GRASS Programming Changes

Michael Shapiro
James Westervelt

March 1, 1991

1 Introduction

This paper introduces some of the changes that have been made to the GRASS compilation process and to the programming libraries for GRASS 4.0. It is assumed that the reader is familiar with programming in GRASS 3.0 and has read the *GRASS 3.0 Programmer's Manual*.

The compilation changes are presented in sections:

§2 Compiling GRASS with gmake4.0

§3 Merging interactive and non-interactive programs

The library changes are presented in sections:

§4 Command line parsing;

§5 Upgrades to better handle large color tables for raster maps;

§6 Upgrades to support latitude-longitude data bases;

§7 New routines to support plotting of vector data;

§8 Miscellaneous new routines; and

§9 Routines which have been deleted.

The libraries effected are the GISLIB (routines that begin with `G_`) and the DISPLAYLIB (routines that begin with `D_`). The programmer should include the "gis.h" header file to get the necessary structures and #defines.

2 Compiling GRASS Programs - gmake4.0

The GRASS compilation process has been upgraded to allow for multiple-architecture compilation from a single copy of the source code. This assumes that the source code is NFS mounted on various machines with differing architectures.

The basic differences between between compiling GRASS 3.0 and compiling GRASS 4.0 are:

1. The *Gmake* command has been renamed. Under GRASS 3.0, this command was called Gmake and lived under $GISBASE/src/CMD. Under GRASS 4.0, the command is called *gmake4.0* and lives in one of the Unix bin directories that is part of your $PATH. At CERL we have been installing it in /usr/local/bin.

2. Object files and library archives are compiled into subdirectories rather than into the same directory with the source files. These subdirectories are named OBJ.*arch* and LIB.*arch* respectively, where *arch* indicates the cpu architecture. For example OBJ.sun4 would be for sun/4 and sparc architectures, LIB.sun386 for 368 machines, etc.

3. The *gmake4.0* no longer produces a *makefile* in the source directory. It now creates a file called *make.rules* in the OBJ.*arch* directory.

This process is 99% compatible with existing Gmakefiles. The only cases where it would fail are (1) if the Gmakefile references the variable $(GIS) and (2) if an explicit compilation rule is given. The $(GIS) variable had been used to reference both the $(SRC) source directories as well as the $(BIN) binary directories. Since the source and binary do not have to be under the same parent directory, the $(GIS) variable has been deleted. Also, since the object files are now compiled into a subdirectory, explicit compilation rules are in direct conflict. These rules must be removed from the Gmakefile.

WARNING: compilation may be run only on one machine at a time. If you try to compile the same source directory on two machines simultaneously, things will not turn out properly. This is your responsibility – gmake can not detect simultaneous compilations.

3 Interactive and Non-Interactive Commands

GRASS 4.0 now merges both the command-line and interactive version of a function under the same name. For example, the GRASS 3.0 commands *reclass* and *Greclass* are now called *r.reclass*.

This merging happens in one of two ways:

1. The programmer writes a single program which uses the new parser capability (see §4). The parser has both a command-line and a rudimentary interactive interface.

2. The programmer writes a command-line version using the parser, but also provides an interactive version as a separate module to override the parser's interactive interface.

The second item requires that both the command-line program and the interactive program be somehow merged into one program. This is accomplished by putting both programs in separate directories under $(GISBASE)/etc/bin and creating a special program in $(BIN).

There are now 6 new directories where programs are placed. These, along with their respective Gmakefile variable, are

etc/bin/main/inter $(BIN_MAIN_INTER)
: Interactive versions of the primary GRASS commands.

etc/bin/main/cmd $(BIN_MAIN_CMD)
: Command-line versions of the primary GRASS commands.

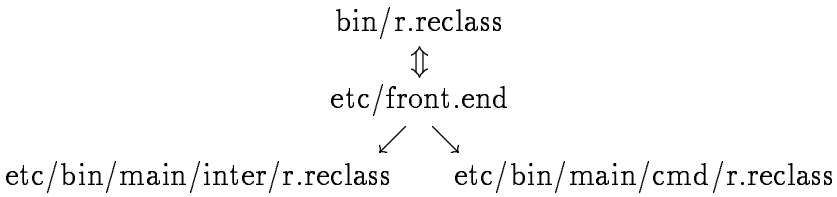
etc/bin/alpha/inter $(BIN_ALPHA_INTER)
: Interactive versions of the alpha-version commands.

etc/bin/alpha/cmd $(BIN_ALPHA_CMD)
: Command-line versions of the alpha-version commands.

etc/bin/contrib/inter $(BIN_CONTRIB_INTER)
: Interactive versions of the contributed commands.

etc/bin/contrib/cmd $(BIN_CONTRIB_CMD)
: Command-line versions of the contributed commands.

The merging is done done by linking program of the same name in $(BIN) to $(GISBASE)/etc/front.end:



The front.end will call the interactive version if the program was run without arguments. Otherwise it will run the command-line version. If only one version exists (for example, there is only a command-line version) then that version is run.

4 GRASS Command Line Parser

This section introduces the GRASS programmer to the use of a standardized command-line parsing capability. All programmers are encouraged to use this parser as it is easy and maintains program look and feel standardization. This results in a family of GRASS programs which are easy for users to learn.

4.1 What and Why

The GRASS parser is a collection of five subroutines which use two structures defined in the GRASS gis.h file. `G_define_option()` and `G_define_flag()` allow a programmer to describe options and flags that make up the command-line options of their program. `G_parser()` generates a complete user interface based on the defined flags and options. If the user provides command-line input, the routine parses the information into the defined categories and then checks to make sure all information is accounted for and that all required options have been addressed. With no command-line arguments, the parser enters a standardized interactive session in which all flags and options are presented to the user for input. `G_usage()` allows a programmer to print the usage message at any time, and `G_disable_interaction()` turns off the default interactive capability

GRASS is a collection of programs that have been generated over many years by many different programmers. As such, the user interface appears haphazard and ahhoc. Each programmer brings different capabilities, understandings, and desires to their work which results in assorted programming styles and user interfaces. With this standardized parsing the programmer is freed from the burden of generating user interface code. Of course the parser limits the programmer to a defined, yet prescribed, look and feel. The small cost in limiting the interface is well worth the shortened user learning curve.

