THE GENERIC MAPPING TOOLS

GMT API Documentation

Release 5.1.1

P. Wessel, W. H. F. Smith,
R. Scharroo, J. Luis, and F. Wobbe

March 01, 2014

1 Preamble 3

2 The i/o abstraction layer 5

3 Our audience 7

4 Definitions 9

5 Recognized resources 11
5.1 Data tables 11
5.2 Text tables 11
5.3 GMT grids 12
5.4 CPT palette tables 12
5.5 GMT images 12
5.6 User data columns (GMT vectors) 12
5.7 User data matrices (GMT matrices) 13

6 Overview of the GMT C Application Program Interface 15

7 The GMT C Application Program Interface 19
7.1 Initialize a new GMT session 19
7.2 Register input or output resources 19
7.3 Create empty resources 22
7.4 Duplicate resources 23
7.5 Get resource ID 24
7.6 Import Data 24
7.7 Manipulate data 28
7.8 Message and Verbose Reporting 30
7.9 Presenting and accessing GMT options 31
7.10 Prepare module options 32
7.11 Calling a GMT module 36
7.12 Adjusting headers and comments 36
7.13 Exporting Data 37
7.14 Destroy allocated resources 40
7.15 Terminate a GMT session 40

8 The GMT FFT Interface 41
8.1 Presenting and Parsing the FFT options 41

8.2 Initializing the FFT machinery 41
8.3 Taking the FFT 42
8.4 Taking the 1-D FFT 42
8.5 Taking the 2-D FFT 42
8.6 Wavenumber calculations 43
8.7 Destroying the FFT machinery 43

9 FORTRAN interfaces **45**
9.1 FORTRAN 77 Grid i/o 45

Index **47**

The Generic Mapping Tools

C/C++ Application Programming Interface

Pål (Paul) Wessel

SOEST, University of Hawai'i at Manoa

Walter H. F. Smith

Laboratory for Satellite Altimetry, NOAA/NESDIS

Remko Scharroo

EUMETSAT, Darmstadt, Germany

Joaquim F. Luis

Universidade do Algarve, Faro, Portugal

Florian Wobbe

Alfred Wegener Institute, Germany

CHAPTER 1

Preamble

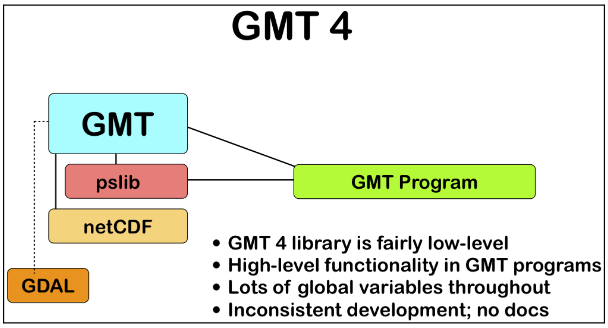

Figure 1.1: GMT 4 programs contain all the high-level functionality.

Prior to version 5, the bulk of GMT functionality was coded directly in the standard GMT C program modules (e.g., `surface.c`, `psxy.c`, `grdimage.c`, etc.). The GMT library only offered access to low-level functions from which those high-level GMT programs were built. The standard GMT programs have been very successful, with tens of thousands of users world-wide. However, the design of the main programs prevented developers from leveraging GMT functionality from within other programming environments since access to GMT tools could only be achieved via system calls [1]. Consequently, all data i/o had to be done via temporary files. The design also prevented the GMT developers themselves from taking advantage of these modules directly. For instance, the tool `pslegend` needed to make extensive use of system calls to `psxy` and `pstext` in order to plot the lines, symbols and text that make up a map legend, making it a very awkward program to maintain.

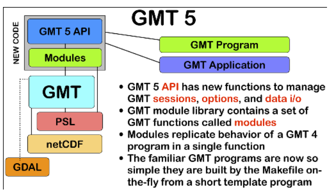

Figure 1.2: GMT 5 programs contain all the high-level functionality.

Starting with GMT version 5, all standard GMT programs have been split into a short driver program (the "new" GMT programs) and a function "module". The drivers simply call the corresponding GMT modules; it is these modules that do all the work. These new functions have been placed in a new GMT high-level API library and can be called from a variety of environments (C/C++, Fortran, Python, Matlab, Visual Basic, Julia, R, etc.) [2]. For example, the main program `blockmean.c` has been reconfigured

[1] or via a very confusing and ever-changing myriad of low-level library functions for bold programmers.

[2] Currently, only C/C++ and Matlab are being tested.

as a high-level function `GMT_blockmean()`, which does the actual spatial averaging and can pass the result back to the calling program (or write it to file). The previous behavior of `blockmean.c` is replicated by a short driver program that simply collects user arguments and then calls `GMT_blockmean()`. Indeed, the driver programs for all the standard GMT programs are identical so that the makefile generates them on-the-fly when it compiles and links them with the GMT library into executables. Thus, `blockmean.c` and others do in fact no longer exist.

CHAPTER 2

The i/o abstraction layer

In order for this interface to be as flexible as possible we have generalized the notion of input and output. Data that already reside in an application's memory may serve as input to a GMT function. Other sources of input may be file pointers and file descriptors (as well as the already-supported mechanism for passing file names). For standard data table i/o, the GMT API takes care of the task of assembling any combination of files, pointers, and memory locations into *a single virtual data set* from which the GMT function may read (a) all records at once into memory, or (b) read one record at a time. Likewise, GMT functions may write their output to a virtual destination, which might be a memory location in the user's application, a file pointer or descriptor, or an output file. The GMT functions are unaware of these details and simply read from a "source" and write to a "destination".

CHAPTER 3

Our audience

Here, we document the new functions in the GMT API library for application developers who wish to call these functions from their own custom programs. At this point, only the new high-level GMT API is fully documented and intended for public use. The structure and documentation of the under-lying lower-level GMT library is not finalized. Developers using these functions may risk disruption to their programs due to changes we may make in the library in support of the GMT API. However, developers who wish to make supplemental packages to be distributed as part of GMT will (other than talk to us) probably want to access the entire low-level GMT library as well. It is unlikely that the low-level library will ever be fully documented.

CHAPTER 4

Definitions

For the purpose of this documentation a few definitions are needed:

1. "Standard GMT program" refers to one of the traditional stand-alone command-line executables known to all GMT users, e.g., `blockmean`, `psxy`, `grdimage`, etc. Prior to version 5, these were the only GMT executables available.
2. "GMT module" refers to the function in the GMT API library that is responsible for all the action taken by the corresponding GMT program. All such modules are given the same name as the corresponding program but carry the prefix `GMT_`, e.g., `GMT_blockmean`.
3. "GMT application" refers to a new application written by any developer and may call one or more GMT functions to create a new GMT-compatible executable.
4. In the API description that follows we will use the type `int` to mean a 4-byte integer. All integers used in the API are 4-byte integers with the exception of one function where a longer integer is used. Since different operating systems have their own way of defining 8-byte integers we use C99's `int64_t` for this purpose; it is guaranteed to yield the correct type that the GMT function expect.

In version 5, the standard GMT programs are themselves specific but overly simple examples of GMT applications that only call the single GMT function they are associated with. However, some programs such as `pslegend`, `gmtconvert`, `grdblend`, `grdfilter` and others call several modules.

CHAPTER 5

Recognized resources

The GMT API knows how to read and write five types of data common to GMT operations: CPT palette tables, data tables (ASCII or binary), text tables, GMT grids and images (reading only). In addition, we present two data types to facilitate the passing of simple user arrays (one or more equal-length data columns of any data type, e.g., double, char) and 2-D or 3-D user matrices (of any data type and column/row organization [1]). We refer to these data types as GMT *resources*. There are many attributes for each of these resources and therefore we use a top-level structure for each type to keep them all in one container. These containers are given or returned by the GMT API functions using opaque pointers (`void *`). Below we discuss these containers in some detail; we will later present how they are used when importing or exporting them to or from files, memory locations, or streams. The first five are the standard GMT objects, while the latter two are the special user data containers to facilitate converting user data into GMT resources. These resources are defined in the include file `gmt_resources.h`; please consult this file to ensure correctness as it is difficult to keep the documentation up-to-date.

5.1 Data tables

Much data processed in GMT come in the form of ASCII, netCDF, or native binary data tables. These may have any number of header records (ASCII files only) and perhaps segment headers. GMT programs will read one or more such tables when importing data. However, to avoid memory duplication or limitations some programs may prefer to read records one at the time. The GMT API has functions that let you read record-by-record by presenting a virtual data set that combines all the data tables specified as input. This simplifies record processing considerably. A `struct GMT_DATASET` may contain any number of tables, each with any number of segments, each segment with any number of records, and each record with any number of columns. Thus, the arguments to GMT API functions that handle such data sets expect this type of variable. All segments are expected to have the same number of columns.

5.2 Text tables

Some data needed by GMT are simply free-form ASCII text tables. These are handled similarly to data tables. E.g., they may have any number of header records and even segment headers, and GMT programs can read one or more tables or get text records one at the time. A `struct GMT_TEXTSET` may contain any number of tables, each with any number of segments, and each segment with any number of records. Thus, the arguments to GMT API functions that handle such data sets expect this type of variable. The user's program may then parse and process such text records as required. This resources is particularly

[1] At the moment, GMT does not have native support for 3-D grids.

useful when your data consist of a mix or data coordinates and ordinary text since regular data tables will be parsed for floating-point columns only.

5.3 GMT grids

GMT grids are used to represent equidistant and organized 2-D surfaces. These can be plotted as contour maps, color images, or as perspective surfaces. Because the native GMT grid is simply a 1-D float array with all the metadata kept in a separate header, we pass this information via a `struct GMT_GRID`, which is a container that holds both items. Thus, the arguments to GMT API functions that handle such GMT grids expect this type of variable.

5.4 CPT palette tables

The color palette table files, or just CPT tables, contain colors and patterns used for plotting data such as surfaces (i.e., GMT grids) or symbols, lines and polygons (i.e., GMT tables). GMT programs will generally read in a CPT palette table, make it the current palette, do the plotting, and destroy the table when done. The information is referred to via a pointer to `struct GMT_PALETTE`. Thus, the arguments to GMT API functions that handle palettes expect this type of variable. It is not expected that users will wish to manipulate a CPT table directly, but rather use this mechanism to hold them in memory and pass as arguments to GMT modules.

