A->me[3],A->n);
/* quick complex inner product */
z_output(__zip__(x->ve,y->ve,x->dim,Z_CONJ));
```

SOURCE FILE: `machine.c, zmachine.c`

Chapter 5

Dense Matrix Factorisation Operations

The following routines are described in the following pages:

To use these routines use the include statement

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

and for the complex routines

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

NAME

BKPfactor, BKPsolve – Bunch–Kaufman–Parlett symmetric indefinite factorise and solve

SYNOPSIS

```
#include "matrix2.h"
MAT     *BKPfactor(MAT *A, PERM *pivot, PERM *blocks)
VEC     *BKPsolve(MAT *A, PERM *pivot, PERM *blocks,
                  VEC *b, VEC *x)
```

DESCRIPTION

The routine **BKPfactor()** forms *in situ* a symmetric indefinite factorisation of the matrix **A** of the form

$$P^T AP = MDM^T$$

where P is a permutation matrix, M is lower triangular, and D is block diagonal, with 1×1 or 2×2 blocks. The matrix P is represented by the permutation **pivot** and D_{ii} is a 1×1 block if and only if **blocks->pe[i] == i**; otherwise **blocks->pe[i]** is the index of the other row/column in the 2×2 block. After the routine the D and M factors are stored in **A** in compact form. This avoids the requirement for additional vectors or matrices for storage.

Note that **pivot** and **blocks** must both be non–NULL and **pivot != blocks** for both **BKPfactor()** and **BKPsolve()**.

The routine **BKPsolve()** solves the equation $Ax = b$ for x. The solve routine **BKPsolve()** is designed specifically to work with **BKPfactor()** as they operate on the same compact storage scheme. Note that the factorisation may succeed when the matrix **A** passed is singular, and that the solve routine may then fail, raising an **E_SING** error. The solve routine may be used *in situ* with **b == x**. If **x** is NULL or too small to hold the result, then a new vector is created of the appropriate size for storing the result. In either case the resulting solution vector is returned.

This factorisation routine, and the accompanying solve routine are derived from "Decomposition of a Symmetric Matrix" by J. Bunch, L. Kaufman and B. Parlett, *Numerische Mathematik* **27**, 95–109 (1976).

Errors will be raised if **A** or **pivot** or **blocks** are NULL, or if **A** is not square, or if the sizes of **A**, **pivot** or **blocks** are not compatible.

EXAMPLE

```
MAT     *A;
PERM    *pivot, *blocks;
VEC     *x, *b;
  ......
A = m_input(MNULL);
```

```
b = v_input(VNULL);
pivot = px_get(A->m);
blocks = px_get(A->m);
/* assuming A symmetric */
BKPfactor(A,pivot,blocks);
x = BKPsolve(A,pivot,blocks,b,VNULL);
```

SEE ALSO

`CHfactor()` and `CHsolve()`

SOURCE FILE: `BKPfactor.c`

NAME

CHfactor, MCHfactor, CHsolve, LDLfactor, LDLsolve – Cholesky factor and solve

SYNOPSIS

```
#include "matrix2.h"
MAT     *CHfactor(MAT *A)
MAT     *MCHfactor(MAT *A, double tol)
VEC     *CHsolve(MAT *A, VEC *b, VEC *x)
MAT     *LDLfactor(MAT *A)
VEC     *LDLsolve(MAT *A, VEC *b, VEC *x)
```

DESCRIPTION

Both `CHfactor()` and `LDLfactor()` factor the matrix `A` *in situ* and returns the factored matrix (in compact form). The Cholesky factorisation routine and the LDL^T routines both use only the lower triangular part of `A`, but the Cholesky factorisation routine fills the upper triangular part of `A` also.

These routines require that `A` is square. The Cholesky factorisation, in particular, requires that `A` be sufficiently positive definite (e.g. lowest eigenvalue of `A` is at least machine epsilon away from zero). If non–positive definiteness is detected during factorisation, then an `E_POSDEF` error will be raised. If you wish to catch such an error, see information on the `catch()` macro. If your matrix is indefinite, then it would be best to use the `BKPfactor()` and `BKPsolve()` routines.

The routine `MCHfactor()` computes a *modified* Cholesky factorisation. This is not a true Cholesky factorisation, but rather the Cholesky factorisation of $A + D$ where D is a diagonal matrix with non-negative entries. Whether the A matrix is modified in this way is determined by the `tol` parameter; the diagonal entry of the Cholesky factorisation is ensured to be $\geq \sqrt{\texttt{tol}}$. The D matrix is guaranteed to be zero in exact arithmetic if $u^T Au \geq \texttt{tol}\, u^T u$ for all u.

EXAMPLE

```
MAT     *A, *LLT, *LDL;
VEC     *b, *x;
double tol;
  ......
A = m_input(MNULL);
b = v_input(VNULL);
input("Input tol for modified Cholesky: ", "%lf", &tol);
LLT = m_copy(A,MNULL);
/* If A positive definite... */
CHfactor(LLT);
x = CHsolve(LLT,b,VNULL);
```

```
/* ...otherwise, get approximate solution... */
LLT = m_copy(A,MNULL);
MCHfactor(LLT,tol);        /* LLT now has factors of A + D */
MCHsolve(LLT,b,x);
/* ...or use LDL factorisation */
LDL = m_copy(A,MNULL);
LDLfactor(LDL);
LDLsolve(LDL,b,x);
```

SEE ALSO

`catch()` and `BKPfactor()`

SOURCE FILE: `CHfactor.c`

NAME

band2mat, mat2band – Band matrix utility routines

SYNOPSIS

```
#include "matrix.h"
MAT  *band2mat(BAND *bdA, MAT *out)
BAND *mat2band(MAT *A, int lb, int ub, BAND *out)
```

DESCRIPTION

The routine **band2mat()** creates an ordinary dense matrix **out** (a Meschach **MAT** structure) that is represented by the band matrix structure **bdA** represents. The returned matrix is square.

The routine **mat2band()** extracts the banded part of **A** with lower bandwidth **lb** and upper bandwidth **ub** and stores the result in the **BAND** structure **out**. The input matrix **A** must be square; if not an **E_SQUARE** error is raised.

For more information about band matrix data structures and storage patterns see the chapter on data structures.

Note that the conversion routines do *not* directly copy the **mat** field of the band structure. If you need efficient storage of band matrices, the routines **band2mat()** and **mat2band()** should probably be avoided.

SEE ALSO

bdLDLfactor() and **bdLUfactor()**.

SOURCE FILE: **bdfactor.c**

NAME

bdLDLfactor, bdLDLsolve – Band Cholesky factorise and solve

SYNOPSIS

```
#include "matrix2.h"
BAND  *bdLDLfactor(BAND *bdA)
VEC   *bdLDLsolve (BAND *bdA, VEC *b, VEC *x)
```

DESCRIPTION

These routines compute the LDL^T factorisation, and solve, a symmetric system of banded equations. These routines only use the lower band and the main diagonal of A.

After the call **bdLDLfactor(A)**, **A** is in factored form which compactly represents both the diagonal matrix D, but also the unit lower triangular matrix L.

If the matrix is exactly singular on factorisation, then an **E_SING** error is raised.

EXAMPLE

To extract a tridiagonal matrix from a dense matrix A, and to factorise and solve a system $Ax = b$:

```
MAT   *A;
VEC   *b, *x;
BAND *bdA;
  ......
/* Note: only need lower triangular part */
bdA = mat2band(A,1,0,(BAND *)NULL);
bdLDLfactor(bdA);
x = bdLDLsolve(bdA,b,VNULL);
```

BUGS

This method can be numerically unstable for matrices that are not positive definite.

The routine **bdLDLfactor()** does not test for symmetry.

SEE ALSO

bdLUfactor(), LDLfactor()

SOURCE FILE: **bdfactor.c**

NAME

LUfactor, LUsolve, LUTsolve, LUcondest, m_inverse, zLUfactor, zLUsolve, zLUAsolve, zLUcondest, zm_inverse – *LU* factorisation (Gaussian elimination) and solve

SYNOPSIS

```
#include "matrix2.h"
MAT      *LUfactor(MAT *A, PERM *pivot)
VEC      *LUsolve (MAT *A, PERM *pivot, VEC *b, VEC *x)
VEC      *LUTsolve(MAT *A, PERM *pivot, VEC *b, VEC *x)
double    LUcondest(MAT *LU, PERM *pivot)
MAT      *m_inverse(MAT *A, MAT *out)

#include "zmatrix2.h"
ZMAT     *zLUfactor(ZMAT *A, PERM *pivot)
ZVEC     *zLUsolve (ZMAT *A, PERM *pivot, ZVEC *b, ZVEC *x)
ZVEC     *zLUAsolve(ZMAT *A, PERM *pivot, ZVEC *b, ZVEC *x)
double    zLUcondest(ZMAT *LU, PERM *pivot)
ZMAT     *zm_inverse(ZMAT *A, ZMAT *out)
```

DESCRIPTION

The routines `LUfactor()` and `zLUfactor()` perform *LU* factorisation, which is otherwise known as Gaussian elimination with implicit scaled partial pivoting. The `zLUfactor()` performs the complex *LU* factorisation. The *LU* factors of `A` are stored in `A` in compact form. Once this is done, the routine `LUsolve()` can be used to solve equations of the form $Ax = b$ for x by forward and back substitution. For real matrices, the system $A^T x = b$ can be solved by using `LUTsolve()`, while for complex matrices $A^* x = b$ can be solved using `zLUAsolve()`. The code for a full factorisation and solving $Ax = b$ and $A^T y = b$ is:

```
/* set up A and b */
  ......
pivot = px_get(A->m);
x = v_get(A->n);
y = v_get(A->m);
LU = m_copy(A,MNULL);
LUfactor(LU,pivot);
x = LUsolve(LU,pivot,b,x);
y = LUTsolve(LU,pivot,b,y);
condition = LUcondest(LU,pivot);
```

A full description of Gaussian elimination with partial pivoting and its numerical behaviour can be found in a number of books, though we refer the reader specifically

to *Matrix Computations* by G.H. Golub and C. van Loan, North Oxford Academic, §§3.2–3.4, pp. 92–122, 2nd Edition (1989). The variant here is that scaling is used *implicitly*. That is, scaling is only used to decide which rows to swap during the partial pivoting process.

Note that the factorisation routine `LUfactor()` may succeed where the solve routine `LUsolve()` fails if, for example, `A` is singular. Also note that *LU* factorisation also succeeds when `A` is not even square, though this is a requirement for the success of `LUsolve()` or `zLUsolve()`. Errors are raised by `LUfactor()` or `zLUfactor()` if `A` or `pivot` is NULL, or if the size of `pivot` is less than the number of rows of `A`. Errors are raised by `LUsolve`, `LUTsolve()`, `zLUsolve()` or `zLUAsolve()` if these conditions occur, if `b` is NULL, or if `A` is not square. Then if `x` is NULL or too small to contain the result a new vector of the appropriate size is created. In either case the solution of $Ax = b$, `x`, is returned. The routines `LUsolve()`, `LUTsolve()`, `zLUsolve()` or `zLUAsolve()` may be used *in situ* (that is, with `b == x`) with version 1.2 or later.

The condition number (relative to the infinity norm) can be *estimated* using the routine `LUcondest()` or the routine `zLUcondest()`. This estimate is not guaranteed to under- or over-estimate the true condition number; however, it can usually be relied on to give an estimate correct to within an order of magnitude, which is usually all that is required.

The routines `m_inverse()` and `zm_inverse()` compute the inverse of `A` and returns the result in `out`. This is carried out using the LU factorisation routines. As is usually noted in numerical analysis texts, inverse matrices should rarely be computed. If a system of equations need to be solved, use the above code calling `LUfactor()` and `LUsolve()`, or `zLUfactor()` and `zLUsolve()` directly.

SOURCE FILE: `lufactor.c, zlufctr.c`

NAME

bdLUfactor, bdLUsolve – Band LU factorise and solve

SYNOPSIS

```
#include "matrix2.h"
BAND  *bdLUfactor(BAND *bdA, PERM *pivot)
VEC   *bdLUsolve (BAND *bdA, PERM *pivot, VEC *b, VEC *x)
```

DESCRIPTION

The routine **bdLUfactor()** computes the LU factorisation of a band matrix A with partial pivoting. This routine performs essentially the same calculations as **LUfactor()**. This operation is done *in situ* in **bdA**. Because partial pivoting is used, the (upper) bandwidth of the matrix being factorised increases. Specifically, the final upper bandwidth is $lb + ub$ where lb is the original lower bandwidth and ub is the original upper bandwidth.

The routine **bdLUsolve()** computes the solution to the banded system $Ax = b$ using the band matrix **bdA** in factored form. Note that only square matrices can be represented as banded matrices. This can be done *in situ* (**x == b**).

These routines raise an **E_NULL** error if either **bdA** or **pivot** is NULL.

EXAMPLE

To factor and solve $Ax = b$:

```
BAND *bdA;
PERM *pivot;
VEC  *x, *b;
  ......
/* set up bdA */
  ......
/* get a random right-hand side */
b = v_rand(v_get(A->mat->n));
/* factor bdA ... */
pivot = px_get(A->mat->n);
bdLUfactor(bdA,pivot);
/* ...and solve system */
x = v_get(b->dim);
bdLUsolve(bdA,pivot,b,x);
```

BUGS

Unless **bdA** is resized to its original size (which can be done very efficiently by **bd_resize()**) repeated calls to **bdLUfactor(bdA,...)** will result in the upper bandwidth increasing until it is $n - 1$ where **bdA** represents an $n \times n$ matrix.

SEE ALSO

`LUfactor()`

SOURCE FILE: `bdfactor.c`

NAME

QRfactor, QRCPfactor, QRsolve, QRCPsolve, QRTsolve, QRcondest, zQRfactor, zQRCPfactor, zQRsolve, zQRCPsolve, zQRAsolve, zQRcondest – QR factorisation and solve

SYNOPSIS

```
#include "matrix2.h"
MAT     *QRfactor(MAT *A, VEC *diag)
MAT     *QRCPfactor(MAT *A, VEC *diag, PERM *pivot)
VEC     *QRsolve(MAT *A, VEC *diag, VEC *b, VEC *x)
VEC     *QRTsolve(MAT *A, VEC *diag, VEC *b, VEC *x)
VEC     *QRCPsolve(MAT *A, VEC *diag, PERM *pivot,
                   VEC *b, VEC *x)
double   QRcondest(MAT *QR)

#include "zmatrix2.h"
ZMAT    *zQRfactor(ZMAT *A, ZVEC *diag)
ZMAT    *zQRCPfactor(ZMAT *A, ZVEC *diag, PERM *pivot)
ZVEC    *zQRsolve (ZMAT *A, ZVEC *diag, ZVEC *b, ZVEC *x)
ZVEC    *zQRAsolve(ZMAT *A, ZVEC *diag, ZVEC *b, ZVEC *x)
ZVEC    *zQRCPsolve(ZMAT *A, ZVEC *diag, PERM *pivot,
                    ZVEC *b, ZVEC *x)
double   zQRcondest(ZMAT *QR)
```

DESCRIPTION

The routines **QRfactor()** and **zQRfactor()** perform straightforward QR factorisations of **A**. The routine **zQRfactor()** computes the complex QR factorisation. For those unfamiliar with the terminology, the QR factorisation of A is a factorisation of the form

$$A = Q\,R$$

where R is upper triangular, and Q is orthogonal in the real case and unitary in the complex case. That is $Q^{-1} = Q^T$ and $Q^TQ = I$ in the real case, and $Q^{-1} = Q^*$ and $Q^*Q = I$ in the complex case. This factorisation exists whether or not A is singular or even square. The QR factorisation is performed using Householder transformations. (These are orthogonal matrices of the form $P_i = I - \alpha_i v_i v_i^T$ (real case) or $P_i = I - \alpha_i v_i v_i^*$ (complex case) where $\alpha_i = 2/v_i^T v_i$ (real case) or $\alpha_i = 2/v_i^* v_i$ (complex case).)

The routines **QRCPfactor()** and **zQRCPfactor()** perform a QR factorisation with column pivoting, which is a factorisation of the form

$$A\,\Pi^T = Q\,R$$

where additionally, Π is a permutation matrix. The Π matrix is represented by **pivot**. This is done exactly as for **QRfactor()** and **zQRfactor()** except for the pivoting.

Both of these factorisations are performed *in situ*, and store the Q and R factors compactly in `A` and `diag`. This compact form is used consistently within this package, and is essentially that of Golub and van Loan's *Matrix Computations*, §5.2, p. 212, 2nd edition, (1989), except that the v's are not normalised in this package. The dimensions of both `diag` must be at least as large as the minimum of the number of rows and columns of `A`.

Once `A` and `diag` contain this compact representation of the QR factors of A, we can use `QRsolve()` to solve systems of linear equations, and indeed, find least square error solutions to overdetermined systems of equations. See *Matrix Computations*, §1.4, p. 11 for an example. Indeed, the code

```
MAT     *QR;
  ......
QR = m_copy(A,MNULL);
QRfactor(QR,diag);
QRsolve(QR,diag,b,x);
```

finds the least squares solution x to

$$A\,x \approx b.$$

Similarly, if `QRCPfactor()` is to be used to factor A, then `QRCPsolve()` can be used to solve the least squares problem $Ax \approx b$. The code to do this is:

```
QR = m_copy(A,MNULL);
QRCPfactor(QR,diag,pivot);
QRCPsolve(QR,diag,pivot,b,x);
```

The corresponding operations for complex matrices simply requires prefixing the functions by a "`z`" and replacing `MAT` by `ZMAT`.

Note that in the real case, `QRTsolve(QR,diag,b,x)` solves the *underdetermined* problem $Ax = b$; that is, it computes the minimum 2-norm x that satisfies $Ax = b$ for $m \leq n$. The corresponding complex routine is `zQRAsolve(QR,diag,b,x)`.

The condition number of a matrix factored using either `QRfactor()` or `QRCPfactor()` can be estimated using `QRcondest()`:

```
printf("2-norm condition no. approx. = %g\n", QRcondest(QR));
```

The corresponding complex function is `zQRcondest()`. The function `QRcondest()` returns a *lower bound* for the least squares condition number of the factored matrix A

$$\kappa_{LS}(A) = \|A\|_2 \|A^+\|_2$$

provided A has full rank. If A is square, then this is exactly equal to the 2-norm condition number

$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2.$$

If the QR factors are exactly singular, then **QRcondest()** will return **HUGE** (**HUGE_VAL** for ANSI C).

The estimate is obtained by obtaining estimates for $\|R\|_2$ and $\|R^{-1}\|_2$. Note that Q and Π do not affect the 2-norm or least squares condition numbers. The estimate of $\|R^{-1}\|_2$ is found using the techniques of **LUcondest()** to obtain a vector y with unit ∞-norm such that $\|R^{-1}y\|_\infty$ is quite small. This is described in Golub and van Loan, 2nd Edition pp. 128–130, (1989). Then the power method is applied to the matrix $(R^TR)^{-1}$ (real case) or $(R^*R)^{-1}$ (complex case) a total of three times with initial vector y. The corresponding estimate of $\|R\|_2$ is obtained by a related method of finding a vector y with unit ∞-norm and $\|Ry\|_\infty$ quite large. The power method is applied to the matrix R^TR (real case) or R^*R (complex case). Taking square root of the estimated eigenvalues gives a lower bound to the 2-norm condition number of R.

A simple, and usually reliable, estimate of the rank of a matrix is to factor the matrix **A** using **QRCPfactor()** (real case) or **zQRCPfactor()** (complex case), and then to count the number of diagonal entries of **A** greater than a certain tolerance in magnitude. A more reliable approach is to use the Singular Value Decomposition. See **svd()**.

SEE ALSO

Householder routines **hhvec()**, **hhtrvec()**, **hhtrrows()** and **hhtrcols()**, **zhhvec()**, **zhhtrvec()**, **zhhtrrows()** and **zhhtrcols()**; **svd()**.

SOURCE FILE: **qrfactor.c**, **zqrfctr.c**

NAME

makeQ, makeR, zmakeQ, zmakeR – explicitly form Q and R factors

SYNOPSIS

```
#include "matrix2.h"
MAT     *makeQ(MAT *QR, VEC *diag, MAT *Qout)
MAT     *makeR(MAT *QR, MAT *Rout)

#include "zmatrix2.h"
ZMAT    *zmakeQ(ZMAT *QR, ZVEC *diag, ZMAT *Qout)
ZMAT    *zmakeR(ZMAT *QR, ZMAT *Rout)
```

DESCRIPTION

The routines **makeQ()** and **zmakeQ()** explicitly forms the real orthogonal Q or complex unitary Q of the QR factorisation from the compact representation in **QR** and **diag**. The result is stored in **Qout**. This routine may not be used to form **Qout** *in situ*.

The routines **makeR()** and **makeR()** explicitly forms the upper triangular R matrix of the QR factorisation. The result is stored in **Rout**. These two routines may be used *in situ*; that is, with **QR == Rout**. (Actually the routine just zeros the strictly lower triangular half of **QR**.)

If **Qout** or **Rout** is NULL or too small to contain the result then a new matrix is created and returned.

EXAMPLE

```
MAT     *A, *QR, *Q, *R;
VEC     *diag;
  ......
diag = v_get(A->m);
QR = m_copy(A,MNULL);
QRfactor(QR,diag);
Q = makeQ(QR,diag,MNULL);
R = makeR(QR,MNULL);
/* makeR(QR,QR); replaces QR with the R matrix */
```

SOURCE FILE: qrfactor.c, zqrfctr.c

NAME

givens, rot_cols, rot_rows, rot_vec, zgivens, zrot_cols, zrot_rows, rot_zvec – Givens' rotations routines

SYNOPSIS

```
#include "matrix2.h"
void     givens(double x, double y, Real &c, Real &s)
MAT     *rot_cols(MAT *A, int i, int k,
                  double c, double s, MAT *out)
MAT     *rot_rows(MAT *A, int i, int k,
                  Real c, Real s, MAT *out)
VEC     *rot_vec (VEC *x, int i, int k,
                  double c, double s, VEC *out)
```

```
#include "zmatrix2.h"
void     zgivens(complex x, complex y, Real &c, complex &s)
ZMAT    *zrot_cols(ZMAT *A, int i, int k,
                   double c, complex s, ZMAT *out)
ZMAT    *zrot_rows(ZMAT *A, int i, int k,
                   double c, complex s, ZMAT *out)
ZVEC    *rot_zvec (ZVEC *x, int i, int k,
                   double c, complex s, ZVEC *out)
```

DESCRIPTION

The routine **givens()** computes a pair (c, s) such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix} \tag{5.1}$$

where $c^2 + s^2 = 1$. The routine **zgivens()** computes a pair (c, s), c real and s complex where

$$\begin{bmatrix} c & -s \\ s^* & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix} \tag{5.2}$$

The matrix formed from the (c, s) pair is a real orthogonal or a complex unitary matrix, and is often referred to as a Givens' rotation. The other routines apply such an orthogonal matrix to vectors and matrices. The actual orthogonal matrix (from

`givens()`)that is applied to vectors and matrices is the matrix

$$J_{ik}(c,s) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \dots & s & & \\ & & \vdots & \ddots & \vdots & & \\ & & -s & \dots & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{5.3}$$

for the real case, and

$$J_{ik}(c,s) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \dots & -s & & \\ & & \vdots & \ddots & \vdots & & \\ & & s^* & \dots & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{5.4}$$

in the complex case.

The routine `rot_cols()` forms $AJ_{ik}(c,s)^T$ and stores the result in `out`. The routine `zrot_cols()` forms $AJ_{ik}(c,s)^*$ and stores the result in `out`.

The routines `rot_rows()` and `zrot_rows()` form $J_{ik}(c,s)A$ and stores the result in `out`.

The routines `rot_vec()` and `rot_vec()` form $J_{ik}(c,s)x$ and stores the result in `out`.

All of the `..rot_...()` routines may be used *in situ* and create a new vector or matrix if the `out` parameter is NULL or is too small to contain the result. The result of the application of the Givens' rotation is returned by each of the `..rot_...()` routines.

Note that $J_{ik}(c,s)^T = J_{ik}(c,-s)$ in the real case, and $J_{ik}(c,s)^* = J_{ik}(c,-s)$ in the complex case. This makes pre- and post-multiplying by transposes of $J_{ik}(c,s)$ easy.

EXAMPLE

```
int     i, k;
VEC     *x;
MAT     *A;
Real    c, s;
  ......
```

```
/* get Givens transformation */
givens(x->ve[i],x->ve[k],&c,&s);
/* apply to x */
rot_vec(x,i,k,c,s);
/* apply symmetrically to A */
rot_cols(A,i,k,c,s);
rot_rows(A,i,k,c,s);
```

BUGS

The `givens()` routine may result in overflow if the `x` and/or `y` parameters are of size greater than $\sqrt{\texttt{HUGE}}$.

SOURCE FILE: `givens.c, zgivens.c`

NAME

hhvec, hhtrcols, hhtrrows, hhtrvec, zhhvec, zhhtrcols, zhhtrrows, zhhtrvec – Householder transformation operations

SYNOPSIS

```
#include "matrix2.h"
VEC  *hhvec(VEC *x, unsigned i0, Real *beta,
            VEC *out, Real *newval)
MAT  *hhtrcols(MAT *A, int i0, int j0, VEC *hh, double beta)
MAT  *hhtrrows(MAT *A, int i0, int j0, VEC *hh, double beta)
VEC  *hhtrvec(VEC *hh, double beta, int i0, VEC *x, VEC *out)

#include "zmatrix2.h"
ZVEC *zhhvec(ZVEC *x, unsigned i0, Real *beta,
             ZVEC *out, complex *newval)
ZMAT *zhhtrcols(ZMAT *A, int i0, int j0, ZVEC *hh,
                double beta)
ZMAT *zhhtrrows(ZMAT *A, int i0, int j0, ZVEC *hh,
                double beta)
ZVEC *zhhtrvec(ZVEC *hh, double beta, int i0, ZVEC *x,
               ZVEC *out)
```

DESCRIPTION

The routines **hhvec()** and **zhhvec()** compute the parameters for a Householder transformation. In particular, given a vector x, a vector v (**== out**) and a real numbers β (**== beta**) and a (possibly complex) number *newval* are computed where the Householder transformation $P = I - \beta vv^*$ satisfies

$$Px = \begin{bmatrix} newval \\ 0 \end{bmatrix}. \tag{5.5}$$

Note that in the case of **x** a real vector, **newval** is real. Note also that **zhhvec()** computes the parameters for a complex vector.

The **x** parameter is not modified. The formulae used are taken from *Matrix Computations* by G. Golub and C. van Loan, p. 40, 1st Edition, (1983), §5.1, pp. 196–196, 2nd Edition, (1989).

If **out** is NULL or too small to hold the v vector, then a new vector is created to store the result. In either case, the result is returned. An error is raised if the **x** vector is NULL.

The routine **hhtrcols()** forms the product AP^T where P is the Householder transformation defined by hh and β (**== beta**). (That is, $P = I - \beta hh\, hh^T$.) The routine **zhhtrcols()** forms the product AP^* where P is the Householder transformation defined by hh and β (**== beta**). (That is, $P = I - \beta hh\, hh^*$.) All rows i with

$i <$ `i0` and columns j with $j <$ `j0` are ignored. The operations are performed *in situ* in `A`.

The routines `hhtrrows()` and `zhhtrrows()` form the product PA where P is the Householder transformation defined by hh and β. Again, all rows i with $i <$ `i0` and columns j with $j <$ `j0` are ignored. The operations is performed *in situ* in `A`.

Finally, the routines `hhtrvec()` and `zhhtrvec()` forms the vector Px where P is the Householder transformation defined by hh and β. The result is stored in `out`. If `out` is NULL or too small to hold the results of the operation, then a new vector is created of the appropriate size. In either case the result is returned.

SOURCE FILE: `hsehldr.c`

NAME

Dsolve, Lsolve, LTsolve, Usolve, UTsolve, zDsolve, zLsolve, zLAsolve, zUsolve, zUAsolve – Basic solve routines

SYNOPSIS

```
#include "matrix2.h"
VEC      *Dsolve (MAT *A, VEC *b, VEC *x)
VEC      *Lsolve (MAT *A, VEC *b, VEC *x, double diag)
VEC      *LTsolve(MAT *A, VEC *b, VEC *x, double diag)
VEC      *Usolve (MAT *A, VEC *b, VEC *x, double diag)
VEC      *UTsolve(MAT *A, VEC *b, VEC *x, double diag)

#include "zmatrix2.h"
ZVEC     *zDsolve (ZMAT *A, ZVEC *b, ZVEC *x)
ZVEC     *zLsolve (ZMAT *A, ZVEC *b, ZVEC *x, double diag)
ZVEC     *zLAsolve(ZMAT *A, ZVEC *b, ZVEC *x, double diag)
ZVEC     *zUsolve (ZMAT *A, ZVEC *b, ZVEC *x, double diag)
ZVEC     *zUAsolve(ZMAT *A, ZVEC *b, ZVEC *x, double diag)
```

DESCRIPTION

The routines `Dsolve()` and `zDsolve()` find and return the solution x of $Dx = b$ where D is the diagonal part of the matrix A (`== A`).

The routines `Lsolve()` and `zLsolve()` find and return the solution x of $Lx = b$ where L is the lower triangular part of A if `diag` is zero; L is the *strictly* lower triangular part of A with `diag` on the diagonal if `diag` is not zero. These routines use forward substitution.

The routines `LTsolve()` and `zLAsolve()` find and return the solutions x of $L^T x = b$ and $L^* x = b$ respectively where L is the lower triangular part of A if `diag` is zero; L is the *strictly* upper triangular part of A with `diag` on the diagonal if `diag` is not zero.

The routines `Usolve()` and `zUsolve()` find and return the solution x of $Ux = b$ where U is the upper triangular part of A if `diag` is zero; U is the *strictly* upper triangular part of A with `diag` on the diagonal if `diag` is not zero. These routines use back substitution.

The routines `UTsolve()` and `zUAsolve()` find and return the solution x of $U^T x = b$ and $U^* x = b$ respectively where U is the upper triangular part of A if `diag` is zero; U is the *strictly* upper triangular part of A with `diag` on the diagonal if `diag` is not zero. These routines use back substitution.

All of these routines may be used *in situ*; that is, they can be used with `b == x`.

If `x` is too small to contain the result then a new vector is created of the appropriate dimension. In either case the solution of the equations is returned.

The rationale behind the use of the **diag** parameter is that often, as in LU factorisation or LDL^T factorisation, the diagonal entry for L is implicit (usually one). The **diag** parameter enables these routines to be used generally, including for the results of QR factorisation, for example.

EXAMPLE

For solving $Ax = b$ using Cholesky factorisation, with only L:

```
MAT     *L;
VEC     *b, *x;
  ......
Lsolve(L,b,x,0.0);    /* use L's diagonal entries */
LTsolve(L,x,x,0.0);
```

For solving $Ax = b$ using LU factorisation with L unit lower triangular and no pivoting:

```
MAT     *L, *U;
VEC     *b, *x;
  ......
Lsolve(L,b,x,1.0);    /* L unit lower triangular */
Usolve(U,b,x,0.0);
```

SEE ALSO

LUsolve(), **zLUsolve()**, **CHsolve()**, **LDLsolve()**, **QRsolve()**, **zQRsolve()**

SOURCE FILE: **solve.c**, **zsolve.c**

NAME

LDLupdate, QRupdate – factorisation update routines

SYNOPSIS

```
#include "matrix2.h"
MAT     *LDLupdate(MAT *LDL, VEC *w, double alpha)
MAT     *QRupdate (MAT *Q, MAT *R, VEC *u, VEC *v)
```

DESCRIPTION

The routine **LDLupdate()** modifies the matrix **LDL** which is assumed to contain (in compact form) the LDL^T factorisation of a matrix A. The L matrix is the strictly lower triangular part of **LDL**, except with ones on the diagonal, while D is the diagonal of **LDL**, so that $A = LDL^T$. The matrix **LDL** is modified *in situ* so that if L_+ and D_+ denote the factors described by **LDL** *after* the routine, then

$$L_+ D_+ L_+^T = A + \alpha w w^T$$

where α is the value of **alpha** and w is **w**. The modified **LDL** matrix is returned.

The method used for updating the factorisation is given in "Methods for modifying matrix factorisations" by P. Gill, G. Golub, W. Murray and M. Saunders, *Mathematics of Computations*, **28**, pp. 505–535 (1974). The particular algorithm used is the algorithm C1 of their paper.

This routine may fail if $A + \alpha w w^T$ is not sufficiently positive definite; if this failure occurs, then an **E_POSDEF** error is raised.

The routine **QRupdate()** updates the QR factorisation of a matrix $A = QR$. Unlike the previous routine, this routine requires the explicit factors Q and R of A. These can be obtained from the compact form by means of the routines **makeQ()** and **makeR()**. If the matrices **Q** and **R** after the routine are denoted Q_+ and R_+ respectively, then

$$Q_+ R_+ = Q(R + uv^T) = A + (Qu)v^T.$$

Setting $u = Q^T w$ gives $Q_+ R_+ = A + wv^T$.

If **Q** is NULL, then only the **R** matrix is modified. The **R** matrix is returned.

The routine is based on one given in *Matrix Computations* by G. Golub and C. van Loan, pp. 437–443, 1st Edition (1983), pp. 593–594, 2nd Edition (1989).

EXAMPLE

Updating LDL^T factorisation:

```
MAT     *A, *LDL;
VEC     *u;
double alpha;
```

```
  ......
LDL = m_copy(A,MNULL);
LDLfactor(LDL);
  ......
/* A <- A + alpha.u.u^T */
LDLupdate(LDL,u,alpha);
```

Updating QR factorisation:

```
MAT     *A, *QR, *Q, *R;
VEC     *diag, *beta, *u, *v, *w;
  ......
QR = m_copy(A,MNULL);
QRfactor(QR,diag,beta);
Q = makeQ(QR,diag,beta,MNULL);
R = makeR(QR,MNULL);
  ......
/* A <- A + w.v^T */
u = v_get(Q->m);
u = vm_mlt(Q,w,u);
QRupdate(Q,R,u,v);
```

SOURCE FILE: `update.c`

NAME

schur, symmeig, trieig, zschur – Eigenvalue routines

SYNOPSIS

```
#include "matrix2.h"
MAT     *schur(MAT *A, MAT *Q)
VEC     *symmeig(MAT *A, MAT *Q, VEC *out)
VEC     *trieig(VEC *a, VEC *b, MAT *Q)