4.2 Details

The use of the parser are demonstrated here with an example. There are four basic steps: 1) Allocate flags and options, 2) Define flags and options, 3) Call the parser, and 4) capture results of the parsing process into variables for use by your program.

Step 1 - Allocate flags and options:
Flags and options are pointers to structures allocated through the parser routines: `G_define_option()` and `G_define_flag()`. Once allocated, the members of the structures can be assigned as desired in step 2.

```
#include "gis.h" ;                  The standard GRASS include file

struct Option *opt ;                Declare an Option pointer for each option
struct Flag *flag ;                 Declare a Flag pointer for each flag

opt = G_define_option() ;           Request pointer to memory for each option
flag = G_define_flag() ;            Request pointer to memory for each flag
```

Step 2 - Define flags and options:

```
opt->key = "option" ;               Key word that user will use
opt->description = "Option test" ;  Description that user sees
opt->type = TYPE_STRING ;           Type of answer
opt->required = YES ;               Must the user give input?

flag->key = 'k' ;                   Single letter used for flag
flag->description = "Flag test" ;   Description that user sees
```

Note: there are other elements in these structures which are described in § 4.3.

Step 3 - Call the parser:

```
if (G_parser(argc, argv) )          Returns 0 if successful
        exit(1) ;
```

Step 4 - Use results:

```
printf("You chose <%s> \n", opt->answer) ;
printf("The flag is %sset\n", flag->answer ? "" : "not ") ;
```

Running the program:
Running such a program would result in the following standard user interface scenarios.

a.out help

This is the standard user call for basic help information on the program. Note how the programmer provided information is captured in the output.

```
Usage:
 a.out [-k] option=name

Flags:
  -k   Flag test

Parameters:
  option   Option test
```

a.out -k

This command line does not contain the required option. Note that the output provides this information along with the standard usage message (as already shown above) .

```
Required parameter <option> not set (Option test).

Usage:
 a.out [-k] option=name

Flags:
  -k   Flag test

Parameters:
  option   Option test
```

a.out option=Hello -k

a.out o=Hello -k

a.out -k option=Hello

These command lines are correct and equivalent. The parser provides no error messages and the program completes normally.

```
You chose <Hello>
The flag is set
```

a.out

Without arguments the parser goes into an interactive mode. In the example below, the user enters `Hello` and a few carriage returns. The parser collects information in a standard if not elegant fashion. The program then completes normally.

```
OPTION:   Option test
     key: option
required: YES
enter option > Hello

You have chosen:
  option=Hello
Is this correct? (y/n) [y] <CR>

FLAG: Set the following flag?
    Flag test?(y/n) [n] <CR>

You chose <Hello>
The flag is not set
```

4.3 Option and Flag Structures

There are more variables associated with the option and flag structures. They are detailed here. Note: the values YES, NO, TYPE_STRING, TYPE_INTEGER, and TYPE_DOUBLE found in the following table are #defined in the "gis.h" header file.

struct Flag				
structure member	C type	required	default	description
key	char	YES	none	Key char used on command line. flag->key = 'f' ;
description	char *	YES	none	String describing flag meaning. flag->description = "run in fast mode" ;
answer	char	NO	NULL	Stores flag state: 0/1
struct Option				
structure member	C type	required	default	description
key	char *	YES	none	Key word used on command line. opt->option = "color" ;
type	int	YES	none	Option type: TYPE_STRING TYPE_INTEGER TYPE_DOUBLE opt->type = TYPE_STRING ;
description	char *	YES	none	String describing option. opt->description = "color to use" ;
answer	char *	NO	NULL	Default answer. opt->answer = "white" ;
required	int	NO	NO	Indication if user MUST provide input. opt->required = YES ;
multiple	int	NO	NO	Indication if user can provide multiple inputs. opt->multiple = NO ;
options	char *	NO	NULL	Approved values or range opt->options = "red,white,blue" ; For integers and doubles, the following format is available: opt->options = "0-1000" ;
key_desc	char *	NO	NULL	Single word describing the key. Commas in this string denote to the parser that several comma separated arguments are expected from the user. For example, if a coordinate is desired, this element might be defined as follows: opt->key_desc = "x,y" ;
gisprompt	char *	NO	NULL	Interactive prompt guidance. There are three comma-separated parts to this argument which guide the use of the standard GRASS file name prompting routines. Read below for details. The following example is NOT appropriate for our color prompt: opt->gisprompt = "old,cell,raster" ;
checker	char *()	NO	NULL	Routine to check answer opt->checker = my_routine() ;

The gisprompt Option structure item requires a bit more description. The three comma-separated (no spaces allowed) sub-arguments are defined as follows:

First argument : "old" results in a call to the GRASS library subroutine `G_ask_old()`, "new" to `G_ask_new()`, "any" to `G_ask_any()`, and "mapset" to `G_ask_in_mapset()`.

Second argument : This is identical to the "element" argument in the above subroutine calls. It specifies a directory inside the mapset that may contain the user's response.

Third argument : Identical to the "element description" argument in the subroutines. This is a string presented to the user that describes the type of data element being requested.

Here are two examples:

"new,cell,raster"	Call to G_ask_new("", name, "cell", "raster")
"old,dig,vector"	Call to G_ask_old("", name, "dig", "vector")

4.4 Questions and Answers

How is automatic prompting turned off? - There are two methods. GRASS 4.0 introduces a new method for driving GRASS interactive and non-interactive programs. For most programs a user runs a front-end program out of the GRASS bin directory which in turn looks for the existence of standard, alpha, and contributed interactive and non-interactive versions of the program. If an interactive version exists and the user provided no command-line arguments, then that version is run. In such a situation, the parser's default interaction will never be seen by the user. Secondly, a programmer using the parser is able to avoid the default interaction by placing a call to `G_disable_interactive()` before calling `G_parser()`.

Can the user mix options and flags? - Yes. Options and flags can be given in any order.

In what order does the parser present options and flags to the user? - Flags and options are presented in the order that the programmer defines them using calls to `G_define_option()` and `G_define_flag()`.

