```
  for (i=0; i<n_varb; i++)
    {
      trial_varb_value[i] = 0.0;
    }

  /* reset the setting to choice 0 */
  trial_choice = 0;

  return(0);
}
```

3.2.5.5 trial_read_window()

This routine should read all the information from the window and write it into the data structure initialized by the routine trial_init(). See Section 3.2.5.4 for more details.

```
/*
 * trial_read_window()
 *
 * routine to read data from all items in the trial window
 */
int
  trial_read_window()
{
  int i, n_varb;
  char *strng;

  /* if no window, then nothing to read */
  if (trial_ip == NULL) return;

  n_varb =  *((int *) pm( GET, "Model.Varb_Dim", NULL ));

  /* read text fields */
  for (i=0; i<n_varb; i++)
    {
      strng = (char *) xv_get(trial_ip->varb_value[i], PANEL_VALUE);
      trial_varb_value[i] = atof(strng);
    }

  /* read setting item */
  trial_choice = xv_get(trial_ip->setting1, PANEL_VALUE);
}
```

3.2.5.6 trial_data_refresh()

This routine should read all the information from the data structure and write it into the window.

```
/*
 * trial_data_refresh()
 *
 * routine to refresh the data in all items in the trial window
 */
```

```
int
  trial_data_refresh()
{
  int format, i, n_varb;
  char strng[20];

  /* if no window, then nothing to refresh */
  if (trial_ip == NULL) return;

  /* read information on precision of doubles */
  format = *((int *) pm(GET, "Defaults.Precision", NULL));

  n_varb =  *((int *) pm( GET, "Model.Varb_Dim", NULL ));

  /* refresh text fields */
  for(i=0; i<n_varb; i++)
    {
      sprintf(strng, "%.*lg",format,trial_varb_value[i]);
      xv_set(trial_ip->varb_value[i], PANEL_VALUE, strng, NULL);
    }

  /* refresh setting items */
  xv_set(trial_ip->setting1, PANEL_VALUE, trial_choice, NULL);
}
```

3.2.5.7 trial_handler()

This is another routine which has changed considerably between Versions 1.1 and 2.0. In Version 1.1, one might have:

```
/*
 * trial_handler()
 *
 * Menu handler for `panelmenu (Trial...)'.
 */
Menu_item
  trial_handler(item, op)
Menu_item item;
Menu_generate op;
{
  if (op == MENU_NOTIFY)
    {
      trial_open();
    }
  return item;
}
```

In Version 2.0, the more complicated form below supports interaction with batch and saveload routines:

```
/*
 * trial_handler()
 *
```

```
 * Menu handler for `panelmenu (Trial...)'.
 */

Menu_item
  trial_handler(item, op)
Menu_item item;
Menu_generate op;
{
  int locn[4];

  if (op == MENU_NOTIFY)
    {
      if (item != (Menu_item) NULL)
trial_open(DEFAULT_WIN_CONFIG,0,0,0,0);
      else
{
  pm(GET_LIST, "Win.Locn.Trial", 0, 3, locn, NULL);
  if ((locn[0] != NO_LOCATION))
    trial_open(SET_WIN_CONFIG,locn[0],locn[1],locn[2],locn[3]);
  else
    trial_open(DEFAULT_WIN_CONFIG,0,0,0,0);
}
    }

  return item;
}
```

3.2.6 Adding a New Library

It is easiest to assemble all the code to implement a new panel into a single library. This time the entry in the file `$DSTOOL_USER/userlib/user_panels.c` will reference the two routines trial_handler() and trial_field_manager(). In the declarations section, declare the handler routine and the initialization routine.

```
extern Menu_item trial_handler();
extern int trial_field_manager();
```

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

```
struct User_Def USR_PANELS[] = {
    {"Trial...", trial_handler, trial_field_manager} };
```

In DsTool 2.0, it is important that the first token in the name of the user panel (*"Trial"* here) be the same as those referred to in the register_win_and_type call of `trial_open()` and the last token in the `pm(GET_LIST,``Win.locn.Trial'', ...)` call within `trial_handler()`.

The computational code which is executed when buttons are pressed should be placed in a file called `trial_go.c`. Note that this file should be completely independent of the windowing system. It should use the data structure written by trial_read_window(). For our example, the code in `trial_go.c` is simply the following:

```
#include <stdio.h>

#include <pm.h>

extern int trial_choice;
extern double *trial_varb_value;

/*
 * trial_button1_go()
 *
 * puts chosen value into postmaster
 */
int
  trial_button1_go()
{
  pm(PUT, "Selected.Varb_Ic", trial_choice, trial_varb_value[trial_choice], NULL);
}
```

There is a general Makefile already written for use with DsTool which should be used to make the new library. To copy it, type

```
cd $DSTOOL_USER/trial
cp $DSTOOL_USER/userlib/Makefile.generic Makefile
```

Now edit the Makefile to include your new code. For our example, the following lines should read:

```
LIBRARY = triallib.a

SOURCES_A = \
        trial_cui.c \
        trial_hnd.c \
        trial.c \
        trial_go.c

DEPS_A = trial_cui.h
```

You will also need to add this new library into the list of libraries contained in the Makefile in the directory `$DSTOOL_USER`. Do this by adding `triallib.a` to the list of `USER_WIN_LIBS`.

```
USER_LIBS = \
        models/modellib.a \
        userlib/userlib.a \
        trial/triallib.a
```

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

Chapter 4

Adding User Transformations to the Multiple Panel

This chapter describes how to add transformations to the multiple panel. These transformations may optionally be applied to sets of initial conditions hence allowing for regions to be selected which are not regular boxes. This is useful, for example, to project the grid of initial conditions generated by the multiple panel onto a specific subspace, or surface.

