The Mathutil library

The Mathutil library

This is edition 0.0.8
Last updated, 26 March 1999

Eleftherios Gkioulekas
Department of Applied Mathematics
University of Washington
lf@amath.washington.edu

This edition of the manual is consistent with Mathutil 0.0.8.
It was last updated, 26 March 1999

Published on the Internet.
http://www.amath.washington.edu/~lf/

Short Contents

Table of Contents

Copying

The *Mathutil* package is "free"; this means that everyone is free to use it and free to redistribute it on a free basis. The *Mathutil* package is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this package that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the programs and documents that relate to *Mathutil*, that you receive source code or else can get it if you want it, that you can change these programs or use pices of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we don't permit you to deprive anyone else of these rights. For example, if you distribute copies of the *Mathutil*-related code, you must give the recipients all the rights that you have. You must make sure that they, too, receive of can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the programs that relate to *Mathutil*. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the programs currently being distributed that relate to *Mathutil* are found in the General Public Licenses that accompany them.

1 Getting started with Mathutil

Mathutil is a library of tools that are indispensable for developing numerical code in C. In order to use C effectively to develop numerical software you need a library oriented to provide for the needs of numerical software. The Mathutil package provides a collection of such libraries.

The features of the Mathutil library include:

- Sophisticated support for reporting errors and exception handling during run-time.
- Support for variadic C functions.
- Support for binary file I/O. Also, automatic handling of endian compatibility.
- Smart arrays in 1d, 2d and 3d. Smart arrays will handle reference counting, stride, and file I/O for you automatically. They are particularly useful for creating C interfaces to Fortran libraries. *More features to be added here as I go*

1.1 Installing Mathutil

Mathutil is an auto-configuring package and can be compiled and installed by following the following simple steps:

1. Download the source code package '`mu-0.0.8.tar.gz`' and unpack it somewhere in your home directory:

   ```
   % gunzip mu-0.0.8.tar.gz
   % tar xf mu-0.0.8.tar
   % cd mu-0.0.8
   ```

   A directory '`mu-0.0.8`' will be created containing the source code.
2. Enter '`mu-0.0.8`' and configure the source code:

   ```
   % ./configure
   ```

   If you would like to install Mathutil in a non-standard location, such as your home directory ('`/home/lf`'), you must use the '`--prefix`' flag:

   ```
   % ./configure --prefix=/home/lf
   ```

   This shell script will automatically configure the package to your platform. Then type

   ```
   % make
   ```

   to build the source code.
3. Validate your build by running the test suite:

   ```
   % make check
   ```

   If all is okey, the go ahead and install by typing:

   ```
   % make install
   ```
4. To print a hard copy of the software documentation that accompanies this package run

   ```
   % make dvi
   ```

   to produce the corresponding '`dvi`' files.

If you installed Mathutil on a non-standard location, make sure that that location is in your path. One way to check this is by attempting to run the '`mu-config`' program:

```
% mu-config --version
```

If this does not work, then your Mathutil installation will also not work. To fix it, if '`/home/lf/bin`' is the directory where you installed '`mathutil-confif`', then add that directory to your shell's path. For the Bash shell, you do this with the following statement in your '`.bashrc`' file:

```
PATH="/home/lf/bin:$PATH"
```

For the csh and tcsh shells:

```
setenv PATH "/home/lf/bin:$PATH"
```

1.2 Writing programs based on Mathutil

Content

2 Error reporting and exception handling

Numerical programs have plenty of opportunities to fail. Failures may happen if a routine receives invalid arguments, if an iteration fails to converge, if the program tries to load a corrupt file, and so on. A robust application should detect these failures when they happen, and then it should *throw an error*. When an error is thrown, we say that *exception* happened. In response to an exception, your program should do something. That something is called *exception handling*.

ANSI C does not have built-in support for exception handling, like higher-level languages. This is why Mathutil provides with routines that you can use in your own programs both to throw errors and to handle the exceptions.

2.1 Error reporting concepts

Mathutil recognizes three levels of errors: *fatal errors*, *warnings*, and *messages*. A *fatal error* is a serious unrecoverable error. When a fatal error happens, the program must immediately do some exception handling. Mathutil can be configured to either completely abort the program, which is the default behaviour, or to do a non-local exit with `setjmp` and `longjmp`. Most of the errors that you have to implement in an application tend to be fatal errors. A *warning* is a message that suggests that a serious error might have happened. The application continuous operating as usual, however the user must follow up on the warning to see if his results are reliable. Mathutil gives you the option to turn warnings on or off. A *message* is not an error. It's is simply a message that the application reports to the user. In numerical simulations it is common to use messages to inform the user about the simulation's progress. However, for convenience, we will speak of "messages" as errors. Note that neither warnings nor messages do any non-local exits or abort the program. Only fatal errors can do that.

Every type of error has a set of parameters associated with it. These parameters are used to form the *error string*, which is either displayed on the users screen, or it is written to a file. Alternatively, the parameters can be passed on to a custom handler, which will display the error information in a different manner. For example, in a graphical user interface application, you would want the custom handler to display the error information using a popup dialog window. The Mathutil routines for reporting errors take the parameters that form the error string from their arguments. The most general form of an error string, as assembled by the default handler, is as follows:

filename`:`*line*`:` *prefix*`:` *standard error*`:` *message*

Here's what the fields mean:

filename, line
: The specific file and line in the source code (or in an input file) that has thrown the error.

prefix
: Indicates whether the string is an error, a warning, or a message. The default prefixes used by Mathutil are `"ERROR"`, `"WARNING"` and `'MESSAGE"` but they can be customized by the main program.

standard error
: The string that corresponds to one of the standard Mathutil error codes. See Section 2.3 [Mathutil error codes], page 6.

message
: An additional string that explains whythis error has been thrown *this* particular time. In this explanation you can often provide a little debugging information too.

2.2 Reporting errors

You can report an error using one of the following functions, depending on whether the error is a fatal error, a warning or a message. Note that as soon as you decide to throw a fatal error, you can kiss your program goodbye (unless you have set up some special kind of exception handling). Throwing warnings or messages does not trigger any exception handling.

`void` **mu_error** (`int` *status*, `int` *errnum*, `char *` *fmt*, `...`) [Function]
: Throw a fatal error, with error code *errnum* (see Section 2.3 [Mathutil error codes], page 6). If *status* is `1`, then the program will abort, **unless** you have pushed a non-local exit to the exception handler. However, if the exception handler itself requests that you continue jumping to outer exception handlers, and you run out of exception handlers, then your program will still abort if *status* is `1`. This is discussed in more detail in Section 2.5 [Exception handling], page 10. The string *fmt* contains a `printf`-like string, and the arguments that follow it are the corresponding `printf`-like arguments. This string and the subsequent arguments are used to form the error string associated with this error.

`void` **mu_error_at_line** (`int` *status*, `int` *errnum*, `char *` *filename*, `int` *line*, `char *` *fmt*, `...`) [Function]
: This function does the exact same thing as `mu_error`, except that it compiles an error string that includes the *filename* and *line* in the source code that causes the error. You can use the C preprocessor macros `__FILE__` and `__LINE__` when to obtain the filename and line number.

`void` **mu_warning** (`int` *status*, `int` *errnum*, `char *` *fmt*, `...`) [Function]
: Throw a warning, with error code *errnum*. The string *fmt* and the subsequent arguments, determine the error string.

`void` **mu_message** (`char *` *fmt*, `...`) [Function]
: Throw a message. The content of the message is defined by the string *fmt* and the subsequent arguments, which follow the *printf* conventions.

2.3 Mathutil error codes

Mathutile has a collection of standard error codes. Every one of these codes corresponds to a specific type of error, and it is associated with an error string that describes in a generic way what type of error is being thrown. The Mathutil error codes should be given as the argument *errnum* to the `mu_error`, `mu_warning` and `mu_error_at_line` routines

described in Section 2.2 [Reporting errors], page 6. Please do not rely on the generic error message however. The generic error should be supplemented with an additional message that describes more specifically, what the cause is of *this* error, in *this* instance. The additional message is in the argument *fmt* and the subsequent variadic arguments of `mu_error`, `mu_warning` and `mu_error_at_line`.

Mathutil recognizes the following error codes:

`MU_EDOM` input domain error, e.g. `sqrt(-1)`

`MU_ERANGE`
: output range error, e.g. `exp(1e100)`

`MU_EFAULT`
: invalid pointer argument

`MU_EINVAL`
: invalid argument (value) supplied by user

`MU_EFAILED`
: generic failure

`MU_ESANITY`
: sanity check failed - shouldn't happen

`MU_ENOMEM`
: memory allocation failed because system is out of memory

`MU_EIO` input/output error

`MU_EBADFUNC`
: problem with user-supplied function

`MU_ERUNAWAY`
: iterative process is out of control

`MU_EMAXITER`
: exceeded maximum number of iterations

`MU_EZERODIV`
: tried to divide by zero

`MU_EBADTOL`
: user specified an invalid tolerance

`MU_ETOL` failed to reach the specified tolerance

`MU_EUNDRFLW`
: underflow

`MU_EOVRFLW`
: overflow

`MU_ELOSS` loss of accuracy

`MU_EROUND`
: failed because of roundoff error

`MU_EBADLEN`
: matrix, vector lengths are not conformant

`MU_ESING` apparent singularity detected

`MU_EUNSUP`
: requested feature is not supported by the hardware

`MU_EUNIMPL`
: requested feature not (yet) implemented

`MU_EINDEX`
: smart array index error

You can convert any one of these error codes to the corresponding error string using the following function:

`char *` **mu_strerror** (`int` *errnum*) [Function]
: Return the generic error string that describes the error that has error code *errnum*.

However, in practice, you don't really get to use this very often.

The standard C library also has an error reporting mechanism of its own. It reports errors by assigning a value to the global variable `errno`. To make this variable available to your programs, add the following at the top of your source file.

```
#include <errno.h>
#ifndef errno
extern int errno;
#endif
```

Then use this routine to decode it:

`char *` **mu_libc_strerror** (`int` *errnum*) [Function]
: Return the error string that corresponds to the Standard C Library error with error code *errnum*. This routine is a wrapper to the not-so-portable `strerror` routine.

A simple example where this is used is in the implementation of `mu_file_close`:

```
void mu_file_close (struct mu_file *file)
{
  /* Close the file descriptor */
 try_again:
  if (close (file->descriptor) < 0)
    {
#ifdef EINTR
      if (errno == EINTR) goto try_again;
#endif
      mu_error (mu_error_status, MU_EIO, "Can't close file %s: %s",
                file->filename, mu_libc_strerror (errno));
    }
  else
    {
      mu_free (file->filename);
      mu_free (file);
      return;
    }
}
```

2.4 Configuring the error reporting handler

Mathutil allows you to customize how your application reports errors. The reason why this flexibility is important is because errors must be handled differently in interactive applications from non-interactive ones.

The following global variable controls whether or not to display warnings. Adventurours users may choose to ignore warnings. Messages however can not be disabled; it is assumed that the user will always want to see them.