How does a programmer query for coordinates? - For any user input that requires a set of arguments (like a map coordinate), the programmer specifies the number of arguments in the key_desc structure element. For example if opt->key_desc="x,y" the parser will require that the user enter a pair of arguments separated only by a comma. See the source code for r.drain or r.cost for examples.

Is a user required to use full option names? - No! Users are required to use only as many characters of an option to make the option choice unambiguous. If, for example, there are two options, *input=* and *output=*, the user may get by with *command i=map1 o=map2*. The first option is often the only required option and is generally the name of a map on which to perform an operation. The option= (often *map=*) can be completely omitted for the first option.

Are options standardized at all? - Yes. There are a few conventions. options which identify a single input map use *map=*, not *raster=* or *vector=*. In the case of an input and output map the convention is *input=xx output=yy*. By running help on the current commands you will likely see other conventions. The desire is to make it as easy as possible for the user to remember (or guess correctly) what the command-line syntax is.

4.5 Sample Code

The following code demonstrates some of the basic capabilities of the parser. It is compiled using the following Gmakefile:

```
sample: sample.o
    $(CC) $(LDFLAGS) -o sample sample.o $(GISLIB)
```

The sample.c code follows. You might experiment with this code to familiarize yourself with the parser.

```
#include "gis.h" ;

main(argc, argv)
    int argc ;
    char **argv ;
{
    struct Option *opt ;
    struct Option *coor ;
    struct Flag *flag ;
    double X, Y ;
    int n ;

    opt = G_define_option() ;
    opt->key          = "option" ;
    opt->description  = "Option test" ;
    opt->type         = TYPE_STRING ;
```

```
    opt->required     = YES ;

    coor = G_define_option() ;
    coor->key         = "coordinate" ;
    coor->key_desc    = "x,y" ;
    coor->description = "One or more coordinates" ;
    coor->type        = TYPE_STRING ;
    coor->required    = YES ;
    coor->multiple    = YES ;

    flag = G_define_flag() ;
    flag->key         = 'k' ;
    flag->description = "Flag test" ;

    if (G_parser(argc, argv))
        exit(-1) ;

    printf("For option you chose: <%s>\n", opt->answer);
    printf("The flag is: %s set\n", flag->answer ? "" : "not") ;

    printf("You specified the following coordinates:\n") ;
    for(n=0; coor->answers[n] != NULL; n+=2)
    {
        G_scan_easting  (coor->answers[n  ], &X, G_projection()) ;
        G_scan_northing (coor->answers[n+1], &Y, G_projection()) ;
        printf("%.3lf,%.2lf\n", X, Y) ;
    }
}
```

5 Raster Colors

The GRASS raster color tables and associated programming interface have undergone a fairly major revision to resolve problems presented by raster maps that have a large range of data values. The previous design used arrays to store a color for each data value between the minimum and maximum values in the raster map. This array structure was also reflected in the format of the color table file – each color stored as a single line in the color file. Because GRASS raster maps can have data values in the range ± 2147483647,[1] this method of storing color information is clearly untenable.

[1] These values are for 32 bit architectures.

The solution to this problem is to represent color tables as linear ramps for intervals of data values. Colors are specified (and stored) for the endpoints of each interval. Colors for values between endpoints are not stored but are computed using a linear interpolation scheme. For example, a single rule that associates black with the minimum data value and white with the maximum data value is all that is required to represent a grey scale.

This method clearly reduces the amount of memory required to hold the color table as well as the size of the color file and the time to read the file. It has a cost of a more complex data structure and a more complex interface to the colors.

The implementation required modifications to the data structure used to hold the color rules in memory, the redefinition of the format of the color file, modifications to the routines that read and write the color file, and the introduction of new routines to provide a programming interface to the new data structure, as well as new routines for the graphic display of raster data.

The GRASS library routines that handle colors are described below, with argument specifications and a short description. Coding examples are presented in another section.[2]

5.1 Reading and Writing Color Files.

These two routines are used to read and write color files.

```
G_read_colors (name, mapset, colors)                    [old]
    char *name;
    char *mapset;
    struct Colors *colors;
```

Reads the color table for a raster map. Returns 1 if successful, 0 if color file missing but default colors generated, or -1 if there was an error reading the color file.

```
G_write_colors (name, mapset, colors)                   [old]
    char *name;
    char *mapset;
    struct Colors *colors;
```

[2] A description of the data structure and the new color file format are beyond the scope of this paper. In part, this is due to the desire to encourage programmers to use the library interface and to discourage them from referencing elements in the color structure.

Writes the color table for a raster map. Returns 1 if successful, or -1 if there was an error writing the color file.

5.2 Setting Colors.

These routines set colors in the color structure.

```
G_init_colors (colors)                                          [old]
    struct Colors *colors;
```

Initializes the color structure. This routine must be called when the color structure is constructed from scratch. It should not be called after `G_read_colors()` because it will undo the effect of reading the color file.

```
G_add_color_rule (data1, r1,g1,b1, data2, r2,g2,b2, colors)     [new]
    CELL data1,data2;
    int r1,g1,b1;
    int r2,g2,b2;
    struct Colors *colors;
```

This is the heart and soul of the new color logic. It adds a color rule to the color structure. The colors defined by the red, green, and blue values `r1,g1,b1` and `r2,g2,b2` are assigned to `data1` and `data2` respectively. Colors for data values between `data1` and `data2` are not stored in the structure, but will be interpolated when queried. The color components `r1,g1,b1` and `r2,g2,b2` must be in the range 0-255.

```
G_set_color (data, r,g,b, colors)                               [old]
    CELL data;
    int r,g,b;
    struct Colors *colors;
```

Sets the color for a single `data` value to `r,g,b`. It is provided for backwards compatibility. *It's use is discouraged because it defeats the purpose of the new logic.*

```
G_free_colors (colors)                                          [old]
    struct Colors *colors;
```

Frees the allocated memory in the colors structure.

