```
static double  variable_min[]={0.,-30.};      /* default varb min for display */
static double  variable_max[]={TWOPI,30};     /* default varb max for display */
```

We remark the `TWOPI` (2π) and `PI` (π) are two constants which the user may use in defining a dynamical system.

Although we have defined labels and initial values for the *spatial* variables, the independent variable (usually thought of as time) is also considered to be a member of every phase space. The code which provides this information is given by:

```
static char    *indep_varb_name="time";  /* name of indep variable            */
static double  indep_varb_min=0.;        /* default indep varb min for display */
static double  indep_varb_max=10000.;    /* default indep varb max for display */
```

In fact, this is the way the code looked when we copied it from `GENERIC.c`, so we do not need to make any changes to the code. If we wanted to call the independent variable "iteration" instead of "time," or if we wanted to change the default plotting range, then the code segment above would have to be appropriately modified.

3.2.5.2 Parameters

When examining bifurcations of our bouncing ball map, it is useful to graphically depict (a subset of) the parameter space. Since $0 < \alpha \leq 1$, we choose the interval $[0, 1]$ as the default range to display α. For γ, we choose $[0, 25]$ as a default range. As initial parameter values, we choose $(\alpha, \gamma) = (0.8, 10)$, since the dynamics for these parameter values are fairly interesting.

To implement these choices, we edit a few more lines in bball_init() so that the code looks like:

```
static int     n_param=2;                /* dim of parameter space          */
static char    *parameter_names[]={"alpha","gamma"}; /* list of param names */
static double  parameters[]={0.8,10.};   /* initial parameter values        */
static double  parameter_min[]={0.,0.};  /* default param min for display   */
static double  parameter_max[]={1.,25.}; /* default param max for display   */
```

3.2.5.3 Auxiliary Functions

For our example, we have only a single function which we will call "KE" for kinetic energy. From the definition of the function (v^2) and from the default plotting range for the variable v, the interval $[0, 1000]$ is probably a suitable range on which to plot the value of v^2. Accordingly, we edit a few more lines in bball_init():

```
static int     n_funct=1;                /* number of user-defined functions */
static char    *funct_names[]={"KE"};    /* list of funct names; {""} if none*/
static double  funct_min[]={0};          /* default funct min for display    */
static double  funct_max[]={1000};       /* default funct max for display    */
```

We remark that if we did not want to monitor any auxiliary functions then we would set `n_funct=0`. The array of function names, however, must contain at least an empty string or else our code will not compile properly. In other words, if there were no auxiliary quantities of interest, then we could write `*funct_names[]=""` but em not `*funct_names[]=`.

3.2.5.4 Periodic Variables

We must tell DsTool whether our phase space contains any periodic variables. The coding of this information is accomplished through two C-language constants: `EUCLIDEAN` or `PERIODIC`. If there are no periodic variables, then the phase space is `EUCLIDEAN`; if there is at least one periodic variable, the phase space is classified as `PERIODIC`. For `PERIODIC` phase spaces, we must also inform DsTool as to which variables are periodic and what interval to use as a fundamental domain.

For our example, since the equations of motion are invariant under the transformation $\phi \to \phi + 2n\pi$ for any integer n, we may consider ϕ to be a periodic variable with period 2π. We may choose *any* interval of length 2π as a fundamental domain for the variable ϕ. Common choices are the intervals $[-\pi, \pi]$ and $[0, 2\pi]$. We make the latter choice. To pass this information into DsTool, we edit yet a few more lines in bball_init():

```
static int    manifold_type=PERIODIC;   /* EUCLIDEAN or PERIODIC               */
static int    periodic_varb[]={TRUE, FALSE}; /* if PERIODIC, which varbs periodic? */
static double period_start[]={0.,0.};   /*if PERIODIC, begin fundamental domain */
static double period_end[]={TWOPI, 1.}; /*if PERIODIC, end of fundamental domain*/
```

We remark on the variables `period_start` and `period_end`. If the jth coordinate is not periodic (ie, the value of `periodic_varb`$[j]$ is `FALSE`) then it does not matter what `period_start`$[j]$ and `period_end`$[j]$ are because the entries are ignored by DsTool. Similarly, if the variable `manifold_type` is `EUCLIDEAN`, then it doesn't matter what values are given for the entries of `periodic_varb`. It is always safe, of course, to set each entry of `periodic_varb` to `FALSE`. As mentioned in Section 3.2.5.1, `TWOPI` is a global constant.