#include "zmatrix2.h"
ZMAT    *zschur(MAT *A, MAT *Q)
```

DESCRIPTION

The routine **schur()** computes the Real Schur decomposition of the matrix **A**. That is, it computes a block upper triangular matrix T and an orthogonal matrix Q such that

$$Q^T A Q = T.$$

The matrix T has diagonal blocks of sizes 1×1 and 2×2. The eigenvalues of these diagonal blocks are the eigenvalues of the original A matrix. The algorithm used to find the eigenvalues of A is the Francis QR algorithm. This algorithm is described in *Matrix Computations* by G. Golub and C. van Loan, pp. 231–236, 1st Edition (1983), pp. 377–381, 2nd Edition (1989).

The matrix **A** is overwritten with T, and if **Q** is not NULL and the correct size, then the Q matrix is stored in it.

The routine **zschur()** computes the complex Schur factorisation of **A**. That is, it computes an upper triangular matrix T and a unitary matrix Q such that

$$Q^* A Q = T.$$

The eigenvalues of A are the diagonal entries of T. The algorithm is a complex version of the Francis QR algorithm, and is, in fact, somewhat simplified in the complex case.

The routine **symmeig()** computes the eigenvalues of a *symmetric* matrix. It also computes an orthogonal matrix Q such that

$$Q^T A Q = \Lambda$$

where Λ is the diagonal matrix of eigenvalues. The algorithm used to find the eigenvalues of A consists of conversion to symmetric Hessenberg (symmetric tridiagonal) form, and then applying **trieig()** to obtain the eigenvalues of the tridiagonal matrix.

The eigenvalues are stored in **out** provided it is not NULL and is sufficiently large to contain all the eigenvalues. The vector containing the eigenvalues is returned. The matrix **A** is *not* overwritten.

The routine `trieig()` computes the eigenvalues of the symmetric tridiagonal matrix

$$T = \begin{bmatrix} a_0 & b_0 & & & \\ b_0 & a_1 & b_1 & & \\ & b_1 & a_2 & \ddots & \\ & & \ddots & \ddots & b_{n-2} \\ & & & b_{n-2} & a_{n-1} \end{bmatrix}. \tag{5.6}$$

The algorithm used is a "chasing" technique described in *Matrix Computations*, pp. 278–281, 1st Edition, pp. 421–424, 2nd Edition. It also accumulates the matrix Q such that Q^TTQ is diagonal. To compute the correct Q matrix, `Q` should be initialised to the identity matrix on entry to `trieig()`. (See `m_ident()`.)

The values in the `a` and `b` vectors are overwritten. At the end of the routine, `a` contains the eigenvalues, and the `b` vector is zero.

In all of the above routines, if the matrix `Q` is NULL on entry, then no calculation of the Q matrices is performed. This should speed up the routines somewhat if only the eigenvalues are needed.

EXAMPLE

Computing real Schur decomposition of (possibly) nonsymmetric A:

```
MAT    *A, *S, *Q, *X_re, *X_im;
VEC    *evals_re, *evals_im;
  ......
S = m_copy(A,MNULL);
Q = m_get(A->m,A->m);
schur(S,Q);
/* get eigenvalues (real, imaginary parts) */
evals_re = v_get(A->m);
evals_im = v_get(A->m);
schur_evals(S,evals_re,evals_im);
/* get eigenvectors (real, imaginary parts) */
X_re = m_get(A->m,A->m);
X_im = m_get(A->m,A->m);
schur_evecs(S,Q,X_re,X_im);
```

Computing eigenvalues and eigenvectors of a real symmetric matrix:

```
MAT    *A, *Q;
VEC    *evals;
  ......
evals = v_get(A->m);
evals = symmeig(A,Q,evals);
```

The Q matrix contains the eigenvectors.

Computing the eigenvalues and eigenvectors of a symmetric tridiagonal matrix defined by the vectors a (the diagonal entries) and b (the off-diagonal entries):

```
MAT     *Q;
VEC     *a, *b;
  ......
Q = m_get(a->dim,a->dim);
m_ident(Q);     /* must initialise Q */
trieig(a,b,Q);
/* a is now the vector of eigenvalues */
```

SEE ALSO

The Hessenberg routines in **hessen.c** and **zhessen.c**.

BUGS

It is up to the caller of **symmeig()** to ensure that the **A** matrix is symmetric. Symmetry of **A** is neither checked nor enforced in **symmeig()**.

SOURCE FILE: **symmeig.c, schur.c, zschur.c**

NAME

schur_evals, schur_vecs – Extracting eigenvalues and eigenvectors from the Schur form

SYNOPSIS

```
#include "matrix2.h"
void schur_evals(MAT *T, VEC *re_evals, VEC *im_evals)
MAT *schur_vecs(MAT *T, MAT *Q, MAT *X_re, MAT *X_im)
```

DESCRIPTION

Both of these routines assume that T is the matrix computed by the **schur()** routine; Q is the orthogonal matrix computed by **schur()**.

The routine **schur_evals()** compute the eigenvalues of a matrix T in Schur form (block diagonal with 1×1 or 2×2 blocks). The kth eigenvalue of $A = QTQ^T$ is **re_evals->ve[k]** $+i$**im_evals->ve[k]**. At worst this requires solving a series of quadratics; however, it does simplify the task of computing eigenvalues. Complex eigenvalues come in complex conjugate pairs.

The routine **schur_vecs()** computes the matrix $X =$ **X_re** $+ i$ **X_im** such that $X^{-1}AX$ is the diagonal matrix of eigenvalues where $T = Q^TAQ$ as computed by the **schur()** routine. The columns of X are computed by means of one step of inverse iteration using the eigenvalues as computed from the Schur form. This method is usually accurate provided the eigenvalues are not too close together. The computed kth column of X is real if the computed kth eigenvalue is real. The ordering of the columns is consistent with the ordering of the eigenvalues generated by **schur_evals()**.

EXAMPLE

See example for **schur()** above.

BUGS

It is a bit difficult to check that the computed X is correct if it is complex.

SEE ALSO

schur()

SOURCE FILE: **schur.c**

NAME

svd, bisvd – Singular Value Decomposition routines

SYNOPSIS

```
#include "matrix2.h"
VEC       *svd(MAT *A, MAT *U, MAT *V, VEC *out)
VEC       *bisvd(VEC *d, VEC *f, MAT *U, MAT *V)
```

DESCRIPTION

The routine **svd()** performs a complete Singular Value Decomposition (SVD) on the matrix A. That is, it computes orthogonal matrices U and V such that UAV^T is diagonal and the diagonal entries are called the *singular values* of the matrix A. The first $\min(m, n)$ singular values are stored in the **out** vector which is also returned. Note that the SVD is defined for nonsquare as well as square matrices.

If NULLs are passed for either or both **U** and **V**, then that orthogonal matrix will not be accumulated. This saves both time and space, if just the singular values are desired and not the U or V matrices. If **out** is NULL on entry to **svd()**, then a vector of the appropriate size is created to store the singular values, which is returned.

The SVD is computed by first transforming the matrix into a bidiagonal matrix (c.f. **schur()** where a matrix is transformed into Hessenberg form for eigenvalue calculations) and then applying **bisvd()**. If a matrix is already in bidiagonal form, then **bisvd()** can be called directly. The vector **d** contains the diagonal entries and **f** contains the super-diagonal entries. As for **svd()**, if NULLs are passed for either or both **U** and **V**, then that (or both) orthogonal matrix will not be accumulated. For correct results using **bisvd()**, you should initialise **U** and **V** to be identity matrices using **m_ident()** before calling **bisvd()**.

The rank of a matrix can be estimated by counting the number of singular values whose magnitude exceeds a specified tolerance. This tolerance for accurately computed matrices should probably be about 100 times **MACHEPS**; otherwise it should about an order of magnitude larger than the errors in the matrix.

The algorithm used follows *Matrix Computations* by Golub and van Loan, pp. 430–435, 2nd Edition (1989).

EXAMPLE

For computing the SVD of A:

```
MAT     *A, *U, *V;
VEC     *svdvals;
  ......
U = m_get(A->m,A->m);
V = m_get(A->n,A->n);
svdvals = svd(A,U,V,VNULL);
```

For computing the SVD of the bidiagonal matrix defined by d (the diagonal entries) and f (the super-diagonal entries):

```
MAT     *U, *V;
VEC     *d, *f;
  ......
U = m_get(d->dim,d->dim);
V = m_get(d->dim,d->dim);
m_ident(U);      /* must initialise U and V */
m_ident(V);
bisvd(d,f,U,V)
/* d now contains the singular values */
```

SOURCE FILE: `svd.c`

NAME

m_exp, m_poly, m_pow – Matrix exponentials, polynomials and powers

SYNOPSIS

```
#include "matrix2.h"
MAT * m_pow(MAT *A, int p, MAT *out)
MAT *_m_pow(MAT *A, int p, MAT *tmp, MAT *out)
MAT * m_exp(MAT *A, double eps, MAT *out)
MAT *_m_exp(MAT *A, double eps, MAT *out,
            int *qout, int *jout)
MAT *m_poly(MAT *A, VEC *a, MAT *out)
```

DESCRIPTION

The routine **m_pow** sets a matrix $A \in R^{n \times n}$ to the power p, where p can be any non-negative integer. (Use **m_inverse()** for negative p.) The result is placed in the matrix **out** = A^p. The routine is based on the binary powering algorithm (see Golub and Van Loan, *Matrix computations*, John Hopkins University Press, Baltimore, 2nd edition,1989). The algorithm requires at most $2\lfloor log_2(p) \rfloor n^3$ flops where n is the dimension of the matrix.
_m_pow it is a variant of the routine **m_pow** which uses **tmp** as a workspace matrix.

The routine **m_exp** computes an approximation of

$$e^A \approx I + A + A^2/2! + \ldots + A^q/q! + \ldots$$

using the Padé approximation

$$\exp(A) \approx R_{qq}(A) = D_q(A)^{-1} N_q(A)$$

where

$$N_q(A) = \sum_{k=0}^{q} c_k A^k, \quad D_q(A) = \sum_{k=0}^{q} c_k (-A)^k,$$

and

$$c_k = \frac{(2q-k)!q!}{(2q)!k!(q-k)!}.$$

The computed exponential is placed in **out**. The degree q is determined from an error tolerance **eps** given by the user. Padé approximation is good for A with a small norm, therefore this condition can be ensured by applying repeated squaring $(R_{qq}(A/2^j))^{2^j}$, where j is chosen so that $\|A/2^j\| \leq 1/2$. The Padé approximate can be more efficient by using special Horner regrouping techniques to evaluate matrix polynomial. The relative error of Padé approximate for a matrix with $\|A\| \leq 0.5$ can be estimated by

$$\frac{\|e^A - (R_{qq}(A/2^j))^{2^j}\|_\infty}{\|e^A\|_\infty} \leq \epsilon(q,q)\|A\|_\infty e^{\epsilon(q,q)\|A\|_\infty},$$

and $\epsilon(q,q) = 2^{3-(2q)}(q!)^2/((2q)!(2q+1)!)$.
In `_m_exp` the degree q is returned in `qout`, and j is returned in `jout`. The routines `m_exp` and `_m_exp` are based on the paper: *"Nineteen Dubious Ways to Compute The Exponential of the Matrix"*, SIAM Rev. 20(4), p.801–836, 1987 by C. Moler and C. Van Loan and the book G.H. Golub, C. Van Loan *"Matrix Computations"*, Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

`m_poly` evaluates the polynomial of a matrix A

$$p(A) = a_0 I + a_1 A + \ldots + a_q A^q,$$

where $a_0, a_1, a_2, \ldots, a_q$ are given by the vector `a` with $q =$ `a->dim-1`. The result is placed in `out`. The algorithm used to compute the matrix polynomials in the Padé approximation and in `m_poly` is based on the paper *"A note on the Evaluation of Matrix Polynomials"*, IEEE Transactions on Automatic Control 24 (1979), p. 209–228 by C.Van Loan. The paper describes a method that is faster and more memory efficient than the standard Horner's method.

SOURCE FILE: `mfunc.c`

NAME

`fft, ifft` – Fast Fourier Transform and inverse

SYNOPSIS

```
#include "matrix2.h"
void      fft(VEC *x_re, VEC *x_im)
void      ifft(VEC *x_re, VEC *x_im)
```

DESCRIPTION

The routine `fft()` performs a fast Fourier transform on the vector $x =$ `x_re` $+$ i`x_im`. The transform is computed *in situ*. It does require that the dimension of x is a power of two.

The routine `ifft()` performs the inverse fast Fourier transform of $x =$ `x_re` $+$ i`x_im`. As with `fft()` it is computed *in situ*, and the dimension of x must be a power of two.

SOURCE FILE: `fft.c`

Chapter 6

Sparse Matrix & Iterative Operations

The following routines are described in the following pages:

To use these routines use the include statement

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

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

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

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

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

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

NAME

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

SYNOPSIS

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

DESCRIPTION

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

The routine **sp_free()** deallocates all memory associated with the sparse matrix structure **A**. The macro **SP_FREE()** calls **sp_free()** to deallocate **A**, but also sets **A** to NULL, which makes this a safer way of freeing a sparse matrix.

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

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

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

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

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

EXAMPLE

```
SPMAT *A;
int    i, j, m, n;
  ......
/* get sparse matrix, with room for 5 entires per row */
A = sp_get(m,n,5);
  ......
sp_set_val(A,i,j,3.1415926);
  ......
/* double size of A matrix */
sp_resize(A,2*m,2*n);
  ......
/* remove entries of size <= 10^{-7} */
sp_compact(A,1e-7);
  ......
/* destroy A matrix */
sp_free(A)
```

SOURCE FILE: `sparse.c`

NAME

sp_copy, sp_copy2 – Spare matrix copy routines

SYNOPSIS

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

DESCRIPTION

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

The routine **sp_copy2()** copies **A** into **OUT**, using all allocated entries in **OUT** in doing so. In this way it avoids memory allocation and preserves the structure of the nonzeros of **OUT** as much as possible.

The routine **sp_copy2()** is especially useful in conjunction with the symbolic and incomplete Cholesky factorisation routines. The idea is that the symbolic Cholesky factorisation allocates all the necessary nonzero entries; if a matrix with the original nonzero pattern is to be factored, it can be copied using **sp_copy2()** into the symbolically factored matrix, and the incomplete Cholesky factorisation routine can then be used to factor the copied matrix without fill-in or memory allocation. See the manual entries on **spICHfactor()** and **spCHsymb()** for more details.

EXAMPLE

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

SEE ALSO

sp_get() and **sp_resize()**

SOURCE FILE: `sparse.c`

NAME

sp_get_val, sp_set_val – Access to entries of a sparse matrix

SYNOPSIS

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

DESCRIPTION

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

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

EXAMPLE

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

SEE ALSO

row_set_val()

BUGS

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

SOURCE FILE: **sparse.c**

NAME

sp_mv_mlt, sp_vm_mlt – sparse matrix–vector multiplication routines

SYNOPSIS

```
#include          "sparse.h"
VEC      *sp_mv_mlt(SPMAT *A, VEC *x, VEC *out)
VEC      *sp_vm_mlt(SPMAT *A, VEC *x, VEC *out)
```

DESCRIPTION

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

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

EXAMPLE

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

SOURCE FILE: **sparse.c**

NAME

`sp_col_access, sp_diag_access` – set up access paths

SYNOPSIS

```
#include "sparse.h"
SPMAT  *sp_col_access (SPMAT *A)
SPMAT  *sp_diag_access(SPMAT *A)
```

DESCRIPTION

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

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

The fields of `A` that are set up by `sp_col_access()` are the `A->start_row[]` and `A->start_idx[]` fields. The values `A->start_row[col]` and `A->start_idx[col]` give the first row, and index into that row where the first allocated entry of column `col`. The other fields set up by `sp_col_access()` are the `nxt_row` and `nxt_idx` fields of each `row_elt` data structure in the sparse matrix `A`. For a more thorough description of how these may be used, see §2.6.

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

EXAMPLE

Using the column access fields to chase the entries in

```
SPMAT *A;
int    i, j, idx;
SPROW *r;
row_elt *e;
  ......
/* set up A matrix */
sp_set_val(A,i,j,3.1415926);
  ......
sp_col_access(A);
/* chase column j of A */
i = A->start_row[j];
idx = A->start_idx[j];
while ( i >= 0 )
{
```

```
    r = &(A->row[i]);
    e = &(r->elt[idx]);
    printf("Value A[%d][%d] = %g\n", i, j, e->val);
    i = e->nxt_row;
    idx = e->nxt_idx;
}
```

Getting diagonal values:

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

BUGS

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

SOURCE FILE: `sparse.c`

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

EXAMPLE

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

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

SOURCE FILE: **sparse.c**

NAME

sp_foutput, sp_output – Sparse matrix output

SYNOPSIS

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

DESCRIPTION

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

The routine **sp_output()** is just a macro

```
#define sp_output(A)    sp_foutput(stdout,(A))
```

which sends the output to **stdout**.

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

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

can be represented as the sparse matrix with printed representation

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

EXAMPLE

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

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

SEE ALSO

`sp_finput()`, `sp_input()`

SOURCE FILE: `sparseio.c`

NAME

sp_finput, sp_input – Input sparse matrix

SYNOPSIS

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

DESCRIPTION

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

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

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

EXAMPLE

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

Example of interactive input session:

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

```
Row 2:
Enter <col> <val> or 'e' to end row
Entry 0: ....
  ......
```

BUGS

Does not allow more than a hundred entries per row.

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

SOURCE FILE: **`sparseio.c`**

NAME

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

SYNOPSIS

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

DESCRIPTION

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

The routine **sprow_get()** allocates and initialises a sparse row data structure (type **SPROW**) with memory for **maxlen** entries.

The routine **sprow_xpd()** reallocates the row **r** to allocate room for at least **newlen** entries. If the current length (**r->len**) is already at least size **newlen**, then the row's allocated memory is approximately double in size. For this routine and the some of the following **sprow_..()** routines the **type** parameter is **TYPE_SPROW** for a stand-alone sparse row, and **TYPE_SPMAT** for a sparse row in a sparse matrix (**SPMAT**) data structure.

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

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

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

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

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

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

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

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

EXAMPLE

Extracting a sparse matrix entry:

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

Shuffling a row:

```
/* build temporary sparse row r1
     containing shuffled entries of r */
r1 = sprow_get(10);
for ( idx = 0; idx < r->len; idx++ )
{
    e = &(r->elt[idx]);
```