5.3 Looking up Colors.

Three routines are provided for extracting color information from the color structure.

```
G_lookup_colors (data, r, g, b, set, n, colors)                    [new]
    CELL *data;
    unsigned char *r, *g, *b, *set;
    int n;
    struct Colors *colors;
```

Translates an array of data values to their respective colors. The colors for the `n` values in the `data` array are stored in the `r,g,b` arrays, with the values in the `set` array indicating if the corresponding data value has a color or not (1 means it does, 0 means it doesn't). The programmer must allocate the `r`, `g`, `b`, and `set` arrays to be at least `n` in length.

```
G_get_color (data, r, g, b, colors)                                [old]
    CELL data;
    int *r, *g, *b;
    struct Colors *colors;
```

Gets the `r,g,b` components of the color associated with a single `data` value.

```
G_get_color_range (min, max, colors)                               [new]
    CELL *min, *max;
    struct Colors *colors;
```

Gets the minimum and maximum data values that have colors associated with them.

5.4 Creating Standard Color Tables.

These routines create entire color tables. They require the minimum and maximum data values as input. Obtaining these values for existing raster maps is illustrated in the examples in § 5.7

```
G_make_aspect_colors (colors, min, max)                    [old]
    struct Colors *colors ;
    CELL min,max;
```

Makes a color table for aspect maps.

```
G_make_random_colors (colors, min, max)                    [old]
    struct Colors *colors ;
    CELL min,max;
```

Makes a random color table.

```
G_make_rainbow_colors (colors, min, max)                   [old]
    struct Colors *colors;
    CELL min, max;
```

Makes a shifted rainbow color table – yellow to green to cyan to blue to magenta to red.

```
G_make_ramp_colors (colors, min, max)                      [renamed]
    struct Colors *colors ;
    CELL min,max;
```

Makes a color table with three parts: black to red; black to green; and black to blue.

```
G_make_gyr_colors (colors, min, max)                       [new]
    struct Colors *colors ;
    CELL min,max;
```

Makes a color table that goes from green to yellow to red.

```
G_make_ryg_colors (colors, min, max)                                    [new]
    struct Colors *colors ;
    CELL min,max;
```

Makes a color table that goes from red to yellow to green.

```
G_make_wave_colors (colors, min, max)                                   [renamed]
    struct Colors *colors ;
    CELL min,max;
```

Makes a color table with three overlapping parts: black to red to black; black to green to black; and black to blue to black.

```
G_make_grey_scale_colors (colors, min, max)                             [renamed]
    struct Colors *colors ;
    CELL min,max;
```

Makes a linear grey scale.

```
G_make_histogram_eq_colors (colors, s)
    struct Colors *colors;
    struct Cell_stats *s;
```

Creates a histogram contrast-stretched grey scale color table from the histogram information in the `Cell_stats` structure `s`. (See §8.5).

5.5 Basic Raster Graphics

GRASS graphics allow two modes for raster color display: fixed and float. Fixed mode loads a predefined color table and translates each raster color to the nearest loaded color. Float mode loads each color directly into the hardware color registers.

These next routines support plotting of GRASS raster maps. They make every effort to reduce the amount of memory required. They also properly handle large color tables when the display is set to float colormode.

Note: this is not the full set of graphics routines. Only those related to the new color logic are described. The issue of placement and scaling of the raster graphics is not discussed.

```
D_set_colors (colors)                                          [new]
    struct Colors *colors;
```

Sets the graphics colors from the color structure. This routine must be called prior to plotting raster data. This routine replaces `D_reset_colors()`, as well as the use of `R_reset_colors()`.

Returns 1 if the colors will fit into the hardware color map, 0 if they do not (in which case a fixed color table translation will be applied). These return codes do not indicate an error. They are informational.

```
D_draw_cell(row, data, colors)                                 [renamed]
    int row ;
    CELL *data ;
    struct Colors *colors;
```

This routine replaces both `D_draw_cell_row` and `D_overlay_cell_row`. It must be passed the same `colors` structure as passed to `D_set_colors`. This routine is used with `D_cell_draw_setup()` that computes resampling and placement of the raster plot. See example in § 5.7

```
D_raster (data, n, repeat, colors)                             [new]
    CELL *data;
    int n, repeat;
    struct Colors *colors;
```

Plots raster data. The `data` array has `n` values. This raster is plotted `repeat` times, one below the other. The `colors` structure must be the same one that was passed to `D_set_colors()`. This routine does not perform any resampling or placement. `D_draw_cell()` which performs resampling and placement used this routine to do the plotting.

Note: the `data` values are changed to their corresponding hardware color number.

```
D_set_overlay_mode (flag)                                    [new]
    int flag;
```

Sets the overlay mode to false if `flag` is zero, true otherwise. The default mode is false. This effects how `D_raster()` behaves. If overlay mode is on, then zero values are not plotted. Otherwise zero values are plotted in whatever color has been assigned to zero.

5.6 Finer Control of Raster Graphics

These next routines provide some additional control of raster graphics.

```
D_reset_color (data, r, g, b)                                [new]
    CELL data;
    int r,g,b;
```

Modifies the hardware colormap provided that the graphics are not using fixed mode colors. The hardware color register corresponding to the raster `data` value is set to `r,g,b`. This routine may only be called after calling `D_set_colors()`. For use by programs such as d.colors.

Returns 1 if the hardware colormap was updated, 0 if not (either because a fixed color table translation is in effect or because the `data` is not in the color range set by the call to `D_set_colors()`).

```
D_check_colormap_size (min, max, ncolors)                    [new]
    CELL min, max;
    int *ncolors;
```

Determines if the range of colors fits into the hardware colormap. If it does, then the colors can be loaded directly into the hardware colormap and color toggling will be possible. Otherwise a fixed lookup scheme must be used and color toggling will not be possible.

If the colors will fit `ncolors` is set to the required number of colors (computed as max-min+2) and 1 is returned. Otherwise `ncolors` is set to the number of hardware colors and 0 is returned.

```
D_lookup_colors (data, n, colors)                                    [new]
    CELL *data;
    int n;
    struct Colors *colors;
```

The `n` `data` values are changed to their corresponding hardware color number. The `colors` structure must be the same one that was passed to `D_set_colors()`.

5.7 Examples

These examples show how to create and use color files for raster maps.

