```
5.6319926e-13  5.6319926e-13  -1.689818e-12  7    6
0.485636829    0.852655       0.500015       23   11
-0.485636322   0.852655       0.500001       59   16
```

According to the header, the first three columns are the x, y, and z values of each fixed point. The fourth column contains color information and the last column contains symbol information. To load this file into DsTool, the user would first set the parameter values a, b, and c, to reflect those parameter values specified in the header. The variable w should also be set to the value 0, since each of these fixed points occur in the plane $w = 0$. Variables and parameters may be explicitly set in the Selected Point Panel. Since the particular set of equations being studied is time independent, the user may choose to ignore the variable time.

The user then selects the unformatted load option from the Load Window, types in a valid directory and filename, and proceeds to choose panels items which tell DsTool that the file to be read contains fixed point data, color information, and symbol information. The user then tells DsTool that it should expect to read the variables x, y, and z by selecting the appropriate rows of the variables list. See the User's Manual for more information on how to use the Load Window.

When the user finally presses the Load command button, DsTool finds the specified file (provided the file exists and is readable), skips over the header, and reads the three fixed points from the file into fixed point memory objects. Associated with each memory object is optional color and symbol data. In the example above, the first fixed point is assigned color 7. This means that whenever a "pick color" colormap is in use, then the color of the fixed point will be the seventh color on whatever colormap is currently in use. (If a "depth" colormap is in use, then the color of the fixed point depends on its coordinates; if an alternating colormap is in use, the color of the fixed point will be whatever the current alternating color was when the fixed point was loaded.) Similarly, the second and third fixed points are assigned the twenty-third and fifty-ninth color, respectively, whenever a "pick color" colormap is in use.

Also associated with each memory object is a symbol. In the example above, the first fixed point was assigned symbol 6, the second was assigned symbol 11, and the third, symbol 16. The table below summarizes what integer is associated with each symbol.

Additional comments about the way DsTool reads unformatted data:

1. The order that variables appear in the variable list (on the Load Window) *is precisely the order in which DsTool expects to encounter variables in a file.* In the above example, since the variables x, y, and z were selected from the Load Window variable list, DsTool will read the first number it encounters into an x position, the next number it meets into the y variable, and the third number it finds into the z variable position. If the user had mistakenly read in a file in which the first column contained values for the z variable, then DsTool would mistakenly place these values into the x variable. The same is true for the order in which parameters appear on the parameter list.

2. All variables are read before any parameters are read. In other words, if a user has

Integer Code	Plotting Symbol
1	Small Dot
2	Medium Dot
3	Large Dot
4	Huge Dot
5	Small Triangle
6	Medium Triangle
7	Large Triangle
8	Huge Triangle
9	Small Cross
10	Medium Cross
11	Large Cross
12	Huge Cross
13	Small Box
14	Medium Box
15	Large Box
16	Huge Box

Table 1.1: Integer codes for plotting symbols within DsTool.

specified that a file contains n variables and k parameters, then the first n numbers DsTool encounters will be stuffed into the specified variable slots, and the next k numbers it finds will be assigned to the specified parameter values.

3. DsTool will read color information (if any) only after reading all specified variables and parameters.

4. DsTool will read symbol information (if any) only after reading all specified variables, parameters, and any color information.

5. Although data files usually contain columns of data, this is not necessary. As long as there is white space (blanks or tabs) between pieces of data, DsTool will assign the data correctly. Thus the data for the above example could also appear within its file as

```
5.6319926e-13   5.6319926e-13  -1.689818e-12  7  6  0.48563682
0.852655  0.500015  23   11  -0.485636322  0.852655
0.500001  59  16
```

Chapter 2

Printing Data

All print files in DsTool are written in the PostScript language. Reference books on PostScript include *PostScript Language Tutorial and Cookbook* and *PostScript Language Reference*, both written by Adobe Systems Incorporated and published by Addison-Wesley.