`int` **mu_warnings_off** [Variable]
: The default value is `0`. If equal to `1`, then all warnings are suppressed. Otherwise, warnings are reported

When an error, warning or message is thrown, the first thing that Mathutil does is to pass on the user the error string. The default handler will send the error string either on the user's screen (the default) or to a file stream. To request that error strings are set to a file stream, use the following routines:

`FILE *` **mu_set_stream** (`FILE *` *stream*) [Function]
: Direct all the error strings to *stream* and return the previous error stream.

`FILE *` **mu_set_stream_by_filename** (`char *` *filename*) [Function]
: Create the file *filename* and direct all the error strings to that file,

The reason why both routines return the old error stream is to allow your application to either close it or restore it at a later time.

Alternatively, you can completely override the default handler alltogether and use a custom handler that is provided by the application. This handler is called the *error reporter*. For example, in a graphical user interface application, you may prefer to report errors and warnings via dialog windows.

mu_error_reporter_t [Data Type]
: The error reporter, in general, is a function that follows the following prototype:

```
typedef void mu_error_reporter_t (char *prefix, int errnum,
  char *filename, int line, char *fmt, va_list args);
```

The arguments passed to the error reporter are obtained directly from the arguments passed on to the error reporting functions. Note that only `mu_error_at_line` passes the arguments *filename* and `line`. Also, `mu_message` does not passes `0` for *errnum*. This means that you should use *errnum* only if it is nonzero. Also, you should use *filename* and *line* only if *filename* is not equal to `NULL`. Finally, don't assume that *prefix* will be non-`NULL`. Check it first.

`void` **mu_error_stream_write** (`char *` *prefix*, `int` *errnum*, `char *` *filename*, `int` *line*, `char *` *fmt*, `va_list` *args*) [Function]
: This is the default error reporting function, provided by Mathutil.

`mu_error_reporter_t *` **mu_set_error_reporter** (`mu_error_reporter_t *` *handler*) [Function]
: Use *handler* as the new error reporting handler and return the old handler back to the user.

Mathutil, for humor value mostly, also allows you to modify the prefixes used to identify errors as fatal errors, warnings or messages. There is very little practical use for this, but it seemed like a good idea at the time.

`void` **mu_error_set_mode** (`char *` *messagestr*`, char *` *warningstr*`, char *` *errorstr*`)` [Function]

Set the message prefix (default `"MESSAGE"`) to *messagestr*, the warning prefix (default `"WARNING"`) to *warningstr*, and the fatal error prefix (default `"ERROR"`) to *errorstr*.

`void` **mu_error_surfer_mode** (`void`) [Function]

Set the prefixes to values that are likely to appeal to individuals that enjoy the use of skateboards.

2.5 Exception handling

When warnings or messages are thrown, bussiness continues as usual. If that is not what you want, in a particular instance, then the error that you throw should be a fatal error instead.

When a fatal error is thrown, Mathutil, by default, will respond in one of two ways, depending on the *status* parameter of the error. If the *status* is `1`, then the program aborts. If the *status* is `0`, then bussiness continues as usual.

Many of the routines in Mathutil can raise fatal errors. You can control what status they use for their errors with the following global variable:

`int` **mu_error_status** [Variable]

This is the error status that is used internally by mathutil when throwing fatal errors. You should set it equal to the error status that you use to report errors in your own applications. If you don't plan on doing any exception handling, then you don't need to worry about the default value of this variable (the default value is `1`).

Your own libraries should use the value of this variable, if you want them to behave consistently with Mathutil. Alternatively, your libraries should have their own global variable for controlling the error status that they use for throwing their errors. We think that general-purpose libraries should be consistent with each other, so we recommend that you use this global variable to obtain the error status when you throw errors in your own libraries. It should be left to the main application to set the policy. However, libraries that are private to your application might want to have different policies from each other, and it is good to control these policies with a global variable, to make it easier to change the policy, if that becomes necessary.

You can change the default way that Mathutil handles fatal errors, and instead ask Mathutil to jump to an error-handling portion of your program if an error happens. Doing that however is very tricky, and to do it right, you should follow the following steps:

1. Call `setjmp` to set the location where you want your program to jump to, if an error happens
2. Call `mu_error_push_jump` to push that location into the *exception stack*.
3. Run the commands that you want to run. If any of them throws an error, then your program will jump to the error handling code.

4. If all the commands run successfully, call `mu_error_pop_jump` to pop the jump location from the exception stack.

Here is a simple example of how to do exception handling:

```
jmp_buf jmp;

if (setjmp (jmp))
  {
    ...jump here in case of error...
  }
else
  {
    mu_error_push_jump (jmp);
    ...run routines that might throw an error...
    mu_error_pop_jump ();
  }
```

This example assumes that this part of the code is the only part that attempts to catch errors, and that none of the routines that you run attempt to catch errors themselves. If this assumption is not true, then we have a major problem. If an inner exception handler catches an error, then none of the outer handlers will ever see the error unless the inner handler specifically requests that control be passed to the outer handler, once it finishes doing its cleanup. This request is made by calling `mu_error_continue_next_jump`.

Here's an example of how you should construct an *inner* error handler:

```
jmp_buf jmp;

if (setjmp (jmp))
  {
    ...jump here in case of error...
    mu_error_continue_next_jump ();
  }
else
  {
    mu_error_push_jump (jmp);
    ...run routines that might throw an error...
    mu_error_pop_jump ();
  }
```

Note that if you use any local variables in the exception handling block of your code (the part that you jump to when an error happens) that have been declared on the same function as the block itself, then you should protect these variables from being optimized away by declaring them with the keyword `volatile`. Otherwise, if these variables are optimized into a register, then `longjmp` (the underlying call that makes the jump) will not be able to restore their values correctly.

For example, suppose that you want to open a file, write something and then close the file. You want an exception handler that will close the file if something goes wrong. Here's one way of doing it:

```
void foo (char *filename)
{
  jmp_buf jmp;
  volatile struct mu_file *file;

  file = mu_file_open (filename)
  if (setjmp (jmp))
```

```
        {
          mu_file_close (file);
          mu_error_continue_next_jump ();
        }
      else
        {
          mu_error_push_jump (jmp);
          write_stuff_to_file (file);
          mu_error_pop_jump ();
        }
      mu_file_close (filename);
    }
```

Note that in this example we don't want to try to close the file if opening it failed. However, we do want to try to close it if we fail writing something to it, so that the corresponding file descriptor is released (file descriptors are a valuable resource). Similarly, we don't try to close the file, if what failed was our attempt to close it! The API for these file handling routines is described in more detail in Chapter 4 [Binary file I/O], page 21.

Mathutil maintains an exception stack where it stores `jmp_buf` values. When you set up an exception handler, you push the location that you want to jump to to the stack. If no errors happen, then you simply pop it from the stack when you are done. If an error however does happen, then Mathutil pops a location from the stack and jumps to that location. Then, a call to `mu_error_continue_next_jump` will pop the next location and jump there. When the stack runs out of locations to jump to, and you still require that the program jumps again, then the error status is used to decide what to do; if the error status is 1, the program is aborted, otherwise it continues.

`void` **mu_error_push_jump** (`jmp_buf` *jmp*) [Function]
: Push the location *jmp* to the exception stack. The location that gets pushed last is the one that we jump to first, when an error happens.

`void` **mu_error_pop_jump** (`void`) [Function]
: Pop the last location pushed on to the exception stack. We do this when we no longer want errors to jump to that location.

`void` **mu_error_continue_next_jump** (`void`) [Function]
: Pop the next (outer) location from the exception stack and jump to it.

Why keep track of all this and make sure that we go through every exception handler? The reason is that exception handlers are usually written for one of two reasons: either to run clean up tasks before aborting the program, or because the program is interactive, and all we want to do is tell the user that an error happened but then continue bussiness as usual. In the first case, we want to go through *all* the cleanup that we should do before aborting. In the second case, we want to climb all the way back to the interactive application's main-loop so that bussiness can indeed continue as usual. In both cases, we want inner exception handlers to pass control on to the outer handlers.

3 Utilities for writing C programs

The Mathutil library provides a few features that make it easier to write. robust numerical applications in C. These features are not directly related to numerical computation, however they prove to be very useful.

3.1 Dynamic memory allocation

Using the ordinary `malloc`, `calloc` and `realloc` routines directly in your programs is not a good idea, because these routines simply return `NULL` pointers when they fail. In a robust program, you must check and make sure that every call to these routines is successful. If any of these routines fail, this is always a fatal error, and your application must handle it.

Mathutil provides wrappers to these routines that you can safely use in your programs without worrying about dealing with error conditions. These wrappers also respond to a few special cases that the standard routines don't handler very well. Finally, one feature of the Mathutil implementation is that we make the interfaces to `malloc` and `calloc` consistent.

`void *` **mu_malloc** (`size_t` *len*, `size_t` *size*) [Function]

Allocate and return a pointer to memory for *len* instances of a type that has size *size*. This will allocate `len*size` bytes of memory. For example, to allocate memory for `N` numbers of `struct foo`:

```
struct foo *a;
a = (struct foo *) mu_malloc (N, sizeof (struct foo));
```

If `size == 0`, then the `MU_EINVAL` error is thrown. If `len == 0`, then no error is thrown; instead a `NULL` pointer is returned. If the underlying call to `malloc` fails, the `MU_ENOMEM` error is thrown.

`void *` **mu_calloc** (`size_t` *len*, `size_t` *size*) [Function]

Does the exact same thing as `mu_malloc` except that the allocated block of memory is initialized by setting every byte of the block equal to zero.

`void *` **mu_realloc** (`void *` *p*, `size_t` *len*, `size_t` *size*) [Function]

Extend an allocated block of memory pointed by *p* such that *len* instances of a type of size *size* can fit in the block. In other words, the block is extended to reserve `len*size` bytes. If possible, the same pointer *p* will be returned, and the operating system will simply reserve more memory. Sometimes, this is not possible however because some of the memory in front of the original block might belong to another block. Then, a completely new block is allocated, the contents of the old block are copied into the new block, and the old block is freed.

If `p == NULL` then, unlike the standard `realloc` in Unix, the Mathutil `mu_realloc` simply behaves like a call to `mu_malloc`. If `p != NULL`, then one of the following may happen: If `len == NULL`, then the block of memory is freed and the NULL pointer is returned. If `size == NULL`, then the `MU_EINVAL` error is thrown. If the call to `realloc` fails, then the `MU_ENOMEM` error is thrown.

`void` **mu_free** (`void *`*p*) [Function]

Frees the block of memory pointed to by *p*. If `p == NULL` then the MU_EFAULT error is thrown.

In general, with numerical applications array allocations should be handled using smart arrays (see Chapter 5 [Smart arrays], page 29. However, you should use these routines to allocate memory for other kinds of data structures.

