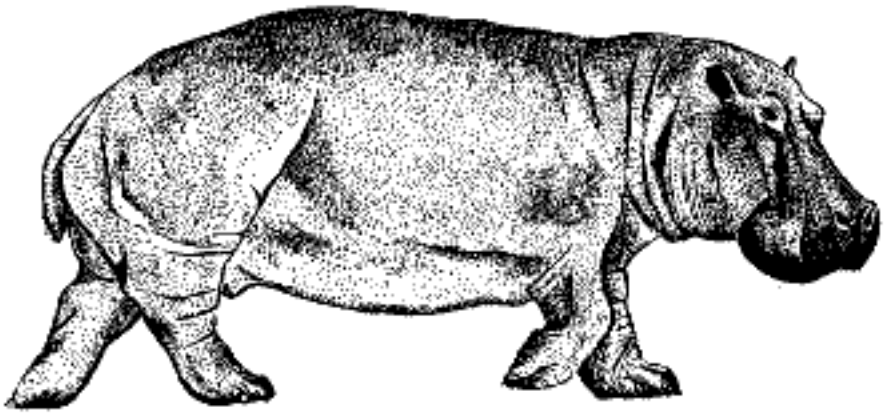
# Hippoplotamus

## Users Guide and Reference Manual (Release 1.1)

**Mike F. Gravina,**
**Paul F. Kunz**
**Paul Rensing**

**Stanford Linear Accelerator Center**
**Stanford University**
**Stanford CA 94309**



## Disclaimer Notice

The items furnished herewith were developed under the sponsorship of the U.S. Government. Neither the U.S., nor the U.S. D.O.E., nor the Leland Stanford Junior University, nor their employees, makes any warranty, express or implied, or assumes any liability or responsibility for accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use will not infringe privately-owned rights. Mention of any product, its manufacturer, or suppliers shall not, nor is it intended to, imply approval, disapproval, or fitness for any particular use. The U.S. and the University at all times retain the right to use and disseminate the furnished items for any purpose whatsoever. Notice 91 02 01

Copyright 1992
by
The Board of Trustees of the
Leland Stanford Junior University.
All rights reserved.

Changes in this document from the previous release (1.0) are marked with a vertical bar in the left hand column such as this paragraph is marked.

This document was produced on a NeXTstation computer using FrameMaker 3.0.1 workstation publishing software. Only Adobe typefaces Courier, Helvetica, and Times were used to allow printing on most PostScript printers.

Work supported by the U.S. Department of Energy under contract DE-AC03-76SF00515.

# Table of Contents

1. Introduction 1

2. N-tuple Creation 3
   - 2.1 N-tuple creation with C, C++, or Objective-C programs 3
   - 2.2 N-tuple creation with FORTRAN program 6
   - 2.3 Plain text to binary conversion 7
   - 2.4 HBOOK4 to Hippo conversion. 8

3. Displays 9
   - 3.1 Code example 9
   - 3.2 Display creation 10
   - 3.3 Bindings to the n-tuple 12
   - 3.4 Axis and bin attributes 12
   - 3.5 Input and Output 13
   - 3.6 Titles and Labels 14
   - 3.7 Plot Drivers 14
   - 3.8 Cuts and Plot Functions 15

4. Reference Manual 17
   - 4.1 N-tuple functions 17
   - 4.2 Display functions 20
   - 4.3 Display axis. 25
   - 4.4 Display Actions 28
   - 4.5 Auxiliary functions 31
   - 4.6 Input and Output functions 33
   - 4.7 Cut and Plot Functions 34
   - 4.8 FORTRAN Binding. 37

5. Installation 39
   - 5.1 Requirements 39
   - 5.2 UNIX Makefile 40
   - 5.3 VMS Installation. 43
   - 5.4 Miscellaneous Files 43
   - 5.5 Problems, Changes and Bug Reports 44

6. History 45

7. References 45

Index

# 1. Introduction

The *Hippoplotamus* package, or *hippo* for short, is a library of data display and histogram functions based on n-tuples. A n-tuple is basically a table of floating point numbers with a fixed number of columns and some indeterminate, perhaps large, number of rows. The entries in the n-tuple, e.g. the rows, could be as simple as a set of x-y points. Such a n-tuple would have only two columns or be said to have a dimension of 2. Or it might be four dimensions containing x, y, error on x, and error on y. *Hippo* can also select one or two columns and produce a histogram of 1 or 2 dimensions. In addition, *hippo* can use additional columns to apply cuts on which rows will be used for 1 or 2 dimensional histograms. Finally, *hippo* can over plot arbitrary functions on its displays.

*Hippo* is logically divided into two parts: the n-tuple package and the display package. The philosophy of the *hippo* design that users will create n-tuples from their own programs, and use an interactive application to view them. Thus the n-tuple part of the *hippo* package is designed to make the creation of the n-tuple as easy as possible. The functions in this part of *hippo* were designed for use by an end-user. On the other hand, the display part of the *hippo* package was designed for someone who implements an interactive application to view and manipulate the displays. It is the basic premise of *hippo* that although the user may have a good idea what data to collect into an n-tuple, he has a much poorer idea of the attributes of their display. Thus, users will use an interactive application to change these attributes. Thus *hippo* is designed to postpone fixing the attributes of displays and histograms until they need to be presented on the users terminal or workstation.

In comparison to other display and histograms packages, *hippo* has the following interesting properties…

- It is written in ANSI C and intended to be used in C programs. As such it will make use of features in C such as casting parameters in function calls, use of pointers and structures, dynamic memory allocation, enumerated data types, etc. However, a FORTRAN binding for the n-tuple part of the package is supported.

- Network support is built into *hippo* by storing files to disk in a form compatible with the industrial standard XDR format[1]. Thus n-tuple files can be generated on one computer and transparently used on another. In a future release, one will be able to have a server on one machine providing n-tuple data to n-tuple viewing application on another.

- *Hippo* can be used with one program generating a n-tuple file and another program or application to view them. In this mode, it is kind of a client/server relationship. However, *hippo* can be easily incorporated into any program that generates the n-tuple data where the client and server are within the same program.

This document is organized first as a users guide and then as a reference manual. The first sections deal with creation of n-tuple data and files. It should be all that a typical user needs to know. The next sections describe the display package for the display programs and applications.

# 2. N-tuple Creation

A *hippo* n-tuple can be created in one of two basic methods. The first is to incorporate the *hippo* n-tuple package in a program. Basic functions are provided to initialize, accumulate, and save the n-tuple into a file. Other functions allow one to give the n-tuple a title and to label the columns. Program binding for the C, C++, Objective-C, and FORTRAN languages are supported. The second method is to create a plain text file representing the n-tuple data, title, and column labels and to use the *hippo* text-to-binary conversion utility.

## 2.1 N-tuple creation with C, C++, or Objective-C programs

The basic steps in creation of an n-tuple file are initialization, accumulation and saving the file. Before making the first call to a *hippo* function one should include the *hippo* header file

```
#include hippo.h
```

and declare a variable to be of type `ntuple`.

```
ntuple   my_tuple;
```

This variable is actually a pointer to a structure where the n-tuple data and other information needed by *hippo* will be stored. The variable must be initialized with the function call

```
my_tuple = h_new( ndim );
```

before it is used. The function `h_new` takes one integer parameter: `ndim` which is the number of variables, i.e. the number of columns, per entry in the n-tuple. There is no need for the user to know about the internals of the n-tuple structure. *Hippo* functions are provided to access any of the information in the structure an advanced user may need to have.

To collect data for a simple x-y plot, the dimension of at least 2 could be used, while for an x-y plot with errors on both x and y, a dimension of at least 4 could be used. For eventual generations of a 1D histogram, `ndim` could be as small as 1, while for weighted 1D histogram `ndim` is 2, one variable for the value (`x`) and the other for the weight (`w`). For a typical 2D histogram generation `ndim` is 2, while for a weight-

ed 2D histogram `ndim` is 3 (`x`, `y`, and `w`). However, any collection of n variables may be collected into the n-tuple.

Attributes normally associated with histograms, such as number of bins, low edge of first bin, bin width, etc. are not defined when the n-tuple is initialized. In *hippo* package, they are considered display attributes and thus defining them is deferred until a projection of the n-tuple is displayed.

Note the *hippo* package does not maintain state. `h_new` allocates memory space for the new n-tuple structure and returns it to the caller. It will not remember that it did that. The user or a higher level package will be responsible for maintaining the list of n-tuples in memory.

To accumulate data into an n-tuple, the user can invoke the *hippo* function `h_fill` as shown below. The first parameter is the n-tuple data variable that was

```
irc = h_fill( my_tuple, x, y, ... );
```

previously declared and initialized. The subsequent parameters to the function call are the data to be accumulated into the n-tuple entry. Their number *must* be equal to the dimension of the n-tuple defined by the `h_new` function call and be of type float.

The function `h_fill` returns the integer return code 0 on success or -1 on failure. The only known failure mode is exhaustion of virtual memory space. Thus, the limit of the number of entries of an n-tuple is determined only by virtual memory space available.

Sometimes it is more convenient organize the data to be accumulated as a floating point array. An alternate *hippo* function accepts such an array as its second parameter. The fragment of code show below illustrates its use. The size of the array

```
float x[10];
...
irc = h_arrayFill( mytuple, x );
```

*must* be at least the dimension of the n-tuple.

Once the n-tuple is collected in memory, it can be written to disk with the `h_write` function. It allows for one or more n-tuples to be written to a file in a single call. It also can write any displays attached to the n-tuple that the user may have defined. The prototype for the `h_write` function is shown below. The first argument is

```
int h_write( const char *filename,
             display dlist[],
             ntuple ntlist[] );
```

```
#include "hippo.h"
main()
{
   ntuple ntlist[2];
   float x; int rc;

   ntlist[0] = h_new(1);
   for ( x = 0; x < 1000.; x++ ) {
      rc = h_fill( ntlist[0], x );
   }
   ntlist[1] = NULL;
   h_write("test", NULL, ntlist);
}
```

Figure 1. Basic n-tuple creation steps.

a string with the name of the file. The remaining two arguments are `NULL` terminated arrays of displays and n-tuples respectively. Passing a `NULL` is the same as an empty list thus a simple example of generating an n-tuple without displays might look as shown in Figure 1. In this example, an n-tuple with 1000 entries containing floating point number from 0. to 999. is created and saved to a file named test.