5.5 GMT images

GMT images are used to represent bit-mapped images typically obtained via the GDAL bridge. These can be reprojected internally, such as when used in grdimage. Since images and grids share the concept of a header, we use the same header structure for grids as for images; however, some additional metadata attributes are also needed. Finally, the image itself may be of any data type and have more than one band (channel). Both image and header information are passed via a `struct GMT_IMAGE`, which is a container that holds both items. Thus, the arguments to GMT API functions that handle GMT images expect this type of variable. Unlike the other objects, writting images has only partial support via `GMT_grdimage` [2].

```
struct GMT_IMAGE {
    enum GMT_enum_type type;          /* Data type, e.g. GMT_FLOAT */
    int                *ColorMap;     /* Array with color lookup values */
    struct GMT_GRID_HEADER *header;   /* Pointer to full GMT header for the image */
    unsigned char   *data;            /* Pointer to actual image */
    /* ---- Variables "hidden" from the API ---- */
    unsigned int id;                  /* The internal number of the data set */
    enum GMT_enum_alloc alloc_mode;   /* Allocation info [0] */
    unsigned int alloc_level;         /* Level of initial allocation */
    const char      *ColorInterp;
};
```

5.6 User data columns (GMT vectors)

Programs that wish to call GMT modules may hold data in their own particular data structures. For instance, the user's program may have three column arrays of type float and wishes to use these as

[2] This may change in later releases.

the input source to the `GMT_surface` module, which normally expects double precision triplets via a `struct GMT_DATASET` read from a file or given by memory reference. Simply create a new `struct GMT_VECTOR` (see section [sec:create]) and assign the union array pointers (see *univector*) to your data columns and provide the required information on length, data types, and optionally range (see Table *vector*). By letting the GMT module know you are passing a data set *via* a `struct GMT_VECTOR` it will know how to read the data correctly.

```
union GMT_UNIVECTOR {
    uint8_t   *uc1;        /* Pointer for unsigned 1-byte array */
    uint8_t   *uc1;        /* Pointer for unsigned 1-byte array */
    int8_t    *sc1;        /* Pointer for signed 1-byte array */
    uint16_t  *ui2;        /* Pointer for unsigned 2-byte array */
    int16_t   *si2;        /* Pointer for signed 2-byte array */
    uint32_t  *ui4;        /* Pointer for unsigned 4-byte array */
    int32_t   *si4;        /* Pointer for signed 4-byte array */
    uint64_t  *ui8;        /* Pointer for unsigned 8-byte array */
    int64_t   *si8;        /* Pointer for signed 8-byte array */
    float     *f4;         /* Pointer for float array */
    double    *f8;         /* Pointer for double array */
};
```

Table 1.1: Definition of the GMT_UNIVECTOR union that holds a pointer to any array type.

```
struct GMT_VECTOR {
    uint64_t            n_columns;     /* Number of vectors */
    uint64_t            n_rows;        /* Number of rows in each vector */
    enum GMT_enum_type  *type;         /* Array with data type for each vector */
    double              range[2];      /* The min and max limits on t-range (or 0,0) */
    union GMT_UNIVECTOR *data;         /* Array with unions for each column */
    unsigned int        id;            /* An identification number */
    enum GMT_enum_alloc  alloc_mode;   /* Determines if we may free the vectors or not */
    unsigned int alloc_level;          /* Level of initial allocation */
};
```

5.7 User data matrices (GMT matrices)

```
struct GMT_MATRIX {
    uint64_t n_rows;                   /* Number of rows in the matrix */
    uint64_t n_columns;                /* Number of columns in the matrix */
    unsigned int n_layers;             /* Number of layers in a 3-D matrix */
    unsigned int shape;                /* 0 = C (rows) and 1 = Fortran (cols) */
    unsigned int registration;         /* 0 for gridline and 1 for pixel registration */
    size_t dim;                        /* Length of dimension for row (C) or column (Fortran) */
    size_t size;                       /* Byte length of data */
    enum GMT_enum_alloc alloc_mode;    /* Determines if we may free the vectors or not */
    double range[6];                   /* The min and max limits on x-, y-, and z-ranges */
    union GMT_UNIVECTOR data;          /* Union with pointers a data matrix of any type */
    /* ---- Variables "hidden" from the API ---- */
    unsigned int id;                   /* An identification number */
    unsigned int alloc_level;          /* Level of initial allocation */
    enum GMT_enum_type type;           /* The matrix data type */
};
```

Likewise, programs may have an integer 2-D matrix in memory and wish to use that as the input grid to the `GMT_grdfilter` module, which normally expects a `struct GMT_GRID` with floating point data via a file or provided by memory reference. As for user vectors, we create a `struct GMT_MATRIX` (see section [sec:create]), assign the appropriate union pointer to your data matrix and provide information on dimensions and data type (see Table *matrix*). Let the GMT module know you are passing a grid via a `struct GMT_MATRIX` and it will know how to read the matrix properly.

The `enum` types referenced in Table *vector* and Table *matrix* and summarized in Table *enums* and Table *types*.

constant	description
GMT_ALLOCATED_EXTERNALLY	Item was *not* allocated by GMT so do not reallocate or free
GMT_ALLOCATED_BY_GMT	GMT allocated the memory; reallocate and free as needed

constant	description
GMT_CHAR	int8_t, 1-byte signed integer type
GMT_UCHAR	int8_t, 1-byte unsigned integer type
GMT_SHORT	int16_t, 2-byte signed integer type
GMT_USHORT	uint16_t, 2-byte unsigned integer type
GMT_INT	int32_t, 4-byte signed integer type
GMT_UINT	uint32_t, 4-byte unsigned integer type
GMT_LONG	int64_t, 8-byte signed integer type
GMT_ULONG	uint64_t, 8-byte unsigned integer type
GMT_FLOAT	4-byte data float type
GMT_DOUBLE	8-byte data float type

CHAPTER 6

Overview of the GMT C Application Program Interface

Users who wish to create their own GMT application based on the API must make sure their program goes through the steps below; details for each step will be revealed in the following chapter. We have kept the API simple: In addition to the GMT modules, there are only 52 public functions to become familiar with, but most applications will only use a small subset of this selection. Functions either return an integer error code (when things go wrong; otherwise it is set to GMT_OK (0)), or they return a void pointer to a GMT resources (or NULL if things go wrong). In either case the API will report what the error is. The layout here assumes you wish to use data in memory as input sources; if the data are simply command-line files then things simplify considerably.

1. Initialize a new GMT session with `GMT_Create_Session`, which allocates a hidden GMT API control structure and returns an opaque pointer to it. This pointer is the first argument to all subsequent GMT API function calls within the session.
2. For each intended call to a GMT module, several steps are involved:
    (a) Register input sources and output destination with `GMT_Register_IO`.
    (b) Each resource registration generates a unique ID number. For memory resources, we embed these numbers in unique filenames of the form "@GMTAPI@-######". When GMT i/o library functions encounter such filenames they extract the ID and make a connection to the corresponding resource. Multiple table data or text sources are combined into a single virtual source for GMT modules to operate on. In contrast, CPT, Grid, and Image resources are operated on individually.
    (c) Enable data import once all registrations are complete.
    (d) Read data into memory. You may choose to read everything at once or read record-by-record (tables only).
    (e) Prepare required arguments and call the GMT module you wish to use.
    (f) Process any results returned to memory via pointers rather than written to files.
    (g) Destroy the resources allocated by GMT modules to hold results, or let the garbage collector do this automatically at the end of the module and at the end of the session.
3. Repeat steps a–f as many times as your application requires.
4. We terminate the GMT session by calling `GMT_Destroy_Session`.

The steps a–d collapse into a single step if data are simply read from files.

Advanced programs may be calling more than one GMT session and thus run several sessions, perhaps concurrently as different threads on multi-core machines. We will now discuss these steps in more detail.

Throughout, we will introduce upper-case GMT C enum constants *in lieu* of simple integer constants. These are considered part of the API and are available for developers via the `gmt_resources.h` include file.

The C/C++ API is deliberately kept small to make it easy to use. Table [tbl:API] gives a list of all the functions and their purpose.

constant	description
GMT_Append_Option	Append new option structure to linked list
GMT_Begin_IO	Enable record-by-record i/o
GMT_Call_Module	Call any of the GMT modules
GMT_Create_Args	Convert linked list of options to text array
GMT_Create_Cmd	Convert linked list of options to command line
GMT_Create_Data	Create an empty data resource
GMT_Create_Options	Convert command line options to linked list
GMT_Create_Session	Initialize a new GMT session
GMT_Delete_Option	Delete an option structure from the linked list
GMT_Destroy_Args	Delete text array of arguments
GMT_Destroy_Cmd	Delete text command of arguments
GMT_Destroy_Data	Delete a data resource
GMT_Destroy_Options	Delete the linked list of option structures
GMT_Destroy_Session	Terminate a GMT session
GMT_Duplicate_Data	Make an identical copy of a data resources
GMT_Encode_ID	Encode a resources ID as a special filename
GMT_End_IO	Disable further record-by-record i/o
GMT_FFT	Take the Fast Fourier Transform of data object
GMT_FFT_1D	Take the Fast Fourier Transform of 1-D float data
GMT_FFT_2D	Take the Fast Fourier Transform of 2-D float data
GMT_FFT_Create	Initialize the FFT machinery
GMT_FFT_Destroy	Terminate the FFT machinery
GMT_FFT_Option	Explain the FFT options and modifiers
GMT_FFT_Parse	Parse argument with FFT options and modifiers
GMT_FFT_Wavenumber	Return wavenumber given data index
GMT_Find_Option	Find an option in the linked list
GMT_Get_Common	Determine if a GMT common option was set
GMT_Get_Coord	Create a coordinate array
GMT_Get_Data	Import a registered data resources
GMT_Get_Default	Obtain as string one of the GMT default settings
GMT_Get_ID	Obtain the ID of a given resource
GMT_Get_Index	Convert row, col into a grid or image index
GMT_Get_Record	Import a single data record
GMT_Get_Row	Import a single grid row
GMT_Get_Value	Convert string into coordinates or dimensions
GMT_Init_IO	Initialize i/o given registered resources
GMT_Make_Option	Create an option structure
GMT_Message	Issue a message, optionally with time stamp
GMT_Option	Explain one or more GMT common options
GMT_Parse_Common	Parse the GMT common options
GMT_Put_Data	Export to a registered data resource given by ID
GMT_Put_Record	Export a data record
GMT_Put_Row	Export a grid row

Continued on next page

Table 6.1 – continued from previous page

constant	description
GMT_Read_Data	Import a data resource or file
GMT_Register_IO	Register a resources for i/o
GMT_Report	Issue a message contingent upon verbosity level
GMT_Retrieve_Data	Obtained link to data in memory via ID
GMT_Set_Comment	Assign a comment to a data resource
GMT_Status_IO	Check status of record-by-record i/o
GMT_Update_Option	Modify an option structure
GMT_Write_Data	Export a data resource

The GMT C Application Program Interface

7.1 Initialize a new GMT session

Most applications will need to initialize only a single GMT session. This is true of all the standard GMT programs since they only call one GMT module and then exit. Most user-developed GMT applications are likely to only initialize one session even though they may call many GMT modules. However, the GMT API supports any number of simultaneous sessions should the programmer wish to take advantage of it. This might be useful when you have access to several CPUs and want to spread the computing load [1]. In the following discussion we will simplify our treatment to the use of a single session only.