A common use of `mu_realloc` is with routines that require some kind of temporary buffer whose size may change from call to call. Rather than continuously allocate and free the block of memory, a working strategy is to simply *realloc* it whenever a need arises for more memory, and keep it around. Here's an example:

```
int mu_file_read_cookie (struct mu_file *file, char *cookie)
{
  static unsigned char *buffer = NULL;
  static int buffer_length = 0;

  int len = strlen (cookie) + 1;
  if (len > buffer_length)
    {
      buffer = mu_realloc (buffer, len, sizeof (unsigned char));
      buffer_length = len;
    }
  mu_file_read (file, buffer, len);
  if (buffer[len-1] != '\0') return 0;
  if (strcmp (cookie, buffer) != 0) return 0;
  return 1;
}
```

This is the implementation of the `mu_file_read_cookie` function used by Mathutil to read in a cookie from a binary file and compare it with an expected string. (see Chapter 4 [Binary file I/O], page 21) The intelligent behaviour of `mu_realloc`, that it reverts to a call to `mu_malloc` when you pass it as an argument the `NULL`, pointer, is very useful in cases such as this, because this way we can correctly handle the *first* call to this function without any extra work. Note, of course, that the memory is never allocated. It is remembered, and it will be reused in the next call to this function.

3.2 Defining variadic functions in C

Variadic functions are C functions that take a variable number or type of arguments. An example of a variadic function is the `printf` function, defined by the C standard library. In this section we describe the Mathutil API for writing new variadic functions.

3.2.1 Concepts with variadic functions

When you invoke a variadic function, the function has no way of knowing the type of each argument; the variable arguments are laid out it in a block of memory but there are no clear marks that indicate where one argument begins and another ends. This is why variadic functions usually take a mandatory string argument that tells the functions what *type* of arguments to expect. This string argument is often called the *format string*. For example, a call to `printf` might look like this:

```
int a = 10;
double b = 20.3;
printf ("%d %lf", a, b);
```

The `%d` token indicates that the first argument is an integer. The `%lf` token indicates that the second argument is a double precision floating point number. Then these two arguments follow.

In general, the logic with variadic functions often is about doing a certain operation with every one of the variable arguments. In the case of `printf`, this operation is printing the values of the arguments. The idea is that rather than explicitly having to call many different functions for each argument, we prefer to have one variadic function that can do that for us concisely. So behind the implementation of a variadic function, there are handlers that do the actual work, which are called once for every argument.

While we were implementing Mathutil, we needed to implement two variadic functions for input and output to binary files (see Section 4.1 [Simple use of binary files], page 21). We've realized then that the developer of numerical software might need to use variadic functions again in a different context. For example, when writing programs for parallel computers, it is often necessary to pass data between processors. One might want to have variadic functions for sending and receiving many different variables from processor to processor in one function call. So we have taken some extra effort to generalize the ideas involved in implementing variadic functions and create a higher-level API that developers can use to write variadic functions. We have used this API in our own implementation of `mu_file_input` and `mu_file_output` and you can use it in your programs as well.

In general, a variadic function has a few fixed arguments, a format string, and the variable arguments. For example, the `mu_file_input` function has the following prototype:

```
void mu_file_input (struct mu_file *file, char *fmt, ...);
```

The *file* argument is a fixed argument. The format string *fmt* describes the number and the type of the arguments that follow in this invokation, and then follow the arguments. The idea is that we want this function to launch the correct input function for each argument (a different input function is used for every different type of data that you want to input), in the order in which the arguments are listed.

The format string *fmt* contains tokens that correspond to the data types that are being read. Each token begins with `%` and ends either by another `%`, or by a space, or by the string's termination. The characters that follw the `%` define the token. Any spaces or other characters between tokens are completely ignored. For each token, there must be a corresponding variadic argument. The correct type of the argument is defined by the token and by the actual handler routine. The functions `mu_file_input` and `mu_file_output` are good examples of how all this works.

The semantics that are supported by our API are not as general as you might think. Because characters that are not part of a token are ignored, it is not possible to use this API to reimplement `printf`. However, it still is powerful enough to handle many useful situations that are not as demanding as `printf`, like I/O of raw data (as opposed to the *formatted* data that `printf` prints).

3.2.2 The variadic handler function

To implement new variadic functions, the most crucial component is to write the *variadic handler function*. This function is responsible for launching the correct routine for each argument, so it is the heart of the implementation. You need to have at least one handler

function, but you can add more handler functions later. If a handler function doesn't know how to handle a specific argument, it passes it on to the next handler function on the list.

mu_var_handler_t [Data Type]

The type for variadic handler functions. These functions have the following declaration:

```
int mu_var_handler_t (void *clientdata, char *spec, va_list *arg);
```

Upon, input:

clientdata contains a pointer to data that are passed on to the handler by the variadic function. These data are usually formed by the fixed arguments that are passed to the variadic function. For example, they may contain the file where you want to write output to, or the processor that you want to send your data to.

spec contains the token that corresponds to the argument that you want to handle. For example, if the token is `"%dm"`, then *spec* contains `"dm"`.

arg is a pointer to the `va_list` structure that points to the block of memory where the variable arguments are being stored. To dereference the current argument, use the `va_arg` macro.

If the handler knows how to handle this argument, and has successfully done so, it returns `1`. This depends on whether the handler knows how to deal with arguments that correspond to the token passed on by *spec*. Otherwise, it returns `0`.

struct mu_var_handler_table [Data Type]

This is a linked list of pointers to variadic handler functions. We use this list to try all the handlers that are available until one of them matches and handles the current argument.

`mu_var_handler_t *handler`
: Pointer to a variadic handler function.

`void *next`
: Pointer to the next `struct mu_var_handler_table` containing the next variadic handler function. If `NULL`, then this is the last handler function that is available.

Here's a (trimmed down) example of a handler function that we use in our implementation of `mu_file_output`. The idea behind this handler function is that we want to allow semantics like this:

```
int i = 10;
float f = 20.2;
double d = 43.23;
mu_file_output (file, "%i %d %f", i, d, f);
```

In this example, note that the variable arguments are passed by value. The handler implementation that can handle these arguments is as follows:

```
static int
mu_file_io_default_output_handler (void *clientdata, char *spec,
                                   va_list *arg)
```

```
{
  struct mu_file *file = (struct mu_file *) clientdata;

  /* Handlers for simple types */
  if (strcmp (spec, "i") == 0)
    {
      int value = va_arg (*arg, int);
      mu_file_write_int (file, &value, 1);
      return 1;
    }
  if (strcmp (spec, "f") == 0)
    {
      /* handle type promotion */
      float value = (float) va_arg (*arg, double);
      mu_file_write_float (file, &value, 1);
      return 1;
    }
  if (strcmp (spec, "d") == 0)
    {
      double value = va_arg (*arg, double);
      mu_file_write_double (file, &value, 1);
      return 1;
    }
  return 0;
}
```

Note that the actual implementation that we use in Mathutil also contains more code to handle smart arrays. There are quite a few points that you need to keep in mind when you write your own handlers that are brought up in this implementation:

1. We use *clientdata* to pass on to the hander the opaque handle to the binary file to which we want to output the arguments. Before using the data passed by *clientdata*, it is necessary to do the appropriate type cast.

2. We use `va_arg` to dereference the current argument from the variable argument list *arg*. However, in order for the dereference to work correctly, we need to know the type of the current argument and pass it on to *va_arg*. The token passed on to us through *spec* is used to determine the type.

3. Since the prototype of a variadic function doesn't specify types for the optional arguments, the following "default argument promotions" are performed on these arguments:

   ```
   char, short int    ↦ int
   unsigned char      ↦ unsigned int
   unsigned short int ↦ unsigned int
   float              ↦ double
   ```

   This means, for example, that when you pass a `float` f to `mu_file_output`, it must be received with `va_arg (*arg, double)`, since f is promoted to `double`. This only affects the types listed above. Pointers and `struct` types are not promoted.

4. If the handler successfully processes an argument, it must return `1`. Otherwise, it must return `0`. It is very important that you don't forget to return the correct value.

3.2.3 Using the variadic function API

Mathutil provides the following two functions that you can use to write the actual variadic function interface to your handler:

`void` **mu_var_register** (`struct mu_var_handler_table *` *table*`, mu_var_handler_t *` *handler*`)` [Function]

Add a new variadic function handler *handler* to the handler table *table*. Use this function to write a function that extends the set of types recognized by your variadic function by adding a new handler. Your function should hide the actual table used from the user.

`void` **mu_var_call** (`struct mu_var_handler_table *` *table*`, void *` *clientdata*`, char *` *routine*`, char *` *fmt*`, va_list *` *arg*`)` [Function]

Pass the variable arguments pointed to by *arg* to the handlers assembled by *table*. For each argument, extract its token from the format string *fmt* and pass it along to the handlers. Also pass to all the handlers the pointer *clientdata*, which points to additional information that the handlers may need to do their job. If none of the handlers succeeds in processing any one of the arguments, then raise a fatal error and report to the user the name of the offending routine, as given in *routine*.

The implementation of `mu_file_input` and `mu_file_output` are a very lively example of how to set up the implementation of variadic functions.

1. On the header file '`mu.h`', we declare the following three functions:

   ```
   void mu_file_input (struct mu_file *file, char *fmt, ...);
   void mu_file_output (struct mu_file *file, char *fmt, ...);
   void mu_file_register (mu_var_handler_t *input,
                          mu_var_handler_t *output);
   ```

   The first two are the actual input and output functions. The third function registers a new pair of handlers. There is a good reason why we want to encourage developers to register new handlers in pairs; every new data type must have both an input and an output handler.

2. On the source file '`mu_file.c`', we first implement the default handlers for *mu_file_input* and *mu_file_output*:

   ```
   mu_file_io_default_input_handler
   mu_file_io_default_output_handler
   ```

   We've already shown you an excerpt of the output handler. For the full source for both handlers, see the file '`mu_file.c`' in the source code.

3. Then we define the tables that contain the current handlers:

   ```
   static struct mu_var_handler_table mu_file_io_input_handlers[] =
   {
     { mu_file_io_default_input_handler, NULL }
   };

   static struct mu_var_handler_table mu_file_io_output_handlers[] =
   {
     { mu_file_io_default_output_handler, NULL }
   };
   ```

Initially, these tables contain only the default handlers and a `NULL` pointer to the *next* fields. When you register more handlers, the contents of these tables will be modified.

4. The routine for registering new handlers, is now very easy to write. All it has to do is issue the appropriate calls to `mu_var_register`:

```
void mu_file_register (mu_var_handler_t *input,
                       mu_var_handler_t *output)
{
  mu_var_register (mu_file_io_input_handlers, input);
  mu_var_register (mu_file_io_output_handlers, output);
}
```

   Note that the actual tables involved are declared `static`. They can not be accessed by other files. So the only way to modify their contents is through this routine. You should follow a practice similar to this in your own code.

5. Finally, we write the implementations of the actual variadic functions. These implementations initialize the variable arguments, and call `mu_var_call` that does the actual work of going throught the arguments and launching the handlers.

```
void mu_file_output (struct mu_file *file, char *fmt, ...)
{
  va_list args;
  va_start (args, fmt);
  mu_var_call (mu_file_io_output_handlers, file,
       "mu_file_output", fmt, &args);
  va_end (args);
}

void mu_file_input (struct mu_file *file, char *fmt, ...)
{
  va_list args;
  va_start (args, fmt);
  mu_var_call (mu_file_io_input_handlers, file,
       "mu_file_input", fmt, &args);
  va_end (args);
}
```

   It is very important to initialize the variable `args` to point to the optional argument memory block by calling `va_start`. After calling `mu_var_call`, we call `va_end` to clean it up.