2.1 Coordinate System

In PostScript, each printed page has a default coordinate system consisting of an origin in the lower left-hand corner of the page. There are 72 PostScript units to the inch, which makes the PostScript unit very close to the standard printer's point (1/72.27 inch). For our purposes, however, it is convenient to work in another coordinate system, namely, the coordinate system employed by the user when he or she selects the Print command button. To avoid anomalies caused by rounding in postscript, the body of a DsTool print data file in Version 2.0 contains coordinates normalized to lie between 0 and 1. For example, suppose a twoD view window displays coordinates x, y and ranges $x_0 \leq x \leq x_1$, $y_0 \leq y \leq y_1$ and suppose that the Print Window is owned by this view window. (The owner of the Print Window may be determined by examining the Owner message on the Print Window. See the entry for the Print Window in the User's Manual.) The default PostScript coordinate system is scaled so that the ranges $x_0 \leq x \leq x_1$, $y_0 \leq y \leq y_1$ fit in a square which is 450 PostScript units on a side. (These dimensions, 450 by 450, can be modified by changing the *Hor Length* and *Ver Height* fields in the print window. See Section 2.4 below.) The point (x_0, y_0) will be located at a point approximately 1 inch to the right and 2 inches up from the lower left-hand corner of the page. These values can also be customized by adjusting the *offset* fields of the print window.

Because of this scaled coordinate system, it is straightforward to modify PostScript files which are written by DsTool (provided the user knows a little bit about PostScript). For example, if the user knows that a special bifurcation point occurs at (α_0, β_0) and wants to place a special symbol at this point, one would refer to the point $((\alpha_0 - x_0)/(x_1 - x_0), (\beta_0 - y_0)/(y_1 - y_0))$ in the body of the print file.

2.2 DsTool Format

The PostScript files generated by DsTool adhere to the PostScript structuring conventions. As such, they consist of several parts:

Header: This consists of a series of comment statements which includes the name of the dynamical system being studied, the date, and the name of the user.

Prologue: The prologue consists of PostScript definitions which are used to draw symbols, bounding boxes, axes, labels, and titles. By default, these definitions are read from a file named `Prolog.ps` which is kept in `$DSTOOL/site_specific`.

Script: The first part of the script defines the coordinate system which will be subsequently used (see above). If the user has chosen to display a bounding box, title, axis labels, and/or system information, then these commands are also executed in the script. The remainder of the script consists primarily of a series of identical PostScript commands of the form:

$$[\textit{red green blue}\ \texttt{setrgbcolor}]\ x\ y\ \textit{size}\ \textit{symbol}$$

Here *red*, *green*, and *blue* are RGB color values (between 0 and 1) which determine the color of the subsequent point. The bracketed expression is optional and is written only when color PostScript is desired. The values x and y are the position of the point (in the normalized coordinate system). The string *size* is actually a PostScript integer variable, whereas *symbol* is a call to a PostScript function (defined in the prologue) which plots a symbol at (x, y) in the size specified by *size*.

Trailer: The trailer is not used by DsTool; it is left empty.

2.3 Piping Data to Printers

Rather than writing data into a PostScript file, it is possible to send data directly to a printer which reads PostScript (for example, the SPARCprinter, the Apple LaserWriter, and the Linotype Linotronic 300 typesetter). This is done by selecting the Printer choice option on the Destination exclusive setting. Type the name of the printer in the Printer text field (referred to below as *printername*) and choose the remaining print options as desired.

When the Print command button is selected, DsTool opens a pipe and spawns the child process `lpr -P`*printername*. The PostScript data is then written into this pipe. If all goes well, the printer will interpret the PostScript data and produce an image.

There are situations in which the child process will fail. If this occurs, the operating system will notify DsTool of the death of the child. Depending upon how the child died, DsTool may or may not be able to recover from the error. Possible errors include: a broken pipe, a bad argument to a system call, and the printer daemons inability to create a temporary file (for example, if the size of your data exceeds the size of temporary file space). If sending data directly to a printer does not work for some reason, we suggest writing the data to a PostScript file and manually sending the file to a printer.

While we cannot foresee all possible sources of error, we have made provisions for the case in which a child process dies because of a typing error. If an incorrect *printername* is given to DsTool, the spawned `lpr` command will terminate with the error message `lpr: `*printername*`: unknown printer` sent to standard error. As DsTool continues to attempt to write into the (now broken) pipe, it will be sent the signal SIGPIPE, indicating an an attempt to write on a pipe with no one to read it. We now describe how this error is handled by DsTool.

When the Print command button is selected, DsTool uses the C library function `signal()` to declare that if a SIGPIPE signal is generated, then execution is to pass immediately to an error function called `broken_pipe_error()`. This function executes a *non-local jump* using the library functions `setjmp()` and `longjmp()`. When a SIGPIPE error is received, DsTool stops writing to the pipe, notifies the user of the error, and then returns control of the program to the window manager.

Some debuggers, including Sabertool, allow you to follow DsTool as it catches the signal and then pases control to `broken_pipe_error()`. It may be necessary to inform the debugger that it should ignore signal SIGPIPE. In sabertool, the command to do this is `ignore SIGPIPE`.