Before writing an n-tuple to a disk file, however, the user may wish to give the n-tuple a title and to label the individual columns. The prototypes for the three *hippo* provided for this purpose are shown below. Each takes as its first parameter the n-tu-

```
int h_setNtTitle( ntuple nt, const char *title );
int h_setNtLabel( ntuple nt, int dim, const char *label );
int h_setAllNtLabels( ntuple nt, const char *label[] );
```

ple data variable. The title of the n-tuple is set by the `h_setNtTitle` function. The `h_setNtLabel` sets the label of the column given in the second parameter, while the `h_setAllNtLabels` function sets the labels of all the columns in one call. These functions can be called any time after the n-tuple has been initialized with the `h_new` function call.

If the user has a n-tuple display program based on *hippo*, then this is all that the user really needs to know about *hippo* function package. The rest of this document is `hippo` reference manual for users that want to handle their own displays, for people who want to write a *hippo* display program, or for users need more sophisticated use of the *hippo* package.

## 2.2 N-tuple creation with FORTRAN program

A limited set of FORTRAN callable routines exist so that n-tuple creation can be done within FORTRAN programs. An example of FORTRAN code using *hippo* is given in Figure 2. For each *hippo* C function that has a FORTRAN binding, the corresponding FORTRAN function has the same name with "`h_`" replaced with "`ip`".

The arguments used by the FORTRAN function are also the same, or have the same meaning as the corresponding C function. However, where the C function argument calls for data of type `ntuple`, an `integer` data type is used for FORTRAN. Thus, `nt = ipnew(ndim)` returns a new n-tuple with `ndim` columns and `irc = ipsetNtTitle(nt, 'A Title')` sets its title. Also, since by default FORTRAN indices start at 1 instead of 0 as in C, the n-tuple column variable, `ndim`, should be equal to 1 for the first column. Since standard FORTRAN doesn't support variable length arguments lists, only the `iparrayFill()` is supported but not `ipfill()`. Finally, trailing blank characters will be removed from any character variables. In the

```
      Real*4 x(10)
      Integer ntlist(4)
      Integer nt
      Integer irc
C
      nt = ipnew(4)! generate the tuple
C
     irc = ipsetNtTitle( nt, 'First Tuple Title' )
C
      irc = ipsetNtLabel( nt, 1, 'First Column' )
      irc = ipsetNtLabel( nt, 2, 'Second Column' )
      irc = ipsetNtLabel( nt, 3, 'Third Column' )
      irc = ipsetNtLabel( nt, 4, 'Fourth Column' )
C
      Do 10 i = 1, 4
         x(i) = i
   10 Continue
C
      Do 20 i = 1,4
         irc = iparrayFill( nt, x )
   20 Continue
C
      ntlist(1) = nt
      ntlist(2) = 0
      irc = ipwrite( 'test.histo', 0, ntlist )
      end
```

Figure 2. Example of Hippo with FORTRAN.

current release, creation of displays with FORTRAN is not supported, so the corresponding argument in the call to `ipwrite()` must be 0. Note also that the n-tuple argument in `ipwrite()` is an array of integers which dimensioned at least one large then the number of n-tuples to be written. The value 0 is used to terminate this array in place of the `NULL` used in the C function.

## 2.3 Plain text to binary conversion

By default, *hippo* generates machine independent binary files when the *hippo* package is incorporated in the user's C or FORTRAN program. An alternate method of generating such files is to first generate a n-tuple in plain text (ASCII) format and use a conversion utility to convert that format to binary. This utility is provided as a line mode command on all platforms supported by *hippo*.

The full details of the conversion program, `text2nt`, is shown below in UNIX syntax. The input and output file names can be specified by switches (`-i` or `-f`, and

```
text2nt [-a hippo_file] [-v] [-n number]
        [-f text_file] [-i text_file]
        [-o hippo_file] [text_file [hippo_file]]
```

`-o`, respectively) or by positional arguments. Thus, the following two commands have the same result…

```
text2nt -i mydata.dat -o mydata.hippo
text2nt mydata.dat mydata.hippo
```

If the input or output file is not specified, then `text2nt` reads from `stdin` or writes to `stdout` respectively. Optionally, conversion can append its output to an existing file. The `-a` switch is used for this purpose and it is followed by the existing filename. When this switch is used, the `-n` may be used to direct the output to a particular n-tuple of the existing file. The number following this switch is used as an index (starting at 0) of the n-tuple to be appended. Finally, the `-v` switch is the verbose flag that causes `ntuple` to print various messages of progress to `stderr`.

The format of the plain text file is very simple. They consist of "lines" of no more than 256 characters that are delimited by the new line character (`'\n'` in C) or a by "`;`". Within a line there can be either numeric fields or string fields. A numeric field starts with a number ("`0-9`", "`-`", or "`+`") are delimited by one or more of "`,`", `<space>`, or `<tab>`. Any letters contained in a numeric fields are ignored except for "`e`" and the remaining digits are read as numbers. Numeric fields in the form "6.626e-34" are allowed. String fields are used for the n-tuple title and column la-

bels. They are delimited by single quote (') or double quote (") characters. An open quote must be matched by an identical close quote (e.g. "matches", 'matches'), but either of the two types can be used in a file.

The title and label fields can be specified anywhere in the file, but the title must be before the labels. The first "line" which does not start with a numeric field and which contains string field (typically, the first line of the file) is used as the title; everything else on that "line" is ignored. The labels come from the second "line" that contains starts with a string field and all labels must be on one "line".

The dimension of the n-tuple is determined from the first line which starts from a numeric field. This line will be scanned until the first field that does not read as a number. The number of valid numeric fields sets the number of columns of the n-tuple. All subsequent lines must have that many (or more) numeric fields; each line corresponding to one row of the n-tuple. Lines which don't contain enough numeric fields, or which don't contain string fields (once the title and labels have been determined) are ignored.

An example of a plain text file readable by the `text2nt` program that contains two rows of three columns is shown below. Note that one can freely mix either style of

```
"This is the title line"
"column0" 'column1' "column2" extra junk ignorned
4.5 32.5 68.3
4.7,29.8, 58.7 extra junk ignored
this line is ignored
-2.18, 66. another line ignroed
4.9, 27.4 55.7 this line used
```

quoted strings and that letters that are not quoted are ignored.

## 2.4 HBOOK4 to Hippo conversion.

The utility program `hb2hippo` will extract n-tuples from file created by the HBOOK4 package[2] and save the n-tuples in *hippo* file format. The program is run as a command with 1 or two arguments as follows…

```
hb2hippo hbook_file [hippo_file]
```

The first argument is the name of the HBOOK4 file. The second argument is optional and is used as the name of the *hippo* file. If not given, then the *hippo* file will be named `out.hippo`. In the current release, any histograms in the HBOOK4 file are ignored.

# 3. Displays

Traditionally, a histogram package is called in three fundamental phases of user code. The first is at initialization when the histogram is defined in terms of its bins, titles and mode of accumulation. The second phase is the data accumulation, where the bin contents are determined and statistics are found. The last phase is display in which the histogram is plotted on some device. At any time after the histogram is defined, the display attributes, but not the bin definition, can be changed.

In this age of interactive computing, there is one drawback to this approach, namely, that one wants the ability to change everything about a histogram, including the bin definition, at any time. This can be looked at in the following way. A one dimensional histogram is a density distribution of one variable. It is not visible until it is projected, somehow, onto two dimensional space, e.g. a piece of paper. Defining a set of bins of finite width and accumulating the number of entries within each bin is the most common way of doing such a projection. Any choice of bins is one display of the histogram. Another choice is another display of the same histogram. There are an infinite number of such possible displays. The traditional 1D histogram is really just one representation of a density distribution in one dimension.

As has already been mentioned in the introduction, *hippo* has been designed with the premise that a user will have access to an interactive application to view his n-tuples. The display component of *hippo* was written with this in mind. The end-user will almost never write code that incorporates this component and the display portion is not designed to be used by an end-user. Rather, the display component is designed for the person who implements an interactive n-tuple viewer. The display functions are mostly low-level functions to change single attributes of a display so that a button or slider in a application can be connected easily to the display attribute.

The rest of this chapter discusses various aspects of the displays. Much of the detail is left for the reference section following this, where all function prototypes are presented.

## 3.1 Code example

Figure 3 is a short example of some display code, which will be helpful to understand the process of creating a display. It is not a complete program. Some steps which may appear to be missing, however, are actually not necessary because of defaults. No error detection is performed.

```
#include "hippo.h"

   display mydisp;
   ntuple mynt;
   float low,high;

   mydisp = h_newDisp(COLORPLOT);     /* create display */
   h_bindNtuple(mydisp, mynt); /* bind display to ntuple */
   h_plot(mydisp);                    /* plot */

   h_bind(mydisp, YAXIS, 3); /* set y-axis to 4-th column of nt */
   h_setTitle(mydisp, "Color Plot");
   h_setLabel(mydisp, XAXIS, "New Label");

   h_plot(mydisp);

   h_getRange(mydisp,XAXIS, &low,&high); /* x-axis range */
   h_setRange(mydisp,low,high/2.0);      /* set new range */
   h_plot(mydisp);

   h_freeDisp(mydisp);
```

Figure 3. Example Hippo display code.

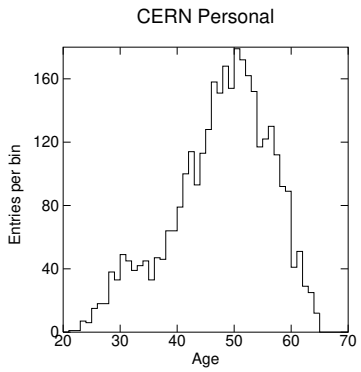
### 3.2 Display creation

The first step of making a display is to create a copy of the structure. The h_newDisp function, whose prototype is shown below, performs this function along with initialize many of the fields. The prototype for this function is shown below.