To initiate the new session we use

```
void *GMT_Create_Session (char *tag, unsigned int pad, unsigned int mode,
                          int (*print_func) (FILE *, const char *));
```

and you will typically call it thus:

```
void *API = NULL;
API = GMT_Create_Session ("Session name", 2, 0, NULL);
```

where `API` is an opaque pointer to the hidden GMT API control structure. You will need to pass this pointer to *all* subsequent GMT API functions; this is how essential internal information is passed from module to module. The key task of this initialization is to set up the GMT machinery and its internal variables used for map projections, plotting, i/o, etc. The initialization also allocates space for internal structures used to register resources. The `pad` argument sets how many rows and columns should be used for padding for grids and images so that boundary conditions can be applied. GMT uses 2 so we recommend that value. The `mode` argument is only used for external APIs that need to replace GMT's calls to a hard exit upon failure with a soft return. Likewise, the *print_func* argument is a pointer to a function that is used to print messages via GMT_Message or GMT_Report from APIs that cannot use the standard printf (this is the case for the Matlab API, for instance). All other uses should simply pass 0 and NULL for these two arguments. Should something go wrong then `API` will be returned as `NULL`.

7.2 Register input or output resources

When using the standard GMT programs, you specify input files on the command line or via special program options (e.g., **-I***intensity.nc*). The output of the programs are either written to standard output (which you redirect to files or pipe to other programs) or to files specified by specific program options (e.g., **-G***output.nc*). Alternatively, the GMT API allows you to specify input (and output) to be associated

[1] However, there is no thread-support yet.

with open file handles or program variables. We will examine this more closely below. Registering a resource is a required step before attempting to import or export data that *do not* come from files or standard input/output.

7.2.1 Resource registration

Registration involves a direct or indirect call to

```
int GMT_Register_IO (void *API, unsigned int family, unsigned int method,
                     unsigned int geometry, unsigned int direction,
                     double wesn[], void *ptr);
```

where *family* specifies what kind of resource is to be registered, `method` specifies how we to access this resource (see Table *methods* for recognized methods, as well as modifiers you can add; these are listed in Table *via*), *geometry* specifies the geometry of the data (see Table *geometry* for recognized geometries), `ptr` is the address of the pointer to the named resource. If `direction` is GMT_OUT and the `method` is not related to a file (filename, stream, or handle), then `ptr` must be NULL. After the GMT module has written the data you can use GMT_Retrieve_Data to assign a pointer to the memory location (variable) where the output was allocated. For grid (and image) resources you may request to obtain a subset via the *wesn* array (see Table *wesn* for information); otherwise, pass NULL (or an array with at least 4 items all set to 0) to obtain the entire grid (or image). The `direction` indicates input or output and is either GMT_IN or GMT_OUT. Finally, the function returns a unique resource ID, or GMT_NOTSET if there was an error.

7.2.2 Object ID encoding

To use registered resources as program input or output arguments you must pass them via a text string that acts as a special file name (Chapter [ch:overview]). The proper filename formatting is guaranteed by using the function

```
int GMT_Encode_ID (void *API, char *filename, int ID);
```

which accepts the unique `ID` and writes the corresponding `filename`. The variable `filename` must have enough space to hold 16 bytes. The function returns 1 if there is an error; otherwise it returns 0.

family	source points to
GMT_IS_DATASET	A [multi-segment] table file
GMT_IS_TEXTSET	A [multi-segment] text file
GMT_IS_GRID	A GMT grid file
GMT_IS_CPT	A CPT file
GMT_IS_IMAGE	A GMT image

method	how to read/write data
GMT_IS_FILE	Pointer to name of a file
GMT_IS_STREAM	Pointer to open stream (or process)
GMT_IS_FDESC	Pointer to integer file descriptor
GMT_IS_DUPLICATE	Pointer to memory we may *duplicate* data from
GMT_IS_REFERENCE	Pointer to memory we may *reference* data from

approach	how method is modified
GMT_VIA_VECTOR	User's data columns are accessed via a GMT_VECTOR structure
GMT_VIA_MATRIX	User's matrix is accessed via a GMT_MATRIX structure

geometry	description
GMT_IS_TEXT	Not a geographic item
GMT_IS_POINT	Multi-dimensional point data
GMT_IS_LINE	Geographic or Cartesian line segments
GMT_IS_POLYGON	Geographic or Cartesian closed polygons
GMT_IS_SURFACE	2-D gridded surface

index	description
GMT_XLO	x_min (west) boundary of grid subset
GMT_XHI	x_max (east) boundary of grid subset
GMT_YLO	y_min (south) boundary of grid subset
GMT_YHI	y_max (north) boundary of grid subset
GMT_ZLO	z_min (bottom) boundary of 3-D matrix subset
GMT_ZHI	z_max (top) boundary of 3-D matrix subset

7.2.3 Resource initialization

All GMT programs dealing with input or output files given on the command line, and perhaps defaulting to the standard input or output streams if no files are given, must call the i/o initializer function `GMT_Init_IO` once for each direction required (i.e., input and output separately). For input it determines how many input sources have already been registered. If none has been registered then it scans the program arguments for any filenames given on the command line and register these input resources. Finally, if we still have found no input sources we assign the standard input stream as the single input source. For output it is similar: If no single destination has been registered we specify the standard output stream as the output destination. Only one main output destination is allowed to be active when a module writes data (some modules also write additional output via program-specific options). The prototype for this function is

```
int GMT_Init_IO (void *API, unsigned int family, unsigned int geometry,
                 unsigned int direction, unsigned int mode,
                 unsigned int n_args, void *args);
```

where `family` specifies what kind of resource is to be registered, `geometry` specifies the geometry of the data, `direction` is either `GMT_IN` or `GMT_OUT`, and `mode` is a bit flag that determines what we do if no resources have been registered. The choices are

GMT_ADD_FILES_IF_NONE means "add command line (option) files if none have been registered already"

GMT_ADD_FILES_ALWAYS means "always add any command line files"

GMT_ADD_STDIO_IF_NONE means "add std* if no other input/output have been specified"

GMT_ADD_STDIO_ALWAYS means "always add std* even if resources have been registered".

GMT_ADD_EXISTING means "only use already registered resources".

The standard behavior is GMT_REG_DEFAULT. Next, `n_args` is 0 if `args` is the head of a linked list of options (further discussed in Section [sec:func]); otherwise `args` is an array of `n_args` strings (i.e., the int argc, char *argv[] model)

Many programs will register an export location where results of a GMT function (say, a filtered grid) should be returned, but may then wish to use that variable as an *input* resource in a subsequent module call. This is accomplished by re-registering the resource as an *input* source, thereby changing the *direction* of the data set. The function returns 1 if there is an error; otherwise it returns 0.

7.2.4 Dimension parameters for user 1-D column vectors

We refer to Table [tbl:vector]. The `type` array must hold the data type of each data column in the user's program. All types other than GMT_DOUBLE will be converted internally in GMT to `double`, thus possibly increasing memory requirements. If the type is GMT_DOUBLE then GMT will be able to use the column directly by reference. The `n_columns` and `n_rows` parameters indicate the number of vectors and their common length. If these are not yet known you may pass 0 for these values and set `alloc_mode` to GMT_ALLOCATED_BY_GMT; this will make sure GMT will allocate the necessary memory to the variable you specify.

7.2.5 Dimension parameters for user 2-D table arrays

We refer to Table [tbl:matrix]. The `type` parameter specifies the data type used for the array in the user's program. All types other than GMT_FLOAT will be converted internally in GMT to `float`, thus possibly increasing memory requirements. If the type is GMT_FLOAT then GMT may be able to use the matrix directly by reference. The `n_rows` and `n_columns` parameters indicate the dimensions of the matrix. If these are not yet known you may pass 0 for these values and set `alloc_mode` to GMT_ALLOCATED_BY_GMT; this will make sure GMT will allocate the necessary memory at the location you specify. Fortran users will instead have to specify a size large enough to hold the anticipated output data. The `registration` and `range` gives the grid registration and domain. Finally, use `dim` to indicate if the memory matrix has a dimension that exceeds that of the leading row (or column) dimension. Note: For GMT_IS_TEXTSET the user matrix is expected to be a 2-D character array with a fixed row length of `dim` but we only consider the first `n_columns` characters. For data grids you will also need to specify the `registration` (see the GMT Cookbook and Reference, *App-file-formats* for description of the two forms of registration) and data domain `range`.

7.3 Create empty resources

If your application needs to build and populate GMT resources in ways that do not depend on external resources (files, memory locations, etc.), then youGMT_Create_Data can obtain a "blank slate" by calling

```
void *GMT_Create_Data (void *API, unsigned int family, unsigned int geometry,
                       unsigned int mode, uint64_t par[], double *wesn,
                       double *inc, unsigned int registration, int pad, void *data)
```

which returns a pointer to the allocated resource. Pass `family` as one of GMT_IS_GRID, GMT_IS_IMAGE, GMT_IS_DATASET, GMT_IS_TEXTSET, or GMT_IS_CPT, or via the modifiers GMT_IS_VECTOR or GMT_IS_MATRIX when handling user data. Also pass a compatible `geometry`. Depending on the family and your particular way of representing dimensions you may pass the additional parameters in one of two ways:

1. Actual integer dimensions of items needed.
2. Physical distances and increments of each dimension.

For the first case, pass the `par` array, as indicated below:

GMT_IS_GRID An empty GMT_GRID structure with a header is allocated; the data array is NULL. The `par` argument is not used.

