Chapter 4

Basic Dense Matrix Operations

The following routines are described in the following pages:

To use these routines use the include statement

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

NAME

`catch`, `catchall`, `catch_FPE`, `tracecatch` – catch errors

SYNOPSIS

```
#include "matrix.h"
catch(err_num,normal_code_to_execute,
        code_to_execute_if_error)
int    err_num;

catchall(normal_code_to_execute,
        code_to_exectue_if_error)

tracecatch(normal_code_to_execute,fn_name)
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 is like this: The old `restart 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.

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

```
"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

cp_ivec, cp_perm, cp_mat, cp_vec – copy objects

SYNOPSIS

```
#include  "matrix.h"
IVEC    *cp_ivec(in,out)
IVEC    *in, *out;

MAT     *cp_mat(in,out)
MAT     *in, *out;

PERM    *cp_perm(in,out)
PERM    *in, *out;

VEC     *cp_vec(in,out)
VEC     *in, *out;
```

DESCRIPTION

All the routines `cp_ivec()`, `cp_mat()`, `cp_perm()` and `cp_vec()` 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 `get_mat()`, `get_perm()` or `get_vec()` as appropriate.

For `cp_mat()`, `cp_vec()` and `cp_ivec()`, 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 `cp_perm()` then a new permutation is created and returned.

There are also "raw" copy routines `_cp_mat(in,out,i0,j0)` and `_cp_vec(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 `_cp_vec()`, `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 `cp_...()` routines all work *in situ* with `in == out`, however, the `_cp_...()` routines will only work *in situ* if `i0` (and also `j0` if this is also passed) is zero.

EXAMPLE

```
/* copy x to y */
cp_vec(x,y);
/* create a new vector z = x */
z = cp_vec(x,VNULL);
/* copy A to the block in B with top-left corner (3,5) */
_cp_mat(A,B,3,5);
```

SEE ALSO

`get_ivec()`, `get_mat()`, `get_perm()`, `get_vec()`

SOURCE FILE: `copy.h, ivecop.h`

NAME

`error` – raise an error

SYNOPSIS

```
#include   "matrix.h"
error(err_num,func_name)
int      err_num;
char     *func_name;

int      set_err_flag(new_flag)
int      new_flag;
```

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);
```

This call does not return.

The call to `ev_err()` prints out a message to stderr indicating that an error has occurred, and where in which function it occurred. For example, it could look like:

```
"tut1.c", line 79: sizes of objects don't match in function f()
```

which indicates that an error was flagged in file "`tut1.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 `ERREXIT()` or `ERRABORT()`.

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 */
```

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  -- this is the default */
EF_ABORT  1  /* abort on error -- dump core for debugging */
EF_JUMP   2  /* do longjmp()   -- see above code */
EF_SILENT 3  /* do not report error, but do longjmp() */
```

EXAMPLE

```
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");
```

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

fin_ivec, fin_mat, fin_perm, fin_vec – input object from a file

SYNOPSIS

```
#include   <stdio.h>
#include   "matrix.h"
MAT     *fin_mat(fp,A)
FILE    *fp;
MAT     *A;
A = fin_mat(fp,MNULL);

PERM    *fin_vec(fp,v)
FILE    *fp;
VEC     *v;
v = fin_vec(fp,VNULL);

PERM    *fin_perm(fp,pi)
FILE    *fp;
PERM    *pi;
pi = fin_perm(fp,PNULL);
```

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 `in_mat(A)`, `in_perm(pi)` and `in_vec(x)`, which are equivalent to `fin_mat(stdin,A)`, `fin_perm(stdin,pi)` and `fin_vec(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.

Note that the format for file input is essentially the same as the output produced by the `fout_...()` and `out_...()` functions. This means that if the output is sent to a file, 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
```

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:

```
MAT  *A;
VEC  *b;
PERM *pi;
  ......
A = in_mat(MNULL);
b = in_vec(VNULL);
pi = in_perm(PNULL);
```

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

```
b = get_vec(m);
in_vec(b);  /* use b's allocated memory */
```

SEE ALSO

`fout_...()` entries, `in_...()` 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.

SOURCE FILE: `matrixio.c`

NAME

fout_ivec, fout_mat, fout_perm, fout_vec – output to a file

SYNOPSIS

```
#include "matrix.h"
fout_mat(fp,A)
FILE    *fp;
MAT     *A;

fout_perm(fp,pi)
FILE    *fp;
PERM    *pi;

fout_vec(fp,v)
FILE    *fp;
VEC     *v;
```

DESCRIPTION

These output 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 fin_...() and in_...() routines.