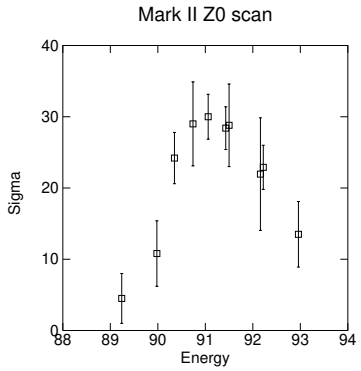
```
display h_newDisp( graphtype_t type );
```

The argument specifies what type of a display it will be (this can be changed later if desired). The possible types are listed below; Figure 3 shows an example of each.

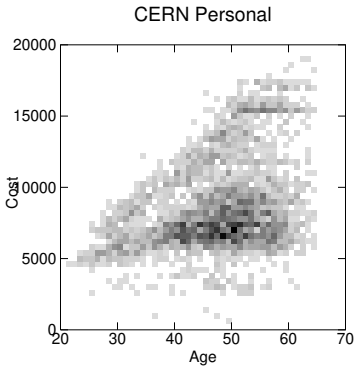
- `HISTOGRAM` - a one dimensional histogram,
- `XYPLOT` - a plot of one column of the ntuple against another, i.e. y versus x,
- `STRIPCHART` - a special form of `XYPLOT` in which the ntuple is considered as a circular buffer,



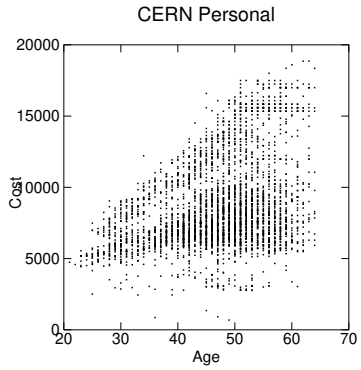

A `HISTOGRAM` display




A `XYPLOT` display




A `COLORPLOT` display




A `SCATTERPLOT` display

A `LEGOPLOT` display

Figure 4. Examples of Hippo displays.

- COLORPLOT - a two dimensional binned plot where the bin values are represented as a color or gray level,
- SCATTERPLOT - a two dimensional scatter plot,
- LEGOPLOT - a two dimensional binned plot drawn in three-D perspective (not implemented yet).

h_newDisp returns a display, which is actually a pointer to the display structure. The returned display is needed in all subsequent calls dealing with this particular display.

## 3.3 Bindings to the n-tuple

The display needs to know where to get the data that it will display, i.e. what ntuple to use. The function h_bindNtuple makes this connection. Its prototype is shown below. The n-tuple used by a display can be changed at any time by calling

```
int h_bindNtuple(display disp, ntuple nt );
```

this function. The display must also know which column of the ntuple to use for a given quantity. The function h_bind is used to specify which column of the ntuple (starting from 0) to use. Its prototype is show below. binding_t is an enumerated

```
int h_bind(display disp, binding_t bt, int nt_col );
```

type which can have the values XAXIS, YAXIS, ZAXIS, WEIGHT, XERROR, or YERROR. The names are hopefully self-explanatory. Not all of the quantities are relevant for all types of plots. For HISTOGRAMs, only XAXIS and WEIGHT are relevant, while for an XYPLOT, XAXIS and YAXIS must be bound (though there always are defaults) while XERROR and YERROR may be bound if desired. To unbind a quantity, set its nt_col to -1.

## 3.4 Axis and bin attributes

By default, *hippo* auto-scales an axis to cover the complete range of whatever ntuple column it is attached to. It tries to pick “nice” values for the upper and lower axis limits. Where bins are appropriate, *hippo* creates 50 bins along an axis. Pres-

ently, the range of bins is fixed to be the same as the axis range. To change these values, there are three functions as show below.

```
int h_setRange(display disp, binding_t axis,
               float low, float high );
int h_setBinNum(display disp, binding_t axis, int nbins);
int h_setBinWidth(display disp, binding_t axis, float width );
```

`h_setBinWidth` will adjust the upper axis limit to be consistent with the number of bins, the width and the lower limit. One can get logarithmic axes using the function `h_setLogAxis` whose prototype is show below, where axis is the desired axis and

```
int h_setLogAxis( display disp, binding_t axis, int flag );
```

flag is either `TRUE` (1) or `FALSE` (0). A log axis has no meaning for a binned axis.

## 3.5 Input and Output

Some of the input/output routines were mentioned in the section on n-tuples. There are four sets of input/output routines, which differ in the specification of the device. There are routines to write to or read from a named file, a C `FILE` structure, an XDR stream or memory. The memory routines have an extra parameter specifying the size of the buffer.

All the routine use the same code to handle the actual I/O. From the `NULL`-terminated lists of displays and n-tuples provided, *hippo* constructs a list of n-tuples which are needed; any ntuple referenced by a display is included, except those flagged as "by reference" using the routine `h_ntByReference`. What is written out is a "magic number" (the string "h_"), a string which gives the version of the structures, and all the n-tuples followed by all the displays. Everything is written out using XDR (external data representation) and is therefore machine portable.

When reading a file, the user passes the read routine display pointer and an ntuple pointer, both by reference. The routine will allocate the display and ntuple structures as it reads them, and will also allocate the lists of displays and n-tuples. Thus, on return, the display pointer points to a `NULL`-terminated list of displays (which, of course, can be access exactly like a C array), and the ntuple pointer to a list of n-tuples. In order to use data in text format, `h_fileParse` is provided to read an ntuple from a text file, while `h_nt2text` will write an n-tuple to a file in plain text format.

## 3.6 Titles and Labels

The display title and labels are set with the routines below In order to access

```
int h_setTitle( display disp, const char *title);
int h_setAxisLabel( display disp, binding_t axis,
                    const char *label);
```

ntuple labels, the following `printf` format-like strings are supported. For example, if "`%t`" appears in a string, then the n-tuple title is substituted in its place. The complete list of these substitution strings is given in section 4.2. Because of these substitutions, if you want a '`%`' in your label, use "`%%`".

## 3.7 Plot Drivers

There are currently six plot drivers supported in *hippo*.

- `NEXT` - using Display PostScript and some NeXT library routines.
- `UNIXPLOT` - using device independent calls. The type of output produced depends on the library linked against. The usual way of operating is to link against the library which send device-independent text to stdout, which is piped to a file. Programs are then used to, for example, display it as Tektronix graphics or send PostScript to a printer. The UNIX Plot libraries can be obtained by get the GNU Graphics package. It is available by anonymous FTP from `qed.rice.edu`.
- `XIVPLOT` - using InterViews X-Windows graphics. InterViews is a X-windows package written in C++ and is available by anonymous FTP from interviews.-stanford.edu.
- `MAC` - using Macintosh graphics routines.
- `X11PLOT` - using X11R4 graphic routines.
- `PSPLOT` - using PostScript to a file routines.

One can compile the graphics drivers or not by setting C macros in the Makefile. One can choose between the compiled drivers using the routine

```
int h_setPlotDrvr( plotdrvr_t drvr, ... );
```

The UNIXPLOT driver expects one extra parameter, a FILE pointer. This is intended as an alternate output file to stdout, but does not work at this moment. In addition, the XIVPLOT driver adds two parameters to h_plot, the plotting routine. The parameters are the "painter" and the "canvas", while the X11PLOT driver takes four additional parameters which are explained in the reference manual section of this document.

New plot drivers are fairly easy to implement. If you wish to write a new driver, it is easiest to work from example. Look at the UNIXPlot driver for the easiest implementation (also, the least pretty).

## 3.8 Cuts and Plot Functions

*Hippo* provides support for making cuts to the ntuple data before displaying and for plotting functions. The cuts are implemented to select or reject a each row of the ntuple data, not by examining the complete ntuple at once. The cuts and plot functions are maintained as linked-lists of a function pointer and an associated parameter block. Thus the user can write any arbitrary function which fits the function prototype in order to implement what she wants.

One problem with using pointers to arbitrary functions is the task of re-establishing the pointer after it has been written to a file and read back in. *Hippo* solves this by maintaining a "function registry" which allows *hippo* to find a function pointer from the name of the function. The user must "register" any function he writes using the function

```
int h_funcReg( void *func );
```

This must be done before using a function as a cut or plot function, and before a display which uses the function is read from a file.

The prototype for a cut function is

```
int cutfunc(float *nt_row, double *param_blk);
```

while that of a plot function is

```
double plotfunc( double x, double *param_blk );
```

*Hippo* provides eight cut functions which implement all possible simple cuts on one tuple dimension, for example, greater than a value or inside a region. These func-

tions are automatically registered. To add one of these cuts to a display, use the function

```
func_id h_addCut( display disp, const char *cutfunc,
                  int cut_dim, double val1, ... );
```

The possible values of cutfunc are "h_cut_ge", "h_cut_gt", "h_cut_le", "h_cut_lt", "h_cut_inside", "h_cut_in_incl", "h_cut_outside", and "h_cut_out_incl". Hopefully, the names are explanatory. cut_dim is the column number of the ntuple used in the cut. The first four cuts require one parameter (val1) while the final four need two parameters (val1 and the next parameter). When there are two parameters, the first is the lesser.

h_addCut returns a pointer to the function structure used by *hippo*. To change the cut, use the function

```
int h_changeCut( display disp, func_id cut_id,
                 double val1, ... );
```

This will change the parameter(s) of the cut, but not the ntuple column used.
Also provided is the function h_addUserCut to use an arbitrary function for cuts.

```
func_id h_addUserCut( display disp, const char *cutfunc,
                      double *paramblk, int nparam );
```

When a cut is added this way, the parameter block is maintained by the user; *hippo* simply maintains a pointer to it. The parameters should all be doubles so that they will be written correctly in a machine independent manner.

In addition, h_deleteCut is provided to delete a cut and h_nextCut is provided to traverse the list of cuts.

For plot functions, the routines h_addPlotFunc, h_deletePlotFunc, h_nextPlotFunc provide similar functionality to h_addUserCut, h_deleteCut and h_nextCut, respectively.

# 4. Reference Manual

In the tradition of C, all the details a programmer would every need to know about the package can be found in the header file. There, the prototypes of all the function calls may be found, as well as all the data types used by *hippo*. Comments in the header file summarize to use of the functions and structures. This reference manual section of this document is little more a a formatted version of the header files.