3.2.5.5 Numerical Algorithms

Continuing with our example, there remain a few pieces of information which we have not yet entered into DsTool. We modify the last lines of code in bball_init() to obtain:

```
static int mapping_toggle=TRUE;         /* this is a map? TRUE or FALSE         */
static int inverse_toggle=EXPLICIT_INV; /* if so, is inverse FALSE, APPROX_INV, */
                                        /* or EXPLICIT_INV? FALSE for vec field */

/*  In this section, input NULL or the name of the function which contains...   */
int     (*def_name)()=bball;            /* the eqns of motion                   */
int     (*jac_name)()=bball_jac;        /* the jacobian (deriv w.r.t. space)    */
int     (*aux_func_name)()=bball_aux;   /* the auxiliary functions              */
int     (*inv_name)()=bball_inv;        /* the inverse or approx inverse        */
int     (*dfdt_name)()=NULL;            /* the deriv w.r.t time                 */
int     (*dfdparam_name)()=NULL;        /* the derivs w.r.t. parameters         */
```

The first line means that our system is a mapping; if it were a vector field, we would set `mapping_toggle` to `FALSE`. The second line means that we have an explicit inverse for our system. In the event that a mapping does not have an explicit inverse, then Newton's method will be used to calculate inverse iterates. In this event, we need to tell DsTool if there exists an approximate inverse (see Section 3.2.3 or the Reference Manual) which can be used as an initial guess for Newton's method. If there is, then `inverse_toggle` is set to `APPROX_INV`; otherwise, it is set to `FALSE`. We remark that if `mapping_toggle` is set to

FALSE (that is, the system is a vector field), then the value of inverse_toggle is ignored by DsTool.

In the last section of code, we provide DsTool with the names of any functions we defined previously. For our example, this means filling in the names of the functions which compute the equations of motion, the jacobian, auxiliary functions, and the inverse. If any of these functions were omitted, then the user should enter NULL instead of a name. For example, the function which defines the inverse will be NULL if you are installing a vector field, and the function which defines the derivative with respect to time should always be NULL for mappings.

At this point, the user should save and exit the file bball_def.c.

3.2.6 Installing a Defined Dynamical System

It is assumed that the reader has completed the steps in the preceding sections of this chapter. In this section, we describe the installation of the dynamical system defined using the above steps. As an illustration, we continue with the bouncing ball example.

Our first task is to modify the file modellib_ds_def.c which should be found in the user's local models directory. There are two steps to this process:

- Inform DsTool about the name of the initialization routine for the dynamical system we desire to install. In our bouncing ball example, this function is bball_init().
- Enter this name into the DsTool list of dynamical systems, along with a title.