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

There are also the routines out_mat(A), out_perm(pi) and out_vec(x) which are equivalent to fout_mat(stdout,A), fout_perm(stdout,pi) and fout_vec(stdout,x).

Note that the in_...() routines are in fact just macros which translate into calls of these fin_...() 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 dump_mat(fp,A), dump_perm(fp,px) and dump_vec(fp,x) where fp is a FILE *, A is a MAT *, px is a PERM * and x is a VEC *. 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 */
out_mat(A);
/* ...or to file junk.out */
if ( (fp = fopen("junk.out","w")) == NULL )
    error(E_EOF,"my_function");
fout_mat(fp,A);
/* ...but for debugging, you may need... */
dump_mat(stdout,A);
```

SEE ALSO

in_...(), fin_...()

SOURCE FILE: matrixio.c

NAME

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

SYNOPSIS

```
#include <stdio.h>
#include "matrix.h"
int      finput(fp,prompt,fmt,var)
FILE     *fp;
char     *prompt, *fmt;
????     *var;

int      input(prompt,fmt,var)
char     *prompt, *fmt;
????     *var;

int      fprompter(fp,prompt)
FILE     fp;
char     *prompt;

int      prompter(prompt)
char     *prompt;
```

DESCRIPTION

The macros `finput()` and `input()` are for general input, allowing for comments as accepted by the `fin_..()` 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()`, `fin_..()`

SOURCE FILE: `matrix.h`

NAME

`freeivec, freemat, freeperm, freevec` – destroy objects and free up memory

SYNOPSIS

```
#include "matrix.h"
freeivec(iv)
IVEC    *iv;

freemat(A)
MAT     *A;

freeperm(pi)
PERM    *pi;

freevec(v)
VEC     *v;
```

DESCRIPTION

These are in fact all macros which result in calls to `iv_free()`, `m_free()`, `px_free()` and `v_free()` respectively. 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 = get_vec(10);
y = x;            /* y and x now point to the same place */
freevec(x);       /* x is now VNULL */
/* y now "dangles" -- using y can be dangerous */
y->ve[9] = 1.0;   /* overwriting malloc area! */
freevec(y);       /* program will probably crash here! */
```

SEE ALSO

`get_...()` routines

BUGS

Dangling pointer problem neither fixed, nor fixable.

SOURCE FILE: `memory.c`

NAME

get_ivec, get_mat, get_perm, get_vec – create and initialise objects

SYNOPSIS

```
#include "matrix.h"
IVEC     *get_ivec(dim)
unsigned dim;

MAT      *get_mat(m, n)
unsigned m, n;

PERM     *get_perm(size)
unsigned size;

VEC      *get_vec(dim)
unsigned dim;
```

DESCRIPTION

All these routines create and initialise data structures for the associated type of object. 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 = get_vec(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 `get_ivec(dim)` routine creates an integer vector of dimension `dim`. Its entries are initialised to be zero. The `get_mat(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 `get_perm(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 `get_vec(dim)` routine creates and returns a vector of dimension `dim`. Its entries are all initialised to zero.

EXAMPLE

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

SEE ALSO

`free...()` routines, `iv_resize()`, `m_resize()`, `px_resize()` and `v_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`

NAME

get_col, get_row – extract columns or rows from matrices

SYNOPSIS

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

VEC     *get_row(A, row_num, v)
MAT     *A;
int     row_num;
VEC     *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.

EXAMPLE

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

SEE ALSO

`set_col()` and `set_row()`.

SOURCE FILE: `matop.c`

NAME

id_mat, ones_mat, ones_vec, rand_mat, rand_vec, zero_mat, zero_vec, mrand, smrand, mrandlist – initialisation routines

SYNOPSIS

```
#include "matrix.h"
MAT     *id_mat(A)

MAT     *ones_mat(A)
VEC     *ones_vec(x)

MAT     *rand_mat(A)
VEC     *rand_vec(x)

MAT     *zero_mat(A)
VEC     *zero_vec(x)
MAT     *A;
VEC     *x;

double  mrand()
void    smrand(seed)
int     seed;

void    mrandlist(a, len)
double  a[];
int     len
```

DESCRIPTION

The routine `id_mat()` 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 `ones_mat()` and `ones_vec()` respectively fill `A` and `x` with ones.

The routines `rand_vec()` and `rand_mat()` respectively fill `A` and `x` with random entries between zero and one as determined by the `rand()` function.