```
    old_col = e->col;
    new_col = ......;
    sprow_set_val(r1,new_col,e->val);
    /* r1 will be expanded if necessary */
}
```

Expanding a temporary row:

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

Printing out a row as a separate structure for debugging:

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

SOURCE FILE: `sparse.c`

NAME

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

SYNOPSIS

```
#include "sparse2.h"
SPMAT  *spCHfactor(SPMAT *A)
VEC    *spCHsolve(SPMAT *LLT, VEC *b, VEC *out)
SPMAT  *spICHfactor(SPMAT *A)
SPMAT  *spCHsymb(SPMAT *A)
```

DESCRIPTION

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

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

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

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

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

```
/* Initialise pattern matrix */
  ......
spCHsymb(pattern);
for ( i = 0; i < num_matrices; i++ )
{   /* set up A matrix -- same nonzero pattern */
      ......
    sp_zero(pattern);
    sp_copy2(A,pattern);
    spICHfactor(pattern);
    /* set up b vector */
      ......
    spCHsolve(pattern,b,x);
      ......
}
```

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

BUGS

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

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

SEE ALSO

`sp_copy2()`, `sp_zero()`, `iter_cg()`, `iter_spcg()`

SOURCE FILE: `spCHfactor.c`

NAME

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

SYNOPSIS

```
#include "sparse2.h"
SPMAT *spLUfactor (SPMAT *A, PERM *pivot, double alpha)
SPMAT *spILUfactor(SPMAT *A, double alpha)
VEC   *spLUsolve (SPMAT *LU, PERM *pivot, VEC *b, VEC *x)
VEC   *spLUTsolve(SPMAT *LU, PERM *pivot, VEC *b, VEC *x)
```

DESCRIPTION

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

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

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

The routine **LUsolve()** solves the system $Ax = b$. The routine **LUTsolve()** solves the system $A^T x = b$. Both of these use the the matrix as factored by **spLUfactor()**. They can also be used *in situ* with **x == b**.

EXAMPLE

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

```
/* Set up A and b */
  ......
pivot = px_get(A->m);
x     = v_get(A->n);
y     = v_get(A->m);
spLUfactor(A,pivot,0.1);
x = spLUsolve(A,pivot,b,x);
y = spLUTsolve(A,pivot,b,y);
```

An example of the use of **spILUfactor()** will be given under the entry for **iter_cg()**, **iter_cgs()** and **iter_lsqr()**.

BUGS

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

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

SEE ALSO

spCHfactor(), **MACHEPS**, **LUfactor()**

SOURCE FILE: **spLUfctr.c**

NAME

spBKPfactor, spBKPsolve – sparse Bunch–Kaufmann–Parlett factorisation

SYNOPSIS

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

DESCRIPTION

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

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

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

EXAMPLE

A simple example of the use of these routines is

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

SEE ALSO

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

SOURCE FILE: `spbkp.c`

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

Indeed the type **Fun_Ax** is defined by

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

That is, an object of type **Fun_Ax** is a function (or equivalently a pointer to a function) which takes a (user-definable) parameter **Ax_par**, the vector x and the destination

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

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

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

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

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

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

SEE ALSO

iter_cg, **iter_cgs** and the other iterative methods

SOURCE FILE: **iter0.c**

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

The routine `iter_cg()` implements the conjugate gradient method. This is for symmetric positive definite matrices only, with symmetric positive definite preconditioners. This is a well-known method for solving such systems since the 1970's. The routine `iter_cg()` implements the standard (pre-conditioned) conjugate gradi-

ent method as presented in Golub and Van Loan's *Matrix Computations*, §10.3, 2nd Edition (1989).

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

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

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

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

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

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

In `iter_spcg()`, `LLT` is the sparse matrix structure containing an approximate Cholesky factorisation of A; If `LLT` is NULL then no preconditioning is used. In

`iter_spcgs()`, `r0` is the auxiliary vector. In `iter_spcgne()`, `iter_spcgs()`, `iter_spgmres()` and `iter_spmgcr()`, `B` is the (explicit) preconditioner. If `B` is NULL then no preconditioning is used. In `iter_splsqr()` there is no preconditioning. In `iter_spgmres()` and `iter_spmgcr()`, `k` is the dimension of the Krylov subspace used.

EXAMPLE

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

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

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

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

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

SEE ALSO

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

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

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXAMPLE

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

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

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

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

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

To go on to compute approximate eigenvectors:

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

SEE ALSO

`iter_get, ..., iter_gmres`

SOURCE FILE: `itersym.c iternsym.c`

Chapter 7

Installation and copyright

7.1 Installation

There are several different forms in which you might receive Meschach. To provide a shorthand for describing collections of files, the Unix convention of putting alternative letters in `[...]` will be used. (So, `fred[123]` means the collection `fred1`, `fred2` and `fred3`.) Meschach is available over Internet/AARnet via netlib, or at the anonymous ftp site `thrain.anu.edu.au` in the directory `pub/meschach`. There are five `.shar` files: `meschach[01234].shar` (which contain the library itself), of which `meschach0.shar` contains basic documentation and machine dependent files for a number of machines. Of the `meschach[1234].shar` files, only `meschach[12].shar` are needed for the basic Meschach library; the third `.shar` file contains the sparse matrix routines, and the the fourth contains the routines for complex numbers, vectors and matrices. There is also this `README` file that you should get directly, or extract it from `meschach0.shar`.

If you need the old iterative routines, the file `oldmeschach.shar` contains the files `conjgrad.c`, `arnoldi.c` and `lanczos.c`.

To get the library from netlib,

```
mail netlib@research.att.com
send all from c/meschach
```

There are a number of other netlib sites which mirror the main netlib sites. These include `netlib@ornl.gov` (Oak Ridge, TN, USA), `netlib@nac.no` (Oslo, Norway), `ftp.cs.uow.edu.au` (Wollongong, Australia; ftp only), `netlib@nchc.edu.tw` (Taiwan), `elib.zib-berlin.de` (Berlin, Germany; ftp only). (For anonymous ftp sites the directory containing the Meschach `.shar` files is `pub/netlib/c/meschach` or similar, possibly depending on the site.)

Meschach is available in other forms on `thrain.anu.edu.au` by ftp in the directory `pub/meschach`. It is available as a `.tar` file (`mesch12a.tar` for version 1.2a), or as a collection of `.shar` files, or as a `.zip` file. The `.tar` and `.zip` versions each contain the entire contents of the Meschach library.

To extract the files from the `.shar` files, put them all into a suitable directory and use

```
sh meschach0.shar
sh meschach1.shar
sh meschach2.shar
sh meschach3.shar
sh meschach4.shar
sh meschach5.shar
```

to expand the files. (Use one `sh` command per file; `sh *.shar` will not work in general.)

For the `.tar` file, use

```
tar xvf mesch12a.tar
```

and for the `.zip` file use

```
unzip mesch12a.zip
```

(Or use `pkunzip mesch12a.zip` if you have `pkunzip`.)

On a Unix system you can use the `configure` script to set up the machine-dependent files. The script takes a number of options which are used for installing different subsets of the full Meschach. For the basic system, which requires only `meschach[012].shar`, use

```
configure
make basic
make clean
```

For including sparse operations, which requires `meschach[0123].shar`, use

```
configure --with-sparse
make sparse
make clean
```

For including complex operations, which requires `meschach[0124].shar`, use

```
configure --with-complex
make complex
make clean
```

For including everything, which requires `meschach[01234].shar`, use

```
configure --with-all
make all
make clean
```

To compile the library in single precision, add the `--with-float` option to `configure` (with `Real` equivalent to `float`); e.g. use

```
configure --with-all --with-float
make all
make clean
```

Some Unix-like systems may have some problems with this due to bugs or incompatibilities in various parts of the system. To check this use `make torture` and run `torture`. In this case use the machine-dependent files from the `machines` directory. (This is the case for RS/6000 machines, the `-O` switch results in failure of a routine in `schur.c`. Compiling without the `-O` switch results in correct results.)

If you want to use the GNU `gcc` compiler, use the `configgnu` configuration script. This works just like the `configure` script, except that it will use `gcc` in preference to other compilers.

If you have problems using `configure`, or you use a non-Unix system, check the `MACHINES` directory (generated by `meschach0.shar`) for your machine, operating system and/or compiler. Save the machine dependent files `makefile`, `machine.c` and `machine.h`. Copy those files from the directory for your machine to the directory where the source code is.

To link into a program `prog.c`, compile it using

```
cc -o prog_name prog.c ...(source files)... meschach.a -lm
```

This code has been mostly developed on the University of Queensland, Australia's Pyramid 9810 running BSD4.3. Initial development was on a Zilog Zeus Z8000 machine running Zeus, a Unix workalike operating system. Versions have also been successfully used on various Unix machines including Sun 3's, IBM RT's, SPARC's and an IBM RS/6000 running AIX. It has also been compiled on an IBM AT clone using Quick C. It has been designed to compile under either Kernighan and Richie, (Edition 1) C and under ANSI C. (And, indeed, it has been compiled in both ANSI C and non-ANSI C environments.)

7.1.1 Installation on non-Unix systems

First look in the `machines` directory for your system type. If it is there, then copy the machine dependent files `machine.h`, `makefile` (and possibly `machine.c`) to the Meschach directory.

If your machine type is not there, then you will need to either compile "by hand", or construct your own makefile and possibly `machine.h` as well. The machine-dependent files for various systems should be used as a starting point, and the "vanilla" version of `machine.h` should be used. Information on the machine-dependent files follows in the next three subsections.

On an IBM PC clone, the source code would be on a floppy disk. Use

```
xcopy a:* meschach
```

to copy it to the `meschach` directory. Then `cd meschach`, and then compile the source code. Different compilers on MSDOS machines will require different installation procedures. Check the directory `meschach\machines` for the appropriate

"makefile" for your compiler. If your compiler is not listed, then you should try compiling it "by hand", modifying the machine-dependent files as necessary.

7.1.2 makefile

This is setup by using the `configure` script on a Unix system, based on the `makefile.in` file. However, if you want to modify how the library is compiled, you are free to change the `makefile`.

The most likely change that you would want to make to this file is to change the line

```
CFLAGS = -O
```

to suit your particular compiler.

The code is intended to be compilable by both ANSI and non-ANSI compilers. To achieve this portability without sacrificing the ANSI function prototypes (which are very useful for avoiding problems with passing parameters) there is a token `ANSI_C` which must be `#define`'d in order to take full advantage of ANSI C. To do this you should do all compilations with

```
#define ANSI_C 1
```

This can also be done at the compilation stage with a `-DANSI_C` flag. Again, you will have to use the `-DANSI_C` flag or its equivalent whenever you compile, or insert the line

```
#define ANSI_C 1
```

in `machine.h`, to make full use of ANSI C with this matrix library.

7.1.3 machine.h

Like `makefile` this is normally set up by the `configure` script on Unix machines. However, for non-Unix systems, or if you need to set some things "by hand", change `machine.h`.

There are a few quantities in here that should be modified to suit your particular compiler. Firstly, the macros `MEM_COPY()` and `MEM_ZERO()` need to be correctly defined here. The original library was compiled on BSD systems, and so it originally relied on `bcopy()` and `bzero()`.

In `machine.h` you will find the definitions for using the standard ANSI C library routines:

```
/*--------------------ANSI C--------------------*/
#include        <stddef.h>
#include        <string.h>
#define MEM_COPY(from,to,size)   memmove((to),(from),(size))
#define MEM_ZERO(where,size)     memset((where),'\0',(size))
```

Delete or comment out the alternative definitions and it should compile correctly. The source files containing `memmove()` and/or `memset()` are available by anonymous ftp from some ftp sites (try archie to discover them). The files are usually called `memmove.c` or `memset.c`. Some ftp sites which currently (Jan '94) have a version of these files are `munnari.oz.au` (in Australia), `ftp.uu.net`, `gatekeeper.dec.com` (USA), and `unix.hensa.ac.uk` (in the UK). The directory in which you will find `memmove.c` and `memset.c` typically looks like `.../bsd-sources/lib/libc/...`.

There are two further machine-dependent quantities that should be set. These are *machine epsilon* or the *unit roundoff* for double precision arithmetic, and the maximum value produced by the `rand()` routine, which is used in `rand_vec()` and `rand_mat()`. The current definitions of these are

```
#define MACHEPS 2.2e-16
#define MAX_RAND 2.147483648e9
```

The value of `MACHEPS` should be correct for all IEEE standard double precision arithmetic.

However, ANSI C's `<float.h>` contains `#define`'d quantities `DBL_EPSILON` and `RAND_MAX`, so if you have an ANSI C compiler and headers, replace the above two lines of `machine.h` with

```
#include <float.h>
/* for Real == float */
#define MACHEPS DBL_EPSILON
#define MAX_RAND RAND_MAX
```

The default value given for `MAX_RAND` is 2^{31}, as the Pyramid 9810 and the SPARC 2's both have 32 bit words. There is a program `macheps.c` which is included in your source files which computes and prints out the value of `MACHEPS` for your machine.

Some other macros control some aspects of Meschach. One of these is `SEGMENTED` which should be `#define`'d if you are working with a machine or compiler that does not allow large arrays to be allocated. For example, the most common memory models for MS-DOS compilers do not allow more than 64Kbyte to be allocated in one block. This limits square matrices to be no more than 90×90. Inserting `#define SEGMENTED 1` into `machine.h` will mean that matrices are allocated a row at a time.

7.1.4 machine.c

The core routines in `machine.c` as they presently are, are adequate on scalar processors. However, they are not designed to make best use of the recent super–scalar processors, or of vector processors. If you wish to make best use of these features of your machine in using the matrix library, then you should re-write these appropriately, possibly in assembly language. This has already been done to some extent, using "loop-unrolling":

```
sum0 = sum1 = sum2 = sum3 = 0.0;

len4 = len / 4;
len  = len % 4;

for ( i = 0; i < len4; i++ )
{
        sum0 += dp1[4*i]*dp2[4*i];
        sum1 += dp1[4*i+1]*dp2[4*i+1];
        sum2 += dp1[4*i+2]*dp2[4*i+2];
        sum3 += dp1[4*i+3]*dp2[4*i+3];
}
sum = sum0 + sum1 + sum2 + sum3;
dp1 += 4*len4;          dp2 += 4*len4;

for ( i = 0; i < len; i++ )
        sum += dp1[i]*dp2[i];
```

It may seem odd to use `dp1[i]*dp2[i]` instead `(*dp1++)*(*dp2++)` in the quest for speed, but optimising compilers cannot be trusted to do what you intend. The expression `dp1[i]*dp2[i]` was recognised for what it is, but `(*dp1++)*(*dp2++)` was not, by the RS/6000 optimising compiler. This may be a matter of taste by the compiler writers, so check it out on your own system before making any terminal decisions about what is fastest on your machine.

Also note that the `__zero__()` routine is defined from `machine.c`. This uses the `MEM_ZERO()` macro in `machine.h` in the standard release. However, if the double precision zero is **not** represented by a bitstring of zeros, the body of this routine would need to be replaced by

```
for ( i = 0; i < len; i++ )
    dp[i] = 0.0;
```

These are the only routines that need be modified, as essentially all other routines rely on these routines and on the `MEM_COPY()` macro, to provide adequate speed.

Such a re-writing effort may be worthwhile on, say, the i860 processor, where the speed of computing inner products in assembly (using special pipeline instructions) is an order of magnitude faster than general arithmetic operations. (See "Personal supercomputing with the Intel i860" by Stephen S. Fried, *Byte*, **16**, no. 1, Jan 1991, pp. 347–358 for an indication of possible performance.) Better use of the IBM RS/6000 super-scalar architecture has been obtained by re-writing some of the routines in `machine.c`. The speed of the core inner product routine on a 20MHz RS/6000 320 went from near the LINPACK rating of 7Mflops to about 20Mflops, half the theoretical peak speed of 40Mflops for a multiply and add each clock cycle.

7.2 Backward compatibility

As with any piece of software that is being modified, there is the problem of being able to use programs written for older versions of the library. This is especially important with Meschach 1.2 as the naming scheme has been made much more uniform and self-consistent. Names such as `get_vec()` (allocate vector) and `cp_vec()` (copy vector) have been changed to `v_get()` and `v_copy()` to be more consistent with `v_add()` (add vectors) and `m_mlt()` (multiply matrices).

The cost of this consistency is inconsistency with programs written for the older versions of Meschach. To deal with this, there is included in Meschach 1.2 a "compatibility" header file `oldnames.h`. Add the line

```
#include "oldnames.h"
```

at the beginning of files using pre-version 1.2 names. This header file consists of a collection of `#define`'s such as