Creating a standard color table

This example creates an aspect color table using one of the standard GRASS color table routines.

```
make_aspect_colors(name,mapset)
    char *name, *mapset;                /* raster map */
{
    struct Colors colors;
    struct Range range;
    CELL min,max;

    G_read_range(name,mapset,&range);   /* get the min,max for the map */
    G_get_range_min_max (&range, &min, &max);

    G_make_aspect_colors (min, max, &colors);
    G_write_colors (name,mapset, &colors);
    G_free_colors (&colors);
}
```

Creating a customized color table

This example creates a customized color table.

```
make_green_to_yellow_to_brown_colors(name,mapset)
    char *name, *mapset;               /* raster map */
{
    struct Colors colors;
    struct Range range;
    CELL min,max,ave;

    G_read_range(name,mapset,&range);   /* get the min,max for the map */
    G_get_range_min_max (&range, &min, &max);

    ave = ((double)min+(double)max)/2;  /* use doubles to avoid int overflow */

/* green    0,255,0
 * yellow 255,255,0
 * brown  192,128,64
 */
    G_init_colors (&colors);
    G_add_color_rule (min,   0,255,0, ave, 255,255, 0, &colors);
    G_add_color_rule (ave, 255,255,0, max, 192,128,64, &colors);
    G_write_colors (name,mapset, &colors);
    G_free_colors (&colors);
}
```

Note: the computation of the average between the min and the max must be done in floating point, not integer. The min and max values may be very large and the sum could overflow an integer.

Plotting a raster map

This next example shows how to plot a raster map.

```
plot_raster_map(name,mapset,overlay)
    char *name, *mapset;
    int overlay;
{
    CELL *data;
```

```
    int row, fd, top, bottom, left, right ;

/* perform all sort of graphics initialization for plotting */
    D_setup (!overlay);                                          [new]
    D_get_screen_window(&top, &bottom, &left, &right) ;          [old]
    if (D_cell_draw_setup(top, bottom, left, right)) {ERROR}     [old]
    data = G_allocate_cell_buf();

/* open the raster map, and read and set the colors */
    if((fd = G_open_cell_old (name, mapset)) < 0) {ERROR}
    if (G_read_colors (name, mapset, &colors) < 0) {ERROR}
    D_set_colors (&colors);                                      [new]

/* plot */
    D_set_overlay_mode (overlay);                                [new]
    for (row = 0; row >= 0;)
    {
        if (G_get_map_row(fd,data,row) < 0) {ERROR}
        row = D_draw_cell(row, data, &colors);                   [new]
    }
    G_close_cell (fd);
    G_free_colors (&colors);
    free (data);
}
```

Note: `D_setup()` is described in § 7.

6 Latitude-Longitude Databases

GRASS now supports databases in a latitude-longitude grid using a projection where the x coordinate is the longitude and the y coordinate is the latitude. This projection is called the Equidistant Cylindrical Projection.[3] It has the property that *where am I* and *row-column* calculations are identical to those in planimetric grids (like UTM[4]). This implies that normal GRASS registration and overlay functions will work without any special considerations or modifications to existing code.

However, the projection is not planimetric. This means that distance and area calculations are no longer Euclidean.

[3] Also known as the Plate Carrée.

[4] Universal Transverse Mercator Projection

Also, since the world is round, maps may not have edges in the east-west direction, especially for global databases. Maps may have the same longitude at both the east and west edges of the display. This feature, called global wraparound, must be accounted for by GRASS programs (particularly vector based functions, like plotting).

This paper describes the GISLIB library routines available to support latitude-longitude databases.

6.1 Coordinates

Latitudes and longitudes are specified in degrees. Northern latitudes range from 0 to 90, and southern latitudes from 0 to -90. Longitudes have no limits since longitudes ± 360 are equivalent.

Coordinates are represented in ASCII using the format `dd:mm:ssN` or `dd:mm:ssS` for latitudes, `ddd:mm:ssE` or `ddd:mm:ssW` for longitudes, and `dd:mm:ss` for grid resolution. For example, 80:30:24N represents a northern latitude of 80 degrees, 30 minutes, and 24 seconds. 120:15W represents a longitude 120 degrees and 15 minutes west of the prime meridian. 30:15 represents a resolution of 30 degrees and 15 minutes.

These next routines convert between the ASCII representation and the machine representation for a coordinate. They work both with latitude-longitude projections and planimetric projections.

Note: in each subroutine, the programmer must specify the projection number. If the projection number is PROJECTION_LL[5], then latitude-longitude ASCII format is invoked. Otherwise standard floating-point to ASCII conversion is used.

```
G_format_easting (east, buf, projection)
    double east;
    char *buf;
    int projection;
```

Converts the double representation of the `east` coordinate to its ASCII representation (in `buf`).

```
G_format_northing (north, buf, projection)
    double north;
    char *buf;
    int projection;
```

[5] defined in "gis.h"

Converts the double representation of the `north` coordinate to its ASCII representation (in `buf`).

```
G_format_resolution (resolution, buf, projection)
    double resolution;
    char *buf;
    int projection;
```

Converts the double representation of the `resolution` to its ASCII representation (in `buf`).

```
G_scan_easting (buf, easting, projection)
    char *buf;
    double *easting;
    int projection;
```

Converts the ASCII representation of the `east` coordinate to its double representation (in `buf`).

```
G_scan_northing (buf, northing, projection)
    char *buf;
    double *northing;
    int projection;
```

Converts the ASCII representation of the `north` coordinate to its double representation (in `buf`).

```
G_scan_resolution (buf, resolution, projection)
    char *buf;
    double *resolution;
    int projection;
```

Converts the ASCII representation of the `resolution` to its double representation (in `buf`).

The following are examples of how these routines are used:

```
double north;
char buf[50];

G_scan_northing (buf, north, G_projection());    /* ASCII to double */
G_format_northing(north, buf, G_projection());   /* double to ASCII */
G_format_northing(north, buf, -1);               /* double to ASCII */
/* this last one forces floating-point ASCII format */
```

6.2 Raster Area Calculations

The following routines perform area calculations for raster maps. They are based on the fact that while the latitude-longitude grid is not planimetric, the size of a grid cell at a given latitude is constant. The first two routines work in any projection.