The routines `zero_mat()` and `zero_vec()` respectively 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;
VEC  *x;
PERM *pi;
  ......
zero_mat(A);  /* A == zero matrix */
id_mat(A);    /* A == identity matrix */
ones_mat(A);  /* A == e.e^T */
rand_mat(A);  /* A[i][j] is random in interval [0,1) */
zero_vec(x);  /* x == zero vector */
ones_vec(x);  /* x == e */
rand_vec(x);  /* x[i] is random in interval [0,1) */
```

BUGS

The routine `id_mat()` "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: `matop.c`

NAME

`in_prod` – inner product

SYNOPSIS

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

DESCRIPTION

The inner product of `x` and `y` is returned by `in_prod`. This will fail if `x` or `y` is NULL.

EXAMPLE

```
VEC  *x, *y;
double x_dot_y;
  ......
x_dot_y = in_prod(x,y);
```

SEE ALSO

`__ip__()` and the core routines.

BUGS

The accumulation is not guaranteed to be done in a higher precision than `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`

NAME

iv_add, iv_sub – Integer vector operations

SYNOPSIS

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

IVEC     *iv_sub(iv1,iv2,out)
IVEC     *iv1, *iv2, *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

iv_resize, m_resize, px_resize, v_resize – Resizing data structures

SYNOPSIS

```
#include "matrix.h"
IVEC     *iv_resize(iv,new_dim)
IVEC     *iv;
int      new_dim;

MAT      *m_resize(A,new_m,new_n)
MAT      *A;
int      new_m, new_n;

PERM    *px_resize(px,new_size)
PERM    *px;
int     new_size;

VEC     *v_resize(x,new_dim)
VEC     *x;
int     new_dim;
```

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` $\times$ `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.

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.

EXAMPLE

```
/* an alternative to workspace arrays */
... my_function(...)
{
    static VEC *x = VNULL;
      ......
```

```
    x = v_resize(x,new_size);
      ......
    cp_vec(..., 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.

SOURCE FILE: `memory.c` and `ivecop.c`

NAME

`MACHEPS` – machine epsilon

SYNOPSIS

```
#include "matrix.h"
double  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 double precision only.

For ANSI C implementations, this is set to the value of the `DBL_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 – matrix addition and multiplication

SYNOPSIS

```
#include "matrix.h"
MAT     *m_add(A,B,C)
MAT     *A, *B, *C;

MAT     *m_mlt(A,B,C)
MAT     *A, *B, *C;

MAT     *m_sub(A,B,C)
MAT     *A, *B, *C;

MAT     *sm_mlt(s,A,OUT)
double  s;
MAT     *A, *OUT;
```

DESCRIPTION

The function `m_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 function `m_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 routine `sm_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 can work *in situ*; that is, `C` need not be different to either `A` or `B`. However, `m_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;
double 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()`

SOURCE FILE: `matop.c`

NAME

m_load, m_save, v_save – MATLAB save/load to file

SYNOPSIS

```
#include "matrix.h"
MAT     *m_load(fp,name)
FILE    *fp;
char    **name;

MAT     *m_save(fp,A,name)
FILE    *fp;
MAT     *A;
char    *name;

VEC     *v_save(fp,x,name)
FILE    *fp;
VEC     *x;
char    *name;

double  d_save(fp,d,name)
FILE    *fp;
double     d;
char    *name;
```

DESCRIPTION

These routines read and write MATLAB^{TM} load/save files. This enables results to be transported between MATLAB^{TM} and Meschach. The routine `m_load()` loads in a matrix from file `fp` in MATLAB^{TM} 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` are 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 = get_vec(Xmat->m);
    for ( i = 0; i < Xmat->m; i++ )
       x->ve[i] = Xmat->me[i][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`

NAME

m_transp, mmtr_mlt, mtrm_mlt – matrix transposes and multiplication

SYNOPSIS

```
#include "matrix.h"
MAT     *m_transp(A,OUT)
MAT *A, *OUT;

MAT     *mmtr_mlt(A,B,OUT)
MAT *A, *B, *OUT;

MAT     *mtrm_mlt(A,B,OUT)
MAT *A, *B, *OUT;
```

DESCRIPTION

The routine `m_transp()` transposes the matrix `A` and stores the result in `OUT`. This routine may be *in situ* (i.e. `A == OUT`) only if `A` is square.

The routine `mmtr_mlt()` forms the product AB^T, which is stored in `OUT`. The routine `mtrm_mlt()` forms the product A^TB, 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 can then be 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`

NAME

m_norm1, m_norm_inf, m_norm_frob – matrix norms

SYNOPSIS