During the execution of DsTool, the user may select one of several installed dynamical systems. The program maintains a list of installed models, along with the procedures which initialize and define them. This list is maintained in the file modellib_ds_def.c. This file contains three blocks of C-code: the first block defines an array of categories used to collect dynamical systems into related classes; the second consists of several lines beginning with the declaration extern int; the last block is a C-language structure which contains titles for the models along with names of the models' initialization procedures. The modellib_ds_def.c delivered with DsTool contains the following code:

```
/* This is a configuration file for dynamical systems */
#include <modellib.h>
struct  DS_DataS DS_model;
char    *DS_Category[] = { "Standard Mappings",         /* category 0 */
                           "Standard Vector Fields",    /* category 1 */
                           "Special Dynamical Systems"};/* category 2 */

/* ------------------------------------------------------------------ */
/*   declare names of initialization routines                         */
/* ------------------------------------------------------------------ */
extern int      lorenz_init(), cubic_init(), vdpol_init();
extern int      standard_init(), ko_init(), henon_init();
extern int      hh_init(), d4_init(), pfork_init();

/* ------------------------------------------------------------------ */
/* fill structure according to: { category, title, init_routine }     */
/* ------------------------------------------------------------------ */
struct DS_DataS DS_Sel[] = {
```

```
      { 0, "Standard Map", standard_init },
      { 0, "Kim-Ostlund Torus Map", ko_init },
      { 0, "Complex Henon Map", henon_init },
      { 1, "Lorenz System", lorenz_init },
      { 1, "Planar Cubic Vector Field", cubic_init },
      { 1, "Forced Van der Pol Eqns", vdpol_init },
      { 2, "D4 Symmetric System", d4_init },
      { 2, "Hodgkin Huxley Equations", hh_init },
      { 2, "Pitchfork NF Vector Field", pfork_init }
    };

/* ------------------------------------------------------------------ */
/* do not edit beyond this line                                       */
/* ------------------------------------------------------------------ */
int     DEFAULT_SYSTEM = 0;
int     N_DS = sizeof(DS_Sel) / sizeof(struct DS_DataS);
int     N_DS_Category = sizeof(DS_Category) / sizeof(char *);
```

To tell DsTool the name of the initialization routine which defines our bouncing ball model, we need to add the line

```
extern int  bball_init();
```

to the second block of code in `modellib_ds_def.c`. To install our model, we add the line of code

```
{2, "Bouncing Ball", bball_init},
```

to the definition of the data structure `DS_Sel` within the third block of code. The variable `DS_Sel` is an array, each element of which is a data structure which defines a dynamical system. The first element of the structure is a number which specifies the category to which the dynamical system belongs. The category is used to group together systems which share similar properties. The default list of categories is defined by the lines in the first block of code:

```
char    *DS_Category[] = { "Standard Mappings",         /* category 0 */
                           "Standard Vector Fields",    /* category 1 */
                           "Special Dynamical Systems"};/* category 2 */
```

This array provides the title for each category. For our example, the category index is 2, so our bouncing ball model belongs to the category named "Special Dynamical Systems." (Word to novice C programmers: remember that arrays in C are indexed from 0 so `DS_Category`[2] is not the second title, but is actually the third.) Every dynamical system must belong to a valid category.

The next element of the dynamical system data structure is the title of the system being defined. We have chosen "Bouncing Ball" as the title of our system. Any descriptive title will do, provided that the length of the title is not larger than the global constant `MAX_LEN_DS_TITLE`. The value of this constant is found in the file `$DSTOOL/src/include/constants.h`.

The last element is the name of the function which contains the initialization routine for the new dynamical system. For our example, the name is `bball_init`.

Notice that each dynamical system data structure is enclosed by braces and is followed by a comma. That is, all except the last. The last element in the array of dynamical systems

(the system entitled "Pitchfork NF Vector Field" in our example code) does not require a trailing comma.

We are now ready to compile the source code we have written and to assemble the code into a library called `modellib.a`. To do this, edit the file `Makefile` in your models subdirectory. Add the file `bball_def.c` to the `SOURCES_A` list of files. In the example described above, the corresponding Makefile entry would be:

```
SOURCES_A =     \
                models.c \
                standard_def.c \
                kotorusmap_def.c \
                henon_def.c \
                lorenz_def.c \
                cubic_def.c \
                vdpol_def.c \
                bball_def.c  \
                d4_def.c  \
                hh_def.c \
                pitchfork_def.c
```

Now save and exit the `Makefile`. To build `modellib.a`, type `make modellib.a`. If the compiler reports errors, correct them by editing `bball_def.c` and attempt to build `modellib.a` again.