```
#define get_vec    v_get
#define freevec    V_FREE
#define cp_vec     v_copy
```

The old iterative routines are still included in release 1.2a of Meschach (`pccg()`, `sp_pccg()`, `cgs()`, `sp_cgs()`, `lsqr()`, `sp_lsqr()`, `lanczos()`, `sp_lanczos()`, `lanczos2()`, `sp_lanczos2()`, `arnoldi()` and `sp_arnoldi()`). However, because of the new data structure for iterative methods, these are being phased out and can be replaced by the newer routines `iter_cg()`, `iter_spcg()` etc. The old iterative routines will not be supported in future.

7.3 Copyright

The copyright provisions for Meschach are intended to follow the lead of the Free Software Foundation in ensuring that the rights of people using and modifying the library cannot take away rights from others, while still enabling commercial use of the library. In that sense Meschach is not entirely "in the public domain". Notice that there is no intention to restrict the possible uses to which Meschach and parts of it are put, or to impede the work of programmers. The intent is only to make sure that users of any derivatives or modified versions of Meschach can still obtain access to the original code, and also to protect the reputations of ourselves and other programmers who modify or use Meschach.

Copyright subsists on the documentation and on the matrix library and source code for same and is held by David Edward Stewart and Zbigniew Leyk. It may be used free of charge provided the following rules are followed:

For legal purposes, in this section "the matrix library" shall refer to the "Meschach matrix library" as copyrighted by David Edward Stewart and Zbigniew Leyk.

1. Anyone to whom software is sold containing part or all of the matrix library in any form, whether modified or not, must have the matrix library source code made available to them in machine readable form at nominal cost.

2. Anyone distributing the library must ensure that copyright notices "Copyright (C) David E. Stewart and Zbigniew Leyk, 1986–1993" are published prominently along with the distribution in whatever form.

3. Anyone making changes to the library must prominently display this fact on any documentation relating to any use of the library (whether the use involves source or compiled code). Also, any such modification must be reflected in the routine `m_version()`, which prints out the current list of modifications to `stdout`.

4. Any code sold in object code form must include `m_version()` so that if the user so desires, he/she can determine what modifications and/or extensions to the original library have been made and who by.

Item (4) is deemed to be satisfied if there is a "version" command which executes the `m_version()` routine.

Finally, there is the usual statement about legal rights if something goes wrong in using the software. Trying to frame conditions under which Meschach can be guaranteed to work is unlikely to be a rewarding task for anyone to undertake, especially with the wide range of software and hardware systems it could work under. This is further complicated by the usual problems of numerical analysis where "proof of correctness" is not a realistic possibility and round-off errors are always present. Finally, due to the non-commercial nature of Meschach, there is unlikely to be any value to persons attempting to sue me for failure of the library in any situation.

Meschach IS PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, THE AUTHOR DOES NOT MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Chapter 8

Designing numerical libraries in C

The purpose of this chapter is to have a bit of a look "under the hood" to see how a library of routines in C can (and we believe, should) be built up. The philosophy here is to make use of the features of C to make programs more flexible and easier to write (and debug), while not sacrificing too much efficiency. There are other ways of designing numerical libraries, but this has been found to be a useful and flexible way of designing numerical libraries in C.

8.1 Numerical programming in C

Numerical and scientific programming has been traditionally associated with Fortran. Indeed, a great deal of software has been written in Fortran, in spite of its well known defects (lack of good data structures, lack of strong typing, reliance on "GOTO", poor lexical characteristics, clumsy input/output). This has led to the "historical" defense of Fortran: "There is so much already written in Fortran that we have to program in Fortran."

However, more sophisticated algorithms need more sophisticated data structures and more structured programs. Sparse matrix data structures and operations on them are one example of this. C is one of a number of languages that easily support such structuring. As well, C is a very flexible language, especially as regards memory management. While it is often argued that C is "merely a systems programming language", several aspects of C seem to indicate otherwise. For example, C has both single and double precision. Sometimes the argument is made that C is not suitable for numerical programming because single precision numbers are automatically converted to double precision whenever they are passed as arguments or used in expressions. This is no longer true in ANSI C. Even with the older C convention, the main drawbacks are the time spent converting between double and single precision numbers. Operations done entirely in double precision are immune to this inefficiency. It is, in any case, a better state of affairs than not having double or extended precision numbers as is the case with Pascal or the original version of Modula-2. Also, the standard Unix^{TM} mathematics library has not only the standard functions (exp, log, and the trigonometric

functions), but also Bessel functions, the Γ function and the error function. Admittedly, C does not have complex numbers, but this is a standard extension to C++.

8.1.1 On efficient compilers

The comment is sometimes made that Fortran *must* be more efficient than C. This is based on the fact that pre-Fortran 90 Fortrans are simpler languages, and that C has a rather more permissive structure. However, with modern compilers the difference in performance is usually fairly small, and is often non-existent. One of the reasons for this is that on many new machines compilers for different languages share common code-generation and optimisation parts. Indeed, the first NAG Fortran 90 compiler is actually a pre-processor that converts Fortran 90 into C — this is a sensible strategy because of the high quality and wide availability of many C compilers. The point that should be made is that efficiency is often a question of how much effort goes into developing the compilers. In the late 1970's the MACLISP compiler developed at MIT could produce machine code for compiled Lisp that rivalled Fortran in efficiency for numerical operations.

There are some inefficiencies that can be introduced in writing C code that would not appear in writing Fortran. But this is due to using a different style of programming. For example, overusing dynamic memory allocation can result in a great deal of overhead. (Beginners to programming in C can easily fall into a trap of writing code that spends most of its time allocating and deallocating temporary objects.) However, with a little care, this overhead can be kept to a negligible level while providing far more flexibility than is possible in Fortran 77.

8.1.2 Strategies for using C

The aspects of C that numerical programmers should make use of include

1. the ability to create self-contained data structures representing meaningful mathematical objects.
2. dynamic memory allocation and de-allocation of data structures and arrays, which often avoids the need for workspace arrays.
3. error and exception handling using `setjmp()` and `longjmp()`.
4. flexible input and output so that self-contained data structures can be read in and printed out.
5. use of pointers to represent user-defined objects whose characteristics are not known at compile time.

Self-contained data structures not only simplify argument lists, but can also be used for internal consistency checks to catch illegal operations. They should also make programs easier to understand in that they correspond closer to mathematical objects, and avoid the need to a plethora of additional length arguments and variables. By

using functions to perform most of the needed operations on these data structures, the chances of misusing the data structures can be greatly reduced.

Dynamic memory allocation and de-allocation not only avoids workspace arrays, but also avoids the need for the strategy of declaring the largest conceivable array sizes in local arrays. With this, memory can be used far more effectively.

A common error/exception handling mechanism means that the usual testing of "`IFLAG`" arguments can be avoided as well. A suitably structured mechanism can be used to provide a safe way of giving control back to the user if an error occurs. The users need to state what error they wish to "catch" and the code in which they wish to "catch" it; if an error occurs executing the code, control passes to the "catch" mechanism which can pass control back to the user's own code for handling the errors. Done properly, it can also provide a partial "backtrace" of the state of the active functions at the time of the error.

Input and output are, of course, very important. After all, a program without output is useless. More than this, by structuring input and output, output can be reused as input. Consider how often have you had to edit data just so that your program can use it as input?

Another aspect of structuring input is that comments can be incorporated into the input. Data, by itself, rarely means much. Including comments makes it much more intelligible to mere mortals. The flexibility of C's input and output has been used to do this.

User-defined objects (of any sort) can be handled by a combination of functions and pointers. Pointers to functions can be arguments to functions, and components of arrays or other data structures. This means that essentially arbitrary user-defined data structures can be used by code without knowing any of their characteristics at compile time. This style of programming has some of the flavour of object-oriented programming.

Meschach in various places makes use of all these aspects of C. We hope that you find this way of programming effective and efficient, not only in terms of CPU time, but your own (programming and debugging) time as well.

8.1.3 Non-C programmers start here!

Before going past this point, you really should read a book on C and programming in C. However, as there are undoubtedly non-C programmers who will want to follow the discussion in this chapter, here are some very brief notes which should help you understand the examples.

C programs consist of collections of functions, one of which is the main program (called "`main()`"). Routines consist of a header followed by a sequence of statements (the body of the routine) inside braces (`{...}`). Statements are either simple statements, which must end with a semi-colon (`;`), or compound statements, which is a collection of simple or compound statements bracketed by braces. The braces work very much like Algol, Pascal and Ada `begin...end` pairs. Comments in C have the form `/* ... */`.

Before a C program is compiled, it is passed through a *pre-processor*. Pre-processor directives must have a # as the first character on that line. The pre-processor can be used to define macros, to include files, and to delete code according to whether macros are defined. Standard header files are almost always included in C programs. Here is an example:

```
#include  <stdio.h>  /* standard input/output header file */
#include  "mydefs.h" /* uses file from current directory */
/* examples of macro definitions */
#define   max(a,b)   ((a) > (b) ? (a) : (b))
#define   DEBUG      TRUE
```

The basic data types in C include `int` ("integer"), `double` ("double precision floating point") and `char` ("character"). A declaration has the name of the data type before a list of variables, as in

```
int     i, j, idx;
double  alpha;
```

A *pointer* to a particular data type is declared by putting a * before the *variable* which is to be a pointer. For example, after the declarations

```
double  d, *pd, **ppd;
```

`d` is a `double`, `pd` is a pointer to `double`, and `ppd` is a pointer to a pointer to `double`.

Consistent with this, accessing the value pointed to by a pointer is simply a matter of putting a * before the variable. For example, the value pointed to by `pd` is `*pd`.

The reverse operation of finding a pointer that points to a variable is done by putting `&` before the variable; e.g. `pd = &d;` now makes `pd` point to the variable `d`.

Arrays are declared using square brackets such as

```
double   x[10];
```

This declares `x` to be an array with 10 entries. However, the starting index is *zero*, not one. So the valid entries of `x` are `x[0]`, `x[1]`, ... , `x[9]`. This is called *zero–relative indexing*. This may appear unusual at first, but is no barrier in practice.

Arrays and pointers are very similar; when arrays are passed to subroutines, only a pointer is passed, and pointers can be used like arrays. For example, `pd[0]` is equivalent to `*pd`; `pd[1]` is the double precision number next to `*pd`. This is called *pointer arithmetic* and can be easily abused. There are two important differences between arrays and pointers: (1) pointers are not necessarily associated with any usable piece of memory, while arrays are, and (2) array names cannot be assigned, but pointers can. So `pd = x;` is legal, but `x = pd;` is not.

Data structures containing (possibly) different kinds of objects are declared using `struct`. For example, complex numbers can be declared as

```
typedef struct cmplx { double real, imag; } complex;
```

(Here we have used `typedef` in order not to use the longer name `struct cmplx`.) Complex numbers can then be declared by

```
complex z1, z2;
```

Structures can be imbedded in structures, and recursive structures (such as linked lists) can be declared using pointers to that structure. For example, here is a linked list structure:

```
struct list { int contents;  struct list *next; };
```

The components of a data structure can be obtained by using ".". The real part of `z1` is `z1.real`. If `pz` is a pointer to a complex number, then the real part of the complex number pointed to is `(*pz).real`, which has the equivalent shorthand form: `pz->real`.

The control structures in C are familiar to most programmers — if–then–else, while, do–while (*cf* Pascal's repeat–until) and for loops. These have a straightforward syntax except for the for loop construct. Before these constructs are described, it should be noted that C has no Boolean or logical data type. Instead, zero or NULL is regarded as "False", while non-zero and non-NULL values are regarded as "True". The results of logical and relational operations are always integers `int`, with 1 representing "True". The comparison operators are equality test (`==`), inequality test (`!=`), and the usual numerical comparison operators (`<`, `>`, `<=`, `>=`). Logical operators include "logical and" (`&&`), "logical or" (`||`), and "logical not" (`!`). (There are also bitwise and, or, not and exclusive or operators.) Expressions involving `&&` and `||` are evaluated left–to–right and evaluation is "short-circuited" so that latter expressions are not evaluated if not needed. This is very useful to avoid performing invalid operations. For example,

```
ok = ( i < array_length ) && item_ok[i];
```

does not evaluate `item_ok[i]` if `i >= array_length`.

If statements have an optional `else` part and can be strung together.

```
if ( condition1 )
    statement1;
else if ( condition2 )
{   statement2; statement3;   }
```

While loops have the form "`while ( condition ) statement;`" or "`while ( condition ) {... }`". The do–while variant has the form "`do statement; while ( condition );`" or "`do { ... } while ( condition );`". The for loop in C is the most flexible and has the form

```
for ( initialisation; test; update )
    statement;
```

where "`statement;`" can be replaced by a compound statment. This is equivalent to a while loop:

```
initialisation;
while ( test )
{   statement;  update;     }
```

The for loop is most commonly used in a standard idiom:

```
for ( i = 0; i < array_length; i++ )
    ..... array[i] .....
```

The expression `i++` returns the value of `i` and then increments the value of `i` by one. (Here, of course, the *value* of the expression is ignored.) This is the *post-increment* operation; `i--` is the *post-decrement* operation. Preceding the variable with `++` or `--` *pre-increments* and *pre-decrements* the value of that variable. Other updates commonly used include incrementing the index by a different *stride*: `i = i+stride`, or with the shorthand `i += stride`.

Inside all loop constructs in C you can put `break` and `continue` statements. The `break` statement causes the loop to exit immediately; the `continue` statement causes control to be passed to just before the end of the loop.

All routines in C are functions. They might have side–effects and they might return `void` (so that the returned value is unusable), but they are functions. It is not necessary to do anything with the returned value, whether or not it has type `void`. Also, all function arguments are passed *by value* rather than *by reference*. Thus if you wish a function to set the value of a variable, you need to pass a pointer to that variable. For example, an integer swap routine would be called like this:

```
int     i, j;
swap(&i,&j);
```

If the type of the returned value from a function is not `int` (i.e. the standard integer type) then it should be declared before use. For example, a routine to add complex numbers together might be declared before use as

```
complex cadd();    /* adds two complex numbers */
```

If this is preceded by `extern` it means that the function is defined in another file. In ANSI C argument types can also be checked if you declare your functions using *function prototypes* such as

```
complex cadd(complex, complex);     /* or */
complex cadd(complex z1, complex z2);
```

Ther are two styles for defining a function: the old way, and ANSI C. Here is the old way:

```
complex cadd(z1, z2)
complex z1, z2;
{   complex z;
    z.real = z1.real + z2.real;
```

```
    z.imag = z1.imag + z2.imag;
    return z;  /* z is the returned value of cadd() */
}
```

And here is the ANSI C way:

```
complex cadd(complex z1, complex z2)
{   complex z;
    z.real = z1.real + z2.real;
    z.imag = z1.imag + z2.imag;
    return z;  /* z is the returned value of cadd() */
}
```

Functions can be passed as parameters, but what is actually passed is a *pointer to a function.* A pointer to a function can be used as other pointers can: arrays of pointers to functions are legal, as are structures containing pointers to functions. Here is declaration of a pointer to a function returning a `double`:

```
double (*f)();
```

Or using ANSI C, the types of the argument(s) can be included:

```
double (*f)(double);
```

Then assigning `f = exp;` is perfectly valid.

8.2 The data structures

C allows for extensive use of data structures. The `struct` and `typedef` facilities provide means whereby heterogeneous structures and primitive types can be combined and used together. As such they provide a *static* way of describing the data structure; they define the way things are stored. Equally important to the way things are stored, is the question of how such information is *used.* This is the *dynamic* part of the data structure. While C is not really set up to deal with complete formal descriptions of both the static and dynamic aspects of a data structures in the way object-oriented languages (such as SmallTalk and C++) are, we can go part way by providing functions that do at least the basic operations on the data structures.

8.2.1 Pointers to struct's

One approach that we have taken throughout the library is to pass only *pointers* to the actual `struct`'s. Passing the actual `struct`'s is useful for relatively small objects, but we believe it is inappropriate to do this for large objects and for objects which contain pointers to allocated memory. For example, complex numbers

```
typedef struct { double real, imaginary; } complex;
```

should be passed as single entities, while vectors

```
typedef struct { int dim, ...; double *ve; } VEC;
```

should not.

Why should this distinction be made?

1. Passing large structures is less efficient.
2. Copying the `struct` itself will only copy the pointers in the `struct`, **not what those pointers are pointing to.**

The second item notes that only a *shallow copy* is made by an assignment of a `struct`. For example, the following code does not do a true copy (at least it is usually not what the writer intends). ***Do not do this!***

```
VEC  x, y;
  ......
y = x;  /* this is an error in pre-ANSI C */
y.ve[1] = 3.0;
/* now x.ve[1] is also 3.0 */
```

Pointers can be copied, but here it is clear that its effect is not a deep copy.