We will assume that the user has already installed a local user version of DsTool (see the DsTool Installation Manual). Let `$DSTOOL_USER` denote the name of the directory of the local user installation. We will be modifying the file `mult_proj.c` in the directory `$DSTOOL_USER/userlib` in order to add a new multiple projection.

There are three steps which must be taken:

1. Write the new transformation.
2. Declare the new transformation.
3. Enter the new transformation and a name into the list of available transformations

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

4.0.7 Writing the Transformation

This section details the requirements of a user multiple transformation. The C code for this transformation may be included at the tail of the file `mult_projs.c`. All the default system multiple transformations are stored in that file and make good examples.

For the purposes of example, we shall examine the included multiple transformation which projects the set of initial points into an ellipse. The declaration for transformation procedures have the following form:

```
/*
 * project onto an ellipsoid
```

```
 */
int
  ellipse_proj(x, fx, x_dim, params, fparams, params_dim, center, radius, alpha)
double *x, *fx, *params, *fparams, *center, *radius, alpha;
int x_dim, params_dim;
```

The information passed into the procedure is the point in phase space `x` and the dimension of the phase space `x_dim`. Time is included as the last coordinate. The parameter values are contained in `params` and the dimension of the parameter space in `params_dim`. The point in the center of the selected region is contained in `center`. This is an augmented array of the phase space (including time) and the parameter space and has dimension `x_dim+params_dim`. Radius contains the array of distances selected in the Multiple window and also has dimension `x_dim+params_dim`. Finally, alpha contains the value of the transformation parameter as specified in the Multiple window.

It is the responsibility of the procedure to fill up the arrays `fx` and `fparams` with the appropriate values, even if it means copying most of them from the input values `x` and `params`. In the example transformation, we consider an ellipsoid in phase space with radii given by the vector `radius` and centered about `center`. If the given point falls within the ellipsoid, then it is not transformed. If it is exterior to the ellipsoid then the point is projected radially towards the center of the ellipsoid onto its surface. Here is the code fragment which accomplishes this:

```
/*
 * project onto an ellipsoid
 */
int
  ellipse_proj(x, fx, x_dim, params, fparams, params_dim, center, radius, alpha)
double *x, *fx, *params, *fparams, *center, *radius, alpha;
int x_dim, params_dim;
{
  double d=0.0;
  int i;

  for (i=0; i<x_dim-1; i++) /* ignore time */
    d += (x[i]-center[i])*(x[i]-center[i])/(radius[i] * radius[i]);
  if (d > 1.0)
    {
      d = sqrt(d);
      for (i=0; i<x_dim-1; i++)
        fx[i] = (x[i]-center[i])/d + center[i];
      fx[x_dim-1] = x[x_dim-1];
    }
  else
    {
      for (i=0; i<x_dim; i++)
        fx[i] = x[i];
    }
  for (i=0; i<params_dim; i++)
    fparams[i] = params[i];
}
```

4.0.8 Declaring the Transformation

In the file mult_projs.c, locate the declaration section and declare your function as in the examples below:

```
/* ******************************************** *
 *                                              *
 * DECLARE your mult projection functions here.  *
 *                                              *
 * ******************************************** */
extern int xyrot();
extern int ellipse_proj();
```

4.0.9 Making the Transformation Available

Now locate the region of mult_projs.c where the MULT_PROJS structure is set up. You should add an element to this struct for your new transformation. The first part of the entry is the name which will appear on the Transformations menu on the Multiple Panel. The second part of the entry is the name of the transformation. Here is an example:

```
/* ******************************************** *
 *                                              *
 * ADD an entry for your mult projection here.   *
 *                                              *
 ********************************************** */

struct Mult_Proj  MULT_PROJS[] = {
  { "ellipse",ellipse_proj},
  {"x-y rotation",xyrot} };
```

Chapter 5

Data Storage

5.1 The Memory Object

This section details the use of the memory object. First we will discuss the structure of the memory object and then we will discuss the routines available to manipulate memory objects.

The memory object is organized as a list of flow objects. A flow object is an array of trajectories grouped together. A trajectory is essentially a list of points. Each point should be considered to have four parts. These are: an array of doubles, an array of integers, an array of double parameters, and an array of integer parameters. We shall refer to the double and integer arrays together as the *body*. The double and integer parameters are associated with the flow object and hence are shared by each point in each trajectory of the flow object. We will refer to these storage arrays together as the *header*. This is the most elementary way to view the memory object since there is no interpretation of what the values mean. We shall call this the *vanilla* view.