4 Binary file I/O

Numerical simulations typically produce large amounts of data, therefore it is efficient to save that data as binary files, instead of as text files. Mathutil provides an flexible interface for reading and writing binary files.

4.1 Simple use of binary files

If you just want to read and write to files created by Mathutil based programs, then all you really need is the information in this chapter. If you would like to write programs that import binary files that are created by other programs, then you should read the entire chapter.

struct mu_file [Data Type]

The Mathutil handle to an opened binary file. The only documented (i.e. public) members of this data structure that you need to know about are:

`int` *`descriptor`*
: The file descriptor corresponding to the file

`char *` *`filename`*
: The file name corresponding to the file.

`int` *`swap_endian`*
: A flag that equals 1 if the file has opposite endianness from the CPU, and 0 if the file has the same endianness as the CPU. See Section 4.4 [Dealing with endian compatibility], page 26, for more details.

Don't modify these members directly. The only manipulations you should make on the binary file handler is through the Mathutil API.

`struct mu_file *` **mu_file_open** (`char *` *`filename`*, `int` *`filemode`*) [Function]

Open the file with filename *filename* and return a binary file handler to the caller. The *filemode* argument defines whether the file is opened for reading, writing or appending. It can have one of the following values:

`mu_file_for_reading`
: Opens the file for reading. To determine if the file is little-endian or big-endian, the `mu_file_open` call expects an endian cookie which is automatically written when Mathutil creates a new file. See Section 4.4 [Dealing with endian compatibility], page 26, for more details about the endian cookie.

`mu_file_for_writing`
: Open the file for writing. If there is any file with the same filename, it's contents are erased. Then, it detects the endian of the current platform and writes an endian cookie to the file.

`mu_file_for_appending`
: First, opens the file for reading and loads the endian cookie to determine if the file is little-endian or big-endian. Then it closes and opens the file

again for appending. If the file does not exist, then an error is thrown. This option should really only be used to append to a file. When you append data to the file, it is written using the endian convention that is compatible with the endian cookie in the beginning of the file.

`void` **mu_file_close** (`struct mu_file *` *`file`*) [Function]
Close the file that corresponds to the file handle *file*.

`void` **mu_file_input** (`struct mu_file *` *`file`*, `char *` *`fmt`*, `...`) [Function]
Read data from file *file*. The string *fmt* indicates what type of data to expect. The variadic arguments that follow *fmt* are pointers to the variables where the data will be stored once it's read in.

For example, the following invocation will read an integer, a float and then a double from the binary file stream:

```
int i;
float f;
double d;
mu_file_input (file, "%i %f %d", &i, &f, &d);
```

The usage is similar in spirit to that of `fscanf` from the Standard C Library, but there are major differences.

The format string *fmt* contains tokens that correspond to the data types that are being read. Each token begins with `%` and ends either by another `%`, or by a space, or by the string's termination. The characters that follow the `%` define the token. Any spaces are completely ignored. For each token, there must be a corresponding variadic argument that is a pointer to the variable where data is being loaded in. This pointer must have the same type as the variable; it should **not** be a `void` pointer.

The following format strings are recognized:

`%d` Read a `double`

`%f` Read a `float`

`%i` Read a `int`

`%da, %fa, %ia`
Read a `struct mu_double_array`, or `struct mu_float_array` or `struct mu_int_array`. See Chapter 5 [Smart arrays], page 29, for more details. First read one `int`, that provides the size of the array. Then read the array. Then read another `int`, which should be the array size again. The array must be initialized to have size consistent with the size of the array on file.

`%daL, %faL, %iaL`
Read a `struct mu_double_array`, or `struct mu_float_array` or `struct mu_int_array`. However, reallocate the arrays to the size indicated by the array record on file.

`%dm, %fm, %im`
Read a `struct mu_double_matrix`, or `struct mu_float_matrix`, or `struct mu_int_matrix`. First we read two `int`, that provide the `X` and

`Y` sizes of the matrix. Then we read the matrix, in `X`-first order. Then we read two `int` again, which should be the same `X` and `Y` sizes of the matrix, as in the beginning. The matrix must be initialized to have size consistent with the size of the matrix on file.

`%dmL, %fmL, %imL`
: Read a `struct mu_double_matrix`, or `struct mu_float_matrix`, or `struct mu_int_matrix`. However, reallocate the arrays to the size indicated by the array record on file.

`%dc, %fc, %ic`
: Read a `struct mu_double_cube`, or `struct mu_float_cube`, or `struct mu_int_cube`. First we read three `int`, that provide the `X`, `Y`, `Z` sizes of the cube . Then we read the cube, in `X`-first, `Y`-second and `Z`-third order. Then we read three `int` again, which should be the same `X`, `Y`, `Z` sizes ofthe cube, as in the beginning. The cube must be initialized to have size consistent with the size of the matrix on file.

`%dcL, %fcL, %icL`
: Read a `struct mu_double_cube`, or `struct mu_float_cube`, or `struct mu_int_cube`. However, reallocate the arrays to the size indicated by the array record on file.

If you use the `*L` tokens to require that the incoming arrays are reallocated according to the size indicated by the input file, then you should initialize these arrays first to the corresponding *null arrays* (see Section 5.3 [Smart array data types], page 32). The `*L` tokens are a matter of convenience, however using them may result in a program that is not robust. To make your program robust, you have to assert that the file that you are reading is not corrupt. It is not possible to assert with absolute certainty. If you do indeed read a corrupt size record, mathutil might attempt an invalid memory reallocation and crash. Still, you can attain some robustness by using *cookies* to assert that the file you are reading is consistent with what you expect to read (see Section 4.3 [Using cookies], page 25).

`void` **mu_file_output** (`struct mu_file *` *`file`*`, char *` *`fmt`*`, ...`) [Function]

Write data from file *file*. The string *fmt* indicates what type of data to expect. The variadic arguments that follow *fmt* are the values of the variables whose contents we want to write to the binary file.

For example, the following invocation will write an integer, a float and then a double to the binary file stream:

```
int i;
float f;
double d;
mu_file_output (file, "%i %f %d", i, f, d);
```

The usage is similar in spirit to that of `fprintf` from the Standard C Library, but there are major differences.

The format string *fmt* contains tokens that correspond to the data types that are being read. Each token begins with `%` and ends either by another `%`, or by a space, or by the string's termination. The characters that follow the `%` define the token.

Any spaces are completely ignored. For each token, there must be a corresponding variadic argument that is a pointer to the variable where data is being loaded in. This pointer must have the same type as the variable; it should **not** be a `void` pointer.

The following format strings are recognized:

`%d` Write a `double`

`%f` Write a `float`

`%i` Write a `int`

`%da, %fa, %ia`
Write a `struct mu_double_array`, or `struct mu_float_array` or `struct mu_int_array`, such that it can be read by a corresponding invocation of `mu_file_input`.

`%dm, %fm, %im`
Write a `struct mu_double_matrix`, or `struct mu_float_matrix`, or `struct mu_int_matrix`, such that it can be read by a corresponding invocation of `mu_file_input`.

`%dc, %fc, %ic`
Write a `struct mu_double_cube`, or `struct mu_float_cube`, or `struct mu_int_cube`, such that it can be read by a corresponding invocation of `mu_file_input`.

4.2 Testing for end of file

When reading a file, Mathutil provides the following routine for detecting whether you have encountered the end of the file while trying to read one more piece of data from the file:

`int` **mu_file_eof_p** (`struct mu_file *`*file*) [Function]
When reading a file, return 1 if an *end of file condition* has been raised. Return 0 otherwise.

Note that Mathutil can *not* detect that you have reached the end of the file, until an attempt to read the file fails because you've run out of the file! This means that you have to do some careful coding when processing files whose length you can not predict.

Consider for example, how you would write a routine that would load a file of `double` and return the sum of its contents:

```
double load_and_sum (char *filename)
{
  struct mu_file *file;
  double sum = 0;
  double a;

  file = mu_file_open (filename, mu_file_for_reading);
 next_element:
  mu_file_input (file, "%d", &a);
  if (mu_file_eof_p (file) == 0)
    {
      sum += a;
      goto next_element;
```

```
    }
  mu_file_close (file);
  return sum;
}
```

Note that our use of `goto`. Some Pascal purists may object to that. However, using `goto` is the best way to convey the idea that the algorithm is *recursive*; that we should recurse, if reading this character worked.

Mathutil makes it possible to define an exception handler for responding to end of file conditions. You should not use this error handler when you are reading files whose size you don't know. You should use it when you are not paying attention to the end of file condition because you believe that the file's size is predictable. If your belief is wrong, if the file, for example, has been truncated during writing because the disc was full, or if your reader simply has a bug, then the exception handler will be invoked when an attempt to read data raises the end of file condition.

mu_file_eof_handler_t [Data Type]

This is the `typedef` definition of the error handler:

```
typedef void mu_file_eof_handler_t (struct mu_file *file);
```

The only arguments that you must pass to the error handler, is the *file* where the end of file condition has happened.

`mu_file_eof_handler_t *` **mu_file_set_eof_handler** (`struct mu_file *`*`file`*`, mu_file_eof_handler_t *`*`handler`*) [Function]

Set the error handler for *file* to *handler*. Return the previoud handler. If the handler is set to `NULL` then it is disabled.

`mu_file_eof_handler_t *` **mu_file_set_default_eof_handler** (`struct mu_file *` *`file`*) [Function]

Set the error handler to a convenient default. The default error handler will raise an error with `mu_error` if your program encounters an end of file condition. Return the previous handler.

In general, the default eof handler should be sufficient.

FIXME: An example is in order.

4.3 Using cookies

Cookies are a null terminated string of characters that is embedded, usually in the beginning or near the beginning, of a binary file. The purpose of cookies is to label the file, so that when an application attempts to read it it can verify that the file is what it think it is.

For example, suppose that you have a simulation that produces three different types of output files: '`X`', '`Y`' and '`Z`', and now you want to write a postprocessing tool for each type of file. If you embed distinct strings, such as `"X"`, `"Y"`, `"Z"` in the beginning of these files, then you can use these strings in your postprocessing tools to detect whether or not you are loading the right type of file.

`void` **mu_file_write_cookie** (`struct mu_file *` *`file`*`, char *` *`cookie`*) [Function]

Label the file corresponding to *file* with the cookie *cookie*.

`int` **mu_file_read_cookie** (`struct mu_file *` *file*`, char *` *cookie*) [Function]
Read in the current contents of the file *file* and check whether the cookie *cookie* is there. If it is, return 1, otherwise return 0. This routine will read in characters from the file until the next terminating `NULL`. When it encounters a terminating `NULL` it compares the string accumulated with the string in *cookie*.

A file may be labeled by more than one cookies. For example, every file contains the endian cookie, so any cookies you write to the file will follow the endian cookie. It is also okey to insert cookies to separate various sections of the binary file. This way you have another safety net for checking if the file that you are reading is in a sane state. It is important however to read the cookie at the right moment, in order to detect it.