GMT_IS_IMAGE An empty GMT_GRID structure with a header is allocated; the image array is NULL. The `par` argument is not used.

GMT_IS_DATASET An empty GMT_DATASET structure consisting of `par[0]` tables, each with `par[1]` segments, each with `par[2]` rows, all with `par[3]` columns, is allocated. The `wesn`, `inc`, and `registration` argument are ignored.

GMT_IS_TEXTSET An empty GMT_TEXTSET structure consisting of `par[0]` tables, each with `par[1]` segments, all with `par[2]` text records (rows), is allocated. The `wesn`, `inc`, and `registration` argument are ignored.

GMT_IS_CPT An empty GMT_PALETTE structure with `par[0]` palette entries is allocated. The `wesn`, `inc`, and `registration` argument are ignored.

GMT_IS_VECTOR An empty GMT_VECTOR structure with `par[0]` column entries is allocated. The `wesn`, `inc`, and `registration` argument are ignored.

GMT_IS_MATRIX An empty GMT_MATRIX structure is allocated. `par[3]` indicates the number of layers for a 3-D matrix, or pass 0, 1, or NULL for a 2-D matrix.

In this case, pass `wesn`, `inc` as NULL (or arrays with elements all set to 0). For the second approach, you instead pass `wesn`, `inc`, and `registration` and leave `par` as NULL (or with all elements equal 0). For grids and images you may pass `pad` to set the padding, or -1 to accept the GMT default. The `mode` determines what is actually allocated when you have chosen grids or images. As for GMT_Read_Data you can pass `GMT_GRID_ALL` to initialize the header and allocate space for the array. Alternatively, you can pass `GMT_GRID_HEADER_ONLY` to just initialize the grid or image header, and call a second time, passing `GMT_GRID_DATA_ONLY`, to allocate space for the array. In that second call you pass the pointer returned by the first call as `data` and specify the family; all other arguments should be NULL or 0. Normally, resources created by this function are considered to be input (i.e., have a direction that is GMT_IN). You can change that to GMT_OUT by adding in the bit flag GMT_VIA_OUTPUT. The function returns a pointer to the data container. In case of an error we return a NULL pointer and pass an error code via `API->error`.

7.4 Duplicate resources

Often you have read or created a data resource and then need an identical copy, presumably to make modifications to. Or, you want a copy with the same dimensions and allocated memory, except data values should not be duplicated. Alternatively, perhaps you just want to duplicate the header and skip the allocation and duplication of the data. These tasks are addressed by

```
void *GMT_Duplicate_Data (void *API, unsigned int family,
                          unsigned int mode, void *data);
```

which returns a pointer to the allocated resource. Specify which `family` and select `mode` from `GMT_DUPLICATE_DATA`, `GMT_DUPLICATE_ALLOC`, and `GMT_DUPLICATE_NONE`, as discussed above (also see `mode` discussion above). For datasets and textsets you can add modifiers `GMT_ALLOC_VERTICAL` or `GMT_ALLOC_HORIZONTAL` if you wish to put all data in a single long table or to paste all tables side-by-side, respectively (thus getting one wide table instead). Additional note for datasets: Normally we allocate the output given the corresponding input dimensions. You can override these by specifying your alternative dimensions in the input dataset variable dim[]. The `data` is a pointer to the resource you wish to duplicate. In case of an error we return a NULL pointer and pass an error code via `API->error`.

7.5 Get resource ID

Resources created by these two methods can be used as in various ways. Sometimes you want to pass them as input to other modules, in which case you need to registration ID of that resource. This task are performed by

```
void *GMT_Get_ID (void *API, unsigned int family,
                              unsigned int direction, void *data);
```

which returns the ID number of the allocated resource. Specify which `family` and select `direction` from `GMT_IN` or `GMT_OUT`. The `data` is a pointer to the resource you whose ID you need. In case of an error we return GMT_NOTSET and pass an error code via `API->error`.

7.6 Import Data

If your main program needs to read any of the five recognized data types (CPT files, data tables, text tables, GMT grids, or images) you will use the GMT_Get_Data or GMT_Read_Data functions, which both return entire data sets. In the case of data and text tables you may also select record-by-record reading using the GMT_Get_Record function. As a general rule, your program development simplifies if you can read entire resources into memory with GMT_Get_Data or GMT_Read_Data However, if this leads to unacceptable memory usage or if the program logic is particularly simple, you may obtain one data record at the time via GMT_Get_Record

All input functions takes a parameter called `mode`. The `mode` parameter generally has different meanings for the different data types and will be discussed below. However, one bit setting is common to all types: By default, you are only allowed to read a data source once; the source is then flagged as having been read and subsequent attempts to read from the same source will result in a warning and no reading takes place. In the unlikely event you need to re-read a source you can override this default behavior by adding GMT_IO_RESET to your `mode` parameter. Note that this override does not apply to sources that are streams or file handles, as it may not be possible to re-read their contents.

7.6.1 Enable Data Import

Once all input resources have been registered, we signal the API that we are done with the registration phase and are ready to start the actual data import. This step is only required when reading one record at the time. We initialize record-by-record reading by calling GMT_Begin_IO This function enables dataset and textset record-by-record reading and prepares the registered sources for the upcoming import. The prototype is

```
int GMT_Begin_IO (void *API, unsigned int family, unsigned int direction,
                        unsigned int mode, unsigned int header);
```

where `family` specifies the resource type to be read or written (see Table [tbl:family]; only GMT_IS_DATASET and GMT_IS_TEXTSET are available for record-by-record handling). The `direction` is either GMT_IN or GMT_out, so for import we obviously use GMT_IN. The function determines the first input source and sets up procedures for skipping to the next input source in a virtual data set. The GMT_Get_Record function will not be able to read any data before GMT_Begin_IO has been called. As you might guess, there is a companion GMT_End_IO function that completes, then disables record-by-record data access. You can use these several times to switch modes between registering data resources, doing the importing/exporting, and disabling further data access, perhaps to do more registration. We will discuss GMT_End_IO once we are done with the data import. The `mode` option is used to allow output to write table header information (`GMT_HEADER_ON`) or not (`GMT_HEADER_OFF`). This is usually on unless you are writing messages and other non-data. The final `header` argument determines if the common header-block should be written during initialization; choose between `GMT_HEADER_ON` and `GMT_HEADER_OFF`. The function returns 1 if there is an error; otherwise it returns 0.

7.6.2 Import a data set

If your program needs to import any of the five recognized data types (CPT table, data table, text table, GMT grid, or image) you will use either the GMT_Read_Data or GMT_Get_Data functions. The former is typically used when reading from files, streams (e.g., `stdin`), or an open file handle, while the latter is only used with a registered resource via its unique ID. Because of the similarities of these five import functions we use an generic form that covers all of them.

Import from a file, stream, or handle

To read an entire resource from a file, stream, or file handle, use

```
void *GMT_Read_Data (void *API, unsigned int family, unsigned int method,
                     unsigned int geometry, unsigned int mode, double wesn[],
                     char *input, void *ptr);
```

- API – None of your business
- *family*
- *method*
- *geometry*
- *wesn*

void *GMT_Read_Data (void *API, unsigned int family, unsigned int method, unsigned int geometry, unsigned int mode, double wesn[], char *input, void *ptr);

Parameters

- **API** – None of your business
- **family** – *family*

Return type None (void)

where `ptr` is NULL except when reading grids in two steps (i.e., first get a grid structure with a header, then read the data). Most of these arguments have been discussed earlier. This function can be called in three different situations:

1. If you have a single source (filename, stream pointer, etc.) you can call GMT_Read_Data directly; there is no need to first register the source with GMT_Register_IO or gather the sources with GMT_Init_IO. However, if you did register a single source you can still pass it via an encoded

filename (see GMT_Encode_ID) or you can instead use GMT_Get_Data using the integer ID directly (see next section).

2. If you want to specify `stdin` as source then use `input` as NULL.
3. If you already registered all desired sources with GMT_Init_IO then you indicate this by passing `geometry` = 0.

Space will be allocated to hold the results, if needed, and a pointer to the object is returned. If there are errors we simply return NULL and report the error. The `mode` parameter has different meanings for different data types.

CPT table `mode` contains bit-flags that control how the CPT file's back-, fore-, and NaN-colors should be initialized. Select 0 to use the CPT file's back-, fore-, and NaN-colors, 2 to replace these with the GMT default values, or 4 to replace them with the color table's entries for highest and lowest value.

Data table `mode` is currently not used.

Text table `mode` is currently not used.

GMT grid Here, `mode` determines how we read the grid: To read the entire grid and its header, pass GMT_GRID_ALL. However, if you need to extract a sub-region you must first read the header by passing GMT_GRID_HEADER_ONLY, then examine the header structure range attributes and to specify a subset via the array `wesn`, and finally call GMT_Read_Data a second time, now with `mode` = GMT_GRID_DATA_ONLY and passing your `wesn` array and the grid structure returned from the first call as `ptr`. In the event your data array should be allocated to hold both the real and imaginary parts of a complex data set you must add either GMT_GRID_IS_COMPLEX_REAL or GMT_GRID_IS_COMPLEX_IMAG to `mode` so as to allow for the extra memory needed and to stride the input values correctly. If your grid is huge and you must read it row-by-row, set `mode` to GMT_GRID_HEADER_ONLY | GMT_GRID_ROW_BY_ROW. You can then access the grid row-by-row using GMT_Get_Row By default the rows will be automatically processed in order. To completely specify which row to be read, use GMT_GRID_ROW_BY_ROW_MANUAL instead.

Import from a memory location

If you are importing via variables or prefer to first register the source, then you should use GMT_Get_Data instead. This function requires fewer arguments since you simply pass the unique ID number of the resource. The function is described as follows:

```
void *GMT_Get_Data (void *API, int ID, unsigned int mode, void *ptr);
```

The `ID` is the unique object ID you received when registering the resource, `mode` controls some aspects of the import (see GMT_Read_Data above), while `ptr` is NULL except when reading grids in two steps (i.e., first get a grid structure with a header, then read the data). Other arguments have been discussed earlier. Space will be allocated to hold the results, if needed, and a pointer to the object is returned. If there are errors we simply return NULL and report the error.

