## 2.13 The Function Settings Window

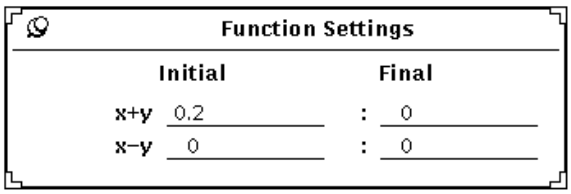


Figure 2.14: The Function window.

**Window title:** Function Settings

**Function:** This is the window which displays the value of the user-defined functions and the values of alternative coordinates.

**Description:** The Function Settings Window is opened by selecting the Function option from the Settings menu button located on the Command Window. This window simply displays information to the user. The initial column displays the values of the functions which correspond to the initial point of the last trajectory. The final column displays the values of the functions at the final point of the last plotted trajectory.

**Panel items:**

- *Functionname* Initial read-only text field: Displays the value of the function at the initial point of the last computed trajectory.
- *Functionname* Final read-only text field: Displays the value of the function at the final point of the last computed trajectory.

## 2.14 The Defaults Settings Window

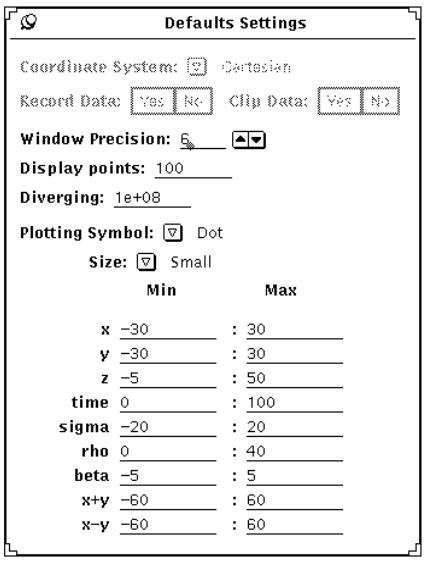


Figure 2.15: The Default Settings Window.

**Window title:** Defaults Settings

**Function:** This window displays the current default ranges for variables, parameters and functions. It also allows the user to turn on and off data clipping and the recording of data. Certain global values (used in DsTool's numerical or display algorithms) can be set via this window.

**Description:** The Defaults Settings Window is opened by selecting the Defaults option from the Settings menu button located on the Command Window. If the record data is off then no data is stored. When record data is on then, if clipping is off, all produced data is stored. When data clipping is on, only data which is displayed in at least one view window (iconized or not) is stored. (This button does nothing if recording data is off.) The other text fields are the default viewing ranges for displaying data.

**Panel items:**

- Coordinate System stack setting: *Not yet implemented.* Allows the user to select a coordinate system. This item is only active when more than one coordinate system has been defined. Presently the only choices are Cartesian and Polar.
- Record Data exclusive setting: Allows the user to turn data recording on and off. The default is on.

- Clip Data exclusive setting: *Not yet implemented.* Allows the user to turn data clipping on and off. The default is off. This setting is inactive if Record Data is turned off.
- Window Precision numeric field: Allows the user to display from 0 to 17 digits of precision in window text fields. The default value is 6. The value of this field determines the precision of the number of decimal places displayed in the DsTool window text entries.
- Display Points read-write text field: During calculations of orbits, DsTool will calculate $n$ points of the orbit before sending the data to be plotted and stored in memory. The integer $n$ is determined by the value in this field. The default is 100.
- Diverging read-write text field: During calculations of orbits, DsTool will stop calculations if orbits appear to diverge to infinity. If the distance from a point to the origin is greater than the number appearing in this field, then the orbit containing that point is deemed to have diverged. The default value is $10^8$.
- Plotting Symbol stack setting: Trajectories are plotted in the symbol determined by this setting. The default is to plot a dot for each point of a trajectory. It is also possible to choose to plot with a cross, a triangle, and a box.
- Size stack setting: Trajectory symbols come in four sizes: small, medium, large, and huge. The default size is medium.
- *Name* Min read-write text field: Shows and allows the user to adjust the default minimum value for viewing this variable, parameter, or function.
- *Name* Max read-write text field: Shows and allows the user to adjust the default high value for viewing this variable, parameter, or function.

# Chapter 3

# User-Defined Dynamical Systems

In most instances, we expect that DsTool will be installed on a network of computers and used by a variety of researchers. It is not required, however, that each user on a network utilize the same executable version of DsTool. We have provided ways in which each user may customize certain features of DsTool to suit his or her needs. An example of DsTool customization is the installation of a user-defined dynamical system. In this chapter we will detail this process; other ways to customize DsTool are discussed in later chapters.