4.4 Dealing with endian compatibility

A `float` and an `int` is made of 4 bytes, and a `double` is made of 8 bytes. The IEEE standard for floating point number representation specifies what the bits in each of these bytes do. However, different machines differ in how they order the bytes themselves. They may order the bytes either in *little endian format* or in *big endian format*. Reversing the order of bytes from `1234` to `4321` in `float` and `int`, (and correspondingly from `12345678` to `87654321` in `double`) switches from one representation to the other. Whether a system will be little-endian or big-endian is not a matter of the operating system. It depends on the CPU of your platform.

This creates a serious problem, when you decide to save data as binary files and share them between two computers that use different endian conventions: a big-endian file will look like garbage to a little-endian computer. Mathutil addresses this problem by handling the numerical data as sequences of bytes and reversing their order for you automatically whenever it is appropriate.

To make this work, Mathutil needs to detect the endianness of the CPU and the endianness of the file. You can detect the endianness of the CPU with the following routines, which you can use in your programs:

`int` **mu_endian_detect** (`void`) [Function]
Return `mu_file_big_endian` if the cpu is big endian and `mu_file_little_endian` if the cpu is little endian.

`char *` **mu_endian_to_string** (`int` *endian*) [Function]
Convert the endian flag *endian* to a string:

```
mu_file_big_endian    ⇒ "big"
mu_file_little_endian ⇒ "little"
```

This is useful when you want to print a message about endianness.

`int` **mu_endian_from_string** (`char *`*str*) [Function]
Convert the string *str* to an endian flag: "big" ⇒ mu_file_big_endian "little" ⇒ mu_file_little_endian This is very useful when parsing command-line arguments.

`int` **mu_endian_valid_filetype_p** (`char *`*str*) [Function]
Check whether the string *str* is equal to `"little"` or `"big"`. Return 1 if yes. Return 0 if no. This is very useful when parsing command-line arguments.

`void` **mu_endian_swap4** (`void *c`) [Function]
Swap the order of the next 4 bytes pointed to by *c*.

`void` **mu_endian_swap8** (`void *c`) [Function]
Swap the order of the next 8 bytes pointed to by *c*.

`void` **mu_endian_swap4_block** (`void *c, size_t len`) [Function]
Swap the order of the next *len* quadraplets of bytes pointed to by *c*. Saves some efficiency by avoiding a function call to `mu_endian_swap4`.

`void` **mu_endian_swap8_block** (`void *c, size_t len`) [Function]
Swap the order of the next *len* octaplets of bytes pointed to by *c*. Saves some efficiency by avoiding a function call to `mu_endian_swap8`.

Mathutil detects the endianness of the file using the *endian* cookie. If you open file for reading using `mu_file_open`, then Mathutil expects your file to begin with the cookie `"endian"`, followed by the character '`l`', if the file is little-endian, or the character '`b`', if the file is big-endian. If you open the file for writing, then Mathutil will create for you an endian cookie that makes the file have the same endianness as the machine on which the program is running at the moment. If you open the file for appending, Mathutil uses the same endianess for writing that the file has been originally created with.

In some cases however, you might want to access binary files written by programs by other people. Such files are not going to have an endian cookie. To bypass the endian cookie mechanism, you should use `mu_file_open_raw` instead to open the file, and then you *must* set the endianness of the file by calling `mu_file_set_endian`. You can read and write to such files with `mu_file_input` and `mu_file_output`, or with the lower level input and output routines (see Section 4.5 [Low-level interface to binary files], page 28). However beware that you might not be able to read and write arrays if the file does not follow the Mathutil conventions. The Mathutil convention is that every array record begins with the a record of the size of the array, then the contents of the array, and then with a repetition of the record of the size of the array. This convention is different from the one used by Fortran, and it may be different from the convention used by your coworkers. Then you will have to provide your own loop and read the elements of the array one by one as single numbers.

`struct mu_file *` **mu_file_open_raw** (`char *filename, int filemode`) [Function]
Open the file *filename* and return a binary file handler to the caller. The *filemode* argument defines whether the file is opened for reading, writing or appending, and it follows the same conventions as the in `mu_file_open`. See Section 4.1 [Simple use of binary files], page 21, for more details. The difference between this routine and `mu_file_open` is that it does not do any automatic handling for the endian cookie.

`void` **mu_file_set_endian** (`struct mu_file *file, int endian`) [Function]
Define the endianness of the file pointed to by the handler *file* to be *endian*. The variable *endian* will take one of the following values:

```
mu_file_little_endian
mu_file_big_endian
```

`void` **mu_file_assert_endian** (`struct mu_file *`*file*`, char *`*routine*`)` [Function]

Assert whether or not the file stream *file* has set its endian. If the file does not have a set endian, then an error is raised. This is already checked for you when you read and write `int`, `float` and `double` via Mathutil. However, if you use the lower level commands `mu_file_read`, and `mu_file_write` then if endianness is important, you should check it yourself by examining `file->swap_endian` (see Section 4.1 [Simple use of binary files], page 21) and doing swaping if appropriate by calling the `mu_endian_swap*` routines. You really shouldn't have to deal with this though.

4.5 Low-level interface to binary files

Mathutil provides the following routines for reading and writing data directly from memory to a binary file. These routines are more robust than the lower-level `read` and `write` system calls in that they check for errors, and try again if the error is recoverable or raise an error.

`void` **mu_file_read** (`struct mu_file *`*file*`, void *`*p*`, size_t` *len*`)` [Function]
Read *len* bytes from the file *file* and store them in the address *p*.

`void` **mu_file_write** (`struct mu_file *`*file*`, void *`*p*`, size_t` *len*`)` [Function]
Write *len* from the memory address *p* to the file *file*.

These routines do not take into account endianness; they just read and write bytes. The following routines can be used to read and write numbers, and will handle endianness automatically:

`void` **mu_file_write_double** (`struct mu_file *`*file*`, double *a, size_t` *n*`)` [Function]
Write *n* doubles to file *file* starting from the address *a*.

`void` **mu_file_write_float** (`struct mu_file *`*file*`, float *a, size_t` *n*`)` [Function]
Write *n* floats to file *file* starting from the address *a*.

`void` **mu_file_write_int** (`struct mu_file *`*file*`, int *a, size_t` *n*`)` [Function]
Write *n* ints to file *file* starting from the address *a*.

`void` **mu_file_read_double** (`struct mu_file **`*file*`, double *a, size_t` *n*`)` [Function]
Read *n* doubles from file *file* and store them in memory starting from the address *a*.

`void` **mu_file_read_float** (`struct mu_file *`*file*`, float *a, size_t` *n*`)` [Function]
Read *n* floats from file *file* and store them in memory starting from the address *a*.

`void` **mu_file_read_int** (`struct mu_file *`*file*`, int *a, size_t` *n*`)` [Function]
Read *n* ints from file *file* and store them in memory starting from the address *a*.

It is probably easier however to use the higher level routines `mu_file_input` and `mu_file_output`. See Section 4.1 [Simple use of binary files], page 21, for details.

5 Smart arrays

5.1 Introduction to smart arrays

Smart arrays are data structures that allow you to represent array, matrix and 3d cube data in a high-level and error free fashion. The word *smart* means that the corresponding array data structures encapsulate the following information:

1. *Array size*. Smart arrays know their size. When you pass a smart array to a function, you are not burdened with providing size information separately.
2. Array reference. It is possible to define arrays that refer to parts of another array. For example it is possible to define an array that refers to a specific row or column of a matrix array. In order to make this work, array structures store and manipulate *stride information* that describes how the data is laid out in memory.
3. Reference counting. When one array refers to the contents of another array, a request to free the memory in one array is delayed until a similar request is turned in for all the refering arrays as well.
4. Error-checking indexing. When activated, the Mathutil library will check every smart array element access for possible index errors. This is extremely useful for debugging numerical software!
5. Input and output. It is possible to read and write array data to and from a file in binary format. Endian-ness issues are also dealt with automagically.

Smart arrays are useful mainly for two reasons: They make APIs to numerical functions that manipulate arrays, matrices and cubes simpler, and their usage less error-prone. And they make it easier to make APIs that interface to Fortran packages by using Fortran conventions and by providing a clean way to deal with the array size and stride.

All of the smart array data structures are based on

```
struct mu_membuf;
```

which implements a reference-counting memory buffer. See Section 5.2 [Memory buffers], page 30, for details. Mathutil defines the following array data structures:

```
struct mu_type_array;
struct mu_type_matrix;
struct mu_type_cube;
```

where `type` can be one of the following:

```
double, float, int
```

Mathutil uses a slick C preprocessor trick to duplicate the smart array API for every one of these types. To keep this documentation concise we will use the word `type` to refer to any one of these types. So when we talk about the interface of:

```
struct mu_type_array mu_type_array_alloc (size_t x)
```

what we really mean is that the following interfaces are all available:

```
struct mu_double_array mu_double_array_alloc (size_t x)
struct mu_float_array mu_float_array_alloc (size_t x)
struct mu_int_array mu_int_array_alloc (size_t x)
```

Here's how we plan to describe the smart array API in this manual:

5.2 Memory buffers

Numerical simulations require big chunks of memory in which to store and manipulate numerical data. In C it is common to allocate memory dynamically during runtime and free it when it is no longer useful. However in numerical simulations this is not always the best approach, because memory allocation and release are often very expensive operations, in terms of CPU time. Instead, numerical simulations often allocate large chunks of memory that they need in advance and use it to store various arrays to it throughout run-time. Such large chunks of memory are often called *workspace* or *memory buffers*. In Mathutil we will prefer to use the term *memory buffer*.

In Mathutil, a memory buffer knows three pieces of information: the memory address that has been reserved for storing data, the number of bytes that have been reserved, and a *reference counter*. Memory buffers can be *allocated*, *attached* to other memory buffers or *released*. They can also be queried to return their size and the memory address that they have reserved.

Here's the data type that encapsulates the memory buffer in Mathutil:

struct mu_membuf [Data Type]

This data type encapsulates a reference counting memory buffer. It contains the following fields:

`void *data`
: The memory address that stores the contents of the memory buffer.

`size_t size`
: The size of the memory available for storing data.

`size_t *refcount`
: The memory address of the reference counter.

Memory buffers can be either *allocated* or *attached* to another memory buffer. It is also possible to defer either action by setting the buffer equal to the *null buffer*.

When you *allocate* a memory buffer, *data* is set to point to a newly allocated block of memory and *size* is set equal to the size of that block. Also a reference counter is allocated and pointed to by *refcount*, and then it is initialize equal to `1`.

When you *attach* a memory buffer to another memory buffer, we call that other memory buffer the *parent buffer*. Then, *data* points to the block of memory that is owned by the parent buffer and *size* copies the corresponding size from the parent buffer. Also *refcount* is set to point to the reference counter used by the parent buffer, and then the counter is incremented by `1`. This reflects on all the memory buffers that are attached together since they all share the same reference counter through the pointer.

When you *release* a memory buffer, the reference counter is decremented by `1`. If, as a result, the reference counter is equal to zero, this means that no other buffers are attached to this buffer, and it is safe to collect the memory allocated to this buffer.

When you *reallocate* or *reattach* a memory buffer, it is released first, and then it is correspondingly allocated or attached again.