## 4.1 N-tuple functions

This subsection describes those *hippo* functions and structures that deal with the n-tuple data sets. They are listed in order of complexity and/or of frequency of use.

```
typedef struct {…} ntuple_t, *ntuple;
```

Contains information on an n-tuple and all the data. One should never access this data directly, but use *hippo* functions calls instead.

```
ntuple h_new( int ndim );
```

Returns a new n-tuple with dimension `ndim` (e.g. number of columns) if successful, otherwise returns `NULL`.

```
int h_freeNt( ntuple nt );
```

Frees all memory associated with the n-tuple and returns 0.

```
int h_clrNt( ntuple nt );
```

Clear all data points from the n-tuple, and reallocates memory for data to default size. Returns -1 on error, 0 on success.

```
int h_fill( ntuple nt,  ...);
int h_arrayFill( ntuple nt, float *data );
```

Adds one entry (e.g. row) to the end of the n-tuple data list. The first parameter for both functions is the n-tuple. The second parameter and subsequent parameters for `h_fill` are the data to be entered and the number of them must be equal to the dimension (e.g. number of columns) of the n-tuple. The second parameter for `h_arrayFill` is an array of data whose size should be at least equal to the dimension of the n-tuple. Returns -1 on error, 0 otherwise.

```
int h_setNtTitle( ntuple nt, const char *title );
```

Sets the title of the n-tuple data set. Returns 0 if successful, -1 otherwise.

```
int h_setNtLabel( ntuple nt, int dim, const char *label );
```

Sets the label of n-tuple dimension (column) specified by `dim`. The first column is specified with `dim = 0`. Returns 0 if successful, -1 otherwise.

```
int h_setAllNtLabels( ntuple nt, const char *label[] );
```

Sets the labels of all columns of the n-tuple from the array of strings `label` whose size must be at least as the dimension (number of columns) of the n-tuple. Returns 0 if successful, -1 otherwise.

```
int h_getNtDim( ntuple nt );
```

Returns the dimension (number of columns) of the specified n-tuple.

```
const char *h_getNtTitle( ntuple nt );
```

Returns the title of the specified n-tuple.

```
const char *h_getNtLabel( ntuple nt, int dim );
```

Returns the label of the n-tuple dimension specified by `dim`, or `NULL` on error.

```
int h_getNtNdata( ntuple nt );
```

Returns the number of entries (rows) of the specified n-tuple.

```
const float *h_getNtAllData( ntuple nt );
```

Returns a pointer to all the data in the specified n-tuple. The data is stored consecutively in the array, first all the columns for the first row, then the subsequent rows. Given this pointer, the user should not attempt to modify the data, but use `h_replData()` or `h_arrayReplData()` instead.

```
const float *h_getNtData( ntuple nt, int i_nt );
```

Returns a pointer to a particular row of the n-tuple data set specified by the second parameter, or returns `NULL` on error. Row numbering starts with the value 0. Given this pointer, the user should not attempt to modify the data, but use `h_replData()` or `h_arrayReplData()` instead.

```
float *h_getNtColumn( ntuple nt, int dim );
```

Returns a pointer to an array of data corresponding to a particular column specified by the second parameter of the specified n-tuple or `NULL` on error. The function `h_getNtNdata()` will return the size of this array. This array is a copy of the data in the n-tuple, therefore the user should free the memory when the array is no longer used. The first column is specified with the value of `dim` equal to 0.

```
const float *h_getNtHigh( ntuple nt );
```

Returns an array of the upper limits of each column of the specified n-tuple. For example, after the statement `f = h_getNtHigh(nt);`, `f[1]` is the largest value in the second column of the n-tuple. The user should not attempt to modify the contents of this array.

```
const float *h_getNtLow( ntuple nt );
```

Returns an array of the lower limits of each column of the specified n-tuple. For example, after the statement `f = h_getNtLow(nt);`, `f[1]` is the smallest value

in the second column of the n-tuple. The user should not attempt to modify the contents of this array

```
int h_arrayReplData( ntuple nt, int i_nt, float *data );
int h_replData( ntuple nt, int i_nt, ... );
```

Replaces the row specified by the second parameter in the n-tuple with the data in the third (or subsequent) parameters. The first row is specified with a parameter equal to 0. The specified row must already exist or an error condition will be returned. In the first form, h_arrayReplData, the parameter data must be an array whose size is at least equal to the dimension of the n-tuple. In the second form, h_replData, the number of third and subsequent parameters must be at least equal to the dimension of the n-tuple. After these functions calls, the upper and lower limit arrays may be in error and the flag extremeBad will be set to indicate that. Each function returns 0 if successful and -1 otherwise.

```
int h_getExtremeBad( ntuple nt);
```

Returns the `extremeBad` flag from the specified n-tuple. The value of this flag is 0 if the upper or lower limits returned by the functions `h_getNtLow()` and `h_getNtHigh()` are correct, or non-zero otherwise.

```
int h_ntSize( ntuple nt);
```

Returns the approximate number of bytes needed to store the specified n-tuple on disk. The exact size is not known until the data is actually encoded for storage, but the value returned by this function will not be smaller than the actual size needed.

## 4.2 Display functions

This subsection describes those *hippo* functions and structures that deal with displays. They are list in order of complexity and/or frequency of use.

```
typedef struct {…} display_t, *display;
```

Contains information of a display, histogram bins if any, and binding information to the n-tuple. One should never access this data directly, but use *hippo* functions calls instead.

```
typedef enum { SCATTERPLOT, LEGOPLOT, COLORPLOT,
       XYPLOT, STRIPCHART, HISTOGRAM } graphtype_t;
```

This enumerated type lists the type of displays available in *hippo*. A short description of this graph types is given below…

- `SCATTERPLOT` graph is normally called a scatter plot. An entry of the bound n-tuple is taken as x and y point pairs and a point is drawn. It is normally used with a large number of entries in the n-tuple.
- `LEGOPLOT` graph is normally called a lego plot for histograms of two dimension. Lego plots for histograms of two dimensions is not yet implemented.
- `COLORPLOT` graph is a form histogram of two dimensions in which color is used to show the number of entries in a bin. Color can be either full color spectrum or a grey scale depending on `drawtype_t` (see below).
- `XYPLOT` graph is similar to `SCATTERPLOT` graph except that error bars can also be drawn. It is normally used when there is relatively few entries in the n-tuple.
- `STRIPCHART` graph is similar to `XYPLOT` except the x-axis is taken as ordered list of points. When displaying a graph of this type, the minimum point on the x-axis is first found and displayed, then subsequent points are drawn to the last entry of the n-tuple. After the last entry, display continues from the first entry until the first point is found again. Thus, the n-tuple is used as a circular buffer for an application creating a strip chart.
- `HISTOGRAM` graph is a one dimensional binned histogram.

```
display h_newDisp( graphtype_t type );
```

Creates and returns a new display of specified type or `NULL` if error. Also sets most of the display attributes to reasonable default values.

```
display h_copyDisp( display olddisp );
```

Creates and returns a new display which is a copy of the display given as a parameter. Returns NULL on error.

```
int h_freeDisp( display disp );
```

Frees all memory associated with the specified display and returns 0.

```
int h_dispSize( display disp );
```

Returns the approximate number of bytes needed to store the specified display on disk. The exact size is not known until the data is actually encoded for storage, but the value returned by this function will not be smaller than the actual size needed.

```
int h_setDispType( display disp, graphtype_t type);
graphtype_t h_getDispType(display disp);
```

The first function sets the type of graph for the display. The second function returns the type of graph type of the display. The graph type must be a member of the enumerated list, graphtype_t, described above.

```
int    h_bindNtuple( display disp, ntuple nt );
ntuple h_getNtuple( display disp );
```

The first function *binds* the specified display to the specified n-tuple and returns 0 if successful and -1 otherwise. The process of binding gives the display its source of data. The display attributes of the display are unaffected by this function call. A display already bound to an n-tuple can be unbound from one and bound to another with this function call. If the display is a histogram, then the data bins of the histogram will be regenerated when the display is plotted. The second function returns the n-tuple to which a display is currently bound to or NULL if none.

```
int   h_setNtByRef( display disp, int flag,
                    const char *filename );
int   h_getNtByRef( display disp );
char *h_getNtFile( display disp );
```

The first function sets a flag to indicate if the n-tuple bound to the display is accessed by reference or not. The flag only has effect when the display is written to disk. If the flag is set (non-zero value), then only the filename, given by the third argument, of the n-tuple is written out with the display structure. If the flag is not set (0 value), then the n-tuple data set is written out with the display structure. The second function returns the by reference flag, while the third function returns the filename if the display is bound by reference. If a display was written out with its n-tuple bound by reference, then the application reading in the display is responsible for re-establishing the binding of the display to the n-tuple.

```
int h_setDrawType(display disp, drawtype_t type);
int h_orDrawType(display disp, drawtype_t type);
drawtype_t h_getDrawType(display disp);
```

The first function sets the drawing type of a display. Several drawing types can be done simultaneously according to which bits of the `drawtype_t` parameter are turned on. The second function does a logical OR of the bits of its `drawtype_t` parameter with the bits already stored. The last function returns the current `drawtype_t` bits of the display. The first two functions return 0 if successful or -1 otherwise. The allowed values of the drawtype_t parameter is given in the following enumerated list.

```
typedef enum { NONE = 0, BOX = 1, POINT = 2, LINE = 4,
                ERRBAR = 8, COLOR =16
} drawtype_t;
```

- `NONE` indicates no drawing.
- `BOX` indicates that bars will be drawn for `HISTOGRAM` type of graph.
- `POINT` indicates that a point will be drawn
- `LINE` indicates that a line connecting neighboring x-y points will be drawn.
- `ERRBAR` indicates for that for `HISTOGRAM`, `XYPLOT`, or `STRIPCHART` type of graph, error bars will be drawn. The error bars are calculated as the square root of the number of entries.