## 3.1 Preliminaries

We presume DsTool is installed according to the instructions provided in the DsTool installation information. This procedure includes

1. Defining `UNIX` environmental variables `DSTOOL`, `DSTOOL_COLOR_DIR`, and `DSTOOL_DATA_DIR`.
2. Creating a local directory for DsTool.
3. Creating a subdirectory for user-defined models.
4. Copying certain files from the DsTool source code into the above subdirectory.

The last two items are completed automatically by running the shellscript `install_dsuser`. Be certain that these steps are complete before proceeding.

## 3.2 Defining Dynamical Systems

The addition of a new dynamical system in DsTool is a two-step process. The first step entails writing a few procedures which define the set of governing equations for the dynamical system (be it a vector field or a mapping) and the initial settings of variables and parameters. If desired, additional procedures may be written which define derivatives (with respect to space, time, and parameters) and define an arbitrary number of auxillary scalar-valued functions. The second step in the process is to install the procedures into the libraries used to construct the executable version of DsTool. To help you complete the necessary steps, we provide the following checklist:

1. Define the dynamical system
   (a) Equations of motion
   (b) Derivatives
   (c) Inverse
   (d) Auxillary equations
   (e) Names and default ranges for variables
   (f) Names and default ranges for parameters
   (g) Names and default ranges for auxillary functions
   (h) Periodic variables
   (i) Names of user-defined functions
2. Install the dynamical system
   (a) Initialization routine
   (b) Title
   (c) Compile source code

The discussion which follows contains details of this process and a description of the files and variables involved. If you are not already there, change directories to your local models directory.

### 3.2.1 Defining the Equations of Motion

We shall use as an example a two-dimensional mapping $f = f_{\alpha,\gamma}$ defined by

$$\begin{aligned} \phi_{j+1} &= \phi_j + v_j, \\ v_{j+1} &= \alpha v_j - \gamma \cos(\phi_j + v_j). \end{aligned} \tag{3.1}$$

These equations are a model for the dynamics of a ball bouncing on a sinusoidally vibrating table. The variable $\phi$ is proportional to the time, but because of the sinusoidal motion of the table, we may think of $\phi$ as belonging to the circle parametrized by $[0, 2\pi)$. The variable $v$ is proportional to the ball's velocity immediately after contact with the table. The parameter $\alpha$, $0 < \alpha \leq 1$ is the coefficient of restitution for the ball; the parameter $\gamma > 0$ is the amplitude of the table oscillations. Essentially, the equations say that given the time of the $j$th impact ($\phi_j$) and the ball's velocity at that time ($v_j$), we know the time that the ball will next strike the table ($\phi_{j+1}$), as well as the ball's velocity immediately after its next bounce ($v_{j+1}$). See [1] for information about the dynamics of this system.

Suppose that a user wishes to study Equation 3.1 numerically using DsTool. The first task is to define the equations. First, copy the file `GENERIC.c` to the file that you want to edit. For this example, call the target file `bball_def.c`. You can use any text editor to edit `bball_def.c`. The beginning of the file looks like:

```
#include <model_headers.h>

/* ------------------------------------------------------------------------
   function used to define the vector field or map
   ------------------------------------------------------------------------ */
int user_ds_func(f,x,p)
double *f,*x,*p;
{
}
```

First change the name of this function from user_ds_func() to bball(). Then edit the function so that it properly defines the mapping. When you are finished, the function should look like:

```
#include <model_headers.h>

/* ------------------------------------------------------------------------
   function used to define the map
   ------------------------------------------------------------------------ */
int bball(f,x,p)
double *f,*x,*p;
{
   double cos();

   f[0] = x[0] + x[1];
   f[1] = p[0] * x[1] - p[1] * cos(x[0] + x[1]);
}
```

The mapping is now defined. We remind novice C-programmers that arrays in C are indexed from 0. When bball() is called, the calling routine is expected to pass in arrays `f`, `x`, and `p`. The input variables are the current state of variables (`x`) and the current parameters (`p`):

$$\begin{aligned} \texttt{x} &= \{\texttt{x[0]}, \texttt{x[1]}\} = \{\phi_j, v_j\}, \\ \texttt{p} &= \{\texttt{p[0]}, \texttt{p[1]}\} = \{\alpha, \gamma\}. \end{aligned}$$

The function bball() places the new state in the array `f`:

$$\texttt{f} = \{\texttt{f[0]}, \texttt{f[1]}\} = \{\phi_{j+1}, v_{j+1}\}.$$

This function *must* be defined in order to begin exploration of dynamics. If the user does not wish to input information about the system's derivative or inverse, and if no auxiliary functions are desired, the user could proceed to Section 3.2.5. For the purpose of illustration, however, we encourage the reader to continue reading.