```
VEC  *x, *y;
  ......
y = x;            /* y and x now point to the same place */
y->ve[1] = 3.0;
/* now x->ve[1] is 3.0 */
```

It is only with C++ that assignment can be forced to result in a deep, rather than a shallow, copy.

8.2.2 Really basic operations

Some operations are so basic that it is absolutely vital that they are implemented first. They are (in order):

1. Allocation and initialisation.
2. Output.
3. De-allocation.
4. Copying.

You might find it strange that output routines appear so soon. However, one thing is sure about developing data structures: you will want to debug them.

Writing allocation and initialisation routines is not difficult, but you should use the discipline that all returned values from `malloc()`, `calloc()` and `realloc()` are

checked. Also, check that the parameters passed make sense. If something goes wrong at this level it is unlikely that you can do much sensible. Passing control to an error handler, such as the `error()` macro does, is probably the most sensible thing to do here. Here is a hypothetical `struct` and the code to do (some) of the allocation and initialisation:

In the file `foo.h` we define the data structure and the new type `foo`:

```
typedef struct { int size; ... double *array; } foo;
```

In the file `foo.c` the basic operations are defined:

```
#include  "foo.h"
  ......
foo *get_foo(size)
int size;
{
    foo *my_foo;

    if ( size <= 0 )
        error(E_BOUNDS,"get_foo");
    /* get foo struct first */
    my_foo = (foo *)calloc(1,sizeof(foo));
    if ( my_foo = (foo *)NULL )
        error(E_MEM,"get_foo");
    /* now set up pointers */
    my_foo->array = (double *)calloc(size,sizeof(double));
    if ( my_foo->array = (double *)NULL )
        error(E_MEM,"get_foo");
    my_foo->size = size; /* now it is safe to set the size */
      ......
    return my_foo;
}
```

The function call `calloc(num_elts, size_elts)` allocates a block of memory for `num_elts` blocks of size `size_elts` characters. What is returned is a pointer to the allocated memory. If `calloc()` returns a NULL pointer, then this indicates that there is insufficient memory. The returned value of `calloc()`, `malloc()` and `realloc()` should always be checked before use. If an error occurs, then the `error()` macro is called, which raises an error at this point, and no further code in this function is executed.

The Meschach macros `NEW(type)` and `NEW_A(num,type)` in `matrix.h` simplify writing this sort of code:

```
if ( (my_foo = NEW(foo)) == (foo *)NULL )
    error(E_MEM,"get_foo");
  ......
```

```
if ( (my_foo->array = NEW_A(size,double)) == (double *)NULL )
    error(E_MEM,"get_foo");
  ......
```

De-allocation should be done using the function `free()` in the reverse order:

```
void free_foo(my_foo)
foo  *my_foo;
{
    if ( my_foo == (foo *)NULL )
        return;
      ......
    if ( my_foo->array != (double *)NULL )
        free(my_foo->array);
    free(my_foo);
}
```

There is not much more error checking that can be done at this stage. Checking that memory heaps are not corrupted can only be part of the design of the memory allocator, not the data structure or its routines.

Note that only pointers to memory that has been allocated by `calloc()`, `malloc()` or `realloc()` can be de-allocated using `free()`, and this can only be done once. Common errors are to try freeing memory more than once.

8.2.3 Output

Output should be structured but human readable. Usually we will want to be able to read the output back in later, so we should try to make the output reasonably machine-readable as well. (Writing input routines is usually much harder and more complex.) Hence the output should contain fore-warnings about what is coming, and how big it is before we get to it. It should also be possible to direct the output to any file or stream that we choose.

In the `foo` example,

```
void fout_foo(fp,my_foo)
FILE *fp;
foo  *my_foo;
{
    int i;

    fprintf(fp,"Foo: ");
    if ( my_foo == (foo *)NULL )
    {
        fprintf(fp,"NULL\n");
        return;
    }
```

```
    fprintf(fp,"size: %d\n",my_foo->size);
      ......
    fprintf(fp,"array: ");
    for ( i = 0; i < my_foo->size; i++ )
    {   /* no more than 6 items on a line */
        if ( (i % 6) == 5 || i == my_foo->size - 1 )
            fprintf(fp,"%g\n",my_foo->array[i]);
        else
            fprintf(fp,"%g ",my_foo->array[i]);
    }
}
```

(Actually, returning `my_foo` at the end would be useful behaviour, although we haven't done this in Meschach.)

Note that care is taken to treat the NULL case separately so that this will not result in failure; instead the message "`Foo: NULL`" is printed. For a proper allocated and initialised the output might look something like this:

```
Foo: size: 10
  ......
array: -3.7 2.5 3.141592 2.2 -1
1.5345 101 25.2321 -3.2 2.5
```

Writing an input routine to read this in is simplified because it can see how big to make the `array` before it has to read any of it in. Writing a routine to output every bit of the `foo` structure (even though most users won't want it) is often useful for debugging purposes. This can be done by writing an additional `foo_dump()` function.

8.2.4 Copying

The purpose of these routines is to provide a *deep copy* which copies all the component parts as well as the `struct` itself. There are two styles of doing this; one is to return a completely new `struct`, created and initialised, and the other is to copy the data structure into an already allocated and initialised one. One way to do both in one routine is to check the target structure pointer; if it is `NULL` then a new target structure should be created:

```
foo *cp_foo(from,to)
foo *from, *to;
{
    int   i;
      ......
    if ( from == (foo *)NULL )
        error(E_NULL,"cp_foo");    /* can't copy NULLs */
    if ( to   == (foo *)NULL )
```

```
        to = get_foo(from->size); /* create a new foo */
    else if ( to->size < from->size )
        /* make sure target is big enough */
        to = foo_resize(to,from->size);

    /* now do copying */
      ......
    for ( i = 0; i < from->size; i++ )
        to->array[i] = from->array[i];
      ......
}
```

The results of using `cp_foo()` can be used without checking as when a failure occurs, there is a call of the `error()` macro which invokes the error handling code. Once the checking is done, the actual copying can proceed as a straightforward loop. The efficiency of copying routines can be improved by using specialised copying routines such as `bcopy()` for BSD, or `memmove()` for ANSI C.

8.2.5 Input

Although this is not one of the "really basic" routines, they are useful and even important. Also, they are also trickier than output routines to write well.

It has been observed that in many software systems that the overall complexity of the code is usually dominated by the user interface. Writing a numerical library avoids a lot of that, and getting other programs/libraries to do your input/output is often a good idea. (Writing routines to output matrices in MATLAB save/load format means that you can use MATLAB to produce three-dimensional plots of "matrices".) However, writing input routines often cannot be avoided, and can also be useful for debugging purposes.

The input and output that is used by Meschach is all character-based. Fancy window-based input/output could also be done, but there the problem is more about standards and the many different ways of graphically displaying and inputting matrices and vectors.

There are two styles of input in Meschach. Interactive (from a "tty" in Unix jargon), or "batch" from a file or other input stream. Interactive input has fewer design rules than batch input, but still can be challenging to write well. (A fully featured input routine would really be an editor.) The basic design rules for batch input are:

1. The format produced by the output routine can be input.

2. Comments which begin with a "#" and continue to the end of the line are ignored.

Writing interactive input has a number of traps. For example, the following code looks fairly respectable:

```
int size = -1;
```

```
  ......
do {
    printf("Input size: ");
} while ( fscanf(fp,"%d", &size) != 1 || size <= 0 )
```

The idea here is that the loop is with the prompt `Input size:` is redisplayed until `size` is correctly scanned as input, and is positive. Note that the call to `scanf()` *must* take place before the test `size <= 0` is evaluated. The variable `fp` is the *file pointer* which indicates from which file `fscanf(fp,...)` reads data. The function `fscanf()` ignores leading and trailing blanks, so inserting leading or trailing blanks does not affect the code.

However, what happens if you input the letter "`x`"? The `fscanf()` routine would read the letter, realise that it cannot be part of a number, and put it back on the input stream. The result: the loop is an infinite loop giving the user no chance to take control as nothing beyond the "`x`" is read.

The way to avoid this is to use line–by–line input by means of `fgets()`. Also output to `stderr` instead of `stdout` means that output file re-direction does not prevent interactive input. Here is a better approach.

```
int size;
  ......
do {
    fprintf(stderr,"Input size: ");
    if ( fgets(line,MAXLINE,fp) == (char *)NULL )
        error(E_INPUT,"in_foo");
} while ( sscanf(line, "%d", &size) != 1 || size < 0 );
```

The idea here is to input a line into a character array, and then scan the character array. Since every failure results in a new line being read, it cannot get stuck. Failure to read a line from the file results in an error being raised so end–of–file situations are caught.

When interactively inputting arrays, it is a good idea to let the user (at the keyboard) know where you are in the array at all times. If the user makes a mistake, then re-display the prompt including the current position. Allowing the user to go back to correct mistakes, and then go forward again, helps to prevent the user from becoming too frustrated at the system. And what could be more frustrating than having hit the return key just after you realise that you made a mistake near the end of a large matrix with over a hundred entries? Here is how the code for inputting the entries of a vector allows for forward and backward motion, and printing out old values where necessary.

```
for ( i = 0; i < dim; i++ )
    do {
      redo:
        fprintf(stderr,"entry %u: ",i);
        if ( !dynamic )
            fprintf(stderr,"old %14.9g new: ",vec->ve[i]);
        if ( fgets(line,MAXLINE,fp) == NULL )
```

```
            error(E_INPUT,"ifin_vec");
        if ( (*line == 'b' || *line == 'B') && i > 0 )
        {    i--;    dynamic = FALSE;     goto redo;     }
        if ( (*line == 'f' || *line == 'F') && i < dim-1 )
        {    i++;    dynamic = FALSE;     goto redo;     }
    } while ( *line == '\0' ||
              sscanf(line, "%lf", &vec->ve[i]) < 1 );
```

By the way, there is only one other place (outside the input routines) where a `goto` is used. Note also that an end-of-file signal will result in an error being raised.

The batch input parts of input routines are relatively easy to write. Comments can be skipped over by using `skipjunk(fp);` and if an error in the input occurs, then an error should be raised. There is no need to try to re-read the input stream. The error handler may try to skip the input until some marker is reached, but this is up to the programmer. Apart from that, all that is necessary is to have enough `fscanf()` calls to skip over the markers that are printed by the output routine. For example, `fscanf(fp,"Foo:");` will skip over the header produced by the `fout_foo()` routine above. Ignoring the return value of `fscanf()` for *this purpose* is acceptable — the result is a less temperamental input routine.

8.2.6 Resizing

Resizing objects is an operation that cannot be done to all data structures, such as those involving hairy user-defined objects and functional arguments. However, allocated arrays can be resized by means of the standard library function `realloc()`. There is a macro `RENEW(var, num, type)` in `matrix.h` which calls `realloc()`, and also handles `NULL` values of `var`. For example, resizing a `foo` data structure could be done something like this:

```
foo *foo_resize(my_foo, new_size)
foo *my_foo;
int new_size;
{
    double *temp;
    if ( my_foo == (foo *)NULL )
        return get_foo(new_size);
    temp = my_foo->array;
    /* actual re-sizing operation: */
    temp = RENEW(temp, new_size, double);
    if ( temp == (double *)NULL )     /* check for failure */
        error(E_MEM,"foo_resize");
    my_foo->array = temp;
    my_foo->size  = new_size;
    return my_foo;
}
```

Note that the result of `RENEW()` is checked immediately. Also, resetting the size is the last thing that is done.

8.3 How to implement routines

The basic rule that should be used is that the more operations that a user wants to use that are provided by the designer of the library, the less the user has to do and the less likely it will be that the user will make mistakes. Finding a good set of kernel operations for a particular data structure is a crucial problem in good library design. Sometimes, not only the obvious operations should be supplied, but also "support" operations should be implemented. (An example of the need for this can be seen with sparse matrices where there are support routines for setting up the column access paths.) The more complex the data structure, the more support routines you will probably need to write to be able to effectively and efficiently use that data structure. Efficiency will often lead to additional routines. For example, even though there are routines for adding vectors `v_add()`, and for computing scalar multiples of vectors `sv_mlt()`, it is more efficient to use the "multiply and add" routine `v_mltadd()` than to use the add and scalar multiply routines separately.

8.3.1 Design for debugging

Arguments should be checked for consistency, except possibly at the lowest level(s) of the library. At the lowest levels it may not be worth doing the checking and losing efficiency. But at almost all other levels which deal with more time-consuming and complex operations, it is well worth checking the arguments. You probably should check at least that

1. none of the input arguments are `NULL`.

2. the sizes of the arguments are compatible.

For example, in a function `foo_bar()`, the following checking should be done:

```
foo *foo_bar(my_foo1, my_foo2, result_foo)
foo *my_foo1, *my_foo2, *result_foo;
{
    /* check that operands are not NULL */
    if ( my_foo1 == (foo *)NULL || my_foo2 == (foo *)NULL )
        error(E_NULL,"foo_bar");
    /* check that they have compatible sizes */
    if ( my_foo1->size != my_foo2->size )
        error(E_SIZES,"foo_bar");
      ......
}
```

Detailed checking for self-consistency of a data structure is not usually necessary; if the programmer using the library is using it properly, then they shouldn't have much opportunity to mess up the data structure. Of course, the library shouldn't mess up the data structure either. If debugging using a good and thorough output routine is not sufficient to debug the library, then maybe a function that checks internal consistency should be written. However, the checking function would probably be most effective when used to help to debug the library than as an automatic argument check.

An example of detailed argument checking that is not worthwhile is checking that a matrix is symmetric before a Cholesky factorisation. If detailed checking of this kind is wanted, then a checking routine would be written, such as a currently non-existent `chk_symm()` function.

There are a number of macros that have been written for error handling which work in conjunction with the function `ev_err()` (short for "evaluation error") in the file `err.c`. The first is clearly the `error()` macro, which calls `ev_err()` with the `__FILE__` and `__LINE__` macros so that the file and line number where the error was raised can be printed out. The file `err.c` and the error-handling macros in `matrix.h` are independent of the rest of the library, and can be used separately.

A tool that is useful for debugging is to use

```
tracecatch(code_to_execute,"function");
```

The effect of this macro is that if `code_to_execute` raises an error, then once the error is processed (which usually means printing out an error message) the error is re-raised at the place of the `tracecatch()`. If the body of each function (excluding the usual initial argument checks) is enclosed in a `tracecatch()`, then what is effectively a stack backtrace would be printed when an error occurs, indicating what functions were active when the error occurred.

A related macro is `catchall(code_to_execute,error_code)`. This macro executes `code_to_execute` normally, but if this raises an error, then `error_code` is executed. This can be used to print out particular information that might be the cause (or result) of the error. You can put a line containing

```
error(_err_num,"catchall");
```

at the end of `error_code` to re-raise the error, and continue the stack backtrace if desired.

For more information about designing for debugging, see §8.6 on debugging.

8.3.2 Workspace

In most Fortran libraries, routines using extra memory require workspace arguments to be passed to the routine. The programmer using the library has to pass a workspace array of a particular size (which the user has to work out before-hand). With C's memory allocation/de-allocation facilities this is not necessary in C, though sometimes it might be useful.

Passing workspace arrays adds to the complexity of using a function, and is usually a headache for the user. Getting the workspace size right is also a way in which errors can occur.

To avoid having to pass workspace arrays, there are two main approaches to making the necessary workspace available. The first is to allocate the workspace on entry (as soon as its size can be worked out) and deallocated on exiting the function. The second is to have a `static` local array which is first allocated and then reallocated.

The first approach keeps the memory available only for as long as is necessary. This is more efficient in memory, but less efficient in time as the workspace has to be reallocated every time the routine is called. The second approach keeps the workspace memory, and so is less memory efficient, but is more time efficient. In one sense, the two methods are two extremes of a range of "compromises" between memory efficiency and time efficiency.

Here's one way of setting up the second sort of internal workspace:

```
foo *foo_bar(...)
{
      ......
    static double *wkspace = NULL;
    static int    wksize = 0;
      ......
    new_wksize = .... ;
    if ( wkspace == (double *)NULL )
      wkspace = (double *)calloc(new_wksize,sizeof(double));
    else if ( wksize < new_wksize )
      wkspace = (double *)realloc(wkspace,
                                  new_wksize,sizeof(double));
    /* check results of calloc() or realloc() before use! */
    if ( wkspace == (double *)NULL )
      error(E_MEM,"foo_bar");
    wksize = new_wksize;
      ......
}
```

(Note that the initialisation of `wkspace` and `wksize` are unnecessary as un-initialised `static` variables are initialised to zero or `NULL`.) This sort of approach is even more convenient with self-contained data structures which can be resized as needed, such as the vectors in the Meschach library:

```
foo *foo_bar(...)
{
      ......
    static VEC *wkspace = VNULL;
      ......
    new_wksize = .... ;
```

```
    wkspace = v_resize(wkspace,new_wksize);
      ......
}
```

Both of these approaches for workspace have their limits.

However, in Meschach, the "compromise" between memory and time efficiency is put in the hands of the user. This involves "registering" workspace arrays so that they can be freed on request by a call outside of the function where the static workspace variable is defined. Registering a static variable is easy:

```
foo *foo_bar(...)
{
      ......
    static VEC *wkspace = VNULL;
      ......
    new_wksize = .... ;
    wkspace = v_resize(wkspace,new_wksize);
    MEM_STAT_REG(wkspace,TYPE_VEC);
      ......
}
```

Note that you can only register *static* variables. If you try to register an automatic variable, the program will most likely crash. There is no way that the variable can be checked for whether it is static or not.

There is a "workspace group number" or "mark" that must be set before (in the dynamic sense, not necessarily in the code sequence) a workspace variable is registered. When a static workspace variable is registered, it is "marked" as belonging to the current workspace group or "mark". This "mark" can be set by, for example,

```
mem_stat_mark(1);
```

This call is usually made in the main calling routine before any routines using static workspace variables are called. The "mark" can be changed by calling `mem_stat_mark()` with a new "mark" or "group number". All of the static workspace variables registered with a particular "mark" can be deallocated and their memory freed with a call `mem_stat_free(mark)`. Note that this unsets the "mark".

Examples of how the `mem_stat_..()` routines work are in chapter 2.

8.3.3 Incorporating user-defined types into Meschach

Meschach 1.2 provides a number of facilities to track memory usage and to control the allocation and deallocation of static workspace arrays. User-defined data structures can be incorporated into these mechanisms so that it can track memory usage and free up workspace variables for your own data structures.

Since related data structures are often defined together, the information about the data structures is passed to the `mem_info_...()` and `mem_stat_...()` routines

by a collection of arrays containing the names of the types, the `.._free()` functions for these data structures, and an array of `long`'s for storing information about the amount of memory used by the various data structures. This collection of arrays is called a *list*, and it describes a family of types. Each family of types known to Meschach has its own list number; the family of standard Meschach types has zero as its list number.

Here is an example taken from `memtort.c`. First there are the definitions:

```
/* the number of a new list */
#define FOO_LIST 2

/* type numbers */
#define TYPE_FOO_1    1
#define TYPE_FOO_2    2

/* new types */
typedef struct {
   int dim;
   int fix_dim;
   Real (*a)[10];
} FOO_1;

typedef struct {
  int dim;
  int fix_dim;
  Real (*a)[2];
} FOO_2;
```

The arrays which contain the information are:

```
char *foo_type_name[] = {
   "nothing",
   "FOO_1",
   "FOO_2"  };

#define FOO_NUM_TYPES \
   (sizeof(foo_type_name)/sizeof(*foo_type_name))

int (*foo_free_func[FOO_NUM_TYPES])() = {
   NULL,
   foo_1_free,
   foo_2_free  };

static MEM_ARRAY foo_info_sum[FOO_NUM_TYPES];
```

Note that the type number `TYPE_FOO_1` and `TYPE_FOO_2` correspond to the position their type names and `.._free()` functions have in the arrays. This list of types is made known to the Meschach routines by the call