2.4 Customizing the Print Environment

Much of the print environment is configurable from within DsTool. On occasion, however, the user may wish to change some of the default settings such as the width of the lines or the size of the bounding box. The default prologue is contained in the file `$DSTOOL/site_specific/Prolog.ps`. The environment variable DSTOOL_PS_PROLOG can be used to specify a different prolog file. The beginning of this file includes a set of constants. These constants and their function are described below. All units are PostScript units (approximately one point).

- BBoxSide (default = 500): The side length of the bounding box. Decreasing this number by one half will result in an image which is one quarter the size of the default image size. Note, however, that the symbols which are plotted are *absolute* in size, so it may be necessary to decrease the values of these symbols.
- TickSize (default = 5): The length of the tick marks.
- xorigin (default = 80): The horizontal position of the lower left hand corner of the bounding box.
- yorigin (default = 145): The vertical position of the lower left hand corner of the bounding box.
- linewidth (default = 0.005): The width of lines used to draw symbols and tick marks. A value of 0 for this constant is interpreted as being the smallest resolution the printer can produce. Because very high resolution printers can produce lines too small to be seen, we have chosen the (arbitrary) default value to be small but positive. If your

printer is incapable of producing a line this thin, it may not print the symbols at all. In that case, change `linewidth` to a larger value such as 0.1. Values over 0.1 are not recommended.

- Sm, Md, Lg, and XL (defaults = 1, 2, 4, 6): These are scaling factors. The symbols (excluding the Dot) are defined in the prologue to fit inside the unit square. The scale factors multiply the lengths so that, for example, the Large (=4) Box is a square 4 units on each size. These symbols are *absolute* in size. The Dot is as above, except that a Small Dot is defined to be a circle with a 0.2 unit radius.

- HeadPtSize (default = 6): The system information is displayed using this font size. This may be too small for some people, and too big for some large-dimensional dynamical systems.

- Yp: The system information will be displayed on the page beginning at this vertical position.

The remainder of the prologue contains PostScript definitions. We do not recommend that you change these definitions.

The default printer used by DsTool will be the value of the LPDEST environment variable if defined. If this is not defined, it will be the value of the PRINTER environment variable. If none of these are defined, the value is taken from the source file defaults.h.

Chapter 3

Creating Custom User Applications

One of the most important features of DsTool is the ease with which new computational routines, constructed by the user, can be added to the program. This chapter describes the methods which may be used to enhance the capabilities of DsTool, providing access to the new computational procedures through a button on the panels menu of the Command Window. The discussion which follows is comprised of two parts. First, the interface between DsTool and a new computational algorithm is described. Second, the techniques used to create a panel through which the new algorithm can be controlled are outlined.

3.1 The DsTool Interface

We will assume that the user has already installed a local user version of DsTool (see the DsTool installation information). In the discussion that follows we shall denote the directory of the local user installation by the label `$DSTOOL_USER`. This section describes how to create a custom user library by going through a sample modification. We note that the DsTool code has already been modified for the following sample customization in order to give the user a working example.

3.1.1 Writing the Required Routines

The addition of new computational code essentially requires writing three new C code routines. The first is an initialization routine which we shall name example_init(). This routine will be called by DsTool each time a new dynamical system is loaded. The second routine, which we shall name example_go(), is the routine to do the analysis. This routine will be called by the user by selecting an item on the Panels Menu of the Command Window. The third routine, which we shall name example_handler(), is a routine required by the windowing system. This is the routine which will actually call the example_go() routine when the menu item is selected. Once these routines have been written, we must modify some DsTool code which allows these routines to be accessed. This code is contained in the

file `$DSTOOL_USER/userlib/user_panels.c`.

For this example, the three new routines will be written in a file named `example.c` in the directory `$DSTOOL_USER/userlib`. We shall now describe each of the three routines in more detail.