Retrieve an allocated result

Finally, if you need to access the result that a GMT module wrote to a memory location, then you must register an output destination with GMT_Register_IO first (passing `ptr` == NULL). The GMT module will then allocate space to hold the output and let the API know where this memory resides. You can then use GMT_Retrieve_Data to get a pointer to the container where the data set was stored. This function requires fewer arguments since you simply pass the unique ID number of the resource. The function is described as follows:

```
void *GMT_Retrieve_Data (void *API, int ID);
```

The ID is the unique object ID you received when registering the NULL resource earlier, Since this container has already been created, a pointer to the object is returned. If there are errors we simply return NULL and report the error.

7.6.3 Importing a data record

If your program will read data table records one-by-one you must first enable this input mechanism with GMT_Begin_IO and then read the records in a loop using

```
void *GMT_Get_Record (void *API, unsigned int mode, int *nfields);
```

where the returned value is either a pointer to a double array with the current row values or to a character string with the current row, depending on mode. In either case these pointers point to memory internal to GMT and should be considered read-only. When we reach end-of-file, encounter conversion problems, read header comments, or identify segment headers we return a NULL pointer. The nfields pointer will return the number of fields returned; pass NULL if your program should ignore this information.

Normally (mode == GMT_READ_DOUBLE), we return a pointer to the double array. To read text records, supply instead mode == GMT_READ_TEXT and we instead return a pointer to the text record. However, if you have input records that mixes organized floating-point columns with text items you could pass mode == GMT_READ_MIXED. Then, GMT will attempt to extract the floating-point values; you can still access the record string, as discussed below. Finally, if your application needs to be notified when GMT closes one file and opens the next, add GMT_FILE_BREAK to mode and check for the status code GMT_IO_NEXT_FILE (by default, we treat the concatenation of many input files as a single virtual file). Using GMT_Get_Record requires you to first initialize the source(s) with GMT_Init_IO. For certain records, GMT_Get_Record will return NULL and sets status codes that your program will need to examine to take appropriate response. Table [tbl:iostatus] list the various status codes you can check for, using GMT_Status_IO (see next section).

7.6.4 Examining record status

Programs that read record-by-record must be aware of what the current record represents. Given the presence of headers, data gaps, NaN-record, etc. the developer will want to check the status after reading the next record. The internal i/o status mode can be interrogated with the function

```
int GMT_Status_IO (void *API, unsigned int mode);
```

which returns 0 (false) or 1 (true) if the current status is reflected by the specified mode. There are 11 different modes available to programmers; for a list see Table [tbl:iostatus]. For an example of how these may be used, see the test program testgmtio.c. Developers who plan to import data on a record-by-record basis may also consult the source code of, say, blockmean.c or pstext.c, to see examples of working code.

mode	description and return value
GMT_IO_DATA_RECORD	1 if we read a data record
GMT_IO_TABLE_HEADER	1 if we read a table header
GMT_IO_SEGMENT_HEADER	1 if we read a segment header
GMT_IO_ANY_HEADER	1 if we read either header record
GMT_IO_MISMATCH	1 if we read incorrect number of columns
GMT_IO_EOF	1 if we reached the end of the file (EOF)
GMT_IO_NAN	1 if we only read NaNs
GMT_IO_GAP	1 if this record implies a data gap
GMT_IO_NEW_SEGMENT	1 if we enter a new segment
GMT_IO_LINE_BREAK	1 if we encountered a segment header, EOF, NaNs or gap
GMT_IO_NEXT_FILE	1 if we finished one file but not the last

7.6.5 Importing a grid row

If your program must read a grid file row-by-row you must first enable row-by-row reading with `GMT_Read_Data` and then use the `GMT_Get_Row` function in a loop; the prototype is

```
int GMT_Get_Row (void *API, int row_no, struct GMT_GRID *G, float *row);
```

where `row` is a pointer to a single-precision array to receive the current row, `G` is the grid in question, and `row_no` is the number of the current row to be read. Note this value is only considered if the row-by-row mode was initialized with GMT_GRID_ROW_BY_ROW_MANUAL. The user must allocate enough space to hold the entire row in memory.

7.6.6 Disable Data Import

Once the record-by-record input processing has completed we disable further input to prevent accidental reading from occurring (due to poor program structure, bugs, etc.). We do so by calling `GMT_End_IO`. This function disables further record-by-record data import; its prototype is

```
int GMT_End_IO (void *API, unsigned int direction, unsigned int mode);
```

and we specify `direction` = GMT_IN. At the moment, `mode` is not used. This call will also reallocate any arrays obtained into their proper lengths. The function returns 1 if there is an error (which is passed back with `API->error`), otherwise it returns 0.

7.7 Manipulate data

[sec:manipulate]

Once you have created and allocated and empty resources, or read in resources from the outside, you will wish to manipulate their contents. This section discusses how to set up loops and access the important variables for the various data families. For grids and images it may be required to know what the coordinates are at each node point. This can be obtained via arrays of coordinates for each dimension, obtained by

```
double *GMT_Get_Coord (void *API, unsigned int family, unsigned int dim, void *data);
```

where `family` must be GMT_IS_GRID or GMT_IS_DATASET, `dim` is either GMT_IS_X or GMT_IS_Y, and `data` is the grid or image pointer. This function will be used below in our example on grid manipulation.

Another aspect of dealing with grids and images is to convert a row and column 2-D reference to our 1-D array index. Because of grid and image boundary padding the relationship is not straightforward, hence we supply

```
int64_t GMT_Get_Index (struct GMT_GRID_HEADER *header, int row, int col);
```

where the `header` is the header of either a grid or image, and `row` and `col` is the 2-D position in the grid or image. We return the 1-D array position; again this function is used below in our example.

7.7.1 Manipulate grids

Most applications wishing to manipulate grids will want to loop over all the nodes, typically in a manner organized by rows and columns. In doing so, the coordinates at each node may also be required for a calculation. Below is a snippet of code that shows how to do visit all nodes in a grid and assign each node the product x * y:

```
int row, col, node;
double *x_coord = NULL, *y_coord = NULL;
< ... create a grid G or read one ... >
x_coord = GMT_Get_Coord (API, GMT_IS_GRID, GMT_X, G);
y_coord = GMT_Get_Coord (API, GMT_IS_GRID, GMT_Y, G);
for (row = 0; row < G->header->ny) {
    for (col = 0; col < G->header->nx; col++) {
        node = GMT_Get_Index (G->header, row, col);
        G->data[node] = x_coord[col] * y_coord[row];
    }
}
```

Note the use of `GMT_Get_Index` to get the grid node number associated with the `row` and `col` we are visiting. Because GMT grids have padding (for boundary conditions) the relationship between rows, columns, and node indices is more complicated and hence we hide that complexity in `GMT_Get_Index`. Note that for trivial procedures such setting all grid nodes to a constant (e.g., -9999.0) where the row and column does not enter you can instead do a single loop:

```
int node;
< ... create a grid G or read one ... >
for (node = 0; node < G->header->size) G->data[node] = -9999.0;
```

Note we must use `G->header->size` (size of allocated array) and not `G->header->nm` (number of nodes in grid) since the latter is smaller due to the padding and a single loop like the above treats the pad as part of the "inside" grid.

7.7.2 Manipulate data tables

Another common application is to process the records in a data table. Because GMT consider the GMT_DATASET resources to contain one or more tables, each of which may contain one or more segments, all of which may contain one or more columns, you will need to have multiple loops to visit all entries. The following code snippet will visit all data records and add 1 to all columns beyond the first two (x and y):

```
int tbl, seg, row, col;
struct GMT_DATATABLE *T = NULL;
struct GMT_DATASEGMENT *S = NULL;

< ... create a dataset D or read one ... >
for (tbl = 0; tbl < D->n_tables; tbl++) {        /* For each table */
  T = D->table[tbl];        /* Convenient shorthand for current table */
  for (seg = 0; seg < T->n_segments; seg++) {    /* For all segments */
    S = T->segment[seg];    /* Convenient shorthand for current segment */
```

```
        for (row = 0; row < S->n_rows; row++) {
          for (col = 2; col < T->n_columns; col++) {
            S->coord[col][row] += 1.0;
          }
        }
      }
    }
```

7.7.3 Manipulate text tables

When data file contain text mixed in with numbers you must open the file as a GMT_TEXTSET and do your own parsing of the data records. The following code snippet will visit all text records and print them out:

```
int tbl, seg, row, col;
struct GMT_TEXTTABLE *T = NULL;
struct GMT_TEXTSEGMENT *S = NULL;

< ... create a textset D or read one ... >
for (tbl = 0; tbl < D->n_tables; tbl++) {   /* For each table */
  T = D->table[tbl];        /* Convenient shorthand for current table */
  for (seg = 0; seg < T->n_segments; seg++) {   /* For all segments */
    S = T->segment[seg];    /* Convenient shorthand for current segment */
    for (row = 0; row < S->n_rows; row++) {
      printf ("T=%d S=%d R=%d : %s\n", tbl, seg, row, S->record[row]);
    }
  }
}
```

7.8 Message and Verbose Reporting

The API provides two functions for your program to present information to the user during the run of the program. One is used for messages that are always written while the other is used for reports that must exceed the verbosity settings specified via **-V**.

```
int GMT_Report (void *API, unsigned int level, char *message, ...);
```

This function takes a verbosity level and a multi-part message (e.g., a format statement and zero or more variables). The verbosity `level` is an integer in the 0–5 range; these are listed in Table [tbl:verbosity]. You assign an appropriate verbosity level to your message, and depending on the chosen run-time verbosity level set via **-V** your message may or may not be reported. Only messages whose stated verbosity level is lower or equal to the **-V***level* will be printed.

constant	description
GMT_MSG_QUIET	No messages whatsoever
GMT_MSG_NORMAL	Default output, e.g., warnings and errors only
GMT_MSG_COMPAT	Compatibility warnings
GMT_MSG_VERBOSE	Verbose level
GMT_MSG_LONG_VERBOSE	Longer verbose
GMT_MSG_DEBUG	Debug messages for developers mostly

```
int GMT_Message (void *API, unsigned int mode, char *format, ...);
```

This function always prints its message to the standard output. Use the `mode` value to control if a time stamp should preface the message. and if selected how the time information should be formatted. See Table *timemodes* for the various modes.

constant	description
GMT_TIME_NONE	Display no time information
GMT_TIME_CLOCK	Display current local time
GMT_TIME_ELAPSED	Display elapsed time since last reset
GMT_TIME_RESET	Reset the elapsed time to 0

7.9 Presenting and accessing GMT options

[sec:parsopt] As you develop a program you may need to rely on some of the GMT common options. For instance, you may wish to have your program present the `-R` option to the user, let GMT handle the parsing, and examine the values. You may also wish to encode your own custom options that may require you to parse user text into the corresponding floating point dimensions, length, coordinates, time, etc. The API provides several functions to simplify these tedious parsing tasks. This section is intended to show how the programmer will obtain information from the user that is necessary to do the task at hand (e.g., special options to provide values and settings for the program). In the following section we will concern ourselves with preparing arguments for calling any of the GMT modules.