Because a newly-declared buffer contains garbage, it is impossible to determine its state. Therefore, such a buffer can not be reallocated, reattached or released. Before you use such

a buffer, you must first allocate it or attach it to another buffer. Alternatively, you can explicitly *mark* the buffer as *uninitialized* by assigning it equal to the *null memory buffer*. This way, the buffer's state can be determined, and when you request it to be reallocated or reattached, if Mathutil notices that the buffer is *unitialized*, it will correspondingly allocate it or attach it instead. This is often very useful in many situations.

Here's the definition of the null memory buffer:

`struct mu_membuf` **mu_membuf_null** [Variable]

The *null* memory buffer contains the following values for its fields:

```
data = NULL;
size = 0;
refcount = NULL;
```

To set a buffer equal to the null memory buffer, just use simple assignment:

```
struct mu_membuf buf;
buf = mu_membuf_null;
```

The following functions will create a new memory buffer either by allocating it or by attaching it to a new buffer. You should only use these functions to initialize a newly declared memory buffer.

`struct mu_membuf` **mu_membuf_allocate** (`size_t` *size*) [Function]

Allocate and return a memory buffer of size *size*.

`struct mu_membuf` **mu_membuf_attach_to** (`struct mu_membuf` *parent*) [Function]

Return a memory buffer that is attached to *parent* and increment the reference counter.

Once a memory buffer has already been initialized, you can reallocate it or reattach it to another memory buffer. You can also do these things if you have assigned the memory buffer to the null memory buffer. To do this, use the following functions:

`void` **mu_membuf_reallocate** (`struct mu_membuf *`*buf*`, size_t` *size*) [Function]

Reallocate the memory buffer *buf* to a block of memory of size *size* and handle reference counting appropriately. The previous block pointed to by *buf* is released.

`void` **mu_membuf_reattach_to** (`struct mu_membuf *`*buf*`, struct mu_membuf` *parent*) [Function]

Reattach the memory buffer *buf* to the parent buffer *parent*, and handle reference counting appropriately. The previous block pointed to by *buf* is released.

Warning: Do not use these routines on a buffer that has only been declared, because such a buffer's contents are garbage and it is impossible to determine its status. Assigning the buffer equal to `mu_membuf_null` will clear that garbage.

Once you've put together a memory buffer, you can use the following functions to obtain information from it:

`int` **mu_membuf_references** (`struct mu_membuf` *buf*) [Function]
Return the total number of buffers that refer to the same block of memory as buffer *buf*. If the buffer *buf* is the null buffer, then this function returns `0`.

`void *` **mu_membuf_address** (`struct mu_membuf` *buf*) [Function]
Return the memory address of the block of memory refered to by buffer *buf*. If the buffer *buf* is the null buffer, then this function returns `NULL`.

Once you are done with using a memory buffer, you can release it with the following function:

`void` **mu_membuf_release** (`struct mu_membuf` *buf*) [Function]
Decrement the reference counter in *buf* and set it equal to the null buffer. If the reference counter drops to zero, free the associated memory block. If you explicitly release a memory buffer, then you can use the functions `mu_membuf_allocate` and `mu_membuf_attach_to` to reinitialize it to something else.

In general, you don't need to use memory buffers directly. Instead you get to work with smart arrays who handle memory buffers for you automatically. Occasionally, it is often useful to explicitly use a memory buffer to refer to a large block of memory that you reserve to be used for temporary storage by an algorithm. Usually such complicated algorithms are encapsulated by a C structure that stores the state of the algorithm. Along with the state, you can also include a few memory buffers as additional members to that structure. When you write the API for creating and releasing such an algorithm structure, you will need to use the memory buffer API to create and release the corresponding member memory buffers. Also, when you write methods for running the algorithm, you will want to attach smart arrays either to pieces of the memory buffer or to the entire memory buffer. *FIXME: Crossreference* for more details. From the user's standpoint it is important to shield the user from having to manipulate memory buffers directly as much as possible.

5.3 Smart array data types

A *smart array* is a memory buffer with additional information that allows you to index some of the contents of that buffer. There are many ways of thinking about these contents. You can have them arranged along a straight line; then you have a *one dimensional array*. For simplicity we will call such things *arrays*. You can also have them arranged along a rectangular block; then you have a *matrix*. Finally you can have them arranged along a three dimensional parallelogram; then you have a *cube*.[1]

It is not necessary for all of the contents of the memory buffer to be accessible through indexing. For example, if you are actually using a memory buffer to store a matrix, you might want to have an array attached to that buffer that will index for you a specific row or column of that matrix. This is why smart arrays, in general, need additional information to describe how the buffer that they are attached to should be indexed.

[1] Note that we are abusing the term *cube*. In mathematics, a cube is a parallelogram whose edges are at right angles and all equal to each other. Here we mean an arbitrary parallelogram whose edges are at right angles, or essentially a *three dimensional array*. We like to use the word because it is cute, short and simple.

Because arrays, matrices and cubes are essentially a memory buffer together with stride information, they follow the same paradigm as memory buffers: they can be *allocated*, *attached* to a parent array, matrix or cube, or set equal to the corresponding *null* array, matrix or cube. Once they have been initialized in one of these ways, they can also be *reallocated* or *reattached*. The API for these operations is described in Section 5.6 [Creating smart arrays], page 38.

Smart array data structures are available for all the C data types mentioned in Section 5.1 [Introduction to smart arrays], page 29. To avoid redundancy, we will use the word `type` to refer to the generic C data type. You should substitute this word with the actual C data type that you are using in all the data structures, variables and API calls.

The smart arrays are encapsulated by the following data types:

struct mu_type_array [Data Type]

Implements an array of `type`s.

`struct mu_membuf` *buf*
: The memory buffer that holds the contents of the array.

`type *`*cache*
: A pointer to the address that holds the contents of the array. This address is maintained in sync with the memory buffer. The sole reason for keeping a separate address like that is to cast the `void` memory pointer, to a `type` pointer that will handle pointer arithmetic correctly.

`size_t` *offset*, *X*, *iX*
: These members describe how to index the array. Let `i` be an index, offset from zero. Then the corresponding element must be accessed from the following memory address:

```
cache + offset + i*iX
```

and the index `i` must satisfy the following inequality:

```
0 <= i < X
```

The integer *iX* is often called the *stride* of the array.

struct mu_type_matrix [Data Type]

Implements a matrix of `type`s.

`struct mu_membuf` *buf*
: The memory buffer that holds the contents of the matrix.

`type *`*cache*
: A pointer to the address that holds the contents of the matrix.

`size_t` *offset*, *X*, *iX*, *Y*, *iY*
: Let `(i,j)` be an index, offset from zero. Then the corresponding element must be accessed from the following memory address:

```
cache + offset + i*iX + j*iY
```

and the index `i` must satisfy the following inequality:

```
0 <= i < X
0 <= j < Y
```

struct mu_type_cube [Data Type]

Implements a cube of `type`s.

`struct mu_membuf` *buf*
: The memory buffer that holds the contents of the matrix.

`type *`*cache*
: A pointer to the address that holds the contents of the matrix.

`size_t` *offset*, *X*, *iX*, *Y*, *iY*
: Let `(i,j,k)` be an index, offset from zero. Then the corresponding element must be accessed from the following memory address:

```
cache + offset + i*iX + j*iY + k*iZ
```

and the index `i` must satisfy the following inequality:

```
0 <= i < X
0 <= j < Y
0 <= k < Z
```

`struct mu_type_array` **mu_type_array_null** [Variable]

Contains the null memory buffer and all the other attributes set equal to zero:

```
buf = mu_membuf_null;
cache = NULL;
X = iX = 0;
```

`struct mu_type_matrix` **mu_type_matrix_null** [Variable]

Contains the null memory buffer and all the other attributes set equal to zero:

```
buf = mu_membuf_null;
cache = NULL;
X = iX = Y = iY = 0;
```

`struct mu_type_cube` **mu_type_cube_null** [Variable]

Contains the null memory buffer and all the other attributes set equal to zero:

```
buf = mu_membuf_null;
cache = NULL;
X = iX = Y = iY = Z = iZ = 0;
```

5.4 Accessing smart arrays

To access the elements of a smart array, you need first to initialize the array (see Section 5.6 [Creating smart arrays], page 38). With a given array, Mathutil provides preprocessor macros for dereferencing the memory address that corresponds to a specific index.

If you define the C preprocessor symbol `MU_CHECK_RANGE`, before including the header file '`mu.h`', then these macros will also error-check you indices for you and abort the program if you attempt to access an invalid index. During development it is very useful to activate this error-checking to debug your program. However during production runs, it's better to deactivate error-checking to allow your program to run more efficiently.

Mathutil provides macros for accessing the same smart array both using indices offset from zero, as well as indices offset from one. Sometimes a mathematical algorithm makes

most sense in terms of offset zero, but there are also many times where it makes most sense in terms of offset one. With Mathutil switching the offset convention is a simple matter of using the appropriate macro to dereference the smart array element.

`type` **A_** (`struct mu_type_array a, size_t i`) [Macro]
Return the *i* element of array. The index *i* is offset from `0`. The overall construct `A_(a,i)` expands to an expression that can be used as an lvalue.

`type` **A1_** (`struct mu_type_array a, size_t i`) [Macro]
Return the *i* element of array. The index *i* is offset from `1`. The overall construct `A1_(a,i)` expands to an expression that can be used as an lvalue.

`type` **M_** (`struct mu_type_matrix a, size_t i, size_t j`) [Macro]
Return the `(i,j)` element of array. The indices *i* and *j* are offset from `0`. The overall construct `M_(a,i,j)` expands to an expression that can be used as an lvalue.

`type` **M1_** (`struct mu_type_matrix a, size_t i, size_t j`) [Macro]
Return the `(i,j)` element of array. The indices *i* and *j* are offset from `1`. The overall construct `M1_(a,i,j)` expands to an expression that can be used as an lvalue.

`type` **C_** (`struct mu_type_cube a, size_t i, size_t j, size_t k`) [Macro]
Return the `(i,j,k)` element of array. The indices *i*, *j* and *k* are offset from `0`. The overall construct `C_(a,i,j,k)` expands to an expression that can be used as an lvalue.

`type` **C1_** (`struct mu_type_matrix a, size_t i, size_t j, size_t k`) [Macro]
Return the `(i,j,k)` element of array. The indices *i*, *j* and *k* are offset from `1`. The overall construct `C1_(a,i,j,k)` expands to an expression that can be used as an lvalue.

Because these macros expand to constructs that can be used as lvalues, they can be used both to access and to modify the contents of an array. For example, if `arr` is an array, `mat` is a matrix and `cub` is a cube, then you can access their contents with statements like the following:

```
a = A_(arr,i);
a = M_(mat,i,j);
a = C_(mat,i,j,k);
```

You can modify them with statements like the following:

```
A_(arr,i) = a;
M_(mat,i,j) = a;
C_(mat,i,j,k) = a;
```

You can infact use them as you would use any dereferenced pointer. So you can use them as part of expressions like:

```
M_(mat1,i,j) = M_(mat2,i,j) + M_(mat3,i,j);
```

Another feature of these macros is that they return an expression that works regardless of the `type` being used. So, you can comfortably use the same macros `A_`, `A1_`, `M`, `M1_`, `C_`, `C1_` for any *type* of smart array.