The standard way to look at a memory object is an interpreted view and gives us seven fundamental elements. These elements are the phase space coordinates, the parameter coordinates, the color coding information, extra doubles, extra integers, extra double parameters, and extra integer parameters. The extra double and extra integer coordinates are stored in the header and hence shared by all the points in the flow object. The extra double and extra integers are stored in the body. The *extra* fields are allowed so that additional information may optionally be stored with points in both the body and the header. The remaining three fields are stored in either the header or the body depending on the type of the memory object. Table 5.1 lists the different memory types and, for each type, shows where these fields are located. The type of a memory object is determined when it is created.

In order to use memory efficiently, the memory object does some of its own memory management. A large block of memory is allocated from the system in order to store data for trajectories. When this space has been filled with trajectory data, then another block is allocated from the system. If the memory object is unable to allocate a block of this size due to memory shortage, then a smaller chunk will be allocated instead. The size of these

Memory Type	Phase Space Data	Parameter Data	Color Data
TRAJ_MEMORY	body	header	header
MULT_MEMORY	body	body	body
SEL_PT_MEMORY	body	body	body
PARAM_MEMORY	—	body	body
FIXPT_MEMORY	body	header	header
CONT_MEMORY	body	body	body

Figure 5.1: Memory Types and Data Field Locations

blocks of data is specified when the memory object is first created.

The color coding information is an array of 3 integers. It contains the information necessary to redraw a point in one of the drawing canvases. The first element of the array contains the alternating color index. The second element contains the pick color index. Finally, the third element specifies the symbol which should be plotted. The available symbol types are in the file `constants.h` in the directory `$DSTOOL/src/include`.

The following sections describe the routines available to manipulate memory objects. The routines are separated into several categories. The first set are those which deal with creating and destroying memory objects. The second set of routines are for writing data into a memory object. The following set of routines are for reading data out of the memory object. There are two sets of read and write routines depending on whether the standard interpreted view or the vanilla view is required. For most applications, the standard ones will be used. The final set of routines are for modifying data in the memory object and for inquiring about properties of the memory object.

The concept of a current point is important for many memory routines. Each memory object maintains internally a record of a current location for write access, as well as a separately stored location for read access.

A number of example code fragments are also included as part of the descriptions of the memory routines. In order to use the memory routines, include the header file `memory.h` from the directory `$DSTOOL/src/include`.

5.1.1 Creating and Destroying Memory Objects

5.1.1.1 memory_create()

```
memory
  memory_create(type, size, def_traj_length, varb_dim, param_dim, color_dim)
int type, size, def_traj_length, varb_dim, param_dim, color_dim;
```

This procedure creates a new memory object. The `type` field specifies the type of data which the memory object will contain as described earlier. The `size` field specifies the number of bytes which will be allocated from the system at a time for point storage. The default trajectory length is specified by `def_traj_length`. The dimensions of the phase space, parameter space and color arrays are given by `varb_dim, param_dim` and `color_dim`, respectively. The usual DsTool value of color_dim is defined as the constant COLOR_DIM. If the creation was successful the memory object is returned, otherwise a NULL pointer is returned.

5.1.1.2 memory_destroy()

```
int
  memory_destroy(m)
memory m;
```

This routine accepts a memory object and frees all memory associated with it. The memory object must not be used after this call.

5.1.2 Writing to Memory Objects

5.1.2.1 memory_start_new_flow()

```
int
  memory_start_new_flow(m, n_trajs, n_extra_dvars, n_extra_ivars, n_points,
                           n_extra_dparams,n_extra_iparams)
memory m;
int n_trajs, n_extra_dvars, n_extra_ivars, n_points,
    n_extra_dparams, n_extra_iparams;
```

This routine is called to allocate space for a new flow inside a memory object. The number of trajectories in the flow is specified by n_trajs. It will use the memory type and known dimensions to decide how to allocate space. Space for additional doubles and integers to be stored with each point can be requested by specifying n_extra_dvars and n_extra_ivars. Additional double and integer parameters to be stored in the flow object are specified by the n_extra_dparams and n_extra_iparams. The expected number of points per trajectory is specified by n_points. If this routine fails, then a non-zero integer is returned.

5.1.2.2 memory_end_current_flow()

```
int
  memory_end_current_flow(m)
memory m;
```

This routine instructs the memory object that the current flow is complete. Any unused space allocated for the flow will be freed up for use by future flows. This routine is called automatically by the routines which start new flows.

5.1.2.3 memory_add_point()

```
int
  memory_add_point(m, p_varb, p_param, p_color, p_extra_doubles, p_extra_ints,
                  p_extra_dparams, p_extra_iparams)
memory m;
double *p_varb, *p_param, *p_extra_doubles, *p_extra_dparams;
int *p_color, *p_extra_ints, *p_extra_iparams;
```