7.9.1 Display usage syntax for GMT common options

You can have your program menu display the standard usage message for a GMT common option by calling the function

```
int GMT_Option (void *API, char *options);
```

where `options` is a comma-separated list of GMT common options (e.g., “R,J,O,X”). You can repeat this function with different sets of options in order to intersperse your own custom options with in an overall alphabetical order; see any GMT module for examples of typical layouts.

7.9.2 Parsing the GMT common options

The parsing of all GMT common option is done by

```
int GMT_Parse_Common (void *API, char *args, struct GMT_OPTION *list);
```

where `args` is a string of the common GMT options your program may use. An error will be reported if any of the common GMT options fail to parse, and if so we return TRUE; if not errors we return FALSE. All other options, including file names, will be silently ignored. The parsing will update the internal GMT information structure that affects program operations.

7.9.3 Inquiring about the GMT common options

The API provide only a limited window into the full GMT machinery accessible to the modules. You can determine if a particular common option has been parsed and in some cases determine the values that was set with

```
int GMT_Get_Common (void *API, unsigned int option, double *par);
```

where `option` is a single option character (e.g., ‘R’) and `par` is a double array with at least a length of 6. If the particular option has been parsed then the function returns the number of parameters passed back via `par`; otherwise we return -1. For instance, to determine if the `-R` was set and what the resulting region was set to you may call

```
if (GMT_Get_Common (API, 'R', wesn)) != -1) {
    /* wesn now contains the boundary information */
}
```

The `wesn` array could now be passed to the various read and create functions for GMT resources.

7.9.4 Parsing text values

Your program may need to request values from the user, such as distances, plot dimensions, coordinates, and other data. The conversion from such text to actual distances, taking units into account, is tedious to program. You can simplify this by using

```
int GMT_Get_Value (void *API, char *arg, double par[]);
```

where `arg` is the text item with one or more values that are separated by commas, spaces, or slashes, and `par` is an array long enough to hold all the items you are parsing. The function returns the number of items parsed, or -1 if there is an error. For instance, assume the character string `origin` was given by the user as two geographic coordinates separated by a slash (e.g., `"35:45W/19:30:55.3S"`). We obtain the two coordinates as decimal degrees by calling

```
n = GMT_Get_Value (API, origin, pair);
```

Your program can now check that `n` equals 2 and then use the values in `pairs`. Note: Dimensions given with units of inches, cm, or points are converted to the GMT default length unit (*PROJ_LENGTH_UNIT*) [cm], while distances given in km, nautical miles, miles, feet, or survey feet are returned in meters. Arc lengths in minutes and seconds are returned in decimal degrees, and date/time values are returned in seconds since the epoch (1970).

7.9.5 Inquiring about a GMT default parameter

If your program needs to determine one or more of the current GMT default settings you can do so via

```
int GMT_Get_Default (void *API, char *keyword, char *value);
```

where `keyword` is one such keyword (e.g., *PROJ_LENGTH_UNIT*) and `value` must be a character array long enough to hold the answer. Depending on what parameter you selected you could further convert it to a numerical value with `GMT_Get_Value` or just use it in a text comparison.

7.10 Prepare module options

[sec:func] One of the advantages of programming with the API is that you have access to the high-level GMT modules. For example, if your program must compute the distance from a point to all other points on the node you can simply set up options and call `GMT_grdmath` to do it for you and accept the result back as an input grid. All the module interfaces are identical are looks like

```
int GMT_Call_Module (void *API, const char *module, int mode, void *args);
```

Here, `module` can be any of the GMT modules, such as `psxy` or `grdvolume`. All GMT modules may be called with one of three sets of `args` depending on `mode`. The three modes differ in how the options are passed to the module:

***mode* == GMT_MODULE_EXIST** Just print a brief one-line summary of the module; args should be NULL. If `module` equals NULL then we list summaries for all the modules.

***mode* == GMT_MODULE_PURPOSE** Just prints the purpose of the module; args should be NULL.

***mode* == GMT_MODULE_OPT** Expects `args` to be a pointer to a doubly-linked list of objects with individual options for the current program. We will see how API functions can help prepare such lists.

***mode* == GMT_MODULE_CMD** Expects `args` to be a single text string with all required options.

mode* > *0 Expects `args` to be an array of text options and `mode` to be a count of how many options are passed (i.e., the `argc, argv[]` model used by the GMT programs themselves).

If no module by the given name is found we return -1.

7.10.1 Set program options via text array arguments

When `mode > 0` we expect an array `args` of character strings that each holds a single command line options (e.g., "-R120:30/134:45/8S/3N") and interpret `mode` to be the count of how many options are passed. This, of course, is almost exactly how the stand-alone GMT programs are called (and reflects how they themselves are activated internally). We call this the "argc–argv" mode. Depending on how your program obtains the necessary options you may find that this interface offers all you need.

7.10.2 Set program options via text command

If `mode == 0` then `args` will be examined to see if it contains several options within a single command string. If so we will break these into separate options. This is useful if you wish to pass a single string such as "-R120:30/134:45/8S/3N -JM6i mydata.txt -Sc0.2c". We call this the "command" mode.

7.10.3 Set program options via linked structures

The third, linked-list interface allows developers using higher-level programming languages to pass all command options via a pointer to a NULL-terminated, doubly-linked list of option structures, each containing information about a single option. Here, instead of text arguments we pass the pointer to the linked list of options mentioned above, and `mode` must be passed as -1 (or any negative value). Using this interface can be more involved since you need to generate the linked list of program options; however, utility functions exist to simplify its use. This interface is intended for programs whose internal workings are better suited to generate such arguments – we call this the "options" mode. The order in the list is not important as GMT will sort it internally according to need. The option structure is defined below.

```
struct GMT_OPTION {
    char                option;    /* Single character of the option (e.g., 'G' for -G) */
    char               *arg;       /* String pointer with arguments (NULL if not used) */
    struct GMT_OPTION *next;       /* Pointer to next option (NULL for last option) */
    struct GMT_OPTION *prev;       /* Pointer to previous option (NULL for first option) */
};
```

7.10.4 Convert between text and linked structures

To assist programmers there are also two convenience functions that allow you to convert between the two argument formats. They are

```
struct GMT_OPTIONS *GMT_Create_Options (void *API, int argc, void *args);
```

This function accepts your array of text arguments (cast via a void pointer), allocates the necessary space, performs the conversion, and returns a pointer to the head of the linked list of program options. However, in case of an error we return a NULL pointer and set `API->error` to indicate the nature of the problem. Otherwise, the pointer may now be passed to the relevant `GMT_module`. Note that if your list of text arguments were obtained from a C `main()` function then `argv[0]` will contain the name of the calling program. To avoid passing this as a file name option, call `GMT_Create_Options` with `argc-1` and `argv+1`. If you wish to pass a single text string with multiple options (in lieu of an array of text strings), then pass `argc` = 0. When no longer needed you can remove the entire list by calling

```
int GMT_Destroy_Options (void *API, struct GMT_OPTION **list);
```

The function returns 1 if there is an error (which is passed back with `API->error`), otherwise it returns 0.

The inverse function prototype is

```
char **GMT_Create_Args (void *API, int *argc, struct GMT_OPTIONS *list);
```

which allocates space for the text strings and performs the conversion; it passes back the count of the arguments via `argc` and returns a pointer to the text array. In the case of an error we return a NULL pointer and set `API->error` to reflect the error type. Note that `argv[0]` will not contain the name of the program as is the case the arguments presented by a C `main()` function. When you no longer have any use for the text array, call

```
int GMT_Destroy_Args (void *API, int argc, char **argv[]);
```

to deallocate the space used. This function returns 1 if there is an error (which is passed back with `API->error`), otherwise it returns 0.

Finally, to convert the linked list of option structures to a single text string command, use

```
char *GMT_Create_Cmd (void *API, struct GMT_OPTION *list);
```

Developers who plan to import and export GMT shell scripts might find it convenient to use these functions. In case of an error we return a NULL pointer and set `API->error`, otherwise a pointer to an allocated string is returned. When you no longer have any use for the text string, call

```
int _GMT_Destroy_Cmd (void *API, char **argv);
```

to deallocate the space used. This function returns 1 if there is an error (which is passed back with `API->error`), otherwise it returns 0.

7.10.5 Manage the linked list of options

Several additional utility functions are available for programmers who wish to manipulate program option structures within their own programs. These allow you to create new option structures, append them to the linked list, replace existing options with new values, find a particular option, and remove options from the list. Note: The order in which the options appear in the linked list is of no consequence to GMT. Internally, GMT will sort and process the options in the manner required. Externally, you are free to maintain your own order.

Make a new option structure

`GMT_Make_Option` will allocate a new option structure, assign it values given the `option` and `arg` parameter (pass NULL if there is no argument for this option), and returns a pointer to the allocated structure. The prototype is

```
struct GMT_OPTION *GMT_Make_Option (void *API, char option, char *arg);
```

Should memory allocation fail the function will print an error message set an error code via `API->error`, and return NULL.

Append an option to the linked list

`GMT_Append_Option` will append the specified `option` to the end of the doubly-linked `list`. The prototype is

```
struct GMT_OPTION *GMT_Append_Option (void *API, struct GMT_OPTION *option, \
                                      struct GMT_OPTION *list);
```

We return the list back, and if `list` is given as NULL we return `option` as the start of the new list. Any errors results in a NULL pointer with `API->error` holding the error type.

Find an option in the linked list

`GMT_Find_Option` will return a pointer `ptr` to the first option in the linked list starting at `list` whose option character equals `option`. If not found we return NULL. While this is not necessarily an error we still set `API->error` accordingly. The prototype is

```
struct GMT_OPTION *GMT_Find_Option (void *API, char option,
                                    struct GMT_OPTION *list);
```

If you need to look for multiple occurrences of a certain option you will need to call `GMT_Find_Option` again, passing the option following the previously found option as the `list` entry, i.e.,

```
list = *ptr->next;
```

Update an existing option in the list

`GMT_Update_Option` will replace the argument of `current` with the new argument `arg` and otherwise leave the option at its place in the list. The prototype is

```
int GMT_Update_Option (void *API, struct GMT_OPTION *current, char *arg);
```

An error will be reported if (a) `current` is NULL or (b) `arg` is NULL. The function returns 1 if there is an error, otherwise it returns 0.