To help out with creating Guile bindings, Mathutil is also providing the following *functions* for accessing and modifying smart arrays. Note however that using the macros is more convenient and more efficient.

`type` **mu_type_array_get** (`struct mu_type_array a, size_t i`) [Function]
Return the *i*th element of the array *a*. The index *i* is offset from `0`.

`void` **mu_type_array_set** (`struct mu_type_array a, size_t i, type val`) [Function]
Assign the *i*th element of the array *a* equal to *val*. The index *i* is offset from `0`.

`type` **mu_type_matrix_get** (`struct mu_type_matrix a, size_t i, size_t j`) [Function]
Return the `(i,j)` element of the array *a*. The indices are offset from `0`.

`void` **mu_type_matrix_set** (`struct mu_type_matrix a, size_t i, size_t j, type val`) [Function]
Assign the `(i,j)` element of the array *a* equal to *val*. The indices are offset from `0`.

`type` **mu_type_cube_get** (`struct mu_type_matrix a, size_t i, size_t j, size_t k`) [Function]
Return the `(i,j,k)` element of the array *a*. The indices are offset from `0`.

`void` **mu_type_cube_set** (`struct mu_type_matrix a, size_t i, size_t j, size_t k, type val`) [Function]
Assign the `(i,j,k)` element of the array *a* equal to *val*. The indices are offset from `0`.

5.5 Looping over smart arrays

In C, the keyword `for` can be used to implement loops. For example, to loop an integer variable from `0` to `N-1` we write:

```
int i;
for (i = 0; i < N; i++)
  {
    ...
  }
```

This can often get cumbersome and error prone. A very common bug is to confuse `<` with `<=`. There is also a lot of tiring redundancy and typing involved when you want to loop over matrices and cubes.

The Mathutil library defines the following C preprocessor macros to make looping easier. To make these macros available to your program, you need to include the header file '`mu.h`'

loop (*counter, begin, end*) [Macro]
Expands to:

```
for (counter = begin; counter <= end; counter ++)
```

You can use this for most numerical loops where you need to increment an integer counter. For example, here is how to evaluate the sum $S_n = 1 + 2 + \ldots + n$

```
double foo (int a)
{
  int i;
```

```
  double a = 0;
  loop (i,1,n)
    {
      a = a + i;
    }
  return a;
}
```

loop_with_step (*counter, begin, end, step*) [Macro]

Expands to:

```
for (counter = begin; counter <= end; counter += step)
```

You can use this for numerical loops where you need to increment an integer counter with a given step. You can also use them to decrement a counter by passing a negative integer for step.

loop_array (*counter, array*) [Macro]

Expands to:

```
loop (counter, 0, array.X-1)
```

Use this to loop over the contents of any type of *array*. For example, to set all the elements of an array equal to zero:

```
int i;
loop_array (i,a) A_(a,i) = 0;
```

loop_matrix (*i, j, matrix*) [Macro]

Expands to:

```
loop (j, 0, matrix.Y-1) loop (i, 0, matrix.X-1)
```

Use this to loop over the contents of any type of *matrix*. The innermost loops `i` because in general the elements of a matrix are arranged such that this is the most rapidly varying index. For example, to copy all the elemnts of a matrix into another:

```
int i,j;
loop_matrix (i,j,a) M_(a,i,j) = M_(b,i,j);
```

loop_cube (*i, j, k, cube*) [Macro]

Expands to:

```
loop (k, 0, cube.Z-1) loop (j, 0, cube.Y-1)
loop (i, 0, cube.X-1)
```

Use this to loop over the contents of any type of *cube*. The innermost loops `i` because in general the elements of a matrix are arranged such that this is the most rapidly varying index. The next outer loop loops `j`. Finally, the outer loop loops `k`. For example, to add all the elements of two cubes and store the result in another cube:

```
int i,j,k;
loop_cube (i,j,k,a) C_(a,i,j,k) = C_(b,i,j,k) + C_(c,i,j,k);
```

5.6 Creating smart arrays

To create a smart array, the first thing that you need to do is declare the smart array. For example:

```
struct mu_type_matrix a;
```

Then you can either *allocate* the smart array, or attach it to an already allocated memory buffer. Once you do either of these two things, the array is initialized. An initialized array can then be *reallocated* or *reattached* to another memory buffer. If you do not want to initialize the array at the moment, then please set it equal to the corresponding null array (see Section 5.3 [Smart array data types], page 32). This way, Mathutil will be able to detect later that the array is uninitialized and do the right thing if you try to reallocate, reattach or release it.

The Mathutil library provides the following calls for initializing arrays:

`struct mu_type_array` **mu_type_array_alloc** (`size_t x`) [Function]
: Allocate a memory buffer of size `x*sizeof(type)` bytes, and return a contiguous array attached to that buffer.

`struct mu_type_matrix` **mu_type_matrix_alloc** (`size_t x, size_t y`) [Function]
: Allocate a memory buffer of size `x*y*sizeof(type)` bytes, and return a contiguous matrix attached to that buffer.

`struct mu_type_cube` **mu_type_cube_alloc** (`size_t x, size_t y, size_t z`) [Function]
: Allocate a memory buffer of size `x*y*z*sizeof(type)` bytes, and return a contiguous cube attached to that buffer.

`struct mu_type_array` **mu_type_array_alloc_from_buffer** (`struct mu_membuf buf, size_t offset, size_t X, size_t iX`) [Function]
: Return an array that is attached to the memory buffer *buf* with the specified attributes *offset*, *X*, *iX*. The reference counter of *buf* is incremented by one.

`struct mu_type_matrix` **mu_type_matrix_alloc_from_buffer** (`struct mu_membuf buf, size_t offset, size_t X ,size_t iX, size_t Y, size_t iY`) [Function]
: Return a matrix that is attached to the memory buffer *buf* with the specified attributes *offset*, *X*, *iX*, *Y*, *iY*. The reference counter of *buf* is incremented by one.

`struct mu_type_cube` **mu_type_cube_alloc_from_buffer** (`struct mu_membuf buf, size_t offset, size_t X ,size_t iX, size_t Y, size_t iY, size_t Z, size_t iZ`) [Function]
: Return a cube that is attached to the memory buffer *buf* with the specified attributes *offset*, *X*, *iX*, *Y*, *iY*, *Z*, *iZ*. The reference counter of *buf* is incremented by one.

`void` **mu_type_array_realloc** (`struct mu_type_array *a, size_t X`) [Function]
: Reallocate the array *a* to a new contiguous memory buffer of size

```
X*sizeof(type)
```

: The previous memory buffer refered to by *a* is released, if it was not the null array.

`void` **mu_type_matrix_realloc** (`struct mu_type_matrix * a, size_t X, size_t Y`) [Function]

Reallocate the matrix *a* to a new contiguous memory buffer of size

```
X*Y*sizeof(type)
```

The previous memory buffer refered to by *a* is released, if it was not the null array.

`void` **mu_type_cube_realloc** (`struct mu_type_cube * a, size_t X, size_t Y, size_t Z`) [Function]

Reallocate the cube *a* to a new contiguous memory buffer of size

```
X*Y*Z*sizeof(type)
```

The previous memory buffer refered to by *a* is released, if it was not the null array.

`void` **mu_type_array_realloc_from_buffer** (`struct mu_array *a, struct mu_membuf buf, size_t offset, size_t X, size_t iX`) [Function]

Reattach the array *a* to the memory buffer *buf* using the specified attributes *offset*, *X*, *iX*. The previous memory buffer refered to by *a* is released, if it was not the null array. The reference counter of *buf* is incremented.

`void` **mu_type_matrix_realloc_from_buffer** (`struct mu_matrix * a, struct mu_membuf buf, size_t offset, size_t X, size_t iX, size_t Y, size_t iY`) [Function]

Reattach the matrix *a* to the memory buffer *buf* using the specified attributes *offset*, *X*, *iX*, *Y*, *iY*. The previous memory buffer refered to by *a* is released, if it was not the null array. The reference counter of *buf* is incremented.

`void` **mu_type_cube_realloc_from_buffer** (`struct mu_cube * a, struct mu_membuf buf, size_t offset, size_t X, size_t iX, size_t Y, size_t iY, size_t Z, size_t iZ`) [Function]

Reattach the cube *a* to the memory buffer *buf* using the specified attributes *offset*, *X*, *iX*, *Y*, *iY*, *Z*, *iZ*. The previous memory buffer refered to by *a* is released, if it was not the null array. The reference counter of *buf* is incremented.

Warning: Don't use these last six calls on smart arrays that have only been declared. Such smart arrays fields contain garbage, and it is impossible to determine their state. It is okey to use these calls however if you clear the garbage. To clear the garbage, assign the array equal to the corresponding null array (see Section 5.3 [Smart array data types], page 32)

5.7 Releasing smart arrays

The following functions can be used to release a smart array. When you release a smart array, this does not necessarily mean that the memory that it uses will be reclaimed. Mathutil will reclaim the memory *only* when all the arrays that use the same block of memory are freed.

`void` **mu_type_array_free** (`struct mu_type_array a`) [Function]

Release the array *a*. The corresponding reference counter is decremented, and memory is collected if it goes to `0`.

`void` **mu_type_matrix_free** (`struct mu_type_matrix a`) [Function]
Release the matrix *a*. The corresponding reference counter is decremented, and memory is collected if it goes to `0`.

`void` **mu_type_cube_free** (`struct mu_type_cube a`) [Function]
Release the matrix *a*. The corresponding reference counter is decremented, and memory is collected if it goes to `0`.

Mathutil also provides an interface for freeing many smart arrays at the same time. For example:

```
struct mu_double_array a,b,c,d,e;
........
mu_double_array_free_many (5,a,b,c,d,e);
```

can be used to free five arrays in one scoop. It saves a lot of typing.

`void` **mu_type_array_free_many** (`int k, ...`) [Function]
Release the *k* arrays that have been passed as variadic arguments.

`void` **mu_type_matrix_free_many** (`int k, ...`) [Function]
Release the *k* matrices that have been passed as variadic arguments.

`void` **mu_type_cube_free_many** (`int k, ...`) [Function]
Release the *k* cubes that have been passed as variadic arguments.

5.8 Smart array aliases

An smart array *alias* is a smart array that behaves exactly like the smart array that it is aliased to. In order for a smart array to be the alias of another, they must share the same memory buffer and have equal attributes.

One way to create an alias is by copying one initialized smart array into another by a direct assignment. The result of such an assignment will be that both smart arrays will share the same memory buffer and attributes. Note that no copying of elements actually takes place! This makes it very efficient to pass smart arrays as arguments to functions or return them from functions.

Making aliases through assignments however is very dangerous because they defeat reference counting. Suppose that you are given an smart array and an alias to that smart array that was created through direct assignment. If you release the alias, and as a result its memory buffer is actually destroyed, then the original array will be left in an inconsistent state.

It is okey to use assignment to pass an smart array as an argument into a function in order to access and modify the contents of that smart array in that function. It is also okey to use assignment to return a smart array from a function whose job is to initialize that smart array in a certain fashion before returning it. There are many functions in the Mathutil library itself that do these things. It is quite reasonable that you may want to write functions similar to those.