- `COLOR` indicates to use full color spectrum for `COLORPLOT` type of graph. If not set, then gray scale is used instead.

```
int         h_setTitle( display disp, const char *title);
const char *h_getTitle( display disp );
```

The first function sets the title of the display and returns 0 if successful or -1 otherwise. The second function returns the current title of the display. In the character string, `title`, one may use `printf()` style substitutions to pickup attributes from the n-tuple bound to the display. The following substitutions are permitted…

- `%t` – title of the n-tuple bound to the display.
- `%x` – label of the n-tuple column bound to the x-axis of the display.
- `%y` – label of the n-tuple column bound to the y-axis of the display.
- `%z` – label of the n-tuple column bound to the z-axis of the display.
- `%w` – label of the n-tuple column bound to the weight-axis of the display.
- `%ex` – label of the n-tuple column bound to the x error-axis of the display.
- `%ey` – label of the n-tuple column bound to the y error-axis of the display.
- `%dx` – width along the x-axis of a histogram bin.
- `%dy` – width along the y-axis of a histogram bin.

```
void h_setPlotSym( display disp, plotsymbol_t p );
plotsymbol_t h_getPlotSym( display disp );
```

The first function sets the plot symbol for points. The attribute is always set but will only have effect if the `drawtype_t POINTS` attribute of the display is set. The second function returns the current plot symbol attribute. The plotsymbol_t argument must be defined in the following enumerated type…

```
typedef enum { SQUARE, SOLIDSQUARE, PLUS, TIMES }
plotsymbol_t;
```

```
void h_setSymSize( display disp, float s );
float h_getSymSize( display disp );
```

The first function sets the size of the plot symbol. The second returns the size of the plot symbol. The units are points, e.g. 1/72 of an inch.

```
void h_setLineStyle( display disp, linestyle_t t );
linestyle_t h_getLineStyle( display disp );
```

The first function sets the style of line to be used by the display. The attribute is always set but will only have effect if the drawtype_t LINE attribute is set. The second function returns the current line style attribute. The linestyle_t argument must be defined in the following enumerated type…

```
typedef enum { SOLID, DASH, DOT, DOTDASH } linestyle_t;
```

## 4.3 Display axis.

This subsection describes *hippo* functions that deal with displays and their binding to axis.

```
typedef enum { XAXIS, YAXIS, ZAXIS, WEIGHT, XERROR, YERROR }
binding_t;
```

The binding_t enumerated type will be used in arguments to *hippo* that require an axis to be specified.

```
int h_bind( display disp, binding_t axis, int dataDim);
int h_getBinding( display disp, binding_t axis );
```

The first function binds an axis of the display to a column of its n-tuple specified by the third argument and returns 0 if successful or -1 otherwise. If the display axis is already bound, its binding will be replaced with the binding specified by the function call. The first column of the n-tuple has the index value 0. The second parameter must be a member of the enumerated type described above. The second function returns the current binding. To unbind a quantity, set its `dataDim` to -1.

```
int h_bindMany(display disp, int n, ...);
```

Binds the specified number (n) of axes to the n-tuple columns. The third and subsequent parameters must be in pairs of binding_t and n-tuple column number. Returns 0 if successful or -1 if error.

```
int h_setAutoScale( display disp, binding_t axis,
                    int onOff );
int h_getAutoScale( display disp, binding_t axis );
h_autoScale( display disp );
```

The first function sets a flag for the specified axis that controls auto-scaling of that axis. If auto-scaling is set on (`onOff` = 1), then the range of the display will be adjusted so that all the data of its n-tuple's column is visible. The second function returns the current status of auto-scale flag for the specified axis. The third function will force auto-scaling to occur immediately (instead of waiting for plot drawing) for axes with the flag set.

```
int h_setRange( display disp, binding_t axis,
                float low, float high );
int h_getRange( display disp, binding_t axis,
                float *low, float *high );
```

The first function sets the low and high values of the displayed range along the specified axis and return 0 if successful or -1 otherwise. The second function returns the currently displayed range in the 3rd and 4th arguments. If the axis is auto-scaled (see below), then the returned values are valid only after the display has been drawn or the binning has been done (for binned axis).

```
int h_setBinNum( display disp, binding_t axis, int n);
int h_getBinNum( display disp, binding_t axis );
```

The first function sets the number of bins for the specified axis of the display and returns 0 if successful or -1 otherwise. The second function returns the current number of bins for the axis of the display. This attribute is always set, although if the display is not a histogram it has no effect.

```
int   h_setBinWidth( display disp, binding_t axis,
                     float width);
float h_getBinWidth( display disp, binding_t axis );
```

The first function sets the width of bins along the specified axis for the display and returns 0 if successful or -1 otherwise. The upper limit of the axis range is adjusted to be consistent with the lower limit, the bin width and the number of bins. The second function returns the bin width of the specified axis of the display. This display attribute is set by this function call, although it has no effect if the display is not a histogram.

```
int h_setLogAxis( display disp, binding_t axis,
                  int onOff );
int h_getLogAxis( display disp, binding_t axis );
```

The first function sets whether the specified axis of the display is to have log base 10 scale (`onOff` = 1) or linear scale (`onOff` = 0). This display attribute is always set, but has no meaning to an axis that is binned. The second function returns 1 if the axis is set to log scale or 0 otherwise.

```
int         h_setAxisLabel( display disp,
                            binding_t axis,
                            const char *label );
const char *h_getAxisLabel( display disp,
                            binding_t axis );
```

The first function sets the label of the specified axis of the display and return 0 if successful or -1 otherwise. The second function returns the label of the specified axis. String substitution as described for `h_setTitle()` on page 24 are valid.

```
char *h_expandLabel( char *dest, const char *src,
                     int max, display disp);
```

This function expands a label `src` which may contain *hippo* format specifiers described on page 24 for the display `disp`. The resulting expansion is copied to string `dest` which is an array of length `max`, and returned by the function.

## 4.4 Display Actions

```
int h_setPlotDrvr( plotdrvr_t drvr, ... );
```

Selects the device used for plotting. The optional arguments are used by some plot drivers. The first parameter must be one of the following enumerated types…

```
typedef enum { NEXT, UNIXPLOT, LPR, XIVPLOT, MAC, X11PLOT
               PSPLOT, EPSPLOT }plotdrvr_t;
```

- `NEXT` type uses Display PostScript and NeXT library graphic routines.
- `UNIXPLOT` type uses UNIXPlot software.
- `LPR` type is intended for line printer output making use only of the standard 96 character ASCII set. If `LPR` type is set, then `h_plot()` is same as calling the `h_print()` function.
- `XIVPLOT` type uses the InterViews X-Windows graphic routines
- `MAC` type uses Apple Macintosh QuickDraw graphic routines.
- `X11PLOT` use X11R4 graphic routines.
- `PSPLOT` use routines to send PostScript to a file. The file stream pointer is given as the second argument in the function call.
- `EPSPLOT` same as `PSPLOT` for now.

```
int h_plot(display disp, ...);
```

Plots the display using the current plot driver. Optional arguments are used by some plot drivers. For example, `XIVPLOT` passes the painter and canvas as optional arguments. The effective prototype for `XIVPLOT` is…

```
int h_plot(display disp, Painter* output, Canvas* canvas);
```

The effective prototype for the `X11PLOT` driver is

```
int h_plot(display disp, Display *dpy, Screen *scrn,
                         Drawable wind, GC gc);
```

where

`disp` is the *hippo* plot to be drawn.

`dsy` is the X display pointer.

`scrn` is the X screen pointer for the screen on which the window resides.

`wind` is the window or Pixmap into which the plot is to be drawn.

`gc` is a graphics context suitable for use with `wind`. The graphics context is used to specify the background color for the plot and can also be used to specify a clip mask (may be useful when handling exposure events).

```
int h_endPage(void);
int h_endDrvr(void);
```

The first function informs the plot driver that it should do an end page operation. The second function informs the plot driver that all PostScript processing is complete. These function is useful, for example, with the plot driver that sends PostScript to a file.

```
void h_fprint(display disp, FILE *file);
void h_print(display disp);
```

Prints an ASCII plain text representation of a display to a file or stdout.

```
int h_bin(display disp);
```

Regenerates the contents of bins if the display is a histogram and returns 0 if successful or -1 otherwise. This function does not re-plot the display.

```
const float *h_getBins( display disp);
```

Returns a pointer to an array with the contents of the bins if a display is a histogram. See the function `h_getBinNum()` for determining the number of bins. The user should not use this pointer to modify the contents of the bins.

```
const float *h_getTotal(display disp, int i, int j);
```

Returns the (weighted) sums of events inside/outside the plot. The index `i` is for x-axis and `j` for y-axis. A value of 0 for these indices means below lower limit, 1 means inside plot's range, and 2 is above range. The user should not use this pointer to modify the contents of the array.

```
const float *h_getVariance( display disp );
```

Returns a pointer to an array of the variances of the bins (the square of the error). See the function `h_getBinNum`() for determining the number of bins. The user should not use this pointer to modify the contents of the array.

```
int h_getBinExtreme( display disp, float *min, float *max );
```

Returns the value of the bins with the minimum and maximum values in its arguments. Returns 0 if successful or -1 otherwise.

```
void h_setDrawTitles( display disp, int flag);
int  h_getDrawTitles( display disp );
```

The first function sets a flag to determine if the title and labels are to be drawn when the display is plotted and returns 0 if successful or -1 otherwise. The second function returns the value of this flag.

```
void h_setDrawAxes( display disp, int flag);
int  h_getDrawAxes( display disp );
```

The first function sets a flag to determine if the axes of the display are to be drawn (flag = 1) or not (flag = 0) when the display is plotted. The second function returns the current value of this flag.

```
void h_setFixedBins( display disp, int flag );
int  h_getFixedBins( display disp );
```

The first function sets a flag which determines if the bins of a histogram display are to be fixed (`flag=1`) or not. If the flag is set, then h_bin() and h_plot() will

no longer do any re-binning and when the display is written out to disk, the contents of the bins will be written as well. The second function returns the value of this flag.