Delete an existing option in the linked list

You may use `GMT_Delete_Option` to remove `option` from the linked `list`. The prototype is

```
int GMT_Delete_Option (void *API, struct GMT_OPTION *current);
```

We return TRUE if the option is not found in the list and set `API->error` accordingly. Note: Only the first occurrence of the specified option will be deleted. If you need to delete all such options you will need to call this function in a loop until it returns a non-zero status.

Specify a file via an linked option

To specify an input file name via an option, simply use < as the option (this is what `GMT_Create_Options` does when it finds filenames on the command line). Likewise, > can be used to explicitly indicate an output file. In order to append to an existing file, use >>. For example the following command would read from file.A and append to file.B:

```
gmtconvert -<file.A ->>file.B
```

These options also work on the command line but usually one would have to escape the special characters < and > as they are used for file redirection.

7.11 Calling a GMT module

Given your linked list of program options (or text array) and possibly some registered resources, you can now call the required GMT module using one of the two flavors discussed in section [sec:func]. All modules return an error or status code that your program should consider before processing the results.

7.12 Adjusting headers and comments

All header records in incoming datasets are stored in memory. You may wish to replace these records with new information, or append new information to the existing headers. This is achieved with

```
int GMT_Set_Comment (void *API, unsigned int family, unsigned int mode
                     void *arg, void *data)
```

Again, `family` selects which kind of resource is passed via `data`. The `mode` determines what kind of comment is being considered, how it should be included, and in what form the comment passed via `arg` is. Table [tbl:comments] lists the available options, which may be combined by adding (bitwise "or"). The `GMT_Set_Comment` does not actually output anything but sets the relevant comment and header records in the relevant structure. When a file is written out the information will be output as well (Note: Users can always decide if they wish to turn header output on or off via the common GMT option `-h`. For record-by-record writing you must enable the header block output when you call GMT_Begin_IO

constant	description
GMT_COMMENT_IS_TEXT	Comment is a text string
GMT_COMMENT_IS_OPTION	Comment is a linked list of GMT_OPTION structures
GMT_COMMENT_IS_COMMAND	Comment is the command
GMT_COMMENT_IS_REMARK	Comment is the remark
GMT_COMMENT_IS_TITLE	Comment is the title
GMT_COMMENT_IS_NAME_X	Comment is the x variable name (grids only)
GMT_COMMENT_IS_NAME_Y	Comment is the y variable name (grids only)
GMT_COMMENT_IS_NAME_Z	Comment is the z variable name (grids only)
GMT_COMMENT_IS_COLNAMES	Comment is the column names header
GMT_COMMENT_IS_RESET	Comment replaces existing information

The named modes (*command*, *remark*, *title*, *name_x,y,z* and *colnames* are used to distinguish regular text comments from specific fields in the header structures of the data resources, such as `GMT_GRID`. For the various table resources (e.g., `GMT_DATASET`) these modifiers result in a specially formatted comments beginning with "Command: " or "Remark: ", reflecting how this type of information is encoded in the headers.

7.13 Exporting Data

If your program needs to write any of the four recognized data types (CPT files, data tables, text tables, or GMT grids) you can use the `GMT_Put_Data`. In the case of data and text tables, you may also consider the `GMT_Put_Record` function. As a general rule, your program organization may simplify if you can write the export the entire resource with `GMT_Put_Data`. However, if the program logic is simple or already involves using `GMT_Get_Record`, it may be better to export one data record at the time via `GMT_Put_Record`.

Both of these output functions takes a parameter called `mode`. The `mode` parameter generally takes on different meanings for the different data types and will be discussed below. However, one bit setting is common to all types: By default, you are only allowed to write a data resource once; the resource is then flagged to have been written and subsequent attempts to write to the same resource will quietly be ignored. In the unlikely event you need to re-write a resource you can override this default behavior by adding GMT_IO_RESET to your `mode` parameter.

7.13.1 Enable Data Export

Similar to the data import procedures, once all output destinations have been registered, we signal the API that we are done with the registration phase and are ready to start the actual data export. As for input, this step is only needed when dealing with record-by-record writing. Again, we enable record-by-record writing by calling `GMT_Begin_IO`, this time with `direction` = GMT_OUT. This function enables data export and prepares the registered destinations for the upcoming writing.

7.13.2 Exporting a data set

To have your program accept results from GMT modules and write them separately requires you to use the `GMT_Write_Data` or `GMT_Put_Data` functions. They are very similar to the `GMT_Read_Data` and `GMT_Get_Data` functions encountered earlier.

Exporting a data set to a file, stream, or handle

The prototype for writing to a file (via name, stream, or file handle) is

```
int GMT_Write_Data (void *API, unsigned int family, unsigned int method,
                    unsigned int geometry, unsigned int mode,
                    double wesn[], void *output, void *data);
```

where `data` is a pointer to any of the four structures discussed previously. Again, the `mode` parameter is specific to each data type:

CPT table `mode` controls if the CPT table's back-, fore-, and NaN-colors should be written (1) or not (0).

Data table If `method` is GMT_IS_FILE, then the value of `mode` affects how the data set is written:

GMT_WRITE_SET The entire data set will be written to the single file [0].

GMT_WRITE_TABLE Each table in the data set is written to individual files [1]. You can either specify an output file name that *must* contain one C-style format specifier for a int variable (e.g., "New_Table_%06d.txt"), which will be replaced with the table number (a running number from 0) *or* you must assign to each table *i* a unique output file name via the `D->table[i]->file[GMT_OUT]` variables prior to calling the function.

GMT_WRITE_SEGMENT Each segment in the data set is written to an individual file [2]. Same setup as for GMT_WRITE_TABLE except we use sequential segment numbers to build the file names.

GMT_WRITE_TABLE_SEGMENT Each segment in the data set is written to an individual file [3]. You can either specify an output file name that *must* contain two C-style format specifiers for two int variables (e.g., "New_Table_%06d_Segment_%03d.txt"), which will be replaced with the table and segment numbers, *or* you must assign to each segment *j* in each table *i* a unique output file name via the `D->table[i]->segment[j]->file[GMT_OUT]` variables prior to calling the function.

GMT_WRITE_OGR Writes the dataset in OGR/GMT format in conjunction with the `-a` setting [4].

Text table The `mode` is used the same way as for data tables.

GMT grid Here, `mode` may be GMT_GRID_HEADER_ONLY to only update a file's header structure, but normally it is simply GMT_GRID_ALL so the entire grid and its header will be exported (a subset is not allowed during export). However, in the event your data array holds both the real and imaginary parts of a complex data set you must add either GMT_GRID_IS_COMPLEX_REAL or GMT_GRID_IS_COMPLEX_IMAG to `mode` so as to export the corresponding grid values correctly. Finally, for native binary grids you may skip writing the grid header by adding GMT_GRID_NO_HEADER; this setting is ignored for other grid formats. If your output grid is huge and you are building it row-by-row, set `mode` to GMT_GRID_HEADER_ONLY | GMT_GRID_ROW_BY_ROW. You can then write the grid row-by-row using `GMT_Put_Row`. By default the rows will be automatically processed in order. To completely specify which row to be written, use GMT_GRID_ROW_BY_ROW_MANUAL instead.

If successful the function returns 0; otherwise we return 1 and set `API->error` to reflect to cause. Note: If `method` is GMT_IS_FILE, `family` is GMT_IS_GRID, and the filename implies a change from NaN to another value then the grid is modified accordingly. If you continue to use that grid after writing please be aware that the changes you specified were applied to the grid.

Exporting a data set to memory

If writing to a memory destination you will want to first register that destination and then use the returned ID with `GMT_Put_Data` instead:

```
int GMT_Put_Data (void *API, int ID, unsigned int mode, void *data);
```

where `ID` is the unique ID of the registered destination, `mode` is specific to each data type (and controls aspects of the output structuring), and `data` is a pointer to any of the four structures discussed previously. For more detail, see `GMT_Write_Data` above. If successful the function returns 0; otherwise we return 1 and set `API->error` to reflect to cause.

7.13.3 Exporting a data record

If your program must write data table records one-by-one you must first enable record-by-record writing with `GMT_Begin_IO` and then use the `GMT_Put_Record` function in a loop; the prototype is

```
int GMT_Put_Record (void *API, unsigned int mode, void *rec);
```

where `rec` is a pointer to either (a) a double-precision array with the current row. Then, `rec` is expected to hold at least as many items as the current setting of `n_col[GMT_OUT]`, which represents the number of columns in the output destination. Alternatively (b), `rec` points to a text string. The `mode` parameter must be set to reflect what is passed. Using `GMT_Put_Record` requires you to first initialize the destination with `GMT_Init_IO`. Note that for families GMT_IS_DATASET and GMT_IS_TEXTSET the methods GMT_IS_DUPLICATE and GMT_IS_REFERENCE are not supported since you can simply populate the GMT_DATASET structure directly. As mentioned, `mode` affects what is actually written:

GMT_WRITE_DOUBLE Normal operation that builds the current output record from the values in `rec`.

GMT_WRITE_TEXT For ASCII output mode we write the text string `rec`. If `rec` is NULL then we use the current (last imported) text record. If binary output mode we quietly skip writing this record.

GMT_WRITE_TABLE_HEADER For ASCII output mode we write the text string `rec`. If `rec` is NULL then we write the last read header record (and ensures it starts with #). If binary output mode we quietly skip writing this record.

GMT_WRITE_SEGMENT_HEADER For ASCII output mode we use the text string `rec` as the segment header. If `rec` is NULL then we use the current (last read) segment header record. If binary output mode instead we write a record composed of NaNs.

The function returns 1 if there was an error associated with the writing (which is passed back with `API->error`), otherwise it returns 0.

7.13.4 Exporting a grid row

If your program must write a grid file row-by-row you must first enable row-by-row writing with `GMT_Read_Data` and then use the `GMT_Put_Row` function in a loop; the prototype is

```
int GMT_Put_Row (void *API, int row_no, struct GMT_GRID *G, float *row);
```

where `row` is a pointer to a single-precision array with the current row, `G` is the grid in question, and `row_no` is the number of the current row to be written. Note this value is only considered if the row-by-row mode was initialized with GMT_GRID_ROW_BY_ROW_MANUAL.