However, for almost every other purpose you should not create aliases without dealing with reference counting properly. This is particularly true if you want to store a reference to

a given smart array in another structure. That reference should be an alias that is properly reference counted so that when that structure attempts to free itself, the smart array alias does not inadvertently destroy the memory buffer while it is still being used by other smart arrays.

Mathutil provides the following two routines for creating aliases that are reference counted. Note that these routines are essentially a special case of the routines that we presented in Section 5.6 [Creating smart arrays], page 38.

`struct mu_type_array` **mu_type_array_alias** (`struct mu_type_array a`) [Function]
: Increment the reference counter of *a* and return an alias to *a*. Use this function to initialize a newly-declared array.

`void` **mu_type_array_realias** (`struct mu_type_array *a, struct mu_type_array b`) [Function]
: Increment the reference counter of *b* and make *a* an alias to *b*. Use this function if *a* has been initialized before *or* if *a* has been assigned equal to `mu_type_array_null`.

`struct mu_type_matrix` **mu_type_matrix_alias** (`struct mu_type_matrix a`) [Function]
: Increment the reference counter of *a* and return an alias to *a*. Use this function to initialize a newly-declared matrix.

`void` **mu_type_matrix_realias** (`struct mu_type_matrix *a, struct mu_type_matrix b`) [Function]
: Increment the reference counter of *b* and make *a* an alias to *b*. Use this function if *a* has been initialized before *or* if *a* has been assigned equal to `mu_type_matrix_null`.

`struct mu_type_array` **mu_type_cube_alias** (`struct mu_type_array a`) [Function]
: Increment the reference counter of *a* and return an alias to *a*. Use this function to initialize a newly-declared cube.

`void` **mu_type_cube_realias** (`struct mu_type_cube *a, struct mu_type_cube b`) [Function]
: Increment the reference counter of *b* and make *a* an alias to *b*. Use this function if *a* has been initialized before *or* if *a* has been assigned equal to `mu_type_cube_null`.

When you release an alias, the memory is not reclaimed if the parent is not released as well.

5.9 Memory information about smart arrays

Sometimes, when you want to create a C wrapper for a Fortran subroutine you want to know the exact memory address that corresponds to a given array.

`type *` **mu_type_array_address** (`struct mu_type_array a`) [Function]
`type *` **mu_type_matrix_address** (`struct mu_type_matrix a`) [Function]
`type *` **mu_type_cube_address** (`struct mu_type_cube a`) [Function]
: Returns the address `a.cache + a.offset`.

If the Fortran subroutine supports arrays with stride then you can get the stride for array `a` and pass it on as `a.iX`. Unfortunately, many Fortran subroutines require the array to be *contiguous*. An array is *contiguous* if all of its elements lie in one compact block of memory. The standard way to handle this situation is to pass the array *if* it is contiguous. If it is not conitguous, then copy its elements into a contiguous array that is attached to some workspace and pass that instead. If the array contents are expected to be modified, then we may need to copy them back to the original array, if we didn't pass on a reference to the original array itself.

The following routines will tell you whether an array is contiguous or not.

`int` **mu_type_array_contiguous_p** (`struct mu_type_array a`) [Function]
Return 1 if the array is contiguous. Return 0 otherwise. An array is contiguous if and only if `a.iX == 1`.

`int` **mu_type_matrix_contiguous_p** (`struct mu_type_array a`) [Function]
Return 1 if the matrix is contiguous. Return 0 otherwise. A matrix is contiguous if and only if

```
a.iX == 1 and a.iY == X
```

`int` **mu_type_matrix_transpose_contiguous_p** (`struct mu_type a`) [Function]
Return 1 if the transpose of this matrix is contiguous. Return 0 otherwise. This predicate is true if and only if:

```
a.iX == Y and a.iY == 1
```

`int` **mu_type_cube_contiguous_p** (`struct mu_type_array a`) [Function]
Return 1 if the cube is contiguous. Return 0 otherwise. A cube is contiguous is and only if

```
a.iX == 1 and a.iY == X and a.iZ == X*Y
```

5.10 Subsidiary smart arrays

A *subsidiary smart array* is an array that is attached to the same memory buffer as the original array that indexes only a *concrete* portion of the original array. In the case of one-dimensional arrays, a concrete portion is any block refered to by consecutive indices. In the case of two-dimensional arrays, a concrete portion is what mathematicians call a *block submatrix*. In general, a subsidiary smart array has the same stride as its parent, but a different offset and different size.

Mathutil offers the following routines for creating subsidiary arrays. Do not apply any of these routines on a newly-declared smart array that hasn't yet been initialized. Instead, initialize such smart arrays to point to the null smart array first. You can safely apply them however to smart arrays that have already been initialized in some other fashion.

`void` **mu_type_array_sub** (`struct mu_type_array * b, struct mu_type_array a, size_t i1, size_t i2`) [Function]
Define b to be an array whose indexing covers the elements `A_(a,i1)` to `A_(a,i2)`. The attributes of b are defined as follows:

```
offset = a.offset + i1*a.iX;
X = i2 - i1 + 1;
iX = a.iX;
```

`void` **mu_type_matrix_sub** (`struct mu_type_matrix * b, struct mu_type_matrix a, size_t i1, size_t i2, size_t j1, size_t j2`) [Function]

Define *b* to be a matrix whose indexing covers the submatrix whose upper left corner element is `M_(a,i1,j1)` and whose lower right corner element is `M_(a,i2,j2)`. The attributes of *b* are defined as follows:

```
offset = a.offset + i1*a.iX + j1*a.iY;
X = i2 - i1 + 1;
Y = j2 - j1 + 1;
iX = a.iX;
iY = a.iY;
```

`void` **mu_type_cube_sub** (`struct mu_type_cube * b, struct mu_type_cube a, size_t i1, size_t i2, size_t j1. size_t j2, size_t k1, size_t k2`) [Function]

Define *b* to be a cube whose indexing covers all the elements `C_(a,i,j,k)` such that

```
i1 <= i <= i2
j1 <= j <= j2
k1 <= k <= k2
```

The attributes of *b* are defined as follows:

```
offset = a.offset + i1*a.iX + j1*a.iY + k1*a.iZ;
X = i2 - i1 + 1;
Y = j2 - j1 + 1;
Z = k2 - k1 + 1;
iX = a.iX; iY = a.iY; iZ = a.iZ;
```

5.11 Interlaced smart arrays

When you want to create an array of complex numbers, it is often useful to interlace the real and imaginary parts in memory. For example, suppose that you want to create an array of `N` complex numbers. A common way of doing this is by allocating an array `a` of `2*N doubles` such that:

```
A_(a,0) + i*A_(a,1)   -> 1st complex number
A_(a,2) + i*A_(a,3)   -> 2nd complex number
....
A_(a,2*N-2) + i*A_(a,2*N-1) -> last complex number
```

While this is a convenient (and, with many packages, standard) way to represent complex arrays, you often want to think of the real and imaginary parts as two separate arrays. In other words, you would like to have two arrays `a_real` and `a_imag` attached to the array `a` such that:

```
A_(a_real, k) ≡ A_(a, 2*k)
A_(a_imag, k) ≡ A_(a, 2*k+1)
```

Mathutil provides the following functions for creating such arrays. Again, these functions assume that your outgoing arrays have already been initialized, either by assigning them to the null array, or by some other way.

`void` **mu_type_array_interlaced** (`struct mu_type_array a, struct mu_type_array *a1, struct mu_type_array *a2`) [Function]

Attach the arrays *a1* and *a2* to the memory buffer of *a* and set up their attributes such that

```
A_(a1, k) ≡ A_(a, 2*k)
A_(a2, k) ≡ A_(a, 2*k+1)
```

`void` **mu_type_matrix_interlaced** (`struct mu_type_matrix a, struct mu_type_matrix *a1, struct mu_type_matrix *a2`) [Function]

Attach the arrays *a1* and *a2* to the memory buffer of *a* and set up their attributes such that

```
M_(a1, i, j) ≡ M_(a, 2*i  , j)
M_(a2, i, j) ≡ M_(a, 2*i + 1, j)
```

`void` **mu_type_cube_interlaced** (`struct mu_type_cube a, struct mu_type_cube *a1, struct mu_type_cube *a2`) [Function]

Attach the arrays *a1* and *a2* to the memory buffer *a* and set up their attributes such that

```
C_(a1, i, j, k) ≡ C_(a, 2*i, j, k)
C_(a2, i, j, k) ≡ C_(a, 2*i + 1, j, k)
```

5.12 Smart array slicing

By *smart array slicing* we mean obtaining a lower-dimensional smart array that is attached to a higher-dimensional array. For example, if you have a matrix, then an array that indexes a row or column of that matrix is an example of a smart array slice.

The main difficulty with dealing with array slices is specifying which slice we want. Mathutil defines the following `enum` type to identify the *orientation* of the desired slices.

mu_slice [Data Type]

A variable of this type can have one of the following values:

```
X_slice,  Y_slice,  Z_slice,
XY_slice, XZ_slice, YX_slice,
YZ_slice, ZX_slice, ZY_slice
```

The meaning of these values depends on context. On certain contexts, some of these values have no meaning.

Mathutil provides the following routines for obtaining array slices:

`void` **mu_type_matrix_to_array** (`struct mu_type_array *b, struct mu_type_matrix a, mu_slice slice, int k`) [Function]

Return *b* such that it is the appropriate row or column of the matrix *a*.

If `slice == X_slice`
make `A_(b,i)` ≡ `M_(a,i,k)`

If `slice == Y_slice`
make `A_(b,j)` ≡ `M_(a,k,j)`

In other words, `X_slice` returns the *row* of the matrix *a* and `Y_slice` returns the *column* of the matrix *a*.

`void` **mu_type_cube_to_array** (`struct mu_type_array *b, struct mu_type_cube a, mu_slice slice, int k1, int k2`) [Function]
Return *b* such that it is the appropriate one dimensional slice of the cube *a*.

If `slice == X_slice`
make `A_(b,i)` ≡ `C_(a,i,k1,k2)`

If `slice == Y_slice`
make `A_(b,j)` ≡ `C_(a,k1,j,k2)`

If `slice == Z_slice`
make `A_(b,k)` ≡ `C_(a,k1,k2,k)`

`void` **mu_type_cube_to_matrix** (`struct mu_type_matrix *b, struct mu_type_cube a, mu_slice slice, int n`) [Function]
Return *b* such that it is the appropriate planar (two-dimensional) slice of the cube *a*.

If `slice == XY_slice`
make `M_(b,i,j)` ≡ `C_(a,i,j,n)`

If `slice == XZ_slice`
make `M_(b,i,k)` ≡ `C_(a,i,n,k)`

If `slice == YX_slice`
make `M_(b,j,i)` ≡ `C_(a,i,j,n)`

If `slice == YZ_slice`
make `M_(b,j,k)` ≡ `C_(a,n,j,k)`

If `slice == ZX_slice`
make `M_(b,k,i)` ≡ `C_(a,i,n,k)`

If `slice == ZY_slice`
make `M_(b,k,j)` ≡ `C_(a,n,j,k)`

5.13 Smart array input and output

5.14 Interfacing to Fortran subroutines

A lot of bla bla bla that I should write only when I have a concrete example.

Index

A

C

L

M

S