## 4.5 Auxiliary functions

```
float h_binVal(display disp, ...);
```

Returns the value of a bin specified by arguments taken as integer index. The number of the second and subsequent arguments must be equal to the dimension of the display, i.e. 1, 2, or 3. The index of the first bin has the value 0.

```
int h_ptToBin(display disp, float f, binding_t axis);
```

Returns the index to the bin which would contain the value `f` along the specified axis, or -1 on error. The index of the first bin has the value 0.

```
int h_setDrawRect(   display disp, rectangle *rect);
int h_setMarginRect( display disp, rectangle *rect);
int h_getDrawRect(   display disp, rectangle *rect);
int h_getMarginRect( display disp, rectangle *rect);
```

The first function sets the size and position of the rectangle in which drawing of the display is done. The second function sets the size and position of the rectangle on which the axes are drawn. The third and fourth functions returns a copy of the rectangle used by their respective “set” functions. For each function, the units used are points, e.g. 1/72 of an inch. The rectangle for the margin is in the same coordinate system as the draw rectangle. Each function returns 0 if successful, -1 otherwise. The rectangle structure is compatible with NeXT’s `NXRect` and defined as follows…

```
typedef struct {
    struct {
        float x, y;
    } origin;
    struct {
        float width, height;
    } size;
} rectangle;
```

```
float h_wPtTogPt(display disp, float wPt, binding_t axis);
float h_gPtTowPt(display disp, float gPt, binding_t axis);
```

The first function converts the value wPt from window coordinates (i.e. the same as the draw rectangle) to graph coordinates (i.e. the same as the axis scale). The second function does the opposite.

```
void h_setDirty( display disp );
```

Sets the dirty flag of a display. A display becomes dirty if some display attribute has changed or if a display is a histogram when the bins need to be regenerated. This function should usually never need to be called as the dirty flag is set automatically by the *hippo* package. However, in the case when a user cut has changed (see section 4.7), it should be called so that re-binning and/or re-plotting will be correctly done.

```
int h_shade(display disp, float low, float high );
```

Shades a region of the plot along the x-axis. The shaded region runs from the low to the high value along the x-axis but does not go past the x-axis range. The shaded region covers the entire range of the y-axis.

## 4.6 Input and Output functions

```
int h_writeStream( FILE *outfile,
                   display dlist[], ntuple ntlist[] );
int h_readStream(  FILE *infile,
                   display **dlist, ntuple **ntlist );
```

These routines write or read, respectively a NULL terminated list of displays and a NULL terminated list of n-tuples to or from a file stream specified by the file pointer. When writing to a file, all the n-tuples that are bound to any of the displays are also written to the file unless they are bound by reference. Thus one can use a NULL argument for the ntuple list and the list will be generated automatically. To write only the n-tuples to a file, use a NULL argument in place of the display list. The following code fragment illustrates how to read a file. Both functions return 0 if successful or -1 otherwise.

```
FILE    *myfile;
display *d_list;
ntuple  *nt_list;
h_readStream( myfile, &d_list, &nt_list )
```

```
int h_write( const char *filename,
             display dlist[], ntuple ntlist[] );
int h_read(  const char *filename,
             display **dlist, ntuple **ntlist );
```

These routines write or read, respectively a NULL terminated list of displays and a NULL terminated list of n-tuples to or from a file specified by filename of the first argument. See `h_writeStream()` for more information as these functions are just layers on top of it.

```
int h_writeMem( char *buf, int len,
             display dlist[], ntuple ntlist[] );
int h_readMem(  char *buf, int len,
             display **dlist, ntuple **ntlist );
```

These routines write or read, respectively a NULL terminated list of displays and a NULL terminated list of n-tuples to or from a memory buffer. See `h_writeStream()` for more information. The second argument is the length of the buffer.

```
int h_writeXDR( XDR *xdrs,
             display dlist[], ntuple ntlist[] );
int h_readXDR(  XDR *xdrs,
             display **dlist, ntuple **ntlist );
```

These routines write or read, respectively a NULL terminated list of displays and a NULL terminated list of n-tuples to or from the specified XDR stream. See h_writeStream() for more information. The XDR stream should be opened with the appropriate ENCODE or DECODE code.

```
ntuple h_fileParse( FILE* ifile,
                    ntuple oldnt, int verbose );
```

Reads an n-tuple represented as a plain text files and creates a new binary n-tuple if oldnt is NULL or append to the n-tuple specified. If the argument verbose is non-zero, the function will print messages to stdout. Returns the n-tuple created or modified.

```
int h_nt2text( FILE *outfile, ntuple nt );
```

Writes a n-tuple as a plain text file (suitable for h_fileParse()) to the file specified.

## 4.7 Cut and Plot Functions

The cut functions of the *hippo* package allows the use of the values of one or more columns of the n-tuple to be used as a selection criteria on whether an entry of the n-tuple (e.g. a row) is to be used for plotting or inclusion into a histogram. Plot functions are use to over-plot the display with a user supplied function.

```
typedef struct {…} func_id_t, *func_id;
```

This is a structure that *hippo* uses to maintain a linked list of functions that are applied as a cut or used as plot functions.

```
int h_func_reg( const char *name, void *func);
void h_funcReg( void *func);
```

The first function assigns the name `name` to the function `func` and registers this name in a function registry. The second function can be used when the function name and the function are the same. Character string names for functions are used for input and output. Built-in *hippo* supported cut functions are automatically registered. User supplied functions must be registered before their first reference.

```
func_id h_addCut( display disp, const char *cutfunc,
                  int cut_dim, double val1, ... );
```

Adds a *hippo*-supported cut to a display. Returns a pointer to the cut structure or `NULL` on error. The standard cuts are available as a character string argument as follows…

"h_cut_le"

"h_cut_lt"

"h_cut_ge"

"h_cut_gt"

"h_cut_inside"

"h_cut_in_incl"

"h_cut_outside"

"h_cut_out_incl"

The first four require one parameter (`val1`) and the remaining four require two parameters. When there are 2 parameters, the first is the lesser of the two.

```
func_id h_addUserCut( display disp, const char *cutfunc,
                      double *param_blk, int nParam );
```

Adds a user-supported cut to a display. The user passes a function name, a pointer to a parameter block (containing all doubles), and the size of the parameter block (number of doubles). It is the responsibility of the user to maintain the parameters; if they are changed, one should call `h_setDirty()` (not necessary if display is not binned) on the display. *Hippo* maintains a pointer to the parameter block, so it should not be moved.

```
int h_changeCut( display disp, func_id cut,
                 double val1, ... );
```

Changes the cutting parameters of the display to the cut defined by the `func_id cut` which was returned by the `h_addCut()` function. It takes a variable number of addition arguments according to what kind of cut is being performed. Returns 0 if successful or -1 otherwise.

```
int h_deleteCut( display disp, func_id cut );
```

Removes a cut from a display. Returns 0 if successful or -1 otherwise. The argument, cut, is one returned from `h_addCut()`.

```
func_id h_nextCut( display disp, func_id thiscut );
```

Returns the next cut function used by a display given the current cut. If the value of the current cut is `NULL`, returns the first cut function. Returns `NULL` if the current cut is the last one.

```
func_id h_addPlotFunc( display disp, const char *plotfunc,
                       double *paramBlk, int nParam
                       linestyle_t ls );
```

Adds a function to the linked list of plot functions that will be drawn over a display when plotted. The name of the function, `plotfunc`, must be registered with `h_func_reg()` before this function is called. The line style for the function is also set (see page 25).

```
func_id h_nextPlotFunc( display disp, func_id thisfunc );
```

Returns the next plot function in the linked list of plot functions applied to a display. Use of this function is similar to `h_nextCut()`.

```
int h_deletePlotFunc( display disp, func_id pfunc );
```

Removes a plot function from the linked list of plot functions applied to a display. Use of this function is the same as `h_deleteCut()`

## 4.8 FORTRAN Binding.

In the current release, the FORTRAN binding support in *hippo* is limited to the handling of n-tuples. This subsection describes the FORTRAN functions that are supported. In general, the FORTRAN function corresponds to the C function with the leading "`ip`" replaced with "`h_`". However, there may be minor differences in the use of the arguments due to differences in the C and FORTRAN languages and their common usage.

```
Integer function ipnew( ndim )
Integer ndim
```

Returns a new n-tuple with dimension `ndim` (e.g. number of columns) if successful, otherwise returns `0`. The integer value returned is actually the address of the C structure that manages the n-tuple. Thus its value must never be modified. From the calling FORTRAN subprogram, the value should be considered as a unique identifier of the n-tuple created with the call to `ipnew()` that will be used as an argument in subsequence calls to the *hippo* package.

```
Integer Function ipfreeNt( nt )
Integer nt
```

Frees all memory associated with the n-tuple and returns 0.

```
Integer Function ipclrNt( nt )
Integer nt
```

Clear all data points from the n-tuple, and reallocates memory for data to default size. Returns -1 on error, 0 on success.

```
Integer Function iparrayFill( nt, x )
Integer nt
Real    x(*)
```

Adds one entry (e.g. row) to the end of the n-tuple data list. The first argument is the n-tuple. The second argument is an array of data whose size should be at least equal to the dimension of the n-tuple. Returns -1 on error, 0 otherwise.

```
Integer Function ipsetNtTitle( nt, title )
Integer       nt
Character*(*) title
```

Sets the title of the n-tuple data set. Trailing blank characters in the character array are removed. Returns 0 if successful, -1 otherwise.

```
Integer Function ipsetNtLabel( nt, ndim, label )
Integer       nt
Integer       ndim
Character*(*) label
```

Sets the label of n-tuple dimension (column) specified by `ndim`. The first column is specified with `dim = 1`. Trailing blank characters in the character array are removed. Returns 0 if successful, -1 otherwise.

```
Integer Function ipwrite( file, dlist, ntlist )
Character*(*) file
Integer       dlist(*)
Integer       ntlist(*)
```

Writes `0` terminated list of displays and a `0` terminated list of n-tuples to a file. However, since the FORTRAN binding to displays are not supported in this release, the second argument should have the value `0`. Figure 2 on page 6 illustrates the use of this function. Returns 0 if successful or -1 otherwise.

# 5. Installation

The Hippoplotamus package was designed to be portable across many different operating systems and different vendor implementations of the UNIX operating system. Also, it also separates out the code to handle a display device (called a plot driver) into separate files. It is only with the installation of *hippo* that one sees the operating system and vendor differences. This section contains instruction for installing *hippo* on systems where it has already been tested and a guide for porting *hippo* to other systems.