7.13.5 Disable Data Export

Once the record-by-record output has completed we disable further output to prevent accidental writing from occurring (due to poor program structure, bugs, etc.). We do so by calling `GMT_End_IO`. This function disables further record-by-record data export; here, we obviously pass `direction` as GMT_OUT.

7.14 Destroy allocated resources

If your session imported any data sets into memory then you may explicitly free this memory once it is no longer needed and before terminating the session. This is done with the `GMT_Destroy_Data` function, whose prototype is

```
int GMT_Destroy_Data (void *API, void *data);
```

where `data` is the address of the pointer to a data container. Note that when each module completes it will automatically free memory created by the API; similarly, when the session is destroyed we also automatically free up memory. Thus, `GMT_Destroy_Data` is therefore generally only needed when you wish to directly free up memory to avoid running out of it. The function returns 1 if there is an error when trying to free the memory (the error code is passed back with `API->error`), otherwise it returns 0.

7.15 Terminate a GMT session

Before your program exits it should properly terminate the GMT session, which involves a call to

```
int GMT_Destroy_Session (void *API);
```

which simply takes the pointer to the GMT API control structure as its only arguments. It terminates the GMT machinery and deallocates all memory used by the GMT API book-keeping. It also unregisters any remaining resources previously registered with the session. The GMT API will only close files that it was responsible for opening in the first place. Finally, the API structure itself is freed so your main program does not need to do so. The function returns 1 if there is an error when trying to free the memory (the error code is passed back with `API->error`), otherwise it returns 0.

CHAPTER 8

The GMT FFT Interface

While the i/o options presented so far lets you easily read in a data table or grid and manipulated them, if you need to do so in the wavenumber domain then this chapter is for you. Here we outline how to take the Fourier transform of such data, perform calculations in the wavenumber domain, and take the inverse transform before writing the results. To assist programmers we also distribute fully functioning demonstration programs that takes you through the steps we are about to discuss; these demo programs may be used as your starting point for further development.

8.1 Presenting and Parsing the FFT options

Several GMT programs using FFTs present the same unified option and modifier sets to the user. The API makes these available as well. If your program needs to present the option usage you can call

```
unsigned int GMT_FFT_Option (void *API, char option, unsigned int dim,
                             char *string);
```

Here, `option` is the unique character used for this particular program option (most GMT programs have standardized on using 'N' but you are free to choose whatever you want except existing GMT common options). The `dim` sets the dimension of the transform, currently you must choose 1 or 2, while the `string` is a one-line message that states what the option does; you should tailor this to your program. If NULL then a generic message is placed instead.

To parse the user's selection you call

```
void *GMT_FFT_Parse (void *API, char option, unsigned int dim, char *args);
```

which accepts the user's string option via `args`; the other arguments are the same as those above. The function returns an opaque pointer to a structure with the chosen parameters.

8.2 Initializing the FFT machinery

Before your can take any transforms you must initialize the FFT machinery. This process involves a series of preparatory steps that are conveniently performed for you by

```
void *GMT_FFT_Create (void *API, void *X, unsigned int dim,
                      unsigned int mode, void *F);
```

Here, `X` is either your dataset or grid pointer, `dim` is the dimension of the transform (1 or 2 only), `mode` passes various flags to the setup, such as whether the data is real, imaginary, or complex, and `F` is the opaque pointer returned by `GMT_FFT_Parse`. Depending on the options you chose to pass to

GMT_FFT_Parse, the data may have a constant or a trend removed, reflected and extended by various symmetries, padded and tapered to desired transform dimensions, and possibly there are temporary files written out before the transform takes place. See the man page for a full explanation of the options presented by GMT_FFT_Option.

8.3 Taking the FFT

Now that everything has been set up you can perform the transform with

```
void *GMT_FFT (void *API, void *X, int direction, unsigned int mode, void *K);
```

which takes as direction either GMT_FFT_FWD or GMT_FFT_INV. The mode is used to specify if we pass a real (GMT_FFT_REAL) or complex (GMT_FFT_COMPLEX) data set, and K is the opaque pointer returned by GMT_FFT_Create. The transform is performed in place and returned via X. When done with your manipulations (below) you can call it again with the inverse flag to recover the corresponding space-domain version of your data. The FFT is fully normalized so that calling forward followed by inverse yields the original data set. The information passed via K determines if a 1-D or 2-D transform takes place; the key work is done via GMT_FFT_1D or GMT_FFT_1D below.

8.4 Taking the 1-D FFT

A lower-level 1-D FFT is also available via

```
int GMT_FFT_1D (void *API, float *data, uint64_t n, int direction, unsigned int mode);
```

which takes as direction either GMT_FFT_FWD or GMT_FFT_INV. The mode is used to specify if we pass a real (GMT_FFT_REAL) or complex (GMT_FFT_COMPLEX) data set, and data is the 1-D data array of length n that we wish to transform. The transform is performed in place and returned via data. When done with your manipulations (below) you can call it again with the inverse flag to recover the corresponding space-domain version of your data. The 1-D FFT is fully normalized so that calling forward followed by inverse yields the original data set. Note that unlike GMT_FFT, this functions does not do any data extension, mirroring, detrending, etc. but operates directly on the data array given.

8.5 Taking the 2-D FFT

A lower-level 2-D FFT is also available via

```
int GMT_FFT_2D (void *API, float *data, unsigned int nx, unsigned int ny,
                int direction, unsigned int mode);
```

which takes as direction either GMT_FFT_FWD or GMT_FFT_INV. The mode is used to specify if we pass a real (GMT_FFT_REAL) or complex (GMT_FFT_COMPLEX) data set, and data is the 2-D data array in row-major format, with row length nx and column length ny. The transform is performed in place and returned via data. When done with your manipulations (below) you can call it again with the inverse flag to recover the corresponding space-domain version of your data. The 2-D FFT is fully normalized so that calling forward followed by inverse yields the original data set. Note that unlike GMT_FFT, this functions does not do any data extension, mirroring, detrending, etc. but operates directly on the data array given.

8.6 Wavenumber calculations

As your data have been transformed to the wavenumber domain you may wish to operate on the various values as a function of wavenumber. We will show how this is done for datasets and grids separately. First, we present the function that returns an individual wavenumber:

```
double GMT_FFT_Wavenumber (void *API, uint64_t k, unsigned int mode, void *K);
```

where `k` is the index into the array or grid, `mode` specifies which wavenumber we want (it is not used for 1-D transform but for the 2-D transform we can select either the x-wavenumber (0), the y-wavenumber (1), or the radial wavenumber (2)), and finally the opaque vector used earlier.

8.6.1 1-D FFT manipulation

To be added later.

8.6.2 2-D FFT manipulation

The number of complex pairs in the grid is given by the header's `nm` variable, while `size` will be twice that value as it holds the number of components. To visit all the complex values and obtain the corresponding wavenumber we simply need to loop over `size` and call `GMT_FFT_Wavenumber`. This code snippet multiples the complex grid by the radial wavenumber:

```
uint64_t k;
for (k = 0; k < Grid->header->size; k++) {
    wave = GMT_FFT_Wavenumber (API, k, 2, K);
    Grid->data[k] *= wave;
}
```

Alternatively, you may choose to be more specific about which components are real and imaginary (especially if they are to be treated differently), and set up the loop this way:

```
uint64_t re, im;
for (re = 0, im = 1; re < Grid->header->size; re += 2, im += 2) {
    wave = GMT_FFT_Wavenumber (API, re, 2, K);
    Grid->data[re] *= wave;
    Grid->data[im] *= 2.0 * wave;
}
```

8.7 Destroying the FFT machinery

When done you terminate the FFT machinery with

```
double GMT_FFT_Destroy (void *API, void *K);
```

which simply frees up the memory allocated by the FFT machinery.

CHAPTER 9

FORTRAN interfaces

FORTRAN 90 developers who wish to use the GMT API may use the same API functions as discussed in Chapter 2. As we do not have much (i.e., any) experience with modern Fortran we are not sure to what extent you are able to access the members of the various structures, such as the GMT_GRID structure. Thus, this part will depend on feedback and for the time being is to be considered preliminary and subject to change. We encourage you to take contact should you wish to use the API with your Fortran 90 programs.

9.1 FORTRAN 77 Grid i/o

Because of a lack of structure pointers we can only provide a low level of support for Fortran 77. This API is limited to help you inquire, read and write GMT grids directly from Fortran 77. To inquire about the range of information in a grid, use

```
int GMT_F77_readgrdinfo (unsigned int dim[], double limits[], double inc[],
                         char *title, char *remark, char *file)
```

where `dim` returns the grid width, height, and registration, `limits` returns the min and max values for x, y, and z as three consecutive pairs, `inc` returns the x and y increment, the `title` and `remark` returns the values of these strings. The `file` argument is the name of the file we wish to inquire about. The function returns 0 unless there is an error. Note that you must declare your variables so that `limits` has at least 6 elements and `inc` and `dime` have at least 2 each.

To actually read the grid, we use

```
int GMT_F77_readgrd (float *array, unsigned int dim[], double wesn[],
                     double inc[], char *title, char *remark, char *file)
```

where `array` is the 1-D grid data array, `dim` returns the grid width, height, and registration, `limits` returns the min and max values for x, y, and z, `inc` returns the x and y increments, the `title` and `remark` returns the values of these strings. The `file` argument is the name of the file we wish to read from. The function returns 0 unless there is an error. Note on input, `dim[2]` can be set to 1 which means we will allocate the array for you; otherwise we assume space has already been secured. Also, if `dim[3]` is set to 1 we will in-place transpose the array from C-style row-major array order to Fortran column-major array order.

Finally, to write a grid to file you can use

```
int GMT_F77_writegrd_(float *array, unsigned int dim[], double wesn[],
                      double inc[], char *title, char *remark, char *file)
```

where `array` is the 1-D grid data array, `dim` specifies the grid width, height, and registration, `limits` may be used to specify a subset (normally, just pass zeros), `inc` specifies the x and y increments, the `title` and `remark` supplies the values of these strings. The `file` argument is the name of the file we wish to write to. The function returns 0 unless there is an error. Note on input, `dim[2]` can be set to 1 which means we will allocate the array for you; otherwise we assume space has already been secured. Also, if `dim[3]` is set to 1 we will in-place transpose the array from Fortran column-major array order to C-style row-major array order before writing. Note this means array will have been transposed when the function returns.

Index

A

API, 1