We remark at this point that although this system is a two-dimensional mapping, the value $x[2]$ contains the current "time." In general, if there are $k$ dependent variables, then `x[0]`, ..., `x[k-1]` contain these variables and `x[k]` contains the independent variable. This is true both for maps and for vector fields.

### 3.2.2 Defining Derivative Information

If available, DsTool can use analytic information in certain computational routines. Jacobian information is used in root-finding algorithms and for implicitly iterating diffeomorphisms backwards (when an explicit inverse does not exist). Detailed information about these algorithms is found in the DsTool Reference Manual.

If derivative information is not provided by the user, then DsTool will use finite difference-approximations. This section will describe how to write a function which provides an explicit Jacobian for Equation 3.1. Because time is discrete for maps, it does not make sense to define a derivative with respect to time. Currently, DsTool does not make use of derivatives with respect to time and parameters, but we include them for the convenience of the user who wishes to extend the capabilities of DsTool.

We now continue with the bouncing ball example by defining the Jacobian of $f$. At the $j$th instant of time, the Jacobian is

$$\begin{pmatrix} \partial f_1/\partial \phi_j & \partial f_1/\partial v_j \\ \partial f_2/\partial \phi_j & \partial f_2/\partial v_j \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ \gamma \sin(\phi_j + v_j) & \alpha + \gamma \sin(\phi_j + v_j) \end{pmatrix}$$

Find the section of the file `bball_def.c` which reads

```
/* ------------------------------------------------------------------------
   function used to define the Jacobian
   ------------------------------------------------------------------------ */
int user_jac(m,x,p)
double  **m, *x, *p;
{
}
```

Using a text editor, modify this code segment to read

```
/* ------------------------------------------------------------------------
   function used to define the Jacobian
   ------------------------------------------------------------------------ */
int bball_jac(m,x,p)
double  **m, *x, *p;
{
   double    sin(), temp;

   temp    = p[1] * sin( x[0] + x[1] );
   m[0][0] = 1.0;
   m[0][1] = 1.0;
   m[1][0] = temp;
   m[1][1] = p[0] + temp;
}
```

The routine which calls bball_jac() sends in a matrix and two arrays which contain the current state and the current parameters for $f_{\alpha,\gamma}$. When bball_jac() returns, it has filled the matrix with the numerical Jacobian for $Df_{\alpha,\gamma}$ evaluated at the current state. As remarked previously, writing a Jacobian routine is optional.

### 3.2.3 Defining Information about an Inverse

In this section we show how to define an inverse or an approximate inverse for a mapping. We recall that the "inverse function" for a vector field is trivial, since integrating the equations of motion backwards in time is the same as integrating forward in time—except we pass in a negative time step to a numerical integrator. Thus if we were installing a vector field, we would skip this section. Moreover, it frequently happens that a mapping does not have an inverse, nor is there any sort of an approximate inverse. In that case, do not edit the inverse routine at all.

For diffeomorphisms, it is sometimes difficult to iterate backwards in time. The Reference Manual discusses Newton's method and implicitly iterating a mapping backwards; it also briefly indicates some of the ways that Newton's method can fail. One of the most important aspects of Newton's algorithm is its dependence on a "good" initial guess. In the present section, we will discuss one way to provide a good guess.

For our bouncing ball example, it turns out that we can find an explicit inverse for $f$ since $\alpha > 0$. This inverse is the map given by

$$\begin{aligned} \phi_{j-1} &= \phi_j - \tfrac{1}{\alpha}(\gamma \cos \phi_j + v_j), \\ v_{j-1} &= \tfrac{1}{\alpha}(\gamma \cos \phi_j + v_j). \end{aligned} \tag{3.2}$$

To enter this inverse into DsTool, locate the section of `bball_def.c` which looks like

```
/* ------------------------------------------------------------------------
   function used to define inverse or approximate inverse
   ------------------------------------------------------------------------ */
int user_inv(y,x,p)
double    *y,*x,*p;
{
}
```

and modify this section to produce

```
/* ------------------------------------------------------------------------
   function used to define inverse
   ------------------------------------------------------------------------ */
int bball_inv(y,x,p)
double    *y,*x,*p;
{
   double     cos(), temp;

   temp = ( p[1] * cos( x[0] ) + x[1] ) / p[0];
   y[0] = x[0] - temp;
   y[1] = temp;
}
```