## 5.1 Requirements

The minimum requirements for installing *hippo* are an ANSI C compliant compiler with the standard C library and the Sun XDR library. The latter is usually available on UNIX systems and is part of the package for TCP/IP support on other operating systems. Optionally the FORTRAN binding can be built, which obviously requires a FORTRAN '77 or FORTRAN '90 compiler. By the way, the bindings have been tested with a FORTRAN '90 compiler.

In some cases, it may be desirable to install only the n-tuple management part of the *hippo* package such as when one generates the n-tuple files on one platform but has a display application on another. In such cases, only the following files need to be complied into the library…

```
hippontuple.c
hippoio.c
hippoxdr.c
hippoutil.c
ntuple-parse.c
```

In addition, the following C header files need to be made available to the programmer…

```
hippo.h
hippostruct.h
hippoxdr.h
hippoutil.h
```

The remaining C source code and header files are only needed for the display part of the *hippo* package.

The *hippo* package comes with three utility programs which can be optionally build and installed. They all main programs that must be linked to the *hippo* library.

The first two are contained in the files `nt2text.c` and `text2nt.c` which perform the conversion to and from plain text file format and the binary XDR format. They only require the C compiler. The third is contained in the file hb2hippo.f which extracts an n-tuple from a HBOOK4 formatted file and produces a *hippo* binary XDR format file. It requires a FORTRAN compiler and the parts of the CERN program library required by HBOOK4 (`packlib`).

## 5.2 UNIX Makefile

This package includes a `makefile` for UNIX machines. The file is built with machine independent parts (`Makefile` and `make.common`) and a machine dependent part i.e. `make.next`, `make.sun4`, etc. The machine architecture is determined by the script "`architecture`" and is set when `make.common` is called from `Makefile`. Architectures on which have tested hippo are Sun, NeXT, Ultrix, SGI, and AIX (on RS/6000). If you are adding a new machine, check that the script `architecture` gives a sensible answer.

Because there are a number of plot drivers available, and one may want more than one driver available on a given architecture, we have designed the makefile to build different libraries for the different (major) drivers. For example, on a NeXT, you would need to build only `libhippoNext.a`, while on a Sun, you might want to build `libhippoX11.a` and `libhippoXIV.a`. If you don't have any of these packages to link to, you could build a library (make target "`hippo`") which only contains the PostScript and line printer drivers, no screen drivers.

There are a number of makefile macros which may need to be tailored to your system. The machine dependency file is included late so that it can override any macro defined in Makefile.

In `make.common` the following makefile macros are defined…

- `SRC_DIR` - this is the directory containing the source code.
- `LIB_INSTALLDIR` - directory in which to install the libraries.
- `INC_INSTALLDIR` - directory in which to install the header files.
- `LIB_DIR` - directory in which the libraries are built. It is included as a `-L` option when linking the utility programs, so it should be defined.

In the system dependent files, e.g. `make.aix6000`, the following makefile macros are defined…

- `CC, CPLUSPLUS` - your compiler commands
- `FC` - your FORTRAN compiler if needed.
- `MACH_DEP_MEMS` - the library members which should be included for this machine. These are essentially only a selection of plot drivers (see below).
- `DRVR_INCS` - the list of include files for the selected drivers.
- `PLOTCFLAGS` - C compile flags for defining plot drivers.
- `LIB_LIST` - list of libraries to build.
- `DEBUGLIB_LIST` - list of debug libraries to build.
- `FBINDINGS` - the FORTRAN binding files. If you don't have a FORTRAN compiler or don't wish to use the FORTRAN bindings, then set this macro to a blank.

The *hippo* package provides a number of plot drivers, which can be selected for inclusion in the library in any combinations. The make.next and make.sun4 give examples of their use. Table 1 summaries the available drivers, their source code files, and required header files. A driver must be selected at compile time in order to

**Table 1: *hippo* plot drivers**

| Name | Device | Source | Header | Macro |
|---|---|---|---|---|
| `LPR` | Line printer | `(none)` | `(none)` | `(none)` |
| `PS` | PostScript to file. | `(none)` | `(none)` | `(none)` |
| `NEXT` | NeXT DPS | `hippoplotNeXT.c` `hippowrapsps.c` | `hippoplotNeXT.h` `hippowrapsps.h` | `_NEXT_PLOT_` |
| `UNIXPLOT` | UNIXPlot | `hippoplotUP.c` | `hippoplotUP.h` | `_UNIXPLOT_` |
| `XIVPLOT` | InterViews | `hippoplotXIV.cc` | `hippoplotXIV.h` | `_XIVPLOT_` |
| `X11PLOT` | X11R4 | `hippoplotX11.c` | `hippoplotX11.h` | `_X11PLOT_` |

be able to use it in a program. By default, on of the screen drivers is included in each library. For example, libhippoX11.a is build using `X11PLOT` driver.

To select a driver to put in a library, take the following steps…

- Add the source code file to `MACH_DEP_MEMS` in the form,

  - `$(LIBRARY)(source_file.o)`

  where `source_file.o` is the compiled object file of the files listed in Table 1.

- Add the corresponding header file to `DRVR_INCS`.

- Add the corresponding C macro definition to `PLOTCFLAGS`, for example,
  `-D_NEXT_PLOT_`

You can select multiple drivers. Finally, you should choose one default plot driver by defining the C macro `DEF_PLOT_DRVR` in `PLOTCFLAGS`; for example
`-DDEF_PLOT_DRVR=NEXT`
One must use one the names listed in Table 1. The driver names, `LPR` and `PS`, are always defined and should be used as a default if you don't want any other driver defined.

Once the makefile and its included file(s) have been customized (if necessary), then one can proceed to build various components of the *hippo* package. The makefile contains the following targets to help with the installation…

- `all` - build the regular and debug libraries and utilities.

- `libs` - build the (default) standard library: "`libhippo.a`".

- `debuglibs` - build the debug library, "`libhippo.debug.a`", e.g. `-g` option used with compiler.

- `<name>` - build library `lib<name>.a`, *e.g.* `libhippoX11.a`.

- `profile` - build the profile library, e.g. `-pg` option used with compiler.

- `util` - build the `nt2text` and `text2nt` converter programs.

- `clean` - delete backup files.

- `install` - copy libraries and certain include files to specified place.

- `hb2hippo` - build the hb2hippo program.

- `test` - run a few tests to see that *hippo* is working correctly.

The default libraries that are built with `libs` target varies with the machine architecture and is control by the individual `make.*` files. For example, on a NeXT, the default is to build `libhippoNext.a`, while on a SUN it is to build both the `libhippoXIV.a` and `libhippoX11.a` libraries.

## 5.3 VMS Installation.

Included in the *hippo* package is an MMS file for VMS under the name `make.vms` (this does not need the file `Makefile`). This file is written to compile with the `X11PLOT` driver. You may need to change the logicals defined under `.First` to point to the correct directories (this is for RPC include files). Otherwise, many of the macro are the same as for the UNIX makefile. We do not change the name of the FORTRAN file `hippof.f`.

When linking a program with *hippo* on VMS, you will need the RPC and Multinet libraries. On the system where *hippo* was tested, they are located at

```
multinet_root:[multinet.library]rpc.olb
multinet_root:[multinet]multinet_socket_library.exe
```

(The socket library is really only needed to resolve `bcopy` and `bset`, something that Multinet should have put in the RPC library). The *hippo* package has not been tested with other vendor's TCP/IP packages. The file `hippo.opt`, which is part of the distributed *hippo* package can be used to specify these libraries.

The C macro `GLOB_QUAL` is defined to be "`noshare`" on VMS. This should allow one to build a sharable library.

## 5.4 Miscellaneous Files

As part of the *hippo* distribution, there are a few other files described below…

- `architecture` - a shell script used by the Makefile to determine which flavor of UNIX is being used.

- `example.hippo` - an example *hippo* n-tuple in binary format and is used by `make test` procedure.

- `example.hiptxt` - an example *hippo* n-tuple file in plain text format and is used by the `make test` procedure.

- `fhippo.c, hippo.f` - the optional FORTRAN binding source files.
- `getarg.c` - a FORTRAN callable function to get the command line arguments. This function is in the FORTRAN run-time library for many systems and supplied as part of the *hippo* package for when it is missing, e.g. NeXT.
- `hippo.opt` - options file for VAX/VMS linker.
- `hshrtnm.h` - header file with short names for most of the *hippo* functions that is used for compiling on IBM mainframes.
- `h_test.c` - a program to test basic *hippo* functions.
- `h_test.out` - reference output of `h_test`.
- `h_testX11.c` - program to exercise the X11 driver.

## 5.5 Problems, Changes and Bug Reports

We are very interested in hearing about bugs and improvements. If you find a bug, please send mail to

```
hippo_bug@ebnextk.slac.stanford.edu
```

Please try to be as specific as possible. Mail to this mailing aliases will be carefully kept and eventually dealt with.

If you have any suggestions or wish to make some contribution (e.g. a new plot driver), please contact the authors. Send mail to

```
hippo_comment@ebnextk.slac.stanford.edu.
```

Mail to this mailing aliases will be kept as a wish list. It is also recommended that if you are using *hippo* that you send mail to this address to register your copy. In that way, you'll be informed of major updates and bug fixes.

# 6. History

The prototype for *hippo* was a package called HandyC written by Benoit Mours while he was at SLAC. It was intended to support the Reason project. HandyC was written in C, and it demonstrated the power of the C programming language for such packages. It followed the calling sequence of SLAC's HandyPak[3] but added a few new features. Features from the DESY package GEP[4], and the CERN package HBOOK[2], which have been incorporated into *hippo*. The authors of *hippo* are indebted to work done Benoit Mours and the authors of these other packages.

HandyC was not only a good test bed to try out new ideas, but also to measure the performance and/or response time the user would feel for these features. In the summer of 1990, Jonas Karlsson, then working as a summer student, suggested that one should really start over again with a better base in order to implement yet further new ideas. William Shipley (another summer student) and Gary Word contributed to the foundation of *hippo* soon thereafter. Jonas Karlsson did the original implementation. The current authors started working on *hippo* in earnest early in 1991. Tom Pavel did the InterViews plot driver and Tony Johnson of Boston University did the X11R4 driver.