```
mem_attach_list(FOO_LIST, FOO_NUM_TYPES, foo_type_name,
                foo_free_func, foo_info_sum);
if ( ! mem_is_list_attached(FOO_LIST) )
    printf("Error: list FOO_LIST is not attached\n");
```

which should be at the beginning of the `main(...)` routine.

Knowing that certain types exists is a start, but to track memory usage, the routines that perform memory allocation, deallocation and resizing need to keep the Meschach system informed about changing memory usage. For example, in `foo_1_get()`:

```
FOO_1 *foo_1_get(dim)
int dim;
{
   FOO_1 *f;

   if ((f = (FOO_1 *)malloc(sizeof(FOO_1))) == NULL)
     error(E_MEM,"foo_1_get");
   else if (mem_info_is_on())
   {
     mem_bytes_list(TYPE_FOO_1,0,sizeof(FOO_1),FOO_LIST);
     mem_numvar_list(TYPE_FOO_1,1,FOO_LIST);  /* 1 more */
   }
   f->dim = dim;
   f->fix_dim = 10;
   if ((f->a = (Real (*)[10])
              malloc(dim*sizeof(Real [10]))) == NULL)
      error(E_MEM,"foo_1_get");
   else if (mem_info_is_on())
      mem_bytes_list(TYPE_FOO_1,0,
                     dim*sizeof(Real [10]),FOO_LIST);

   return f;
}
```

The routine that actually notifies the Meschach system about the change in the *amount* of memory usage is `mem_bytes_list()`, and the routine that notifies Meschach about the *number* of allocated structures is `mem_numvar_list()`. For `mem_bytes_list()` the first argument is the type number, the second is the old size in bytes, the third is the new size in bytes, and the last parameter is the list number of the family of types. It is not important that the absolute values of old and new sizes are correct, other than being non-negative; rather it is the difference between them

that is important. For `mem_numvar_list()` the *change* in the number of allocated structures is passed.

The corresponding `.._free()` routine also needs to call `mem_byte_list()`:

```
int foo_1_free(f)
FOO_1 *f;
{
   if ( f != NULL) {
      if (mem_info_is_on())
      {
          mem_bytes_list(TYPE_FOO_1,
              sizeof(FOO_1)+f->dim*sizeof(Real [10]),0L,2);
          mem_numvar_list(TYPE_FOO,-1,2);  /* 1 less */
      }
      free(f->a);
      free(f);
   }
   return 0;
}
```

Similarly, `.._resize()` routines need to call `mem_bytes_list()` if there is any actual memory allocation, deallocation or resizing. If the argument is NULL, then the main `.._get()` routine should be called; otherwise there is no change in the number of `FOO_1` structures, and so there is no need to call `mem_numvar_list()`. Merely re-arranging the internal structure doesn't have to be reported via `mem_bytes_list()`.

User-defined data structures can be used as static workspace arrays, just like the standard Meschach data structures. They can be registered as workspace variables just like the standard Meschach data structures, except that the list number of the family of types needs to be given, and is positive. For example,

```
hairy1(...)
{
    static FOO_1 *f;   /* initially NULL */
      ......
    if ( ! f )  f = foo_1_get();  /* allocate if f NULL */
    /* ...or could use a .._resize() routine */
    mem_stat_reg_list(&f, TYPE_FOO_1, FOO_LIST);
      ......
}
```

These static workspace variables will be deallocated using a call to `mem_stat_free_list()`. Note that unlike the `MEM_STAT_REG()` macro, you have to explicitly take the address of `f`; `MEM_STAT_REG()` is a macro.

This is an example of how to use this to free `f`:

```
main(...)
```

```
{
      ......
    mem_stat_mark(1);
      ......
    for ( i = 0; i < 1000; i++ )
        hairy1(...);
      ......
    /* now free up FOO_1 and FOO_2 workspace structures */
    mem_stat_free_list(1,FOO_LIST);
    /* now free up standard Meschach workspace structures */
    mem_stat_free(1);
    /* which is equivalent to: mem_stat_free_list(1,0); */
      ......
}
```

If you have a family of types, where creating one type involves creating another in the same family, care should be taken to avoid double counting. In this case a "main-type" contains a pointer to a "sub-type", say. There are two ways around this: one is to call `mem_bytes_list()` and `mem_numvar_list()` only for those parts of the data structure not in the "sub-type". The other, more complex approach, is to inform the routines that create the "sub-type" that it is created as part of the "main-type", and to account for all of the memory and structure allocation as part of the "main-type". This second approach is only really of use if the "sub-type" is understand as being only of use as part of the larger "main-type". This approach *is* used in Meschach for sparse rows in sparse matrices. Stand alone sparse rows can be created, destroyed, etc., but are almost never used in this way.

8.3.4 Output and object resizing

While it is quite possible to create a new data structure and allocate new memory for every new result, this reduces the efficiency of the algorithms and rapidly loses memory. As there is no garbage collection in C, the memory that is "lost" is unrecoverable. Also, numerical analysts and applications people are often working with large problems on the limits of the machine(s) that they use. So it is rather important that the programmer using a library will want control over memory allocation, or at least over the allocation of the large objects.

The standard used in Meschach is that whenever a large or composite object results from a computation, there is an extra parameter in which the result is to be put. As before, this parameter is a pointer to a data structure. If this pointer is `NULL`, then the output data structure is allocated and initialised. This allows for the creation of the output when the user desires, but still gives control over memory allocation.

If the output object is not `NULL`, but is not of the correct size, then a resizing function should be used. An example of this might be:

```
foo *foo_bar(my_foo1, my_foo2, out_foo)
```

```
foo *my_foo1, *my_foo2, *out_foo;
{
      ......
    if ( out_foo == NULL  || out_foo->size != my_foo1->size )
        out_foo = foo_resize(out_foo, my_foo1->size);
      ......
}
```

The call to `get_foo()` is not necessary if the resizing function (here `foo_resize()`) allocates and initialises a new `foo` data structure if it is passed a `NULL`.

If you cannot write a resizing function, then raise an error if the sizes are incompatible. In such a case, it is better to get the user to create the thing with the right size to start with. The alternative approach to that of creating a new object when the output data structure has the wrong size will result in "memory leaks" with code such as

```
foo *my_foo1, *my_foo2, *out_foo;
  ......
out_foo = foo_bar(my_foo1,my_foo2,out_foo);
```

If `out_foo` is the wrong size, then creating a new data structure will result in the original `out_foo` data structure being lost, and being replaced by a newly created data structure. This memory would be lost until the program terminates.

To repeat: the output parameter should be resized if it is the wrong size, or raise an error.

8.4 User-defined functions

When data structures of a conventional sort cannot explicitly and easily cope with the complexities of a problem, it is usual for programmers to use functional parameters — especially numerical and scientific programmers. In C these are not difficult to use: just remember that you are actually passing pointers to functions, rather than the code itself!

A standard example used is working out the definite integral

$$\int_a^b f(x)\,dx$$

using a quadrature (integration) rule of some kind. The function that computed the integral might look like this:

```
double integrate(f, a, b, n)
double (*f)();      /* function to integrate */
double a, b;        /* lower and upper limits */
int    n;           /* number of sub-intervals to use */
{
    int i;
```

```
    double sum;
      ......
    sum += (*f)(a+i*(b-a)/n);
      ......
    return sum/n;
}
```

Then `integrate(sin, 0.0, PI, 100)` would give an approximation to $\int_0^\pi \sin(x)\,dx$. If you want to integrate a particular function, then you have to write it yourself. So far, so good. However, the function `f` in `integrate()` must be a function of only one variable — the variable that is integrated. Usually functions have parameters, and usually those parameters are changed from run to run, or call to call. These parameters are outside this model of how `f` works as a function.

The standard way of dealing with this in C is to set up some global variables containing the parameters and then modifying them as necessary from run to run, or call to call, of `integrate()`. This is not a very good way of dealing with parameters: as a general rule, the more global variables, and "pathological" (i.e. hidden) connections between routines, the more unpredictable a piece of code becomes.

The alternative that we would recommend here is to allow for an extra parameter in `f` of the type `void *`. This could be a pointer to a `struct` containing the relevant parameters, or even much larger, more complex, data structures. The code for the integration function would then look like:

```
double integrate2(f, fparams, a, b, n)
double (*f)();      /* function to integrate */
void   *fparams;    /* extra parameters for f */
double a, b;        /* lower and upper limits */
int    n;           /* number of sub-intervals to use */
{
      ......
    sum += (*f)(fparams,a+i*(b-a)/n);
      ......
}
```

Then, for example, for a general quadratic $f(x) = ax^2 + bx + c$, the following code could be used:

```
struct PQ { double a, b, c; };

double quadratic(params, x)
struct PQ *params;
double x;
{   /* using Horner's nested multiplication scheme */
    return x*(params->a*x + params->b) + params->c;
}
```

This could be used in something like the following:

```
{
    struct PQ par_quad;
      ......
    par_quad.a = 5.0;
    par_quad.b = -3.7;
    par_quad.c = 101.433445;
    printf("Integral = %g\n",
      integrate2(quadratic,(void *)&par_quad,0.0,1.0,100));
      ......
}
```

What if you want to integrate a function that really is just of one variable, with no additional parameters? At the cost of an extra layer of function calls it can be done using

```
double apply(f, x)
double (*f)(), x;
{       return (*f)(x);       }
```

so that $\int_0^\pi \sin(x)\,dx$ can be computed (approximately) by the call

```
int_val = integrate2(apply, sin, 0.0, PI, 100);
```

Ideally, both styles should probably be implemented, but the additional flexibility in having a `void *` parameter for functional parameters is well worth the effort of writing them into a library.

This approach is an alternative to the "*reverse communication*" path that is taken in most Fortran libraries. The disadvantage of reverse communication is the complexity needed to handle a routine that uses reverse communication. There are possibly some particularly complex things for which reverse communication is still the best technique. However, implementing a number of separate routines which act on the same data structure might still be a more convenient way of doing things than reverse communication.

8.5 Building the library

Building up a library of routines to be generally useful, or even to solve a single problem, usually takes a few steps. The best advice here is summed up in the term "*incremental testing*". As routines are added to the collection that forms your library or problem solver, they should be tested. There is very little more disheartening than to spend a week trying to find an unexpected bug buried somewhere deep in the code. Keep the argument checking and debugging tools (e.g. print-out routines) around — they are still useful.

Build new data structures as you need them, and test them and their routines before going on to the next level. Even if you decide later that you would prefer to use a different way of doing the sub-problems, the interface to a modified data structure should probably stay pretty much the same as for the original data structure used. Use previous (debugged) data structures and their routines. This prevents a lot of errors and simplifies programming; they start to work more like building blocks than isolated bits of code. For example, if you are a control systems designer, you might want to have a "rational function" data structure representing ratios of polynomials:

$$R(x) = \frac{P(x)}{Q(x)}.$$

Each of the polynomials $P(x)$ and $Q(x)$ can be represented by vectors of coefficients. The data structure for $R(x)$ might be

```
typdef struct { int deg_P, deg_Q;     VEC *P, *Q; } rational;
```

There is some redundancy in this data structure since `deg_P` should be one more than the dimension of the vector `P`. Whether or not this degree of redundancy is acceptable will depend on whether users of the library will want to have direct access to `deg_P` and `deg_Q`, and whether routines are written to rely on `deg_P` and `deg_Q` or `P->dim` and `Q->dim`.

Before defining the operations to be performed on objects of type `rational`, the basic operations on polynomials should be defined: adding, subtracting, multiplying and normalising polynomials; synthetic division of polynomials, and evaluating a polynomial at a real or complex value of x. Some of these can be defined in terms of operations on `VEC`'s. Then the operations on rational functions can be defined in terms of the polynomial operations.

8.5.1 Numerical aspects

An important issue in numerical computations is that of the accumulation and magnification of roundoff error. That is, the computations should be *numerically stable*, and avoid accumulating or magnifying roundoff errors. While it can, in general, be very difficult to predict the effects of roundoff error, some situations are more likely to lead to bad results than others. For example, polynomials can be rather badly behaved in this regard. An example can be found in K. Atkinson's *Introduction to Numerical Analysis*, 1st Edition, pp. 80–84 (1979). The designers of MATLAB's polynomial root finding algorithm in fact avoid polynomials altogether in their approach: they find instead the eigenvalues of the companion matrix of the polynomial $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$,

$$C = \begin{bmatrix} 0 & 1 & 0 & \dots & \dots & 0 \\ 0 & 0 & 1 & \dots & \dots & 0 \\ 0 & 0 & 0 & \ddots & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ -a_0 & -a_1 & -a_2 & \dots & \dots & -a_{n-1} \end{bmatrix}.$$

Since rootfinding of polynomials can be badly conditioned, setting up the companion matrix would lead to an equally ill conditioned eigenproblem. However, generally eigenproblems are apparently less likely to suffer such ill conditioning as extreme as for polynomial rootfinding. A control system designer might take this as a hint and deal with control systems in $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$ matrix form, using companion matrices to represent polynomial systems.

A rule of thumb that seems to work for a great many applications for keeping good numerical stability, is to keep intermediate computations in a form close to the form of the original data. Elaborate transformations might give exactly equivalent problems, but the introduction of noise and rounding errors can make some methods far better or far worse than others.

There are a number of hard-won rules which numerical analysts have discovered over the years (and re-discovered far too many times!). In relation to matrix computations the oldest and most important one is:

Don't compute the inverse of a matrix if all you want is to solve some equations.

Computing the inverse of a matrix does not make any of the subsequent calculations for solving a system of equations faster than using its LU factors, the accuracy is slightly worse usually, and it takes longer to compute the inverse in the first place. For sparse matrices it is even more important. The LU factors of a sparse matrix are usually fairly sparse, but the inverse is almost never sparse for practical problems. Forming the inverse of a large sparse matrix may be an impossible undertaking on a machine, even though solving the system of equations can be accomplished quite quickly on that same machine.

Another problem that one would do well to avoid is "finding all eigenvectors of a large matrix". Finding all the eigen*values* of a large symmetric matrix is not an unreasonable task (use the Lanczos routines). Generating the eigenvectors can then often be done using inverse iteration (see K. Atkinson's *An Introduction to Numerical Analysis*, 1st Edition, pp. 548–553 (1979)) on demand for large sparse matrices. Remember: just storing all the eigenvectors of a 10 000 × 10 000 matrix will take up 800Mbyte — not a small amount on any current computer!

8.6 Debugging

While the `error()` macro will save many types of errors, it cannot save you from all of them. If your program is crashing, then put

```
setbuf(stdout,(char *)NULL);
```

at the start of your `main()` program (at least on Unix systems) to ensure that you are seeing all your output. Use liberal `printf()` and `..._output()` calls to check the values of your data types, and to "checkpoint" your program. This also means you should *write* `..._output()` routines for any new data structures that you define.

Potential bugs can sometimes be spotted by automatic tools, such as `lint` on Unix machines, which can detect things like unreachable code, unportable pointer conversions, and function argument incompatibilities for non-ANSI C code. The GNU compiler `gcc` can detect potential portability and related problems in a similar way to `lint` if you use the `-Wall` option (which reports *all* warnings).

Try using open-ended test programs so that you can input any object of a particular data structure, and checking the result. Avoid tests which only give you a "yes/no" answer. If it got the answer by chance, then it has a 50% chance of fooling you. Compute residuals. For systems of equations this means printing out $\|Ax - b\|$; for eigenvalues/eigenvectors this means $\|Ax - \lambda x\|/\|x\|$; for solving $f(x) = 0$ this means printing $\|f(x)\|$; for least squares problems this means printing $\|A^T(Ax - b)\|$. Whatever your problem is, try to compute sufficient information that it is easy to *verify* the complete computed results. For optimisation problems, this would mean checking the first order necessary conditions at least. Use the routines that you have available, not just for doing the computations, but also for helping you to do the verification as well (such as `v_norm2()`).

If a program has a problem, try to find out *where* the problem is. If the program crashes at an unknown point for some reason, put in checkpoints in you main program. Once you've narrowed down the range in which the error occurs there to a single statement, the chances are that it will be a function call. "Open up" that function, putting in checkpoint statements, and printing any relevant quantities until the problem can be located in that function, continuing until the problem is localised.

8.6.1 Memory allocation bugs

These bugs occur when the memory allocation heap has been corrupted. This can occur when an allocated array is written to at an invalid location, or `free()` is called with an invalid address (that is, an address that wasn't returned by `malloc()`, `calloc()` or `realloc()`). Either way the memory heap's headers are corrupted. The results of memory heap corruption can be unpredictable, sometimes resulting in the program crashing, sometimes resulting in apparently "intermittent" bugs. The rules given above for localising bugs don't work for these sorts of bugs, since the corruption is not evident until a call to `malloc()` or `free()` etc. Most programmers could use some help with these sorts of memory heap corruption bugs.

As of version 1.2 of Meschach, there are some built-in routines for keeping a watch on memory usage which are `mem_info_on()`, `mem_info_file()` and `mem_info_type()`. These routines respectively turn the "`mem_info_...`" system on or off, printout a summary of the memory used in Meschach data structures to a file or stream, and return the amount of memory used for a particular Meschach data type. They can be used as follows to check for memory leaks, here in a function `hairy()`:

```
main()
{
        .....
    mem_info_on(TRUE);
```

```
    hairy(....);
    mem_info_f(stdout);  /* print out summary */
    printf("Memory used for vectors by hairy(): %d\n",
           (int)mem_info_type(TYPE_VEC));
}
```

If you get negative amounts of memory in use then something has gone wrong. If static workspace arrays are used you may need to use the `MEM_STAT_REG()` and `mem_stat_...()` routines. The routine `mem_stat_dump()` can also be useful in determining the status of workspace variables.

If you suspect that there is a subtle memory over-writing error, then you should use a package that replaces the standard (fast) memory allocation package `malloc()` and `free()` etc, with something like the public domain package by Conor P. Cahill (uunet address: `uunet!virtech!cpcahil`). This provides a drop-in replacement for the standard library routines: compile your program as in