```
#include "matrix.h"
double   m_norm1(A)
MAT      *A;

double   m_norm_inf(A)
MAT      *A;

double   m_norm_frob(A)
MAT      *A;
```

DESCRIPTION

These routines compute matrix norms. The routine `m_norm1()` computes the matrix norm of `A` in the matrix 1–norm;
`m_norm_inf()` computes the matrix norm of `A` in the matrix ∞–norm; `m_norm_frob()` computes 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()`

BUGS

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

SOURCE FILE: `norm.c`

NAME

mv_mlt, vm_mlt, mv_mltadd, vm_mltadd – matrix–vector multiplication

SYNOPSIS

```
#include "matrix.h"
VEC     *mv_mlt(A,x,out)
MAT     *A;
VEC     *x, *out;

VEC     *vm_mlt(A,x,out)
MAT     *A;
VEC     *x, *out;

VEC     *mv_mltadd(v1,v2,A,alpha,out)
VEC     *v1, *v2, *out;
MAT     *A;
double  alpha;

VEC     *vm_mltadd(v1,v2,A,alpha,out)
VEC     *v1, *v2, *out;
MAT     *A;
double  alpha;
```

DESCRIPTION

The routines mv_mlt() and vm_mlt() form Ax and $A^Tx = (x^TA)^T$ and store it in out. The routines mv_mltadd() and vm_mltadd() form $v_1 + \alpha A v_2$ and $v_1^T + \alpha v_2^T A$ 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;
double 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

NAME

px_id, px_inv, px_mlt – permutation identity, inverse and multiplication

SYNOPSIS

```
#include "matrix.h"
PERM    *px_id(pi)
PERM    *pi;

PERM    *px_mlt(pi1,pi2,out)
PERM    *pi1, *pi2, *out;

PERM    *px_inv(pi,out)
PERM    *pi, *out;

PERM    *trans_px(pi,i,j)
PERM    *pi;
int     i, j;
```

DESCRIPTION

The routine `px_id()` initialises `pi` to be the identity permutation of the size of `pi` 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 `trans_px()` swaps `pi->pe[i]` and `pi->pe[j]`; it is a multiplication by the transposition $i \leftrightarrow j$.

EXAMPLE

```
PERM  *pi1, pi2, pi3;
  ......
pi1 = get_perm(10);
px_id(pi1);           /* sets pi1 to identity */
trans_px(pi1,3,5);    /* pi1 is now a transposition */
px_inv(pi1,pi1);      /* invert pi1 -- in situ */
px_mlt(pi1,pi2,pi3);  /* pi3 = pi1.pi2 */
```

SOURCE FILE: `pxop.c`

NAME

px_cols, px_rows, px_vec, px_invvec – permute rows or columns of a matrix, or permute a vector

SYNOPSIS

```
#include "matrix.h"
MAT     *px_rows(pi,A,OUT)
PERM    *pi;
MAT     *A, *OUT;

MAT     *px_cols(pi,A,OUT)
PERM    *pi;
MAT     *A, *OUT;

VEC     *px_vec(pi,x,out)
PERM    *pi;
VEC     *x, *out;

VEC     *px_invvec(pi,x,out)
PERM    *pi;
VEC     *x, *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()` permutes 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, `px_invvec()` permutes `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()`.

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;
MAT   *A, *B;
  ......
/* permute x to give tmp */
tmp = px_vec(pi,x,tmp);
/* restore x */
x = px_invvec(pi,tmp,x);
/* symmetric permutation */
B = px_rows(A,MNULL);
A = px_cols(B,A);
```

SEE ALSO

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

SOURCE FILE: `pxop.c`

NAME

set_col, set_row – set rows and columns of matrices

SYNOPSIS

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

MAT     *set_row(A,k,out)
MAT     *A;
int     k;
VEC     *out;
```

DESCRIPTION

The routine `set_col()` above sets the value of the kth 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 kth 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` data structure is a row-oriented data structure, 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`

NAME

sv_mlt, v_add, v_mltadd, v_sub – scalar–vector multiplication and addition

SYNOPSIS

```
#include "matrix.h"
VEC     *sv_mlt(s,x,out)
double  s;
VEC     *x, *out;

VEC     *v_add(v1,v2,out)
VEC     *v1, *v2;
VEC     *out;

VEC     *v_mltadd(v1,v2,s,out)
VEC     *v1, *v2, *out;
double  s;

VEC     *v_sub(v1,v2,out)
VEC     *v1, *v2;
VEC     *out;
```

DESCRIPTION

The `sv_mlt()` routine performs the scalar multiplication of the scalar `s` and the vector `x` and the results are placed in `out`.

The routine `v_add()` adds the vectors `v1` and `v2`, and the result is returned in `out`.