`G_begin_cell_area_calculations()`

> This routine must be called once before any call to `G_area_of_cell_at_row()`. It can be used in either planimetric projections or the latitude-longitude projection. It returns 2 if the projection is latitude-longitude, 1 if the projection is planimetric, and 0 if the projection doesn't have a metric (e.g. imagery). If the return value is 1 or 0, all the grid cells in the map have the same area. Otherwise the area of a grid cell varies with the row.

```
double
G_area_of_cell_at_row (row)
    int row;
```

> This routine returns the area in square meters of a cell in the specified `row`. This value is constant for planimetric grids and varies with the row if the projection is latitude-longitude.

These next four routines are for latitude-longitude only. Two are for the ellipsoidal form and two are for the spherical form of the Earth. They calculate the area within a latitude zone (the region between two latitudes).

```
G_begin_zone_area_on_ellipsoid (a, e2, s)
    double a, e2, s;
```

Initializes raster area calculations for an ellipsoid where `a` is the semi-major axis of the ellipse (in meters), `e2` is the ellipsoid eccentricity squared, and `s` is a scale factor to allow for calculations of part of the zone (`s=1.0` is full zone, `s=0.5` is half the zone, `s=360/ew_res` is for a single grid cell).

Note: `e2` must be positive. A negative value makes no sense, and zero implies a sphere.

```
double
G_area_for_zone_on_ellipsoid (north, south)
    double north, south;
```

Returns the area between latitudes `north` and `south` scaled by the factor `s` passed to the call to `G_begin_zone_area_on_ellipsoid()`.

```
G_begin_zone_area_on_sphere (r, s)
    double r, s;
```

Initializes raster area calculations for a sphere. The radius of the sphere is `r` and `s` is a scale factor to allow for calculations of part of the zone (see `G_begin_zone_area_on_ellipsoid()` above).

```
double
G_area_for_zone_on_sphere (north, south)
    double north, south;
```

Returns the area between latitudes `north` and `south` scaled by the factor `s` passed to the call to `G_begin_zone_area_on_sphere()`.

6.3 Polygonal Area Calculations

These next routines provide area calculations for polygons. Some of the routines are specifically for latitude-longitude while others work in all projections.

However, there is an issue for latitude-longitude that does not occur with planimetric grids. Vector/polygon data is described as a series of x,y coordinates. The lines connecting the points are not stored but are inferred. This is a simple straightforward process for planimetric grids, but it is not simple for latitude-longitude. What is the shape of the line that connects two points on the surface of a globe?

One choice (among many) is the shortest path from x1,y1 to x2,y2 known as the geodesic. Another is a straight line on the grid. The area routines described below assume the latter. Routines to work with the former have not yet been developed.

```
G_begin_polygon_area_calculations()
```

This initializes the polygon area calculation routines. It is used both for planimetric and latitude-longitude projections.

It returns 2 if the projection is latitude-longitude, 1 if the projection is planimetric, and 0 if the projection doesn't have a metric (e.g. imagery).

```
double
G_area_of_polygon(x,y,n)
    double *x, *y;
    int n;
```

Returns the area in square meters of the polygon described by the n pairs of x,y vertices. It is used both for planimetric and latitude-longitude projections.

Note: if the database is planimetric with a non-meter grid, this routine performs the required unit conversion to produce square meters.

```
double
G_planimetric_polygon_area(x,y,n)
    double *x, *y;
    int n;
```

Returns the area in coordinate units of the polygon described by the `n` pairs of `x,y` vertices for planimetric grids. If the units for `x,y` are meters, then the area is in square meters. If the units are feet, then the area is in square feet, etc.

```
G_begin_ellipsoid_polygon_area (a, e2)
    double a, e2;
```

This initializes the polygon area calculations for the ellipsoid with semi-major axis `a` (in meters) and eccentricity squared `e2`.

```
double
G_ellipsoid_polygon_area (lon, lat, n)
    double *lon, *lat;
    int n;
```

Returns the area in square meters of the polygon described by the `n` pairs of `lon,lat` vertices for latitude-longitude grids.

Note: this routine assumes grid lines connecting the vertices (as opposed to geodesics).

6.4 Distance Calculations

Two routines perform distance calculations for any projection.

```
G_begin_distance_calculations()
```

Initializes the distance calculations. It is used both for planimetric and latitude-longitude projections.

It returns 2 if the projection is latitude-longitude, 1 if the projection is planimetric, and 0 if the projection doesn't have a metric (e.g. imagery).

```
double
G_distance (x1,y1,x2,y2)
    double x1,y1,x2,y2;
```

Computes the distance, in meters, from `x1,y1` to `x2,y2`. If the projection is latitude-longitude, this distance is measured along the geodesic.

Two routines perform geodesic distance calculations.

```
G_begin_geodesic_distance (a, e2)
    double a, e2;
```

Initializes the distance calculations for the ellipsoid with semi-major axis `a` (in meters) and eccentricity squared `e2`. It is used only for the latitude-longitude projection.

```
double
G_geodesic_distance (lon1, lat1, lon2, lat2)
    double lon1, lat1, lon2, lat2;
```

Calculates geodesic distance from `lon1,lat1` to `lon2,lat2` in meters.

The calculation of the geodesic distance is fairly costly. These next three routines provide a mechanism for calculating distance with two fixed latitudes and varying longitude separation.

```
G_set_geodesic_distance_lat1(lat1)
    double lat1;
```

Set the first latitude.

```
G_set_geodesic_distance_lat2(lat2)
    double lat2;
```

Set the second latitude.

```
double
G_geodesic_distance_lon_to_lon (lon1, lon2)
    double lon1, lon2;
```

Calculates geodesic distance from `lon1,lat1` to `lon2,lat2` in meters, where `lat1` was the latitude passed to `G_set_geodesic_distance_lat1()` and `lat2` was the latitude passed to `G_set_geodesic_distance_lat2()`

6.5 Global Wraparound

These next routines provide a mechanism for determining the relative position of a pair of longitudes. Since longitudes ± 360 are equivalent, but GRASS requires the east to be bigger than the west, some adjustment of coordinates is necessary.

```
double
G_adjust_easting (east, region)
    double east;
    struct Cell_head *region;
```

If the region projection is PROJECTION_LL, then this routine returns an equivalent `east` that is bigger, but no more than 360 degrees bigger, than the coordinate for the western edge of the `region`, Otherwise no adjustment is done and the original `east` is returned.

```
double
G_adjust_east_longitude (east, west)
    double east, west;
```

This routine returns an equivalent `east` that is bigger, but no more than 360 degrees bigger, the `west` coordinate.