```
cc  -o my_prog my_prog.c ..... meschach.a libmalloc.a -lm
```

and use his `malloc_chain_check(0)` to check for corruption of the `malloc()` heap. There may be other "debugging" memory allocation/deallocation packages that you have access to.

There are also tools that come with the GNU C compiler for tracking bugs that affect the memory heap.

These are also useful tools to determine if your program has a "memory leak" that results in memory being allocated and then thrown away, although `mem_info()` should be enough to track down memory leaks.

8.6.2 If all else fails

Beyond these things, there are two ways of dealing with these problems.

1. Look at the source code. No-one's code is perfectly readable but we believe that it is not too difficult to follow, especially for experienced C programmers.

2. Contact us. This is best done by e-mail; a current e-mail address is

   ```
   david.stewart@anu.edu.au
   zbigniew.leyk@anu.edu.au
   ```

We cannot guarantee to even look at your problem as we are not employed as programmers, but as academic mathematicians. Our e-mail addresses are also subject to change without notice.

8.7 Suggestions for enthusiasts

There are a number of areas which seem to be particularly ripe for additions. Porting to C++ and making use of classes and operator overloading in itself would be a useful project.

Sets can be implemented a number of ways using permutations and/or integer vectors.

Some extensions that have been considered (and maybe something will be released eventually) include linear programming extensions, ODE solvers, and maybe some nonlinear equation solvers. But there is much more that can be done. One item conspicuously absent are sparse matrix re-ordering routines. A good minimum degree algorithm should be implemented for Meschach.

8.8 Pride and Prejudice

This section is about our own personal beliefs and prejudices. These opinions are nobody's but our own. If you find them obnoxious or frivolous, remember, you have been warned!

8.8.1 What about Fortran 90?

We might have started thinking about it if it had been around when we started on this project six years ago. As it is, we still haven't seen a Fortran 90 compiler, although we have seen a very near miss in the Connection Machine Fortran.

Learning Fortran 90, especially the parts of interest to us, would involve learning a whole new language. When it comes to pointers, dynamic memory allocation and de-allocation, structures/records etc, it *is* a completely new language. We doubt that many future users of Fortran 90 will use the full power of the language for a good many years yet. And then, the people who do make full use of it will be people who have programmed before in C, C++, Ada, Modula-2 (or perhaps Modula-3) and the like. They will know the benefit of using these advanced features.

Porting it to Fortran 90 might be a possibility someday. We don't want to do that job. Porting to C++ would be a much more useful task in the near future. (Meschach has already been used within a C++ program.)

8.8.2 Why should people writing numerical code care about good software?

Numerical analysts and scientists often write unreadable programs.

One of us remembers trying to translate Bill Gear's DIFSUB program from Fortran 77 to C. And failed. He got lost in the spaghetti. So he looked at his description of what it was supposed to do, and implemented *that*. And the result worked.

Quite a few older programmers find this situation normal or even desirable, almost as a sort of job security, or a sense of machismo: "Real programmers don't document

their code; if it was hard to write, it should be hard to read." It wasn't academic politics that made this attitude unacceptable in any modern computer science department, but practical experience combined with the urgency of the "software crisis" of the late sixties and seventies. This "software crisis" still hasn't gone away; big, complex systems (such as commercial and military aircraft) rely more than ever on good, bug-proof software.

On a more personal level, not being a masochist, we much prefer being able to write programs and modify them without having to remember to juggle a dozen flags, set and reset global variables, and so on. Modifying programs is the nature of research. You need to be able to modify the code to do things in different, but meaningful, ways. Trying to do this without helpful software underneath is painful; usually we find that the same underlying operation needs to be re-implemented for the nth time.

Routines which are general purpose, and are designed with flexibility in mind, make an enormous difference when it comes to programming and designing new algorithms. This is why people use numerical libraries. And that is why we wrote this library. The state of the art moves on, and instead of waiting for one's favourite numerical library to be updated with something you would like to see, this library enables you to implement new algorithms. The code is there for inspection, use and modification. (But, please, don't modify old routines unless they have bugs in them — real bugs — but modify the code to create *new* routines.) In doing so, you can provide a platform for further development by yourself or others. Thus the computer can be used not just to crunch numbers, but also to improve your "personal productivity" as the advertisements say. After all, if computers can't make life easier, or more productive, what good are they?

For Further Reading . . .

A full and detailed discussion of the properties and behaviour of the numerical methods in this library and numerical methods in general is beyond the scope of a book such as this. Fuller treatments of numerical methods can be found in numerous numerical analysis texts, which cover a range of different levels from beginning to advanced, and different aspects of numerical analysis.

The text which has been of greatest use to the authors is

Matrix Computations, by G.H. Golub and C. van Loan, 1st Edition published 1983 by North Oxford Academic Publ., Oxford, 2nd Edition published 1989 by John Hopkins University Press, Baltimore and London.

Other general numerical analysis texts that may be useful are

An Introduction to Numerical Analysis, by K.E. Atkinson, 1st Edition published 1978, 2nd Edition published 1989, by John Wiley and Sons, New York, Chichester, Brisbane and Toronto.

Numerical Analysis, by R.L. Burden and J.D. Faires, 4th Edition published 1989 by Prindle, Weber & Schmidt, Boston, Massachusetts. (First edition co-authored by A.C. Reynolds and published in 1978.)

Numerical mathematics and computing, by E.W. Cheney, D. Kincaid, 2nd Edition published in 1985 by Brooks/Cole, Monterey, California.

Some other books on the implementation of numerical algorithms that may be useful are:

The Engineering of Numerical Software, by Webb Miller, published 1984 by Prentice–Hall, Englewood Cliffs, New Jersey.

Numerical Recipes in C: The Art of Scientific Computing, by W.H. Press, B.P. Flannery, S.A. Teulkolsky and W.T. Vetterling, published in 1988 by Cambridge University Press, Cambridge, England.

Index

Function index

In the descriptions below, matrices are represented by capital letters, vectors by lower case letters and scalars by greek lower case letters.

Function	Description	Page
`MCHfactor()`	Modified Cholesky factorisation (actually factors $A+D$, D diagonal, instead of A)	118
`m_copy()`	Copy dense matrix	59
`m_dump()`	Dump matrix data structure to a stream	65
`mem_attach_list()`	Adds a new family of types	83
`mem_bytes()`	Notify change in memory usage (macro)	83
`mem_bytes_list()`	Notify change in memory usage	83
`mem_free_list()`	Frees a family of types	83
`mem_info_bytes()`	Number of bytes used by a type	83
`mem_info_numvar()`	Number of structures of a type	83
`mem_info_file()`	Print memory info to a stream	83
`mem_info_is_on()`	Is memory data being accumulated?	83
`mem_info_on()`	Turns memory info system on/off	83
`mem_is_list_attached()`	Is list of types attached?	83
`mem_numvar()`	Notify change in number of structures allocated (macro)	83
`mem_numvar_list()`	Notify change in number of structures allocated	83
`mem_stat_dump()`	Prints information on registered workspace	88
`mem_stat_free()`	Frees (deallocates) static workspace	88
`mem_stat_mark()`	Sets mark for workspace	88
`MEM_STAT_REG()`	Register static workspace (macro)	88
`mem_stat_show_mark()`	Current workspace group	88
`m_exp()`	Computes matrix exponential	145
`m_finput()`	Input matrix from a stream	62
`m_foutput()`	Output matrix to a stream	65
`M_FREE()`	Free (deallocate) a matrix (macro)	68
`m_free()`	Free (deallocate) matrix (function)	68
`m_free_vars()`	Free a list of matrices	68
`m_get()`	Allocate and initialise a matrix	70
`m_get_vars()`	Allocate list of matrices	70
`m_ident()`	Sets matrix to identity matrix	73
`m_input()`	Input matrix from `stdin` (macro)	62
`m_inverse()`	Invert matrix	122
`m_load()`	Load matrix in MATLAB format	91
`m_mlt()`	Multiplies matrices	81
`mmtr_mlt()`	Computes AB^T	93
`m_norm1()`	Computes $\|A\|_1$ of a matrix	94
`m_norm_frob()`	Computes the Frobenius norm of a matrix	94

Function	Description	Page
`sp_copy()`	Copies a sparse matrix	149
`sp_copy2()`	Copies a sparse matrix into another	149
`sp_diag_access()`	Sets up diagonal access paths for a sparse matrix	155
`sp_dump()`	Dump sparse matrix data structure to a stream	158
`sp_finput()`	Input sparse matrix from a stream	160
`sp_foutput()`	Output a sparse matrix to a stream	158
`sp_free()`	Free (deallocate) a sparse matrix	149
`sp_get()`	Allocate and initialise a sparse matrix	149
`sp_get_val()`	Get the (i, j) entry of a sparse matrix	153
`spICHfactor()`	Sparse incomplete Cholesky factorisation	165
`sp_input()`	Input a sparse matrix form `stdin`	160
`spLUfactor()`	Sparse LU factorisation using partial pivoting	167
`spLUsolve()`	Solves $Ax = b$ using sparse LU factors	167
`spLUTsolve()`	Solves $A^T x = b$ using sparse LU factors	167
`sp_mv_mlt()`	Computes Ax for sparse A	154
`sp_output()`	Outputs a sparse matrix to a stream (macro)	158
`sp_resize()`	Resize a sparse matrix	149
`sprow_add()`	Adds a pair of sparse rows	162
`sprow_foutput()`	Output sparse row to a stream	162
`sprow_get()`	Allocate and initialise a sparse row	162
`sprow_get_idx()`	Get location of an entry in a sparse row	162
`sprow_merge()`	Merge two sparse rows	162
`sprow_mltadd()`	Sparse row vector multiply-and-add	162
`sprow_set_val()`	Set an entry in a sparse row	162
`sprow_smlt()`	Multiplies a sparse row by a scalar	162
`sprow_sub()`	Subtracts a sparse row from another	162
`sprow_xpd()`	Expand a sparse row	162
`sp_set_val()`	Set the (i, j) entry of a sparse matrix	153
`sp_vm_mlt()`	Compute $x^T A$ for sparse A	154
`sp_zero()`	Zero (but do not remove) all entries of a sparse matrix	157
`svd()`	Compute the SVD of a matrix	143
`sv_mlt()`	Scalar–vector multiply	102
`symmeig()`	Compute eigenvalues/vectors of a symmetric matrix	139
`tracecatch()`	Catch and re-raise errors (macro)	51
`trieig()`	Compute eigenvalues/vectors of a symmetric tridiagonal matrix	139

Function	Description	Page
`Usolve()`	Solve $Ux = b$ where U is upper triangular	135
`UTsolve()`	Solve $U^T x = b$ where U is upper triangular	135
`v_add()`	Add vectors	102
`v_conv()`	Convolution product of vectors	104
`v_copy()`	Copy vector	59
`v_dump()`	Dump vector data structure to a stream	65
`v_finput()`	Input vector from a stream	62
`v_foutput()`	Output vector to a stream	65
`V_FREE()`	Free (deallocate) a vector (macro)	68
`v_free()`	Free (deallocate) vector (function)	68
`v_free_vars()`	Free a list of vectors	68
`v_get()`	Allocate and initialise a vector	70
`v_get_vars()`	Allocate list of vectors	70
`v_input()`	Input vector from `stdin` (macro)	62
`v_lincomb()`	Compute $\sum_i a_i x_i$ for an array of vectors	107
`v_linlist()`	Compute $\sum_i a_i x_i$ for a list of vectors	107
`v_map()`	Apply function componentwise to a vector	104
`v_max()`	Computes max vector entry & index	104
`v_min()`	Computes min vector entry & index	104
`v_mltadd()`	Computes $y \leftarrow \alpha x + y$ for vectors x, y	102
`vm_mlt()`	Computes $x^T A$	96
`vm_mltadd()`	Computes $y^T \leftarrow y^T + x^T A$	96
`v_norm1()`	Computes $\|x\|_1$ for a vector	109
`v_norm2()`	Computes $\|x\|_2$ (the Euclidean norm) of a vector	109
`v_norm_inf()`	Computes $\|x\|_\infty$ for a vector	109
`v_ones()`	Set vector to all 1's	73
`v_output()`	Output vector to `stdout` (macro)	65
`v_pconv()`	Periodic convolution of two vectors	104
`v_rand()`	Randomise entries of a vector	73
`v_resize()`	Resize a vector	77
`v_resize_vars()`	Resize a list of vectors	77
`v_save()`	Save a vector in MATLAB format	91
`v_slash()`	Computes componentwise ratio of vectors	104
`v_sort()`	Sorts vector components	104
`v_star()`	Componentwise vector product	104
`v_sub()`	Subtract two vectors	102
`v_sum()`	Sum of components of a vector	104

Function	Description	Page
`zm_free()`	Free (deallocate) complex matrix (function)	68
`zm_free_vars()`	Free a list of complex matrices	68
`zm_get()`	Allocate complex matrix	70
`zm_get_vars()`	Allocate a list of complex matrices	70
`zm_input()`	Input complex matrix from `stdin`	62
`zm_inverse()`	Compute inverse of complex matrix	122
`zm_load()`	Load complex matrix in MATLAB format	91
`zmlt()`	Multiply complex numbers	111
`zmma_mlt()`	Computes AB^* (complex)	93
`zm_mlt()`	Multiply complex matrices	81
`zm_norm1()`	Complex matrix 1–norm	94
`zm_norm_frob()`	Complex matrix Frobenius norm	94
`zm_norm_inf()`	Complex matrix ∞-norm	94
`zm_rand()`	Randomise complex matrix	73
`zm_resize()`	Resize complex matrix	77
`zm_resize_vars()`	Resize a list of complex matrices	77
`zm_save()`	Save complex matrix in MATLAB format	91
`zm_sub()`	Subtract complex matrices	81
`zmv_mlt()`	Complex matrix–vector multiply	96
`zmv_mltadd()`	Complex matrix–vector multiply and add	96
`zm_zero()`	Zero complex matrix	73
`zneg()`	Computes $-z$ (complex)	111
`z_output()`	Print complex number to `stdout`	65
`zQRCPfactor()`	Complex QR factorisation with column pivoting	126
`zQRCPsolve()`	Solve $Ax = b$ using complex QR factorisation	126
`zQRfactor()`	Complex QR factorisation	126
`zQRAsolve()`	Solve $A^*x = b$ using complex QR factorisation	126
`zQRsolve()`	Solve $Ax = b$ using complex QR factorisation	126
`zrot_cols()`	Complex Givens' rotation of columns	130
`zrot_rows()`	Complex Givens' rotation of rows	130
`z_save()`	Save complex number in MATLAB format	91
`zschur()`	Complex Schur factorisation	139
`zset_col()`	Set column of complex matrix	101
`zset_row()`	Set row of complex matrix	101

Function	Description	Page
`zsm_mlt()`	Complex scalar–matrix product	81
`zsqrt()`	Square root $\sqrt{z}$ (complex)	111
`zsub()`	Subtract complex numbers	111
`zUAsolve()`	Solve $U^*x = b$, U complex upper triangular	135
`zUsolve()`	Solve $Ux = b$, U complex upper triangular	135
`zv_add()`	Add complex vectors	102
`zv_copy()`	Copy complex vector	59
`zv_dump()`	Dump complex vector to a stream	65
`zv_finput()`	Input complex vector from a stream	62
`ZV_FREE()`	Free (deallocate) complex vector (macro)	68
`zv_free()`	Free (deallocate) complex vector (function)	68
`zv_free_vars()`	Free a list of complex vectors	68
`zv_get()`	Allocate complex vector	70
`zv_get_vars()`	Allocate a list of complex vectors	70
`zv_input()`	Input complex vector from a `stdin`	62
`zv_lincomb()`	Compute $\sum_i a_i x_i$ for an array of vectors	107
`zv_linlist()`	Compute $\sum_i a_i x_i$ for a list of vectors	107
`zv_map()`	Apply function componentwise to a complex vector	104
`zv_mlt()`	Complex scalar–vector product	102
`zv_mltadd()`	Complex scalar–vector multiply and add	102
`zvm_mlt()`	Computes A^*x (complex)	96
`zvm_mltadd()`	Computes $A^*x + y$ (complex)	96
`zv_norm1()`	Complex vector 1–norm	109
`zv_norm2()`	Complex vector 2– (or Euclidean) norm	109
`zv_norm_inf()`	Complex vector ∞– (or supremum) norm	109
`zv_rand()`	Randomise complex vector	73
`zv_resize()`	Resize complex vector	77
`zv_resize_vars()`	Resize a list of complex vectors	77
`zv_save()`	Save complex vector in MATLAB format	91
`zv_slash()`	Componentwise ratio of complex vectors	104
`zv_star()`	Componentwise product of complex vectors	104

Function	Description	Page
`zv_sub()`	Subtract complex vectors	102
`zv_sum()`	Sum of components of a complex vector	104
`zv_zero()`	Zero complex vector	73

Low level routines

Function	Description	Page
`__add__()`	Add arrays	113
`__ip__()`	Inner product of arrays	113
`MEM_COPY()`	Copy memory (macro)	113
`MEM_ZERO()`	Zero memory (macro)	113
`__mltadd__()`	Forms $x + \alpha y$ for arrays	113
`__smlt__()`	Scalar–vector multiplication for arrays	113
`__sub__()`	Subtract an array from another	113
`__zadd__()`	Add complex arrays	113
`__zconj__()`	Conjugate complex array	113
`__zero__()`	Zero an array	113
`__zip__()`	Complex inner product of arrays	113
`__zmlt__()`	Complex array scalar product	113
`__zmltadd__()`	Complex array saxpy	113
`__zsub__()`	Subtract complex arrays	113
`__zzero__()`	Zero a complex array	113

ISBN 0-7315-1900-0