The `v_mltadd()` routine sets `out` to be the linear combination `v1+s.v2`.

The routine `v_sub()` subtracts `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;
double alpha;
  ......
tmp = get_vec(x->dim);
z = get_vec(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);
```

SOURCE FILE: `vecop.c`

NAME

v_map, v_max, v_min, v_star, v_slash, v_sort, v_sum – componentwise operations

SYNOPSIS

```
#include  "matrix.h"
VEC    *v_map(fn, x, out)
double (*fn)();
VEC    *x, *out;

double v_max(x, index)
VEC    *x;
int    *index;

double v_min(x, index)
VEC    *x;
int    *index;

VEC    *v_star(x, y, out)
VEC    *x, *y, *out;

VEC    *v_slash(x, y, out)
VEC    *x, *y, *out;

VEC    *v_sort(x, order)
VEC    *x;
PERM   *order;

double v_sum(x)
VEC    *x;
```

DESCRIPTION

The routine `v_map()` applies the function `(*fn)()` to the components of `x` to give the vector `out`. That is, `out->ve[i] = (*fn)(x->ve[i])`. There is also a version

```
VEC    *_v_map(double (*fn)(void *,double), void *fn_params,
               VEC *x, VEC *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 index 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()`.

The routine `v_star()` computes 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 routine `v_slash()` computes 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 routine `v_sum()` returns the sum of the entries of `x`.

EXAMPLE

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

```
VEC    *x, *y, *z;
PERM   *order;
double 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 `pccg()`, `cgs()`, `lsqr()`.

SOURCE FILE: `vecop.c`

NAME

v_lincomb, v_linlist – linear combinations

SYNOPSIS

```
#include "matrix.h"
VEC     *v_lincomb(n,v_list,a_list,out)
int     n;
VEC     *v_list[];
double  a_list[];
VEC     *out;

VEC     *v_linlist(out,v1,a1,v2,a2,...,VNULL)
VEC     *out;
VEC     *v1, *v2, ...;
double  a1,  a2,  ...;
```

DESCRIPTION

The routine `v_lincomb()` computes 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 routine `v_linlist()` is a variant of the above which does 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;
double 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) */
zero_vec(out);
out = v_linlist(out, v1, h/6.0, v2, h/3.0,
                     v3, h/3.0, v4, h/6.0,
                     VNULL);
```

SEE ALSO

`v_smlt()`, `v_mltadd()`

BUGS

The routine `v_linlist()` is implemented as having arguments `out, v1, a1, v2, a2, v2, a2, v3, a3, v4, a4, v5, a5, v6, a6, v7, a7, v8, a8, v9, a9, v10, a10`. There is therefore a limit of 10 vectors. This routine should be implemented using `va_args()`.

SOURCE FILE: `vecop.c`

NAME

v_norm1, v_norm2, v_norm_inf – vector norms

SYNOPSIS

```
#include "matrix.h"
double  v_norm1(x)
VEC     *x;

double  v_norm2(x)
VEC     *x;

double  v_norm_inf(x)
VEC     *x;
```

DESCRIPTION

These functions compute vector norms. In particular, `v_norm1()` gives the 1–norm, `v_norm2()` gives the 2–norm or Euclidean norm, and `v_norm_inf()` computes 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()`. 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`. If the corresponding component of `scale` is zero, 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()`

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

`__add__`, `__ip__` , `__mltadd__`, `__smlt__`, `__sub__`, `__zero__` – core routines

SYNOPSIS

```
#include "machine.h"
void     __add__(dp1,dp2,out,len)
double   dp1[], dp2[], out[];
int      len;

double   __ip__(dp1,dp2,len)
double   dp1[], dp2[];
int      len;

void     __mltadd__(dp1,dp2,s,len)
double   dp1[], dp2[], s;
int      len;

void     __smlt__(dp,s,out,len)
double   dp[], s, out[];
int      len;

void     __sub__(dp1,dp2,out,len)
double   dp1[], dp2[], out[];
int      len;

void     __zero__(dp,len)
double   dp[];
int      len;
```

DESCRIPTION

These routines are the underlying routines for all 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 `double`'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.

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

The routine `__smlt__()` sets `out[i] = s*dp[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 `__zero__()` sets `out[i] = 0.0` for `i` ranging from zero to `len-1`. This routine should be used instead of the macro `mem_zero()` or the ANSI C routine `memset()` for portability, in case the double precision zero is not represented by a bit string of zeros.

EXAMPLE

```
MAT    *A, *B;
```

```
double 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);
```

SOURCE FILE: `machine.c`

Contents