3.1.1.1 example_init()

The init routine is called when a new dynamical system is loaded. It is not passed any parameters and may return an integer, however any value it returns will be ignored. This routine is generally useful for initializing data structures whose size is dependent on the dynamical system. The following code is an example which does nothing except print a message that the routine was called.

```
#include <stdio.h>    /* include stdio.h declarations for stdout */

/*
 * example_init()
 *
 * example routines initialization
 * called whenever a new dynamical system is loaded
 */
int
example_init()
{
  fprintf(stdout, "EXAMPLE routines initialized.\n");
}
```

3.1.1.2 example_go()

The example_go() routine is the one which should do the computational analysis. It is called without any arguments and may return an integer. Any value returned by the routine example_go() will be ignored by DsTool. This routine will be called by the handler routine when the appropriate menu item has been selected by the user. The sample code below simply prints a message that it has been called.

```
#include <stdio.h>    /* include stdio.h declarations for stdout */

/*
 * example_go()
 *
 * example analysis begin
 * called when the "Example" menu item is selected
 */
int
example_go()
{
  fprintf(stdout, "Beginning EXAMPLE analysis...\n");
  /* this is where you must put your own code */
  fprintf(stdout, "EXAMPLE analysis complete.\n");
}
```

3.1.1.3 example_handler()

The handler routine is required by the windowing system and must be declared in the following way. It will be passed in two objects - a Menu_item and a Menu_generate. These are special XView window objects and the user should refer to The XView Programming Manual [2] for more information. The important fact for this routine is that when the Menu_generate item has value equal to MENU_NOTIFY then the user has selected this option from the menu so the computational procedure should be called. The menu object types are declared in the file `<xview/xview.h>` which must be included into this C code file. The handler routine should also return the Menu_item object when it exits. The following code is a sample handler routine.

```
#include <xview/xview.h>     /* include the xview declarations for        */
                             /* Menu_item, Menu_generate, and MENU_NOTIFY */

/*
 * example_handler()
 *
 * Menu handler for 'panelmenu (Example)'.
 */
Menu_item
  example_handler(item, op)
Menu_item item;
Menu_generate op;
{
  if (op == MENU_NOTIFY)
    {
      example_go();
    }
  return item;
}
```

3.1.2 Modifying user_panels.c

To declare the new routines, edit the file `$DSTOOL_USER/userlib/user_panels.c`. In the declarations section, declare the handler routine and the initialization routine.

```
extern Menu_item example_handler();
extern int example_init();
```

In the structure definition section, add a line for the new menu item:

```
struct User_Def USR_PANELS[] = {
    {"Example", example_handler, example_init} };
```

The string "Example" is the word which will appear on the Command Window Panels menu. If the example_init() procedure is not needed then that entry may be replaced with `NULL`. To install multiple custom routines, the structure definition section can contain multiple entries. For example:

```
struct User_Def USR_PANELS[] = {
    {"Custom1...", custom1_handler, NULL},
    {"Example", example_handler, example_init},
    {"Debug...", debug_handler, debug_init} };
```

These are all the changes which need to be made to the `user_panels.c` file.

3.1.3 Updating the User Library Makefile

Finally, the `Makefile` for the local user library needs to be updated. To continue the example from the previous section, assume that the three new routines and any additional code are contained in a file named `example.c`. This file should be moved to the `$DSTOOL_USER` subdirectory and needs to be included in the user library. Edit the file `$DSTOOL_USER/Makefile` by including the new source code file in the `SOURCES_B` list. This becomes:

```
SOURCES_B = \ example.c
```

When the user's local version of DsTool is recompiled, the new code will be incorporated into DsTool. Recall that the compilation is initiated from the `$DSTOOL_USER` directory with the command `make dstool_u`.

3.2 Creating a Custom Window

The last section described the method for interfacing new code into DsTool. If the new code requires any substantial user interaction, then the implementation of a new window is warranted. This section gives instructions on how to build windows in the same style as the current DsTool windows. The interfaces into DsTool are the same ones described in the previous section, so it is not necessary that a new window be built to these specifications. This section is simply a set of suggestions for one way to develop a new DsTool window. In order to add a custom window to DsTool you will need to make sure that DsTool was installed with the **full** option.

We will assume that the reader has available the OpenWindows Developer's GUIDE (Graphical User Interface Design Editor) [6]. This package will greatly facilitate the writing of windowing code using the XView toolkit. Instructions on the use of this development tool is best left to the GUIDE User's Manual [6]. Similarly, we will not attempt to instruct the reader in the XView toolkit. For this we recommend The XView Programming Manual [2].

3.2.1 The Example

In addition to outlining a method for adding a new window, we shall give code fragments for a trivial example. We shall call this sample implementation the Trial panel. The Trial panel will simply be a set of text fields, one for each phase space variable. The user will be able to enter a value for each variable. There will be a single settings item which will allow the user to select a single phase space variable. Finally, there will be a button. When the user presses the button, the new code should set the value of the selected variable to be the value given in the Trial window. The change in value of this coordinate should be reflected in DsTool's