This routine should be used only with latitude-longitude coordinates.

```
G_shortest_way (east1, east2)
    double *east1, *east2;
```

If the database projection is PROJECTION_LL, then `east1,east2` are changed so that they are no more than 180 degrees apart. Their true locations are not changed.

If the database projection is not PROJECTION_LL, then `east1,east2` are not changed.

6.6 Miscellaneous

```
char *
G_ellipsoid_name (n)
    int n;
```

This routine returns the name for the nth ellipsoid in the GRASS ellipsoid table, NULL when n is too large. It can be used as follows:

```
int n;
char *name;

for (n=0; name = G_ellipsoid_name(n); n++)
    printf ("%s\n", name);
```

```
G_get_ellipsoid_by_name (name, a, e2)
    char *name;
    double *a, *e2;
```

This routine returns the semi-major axis a (in meters) and eccentricity squared e2 for the named ellipsoid.

```
G_get_ellipsoid_parameters (a, e2)
    double *a, *e2;
```

This routine returns the semi-major axis a (in meters) and eccentricity squared e2 for the ellipsoid associated with the database.

```
double
G_meridional_radius_of_curvature (lon, a, e2)
    double lon, a, e2;
```

Returns the meridional radius of curvature at a given longitude:

$$\rho = \frac{a(1-e^2)}{(1-e^2 sin^2 lon)^{\frac{3}{2}}}$$

```
double
G_transverse_radius_of_curvature (lon, a, e2)
    double lon, a, e2;
```

Returns the transverse radius of curvature at a given longitude:

$$\nu = \frac{a}{(1 - e^2 sin^2 lon)^{\frac{1}{2}}}$$

```
double
G_radius_of_conformal_tangent_sphere (lon, a, e2)
    double lon, a, e2;
```

Returns the radius of the conformal sphere tangent to ellipsoid at a given longitude:

$$r = \frac{a(1 - e^2)^{\frac{1}{2}}}{(1 - e^2 sin^2 lon)}$$

```
G_pole_in_polygon(x, y, n)
    double *x, *y;
    int n;
```

For latitude-longitude coordinates, this routine determines if the polygon defined by the `n` vertices `x,y` contains one of the poles

Returns -1 if it contains the south pole, 1 if it contains the north pole, 0 if no pole.

Note: use this routine only if the projection is PROJECTION_LL.

7 Vector Graphics/Plotting Routines

This paper describes some new routines that have been added to the GISLIB and the DISPLAYLIB to support plotting of vector data. The best source for an example of how they are used is the GRASS *d.vect* module.

7.1 DISPLAYLIB routines

```
D_setup (clear)
    int clear;
```

Performs a full setup for the current graphics frame: (1) Makes sure there is a current graphics frame (will create a full-screen one, if not); (2) Sets the region coordinates so that the graphics frame and the active program region agree (may change active program region to do this); and (3) Performs graphic frame/region coordinate conversion initialization.

If `clear` is true, the frame is cleared (same as running *d.erase*). Otherwise it is not cleared.

```
D_set_clip_window (top, bottom, left, right)
    int top, bottom, left, right;
```

Sets the clipping window to the pixel coordinate window defined by `top,bottom,left,right`. The routines `D_cont_abs()` and `D_cont_rel()` will not draw outside this clipping window.

```
D_set_clip_window_to_map_window()
```

Sets the clipping window to the pixel window that corresponds to the current database region. This is the default.

```
D_cont_abs(x,y)
    int x,y;
```

Draws a line from the current position to pixel location `x,y`. Any part of the line that falls outside the clipping window is not drawn. Note: the new position is `x,y`, even if it falls outside the clipping window.

Returns 0 if the line was contained entirely in the clipping window, 1 if the line had to be clipped to draw it.

```
D_cont_rel(x,y)
    int x,y;
```

Equivalent to `D_cont_abs(curx+x,cury+y)` where `curx,cury` is the current pixel location.

```
D_move_abs(x,y)
    int x,y;
```

Move without drawing to pixel location `x,y`, even if it falls outside the clipping window.

```
D_move_rel(x,y)
    int x,y;
```

Equivalent to `D_moved_abs(curx+x,cury+y)` where `curx,cury` is the current pixel location.

7.2 GISLIB routines

The following routines are a general purpose line and polygon plotting capability.

```
G_setup_plot (t, b, l, r, Move, Cont)
    double t, b, l, r;
    int (*Move)();
    int (*Cont)();
```

Initializes the plotting capability. This routine must be called once before calling the `G_plot_*()` routines described below.

The parameters `t,b,l,r` are the top, bottom, left, right of the output x,y coordinate space. They are not integers, but doubles, to allow for subpixel registration of the input and output coordinate spaces. The input space is assumed to be the current GRASS region.

`Move,Cont` are subroutines that will draw lines in x,y space. They will be called as follows:

Move(x,y) move to x,y (no draw)

Cont(x,y) draw from previous position to x,y. Cont() is responsible for clipping.

```
G_plot_line (east1, north1, east2, north2)
    double east1, north1, east2, north2;
```

Line from `east1,north1` to `east2,north2` is plotted in output x,y coordinates (e.g. pixels for graphics). This routine handles global wrap-around for latitude-longitude databases.

```
G_plot_polygon (east,north,n)
    double *east, *north;
    int n;
```

The polygon described by the `n` vertices `east,north` is plotted, in the output x,y space, as a filled polygon.

```
G_plot_where_en (x, y, east, north)
    int x, y;
    double *east, *north;
```

The pixel coordinates `x,y` are converted to map coordinates `east,north`.

```
G_plot_where_xy (east, north, x, y)
    double east, north;
    int *x, *y;
```

The map coordinates `east,north` are converted to pixel coordinates `x,y`.

```
G_plot_fx (f, east1, east2)
    double (*f)();
    double east1, east2;
```

The function `f(east)` is plotted from `east1` to `east2`. The function `f(east)` must return the map northing coordinate associated with `east`.

