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 $\sqrt{\mathbf{HUGE}}$.

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 $A^T x = (x^T A)^T$ 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 $v_1 + sA^T v_2$ 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 $z_i = \sum_{0\leq j\leq i} x_j y_{i-j}$. 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 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()`. 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 $\sqrt{\mathbf{HUGE}}$.

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**