# 7. References

**[1]** XDR External Data Representation Standard, RFC1014, Sun Microsystems, Inc., USC-ISI (see also man pages on UNIX systems).

**[2]** R. Brun, D. Lienart, HBOOK USER GUIDE: CERN COMPUTER CENTER PROGRAM LIBRARY LONG WRITEUP:VERSION 4, CERN-Y250, Oct 1987. 108pp.

**[3]** A. Boyarski, HANDYPAK: A HISTOGRAM AND DISPLAY PACKAGE (RELEASE 6.5), SLAC-0234-Rev-2, Sep 1988. 132pp. Revised version.

**[4]** E. Bassler, THE GRAPHICAL EDITOR PROGRAM: GEP, COMPUT. PHYS. COMMUN. 45 (1987) 201-205.

# Index


% subsitutions 14, 24

**A**
accumulate, see fill
Apple 28
architecture 40, 43
auto-scaling 26
axes
  auto-scaling 26
  binding 25, 26
  drawing 30
  linear scale 27
  log scale 13, 27

**B**
binding
  axes 25, 26
  by reference 23
  display to n-tuple 12, 22
binding_t 25
bins 13, 29
  fixed 31
  index of 31
  maximum 30
  minimum 30
  number of 26
  value of 31
  width of 27
BOX 23
buffer 33
bugs 44
by reference flag 13

**C**
canvas 15
clear n-tuple
  with C 17
  with FORTRAN 37
COLOR 23
color 24
color plot 21
COLORPLOT 21
conversion
  from HBOOK 8
  from plain text 7
copy display 22
creation of n-tuple 3
cuts
  add 16, 35
  change 16, 36
  delete 16, 36
  function 15
  next 36
  user 35

**D**
DASH 25
delete
  cut 16, 36
  plot function 16, 36
dirty flag 32
display 21
Display Postscript 14
displayed range 26
displays with FORTRAN 7
DOT 25
DOTDASH 25
drawtype_t 23

**E**
e-mail addresses 44
Encapsulated PostScript 28
enumerated types
  binding_t 25
  drawtype_t 23
  graphtype_t 21
  linestyle_t 25
  plotdrvr_t 28
  plotsymbol_t 24
EPSPLOT 28
ERRBAR 23
error bars 23
example files 43
expand 27

**F**
fill n-tuple
  C 4, 18
  FORTRAN 6, 37
flags
  auto-scale 26
  by reference 13, 23
  dirty 32
  draw axes 30
  draw labels 30
  draw titles 30
  extremeBad 20
  fixed bins 30
  log axis scale 13
FORTRAN 6, 37, 39, 41
free
  display 22
  n-tuple
    C 17
    FORTRAN 37
full color 24
func_id_t 34
function
  cut 15
  delete 36
  next plot 36
  plot 15, 36
function registry 15, 35
functions 15

**G**
GEP 45
getarg.c 44
GLOB_QUAL 43
GNU 14
graph coordinates 32
graphtype_t 21
gray scale 24

**H**
h_addCut 16, 35
h_addPlotFunc 16, 36
h_addUserCut 16, 35
h_arrayFill 4, 18
h_arrayReplData 20
h_autoScale 26
h_bin 29
h_bind 12, 25
h_bindMany 26
h_bindNtuple 12, 22
h_changeCut 16, 36
h_clrNt 17
h_copyDisp 22
h_cut_ge 35
h_cut_gt 35
h_cut_in_incl 35
h_cut_inside 35
h_cut_le 35
h_cut_lt 35
h_cut_out_incl 35
h_cut_outside 35
h_deleteCut 16, 36
h_deletePlotFunc 16, 36
h_dispSize 22
h_endDrvr 29
h_endPage 29
h_expandLabel 27
h_fileParse 13, 34
h_fill 4, 18
h_fprint 29
h_freeDisp 22
h_freeNt 17
h_func_reg 35
h_funcReg 15, 35
h_getAutoScale 26
h_getAxisLabel 27
h_getBinding 25
h_getBinExtreme 30
h_getBinNum 26
h_getBins 29

h_getBinWidth 27
h_getDispType 22
h_getDrawAxes 30
h_getDrawRect 31
h_getDrawTitles 30
h_getDrawType 23
h_getExtremeBad 20
h_getFixedBins 30
h_getLineStyle 25
h_getLogAxis 27
h_getMarginRect 31
h_getNtAllData 19
h_getNtByRef 23
h_getNtColumn 19
h_getNtData 19
h_getNtDim 18
h_getNtFile 23
h_getNtHigh 19
h_getNtLabel 18
h_getNtLow 19
h_getNtNdata 19
h_getNtTitle 18
h_getNtuple 22
h_getPlotSym 24
h_getRange 26
h_getSymSize 25
h_getTitle 24
h_getTotal 30
h_getVariance 30
h_gPtTowPt 32
h_new 3, 17
h_newDisp 10, 21
h_nextCut 16, 36
h_nextPlotFunc 16, 36
h_nt2text 13, 34
h_ntByReference 13
h_ntSize 20
h_orDrawType 23
h_plot 15, 28
h_print 29
h_ptToBin 31
h_read 33
h_readMem 33
h_readStream 33
h_readXDR 34
h_replData 20
h_setAllNtLabels 5, 18
h_setAutoScale 26
h_setAxisLabel 14, 27
h_setBinNum 13, 26
h_setBinWidth 13, 27
h_setDirty 32
h_setDispType 22
h_setDrawAxes 30
h_setDrawRect 31
h_setDrawTitles 30
h_setDrawType 23
h_setFixedBins 30
h_setLineStyle 25
h_setLogAxis 13, 27
h_setMarginRect 31
h_setNtByRef 23
h_setNtLabel 5, 18
h_setNtTitle 5, 18
h_setPlotDrvr 14, 28
h_setPlotSym 24
h_setRange 13, 26
h_setSymSize 25
h_setTitle 14, 24
h_shade 32
h_test.c 44
h_testX11.c 44
h_wPtTogPt 32
h_write 4, 33
h_writeMem 33
h_writeStream 33
h_writeXDR 34
HandyC 45
HandyPak 45
hb2hippo 40
hb2hippo utility 8
HBOOK 8, 40, 45
hippo.opt 44
histogram 21
hshrtnm.h 44

**I**
IBM 44
index to a bin 31
InterViews 14, 28
iparrayFill 6, 37
ipclrNt 37
ipfill 6
ipfreeNt 37
ipnew 6, 37
ipsetNtLabel 6, 38
ipsetNtTitle 6, 38
ipwrite 7, 38

**L**
labels
  axes 27
  C 14, 18
  display 30
  FORTRAN 6, 38
  plain text 7
labels of columns 5
lego plot 21
LEGOPLOT 21
libraries 40
limits of n-tuple data 19, 20
LINE 23
line style 25, 36
linear scale 27
linestyle_t 25
list of functions 34
list of n-tuples 4
log scale 27
LPR 28

**M**
MAC 14, 28
Macintosh 14, 28
Makefile 14
makefile 40, 42
maximum bin 30
memory buffer 33
minimum bin 30
Multinet 43

**N**
network support 1
new
  display 21, 22
  n-tuple
    C 17
    FORTRAN 6, 37
NEXT 14, 28
NeXT 14
next cut 36
noshare 43
nt2text 40
ntuple 3, 17
n-tuple, definition of 1
ntuple_t 17
number of bins 26
number of columns 18
number of entries 19

**P**
painter 15
plain text 7, 13
plot
  color scatter 21
  delete function 36
  drivers 14, 28
  function 15, 36
  lego plot 21
  next function 36
  plain text 29
  scatter plot 21
  symbol 24
plot libraries 14
plotdrvr_t 28
plotfunc 15
plotsymbol_t 24
PLUS 24
POINT 23
PostScript 14, 28
printer 28
PSPLOT 14, 28

**Q**
QuickDraw 28

**R**
range of display 13, 26
read
  buffer 33
  file 33
  plain text 34
  stream 33
  XDR 34
reading a file 13
re-binning 31
rectangle 31
`rectangle` 32
register functions 15, 35

**S**
scatter plot 21
`SCATTERPLOT` 21
shading 32
size of display 22
size of n-tuple 20
size of plot symbol 25
`SOLID` 25
`SOLIDSQUARE` 24
`SQUARE` 24
structures
  display 21
  functions 34
  n-tuple 17
  rectangle 31
style of line 25
substitution strings 14
substitutions in labels 14, 24
sums 30
symbol for plot 24

**T**
`text2nt` 40
`text2nt` utility 7
`TIMES` 24
title
  display 24, 30
  n-tuple
    C 14, 18
    FORTRAN 6, 38
  plain text 7
title of n-tuple 5
totals 30
type of display 22
type of drawing 23
`typedef`
  `binding_t` 25
  `display` 21
  `display_t` 21
  `drawtype_t` 23
  `func_id` 34
  `func_id_t` 34
  `graphtype_t` 21
  `linestyle_t` 25
  `ntuple` 17
  `ntuple_t` 17
  `plotdrvr_t` 28
  `plotsymbol_t` 24
  `rectangle` 32

**U**
unbind 12, 25
`UNIXPLOT` 14, 28
user cut 35
utilities 39
  `hb2hippo` 8
  `text2nt` 7

**V**
value of a bin 31
variance 30
VAX/VMS linker 44
VMS 43

**W**
`WEIGHT` 25
width of bins 13, 27
window coordinates 32
write
  buffer 13, 33
  file 33
  FORTRAN 7, 38
  plain text 34
  stream 33
  XDR 34
writing n-tuple 13

**X**
`X11PLOT` 14, 28
X11R4 28
`XAXIS` 25
XDR 1, 13, 34, 39, 45
`XERROR` 25
`XIVPLOT` 14, 28
X-Windows 14, 28
x-y plot 21
`XYPLOT` 21

**Y**
`YAXIS` 25
`YERROR` 25

**Z**
`ZAXIS` 25