8 Miscellaneous Routines

This section describes an assortment of new routines.

8.1 Forming layer names

```
char *
G_fully_qualified_name (name, mapset)
    char *name;
    char *mapset;
```

Returns a fully-qualified name. A fully-qualified name is a string that contains both the `name` and the `mapset`. Currently, this string is `name@mapset`, but the programmer should pretend not to know this and always call this routine to get the fully-qualified name.

This routine is used by interactive programs that pass names to command-line programs. For example, and interactive version of *d.rast* might look like this:

```
#include "gis.h"
main(argc,argv) char *argv[];
{
    char name[100], *mapset;
    char command[1024];

    G_gisinit(argv[0]);       /* initialize GRASS */
    mapset = G_ask_cell_old("Enter raster map to be plotted",name,mapset);
    if (mapset != NULL)
    {
sprintf (command, "d.rast map='%s'", G_fully_qualified_name(name,mapset));
system(command);
    }
}
```

Note: `G_fully_qualified_name()` returns a pointer to static memory that is overwritten each call.

8.2 Cell header/window routines

These routines work with the GRASS cell header structure. This structure is used to hold both the header file for raster maps as well as the region[6] information.

[6]This used to be called the database *window* which GRASS now calls the *region*. However, the names of the library routines still use (and will continue to use) the term window.

```
char *
G_adjust_Cell_head (cellhd, have_rows, have_cols)
    struct Cell_head *cellhd;
    int have_rows, have_cols;
```

This function fills in missing parts of the input cell header. It also makes projection-specific adjustments. The `cellhd` structure must have its *north, south, east, west,* and *proj* set. If `have_rows` is true, then the north-south resolution is computed from the number of rows in the header. Otherwise the number of rows is computed from the north-south resolution in the header. Similarly for the columns and the east-west resolution.

This routine returns NULL if ok, otherwise it returns an error message.

```
char *
G_align_window (window, ref)
    struct Cell_head *window, *ref;
```

Modifies `window` to have the same resolution as `ref` and aligns the `window` edges (north, south, east, west) to align with the `ref` grid.

The `window` is enlarged if necessary – the north is rounded "up", the south "down", the east "right" and the west "left."

This routine returns NULL if ok, otherwise it returns an error message.

```
double
G_col_to_easting (column, window)
    double column;
    struct Cell_head *window;
```

Converts a `column` relative to the `window` to an easting.

Note: `column` is a double: column+0.5 will give the coordinate for the center of the column; column+0.0 will give the coordinate for the western edge of column; and column+1.0 will give the coordinate for the eastern edge of column.

```
double
G_row_to_northing (row, window)
    double row;
    struct Cell_head *window;
```

Converts a `row` relative to the `window` to a northing.

Note: `row` is a double: row+0.5 will give the coordinate for the center of the row; row+0.0 will give the coordinate for the northern edge of row; and row+1.0 will give the coordinate for the southern edge of row.

```
double
G_easting_to_col (east, window)
    double east;
    struct Cell_head *window;
```

Converts an easting to a column position relative to the `window`.

Note: the result is a double. Casting it to an integer will give the column number.

```
double
G_northing_to_row (north, window)
    double north;
    struct Cell_head *window;
```

Converts an northing to a row position relative to the `window`.

Note: the result is a double. Casting it to an integer will give the row number.

8.3 Database projection and units

```
char *
G_database_projection_name()
```

Returns a string describing the database projection.

```
char *
G_database_unit_name(plural)
    int plural;
```

Returns a string describing the database grid units. It returns a plural form (e.g. feet) if `plural` is true. Otherwise it returns a singular form (e.g. foot).

```
double
G_database_units_to_meters_factor()
```

Returns a factor which convert the grid unit to meters. If the database is not a metric database (like imagery), then 0.0 is returned.

8.4 Bresenham line algorithm

```
G_bresenham_line (x1,y1,x2,y2,point)
    int x1, y1 ;
    int x2, y2 ;
    int (*point)();
```

Draws a line from `x1,y1` to `x2,y2` using Bresenham's algorithm. A routine to plot points must be provided, and is defined as

point(x,y) plot a point at x,y.

8.5 Cell stats

This next set of routines provides a relatively efficient mechanism for producing a histogram of raster data. They use the `Cell_stats` structure to hold the histogram information.

```
G_init_cell_stats(s)
    struct Cell_stats *s;
```

Initialize the Cell_stats structure `s`.

```
G_update_cell_stats (data, n, s)
    CELL *data;
    int n;
    struct Cell_stats *s;
```

Update `s` from the `n` values in the `data` array.

```
G_find_cell_stat (cat, count, s)
    CELL cat;
    long *count;
    struct Cell_stats *s;
```

Find the count associated with the cat value. Returns 1 if found, 0 otherwise.

```
G_rewind_cell_stats (s)
    struct Cell_stats *s;
```

This routine rewinds the structure so that sequential access can begin.

```
G_next_cell_stat (cat, count, s)
    CELL *cat;
    long *count;
    struct Cell_stats *s;
```

Retrieves the next cat and its count from the structure. Returns 0 if there were no more items. Be sure to call `G_rewind_cell_stats()` once before calling this routine:

```
G_rewind_cell_stats(&s);
while (G_next_cell_stat(&cat,&count,&s))
{ /* do something }
```

```
G_free_cell_stats (s)
    struct Cell_stats *s;
```

Frees the memory allocated to the structure by `G_update_cell_stats()`.

9 Deleted Routines

The following routines have been deleted from the GRASS libraries:

`G_parse_command()`
`G_parse_command_usage()`
`G_set_parse_command_usage()`
`D_usage()`
`D_parse_command()`

Replaced by `G_parser()`.

`D_reset_colors()`

Replaced by `D_set_colors()` and `D_reset_color()`.

`D_draw_cell_row()`
`D_overlay_cell_row()`

Replaced by `D_draw_cell()` and `D_set_overlay_mode()`.

`G_make_histo_grey_scale()`

Replaced by `G_make_histogram_eq_colors()`.