Suppose that our map did not have an explicit inverse (or we could not calculate it). Then we would have to resort to using Newton's method to numerically solve for the inverse iterate at each backward step. To do so requires that we provide an initial guess for the inverse image of the current state. If we provide a good guess for the inverse image, then possibly we will require only a few Newton steps to "sharpen" our guess and converge to

a solution (See the Reference Manual). Suppose, for the sake of an example, that we were only interested in the bouncing ball map $f_{\alpha,\gamma}$ in the special case that $\gamma$ is small. Then $f$ can be thought of as a perturbation of the linear map $g$ defined by

$$\begin{aligned} \phi_{j+1} &= \phi_j + v_j, \\ v_{j+1} &= \alpha v_j, \end{aligned} \tag{3.3}$$

and $g^{-1}$ can be written as:

$$\begin{aligned} \phi_{j-1} &= \phi_j - \tfrac{1}{\alpha} v_j, \\ v_{j-1} &= \tfrac{1}{\alpha} v_j. \end{aligned} \tag{3.4}$$

In this case, we could code $g^{-1}$ into DsTool in place of $f^{-1}$. If we were to do this, we would modify `bball_def.c` to read:

```
/* ------------------------------------------------------------------------
   function used to define approximate inverse
   ------------------------------------------------------------------------ */
int bball_approx_inv(y,x,p)
double    *y,*x,*p;
{
   y[0] = x[0] - x[1] / p[0];
   y[1] = x[1] / p[0];
}
```

Of course, we must inform DsTool whether to interpret the procedure as a definition of the actual inverse or merely as an approximation; Section 3.2.5 explains how this is done.

### 3.2.4 Defining Auxiliary Functions

We continue with the above example. Suppose that there is some scalar function of the phase space variables, time, and parameters which we wish to monitor. The scalar function may be a physically meaningful quantity such as the total energy of an (almost) conservative system, or it could be a mathematically interesting quantity such as the largest eigenvalue of the Jacobian matrix. Sometimes it is useful to have functions which represent coordinate transformations. For example, a particular dynamical system may be best described in terms of spherical coordinates. Auxiliary functions can also be used to monitor the distance from some set of interest. For example, if a system spends most of its time close to the sphere $|x| = 1$ then it may be interesting to define a function $|x|$.

For our bouncing ball example, we will monitor the quantity $v^2$. Since $v_j$ is proportional to the ball's velocity just after it strikes the table for the $j$th time, we can think of $v_j^2$ as being a measure of the ball's kinetic energy at time $j$. To define this function, find the location in the file `bball_def.c` which reads

```
/* ------------------------------------------------------------------------
   function used to define aux functions of the varbs, time, or params
   ------------------------------------------------------------------------ */
int user_aux_func(f,x,p)
double *f,*x,*p;
{
}
```

and edit it to produce

```
/* ------------------------------------------------------------------------
   function used to define aux functions of the varbs, time, or params
   ------------------------------------------------------------------------ */
int bball_aux(f,x,p)
double *f,*x,*p;
{
   f[0] = x[1] * x[1];
}
```

The number of auxiliary functions is completely arbitrary; it is not necessary to have any auxiliary functions. When this equation is called, the calling routine passes in the current state of the phase space variables (in `x`), the current parameters (in `p`), and an array (`f`) which is filled up by this function.

### 3.2.5 Defining Labels and Initial Conditions

This section explains how to complete the definition of a dynamical system by giving DsTool information about the structure of phase space, the names of variables, and initial conditions.

This section is split into subsections. The subsections describe how to provide information on variables, parameters, auxiliary functions, periodic variables, and numerical algorithms.

Before we begin, find the code segment of `bball_def.c` which looks like

```
/* ------------------------------------------------------------------------
   Procedure to define default data for the dynamical system. NOTE: You
   may change the entries for each variable but DO NOT change the list of
   items.  If a variable is unused, NULL or zero the entry, as appropriate.
   ------------------------------------------------------------------------ */
int user_init()
{
/* ------------ define the dynamical system in this segment --------------- */
...
```

and change the name of the function from user_init() to bball_init().

#### 3.2.5.1 Variables

For our example, we have two spatial phase space variables, namely $\phi$ and $v$. Because the equations of motion (Equation 3.1) are invariant under the coordinate transformation $\phi \to \phi + 2n\pi$, $n \in \mathbb{Z}$, it is natural to display $\phi$ on the interval $[0, 2\pi]$. There is no "natural" interval to use in displaying the $v$ coordinate, since $v$ can be any real number, but we will choose the interval $[-30, 30]$ as a default range on which to display $v$. We will also need to choose a default initial condition $(\phi_0, v_0)$, which we arbitrarily select to be the origin, $(0, 0)$.

After implementing these choices, the relevant code in bball_init() looks like:

```
static int     n_varb=2;                      /* dim of phase space          */
static char    *variable_names[]={"phi","v"}; /* list of phase varb names    */
static double  variables[]={0.,0.};           /* default varb initial values */
```