Once `modellib.a` is constructed without error, we can create a custom version of DsTool called `dstool_u`. To do this, change to your local DsTool directory and make a custom version of DsTool with the command

```
make dstool_u
```

If the file `dstool_u` is made, then this local (user) version of DsTool may be executed by the command

```
dstool_u
```

This version of DsTool will include the bouncing ball equations among its installed dynamical systems.

3.3 A Vector Field Example

In this section we present an example of installing a user-defined vector field into DsTool. It is assumed that the reader is familiar with the material in Section 3.2.

Our task is to install the vector field defined by the equations

$$\begin{aligned} \dot{u} &= v, \\ \dot{v} &= u - u^3 - \delta v - \gamma \cos \omega t. \end{aligned} \tag{3.5}$$

This system is known as Duffing's equations and we direct the reader to [1] and references therein for an exposition of the dynamics of this system.

To define and install this system, change to your local models directory and copy the file `GENERIC.c` to the file `duffing_def.c`. Use any text editor to edit `duffing_def.c`.

The segment of code which defines the equations of motion (Equation 3.5) will be called duffing(). This function is defined as follows:

```
/* ------------------------------------------------------------------------
   function used to define the vector field
   ------------------------------------------------------------------------ */
int duffing(f,x,p)
double *f,*x,*p;
{
   double  cos();

   f[0] = x[1];
   f[1] = x[0] - x[0]*x[0]*x[0] - p[0]*x[1] - p[1]*cos( p[2]*x[2] );
}
```

Here we have defined the variables and parameters as

$$\begin{aligned} \texttt{x} &= \{\texttt{x[0]}, \texttt{x[1]}\} = \{u, v\}, \\ \texttt{p} &= \{\texttt{p[0]}, \texttt{p[1]}, \texttt{p[2]}\} = \{\delta, \gamma, \omega\}. \end{aligned}$$

As usual, the independent variable is stored after the spatial variables so that `x[2]` is time. The function duffing() returns in the array `f` the strength of the vector field with parameters `p`, evaluated at the point `x`.

We now define the Jacobian of Equation 3.5 by writing the function duffing_jac():

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

   m[0][0] = 0.0;
   m[0][1] = 1.0;
   m[1][0] = 1.0 - 3.0 * x[0] * x[0];
   m[1][1] = -p[0];
}
```

Since our equations are for a vector field, we do not need to define an inverse function. Our vector field is time-dependent, so we can define a function which returns the temporal derivatives of the vector field. This function is not yet used by DsTool, and there is no template for such a function in `GENERIC.c`, but it is easy enough to write:

```
/* ------------------------------------------------------------------------
   function used to define temporal derivatives
   ------------------------------------------------------------------------ */
int duffing_dfdt(m,x,p)
double  *f, *x, *p;
{
   double    sin();

   f[0] = 0.0;
   f[1] = -p[1] * p[2] * sin( p[2] * x[2] );
}
```

Since our vector field is Hamiltonian for $\delta = 0$, we choose

$$H(u,v) = \frac{v^2}{2} - \frac{u^2}{2} + \frac{u^4}{4}$$

as an auxiliary function. We will also define a second auxiliary function. Since our vector field is periodic in time (with period $2\pi/\omega$), it is a common technique to look at the time-$2\pi/\omega$ map in order to better understand the dynamics. This "stroboscopic map" can also be thought of as a Poincaré map for the extended phase space. We are thus interested in the times t for which $\sin(wt + t_0) = 0$. Choosing $t_0 = 0$ we define the procedure duffing_aux() by

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

   temp = 0.5 * x[0] * x[0];
   f[0] = 0.5 * x[1] * x[1] - temp + temp * temp;
   f[1] = sin( p[2] * x[2] );
}
```

We are now ready to define the labels and initial conditions for Duffing's equations by writing the function duffing_init(). We choose $[-2, 2]$ as default plotting ranges for both u and v. Initializing the variables is straightforward:

```
int duffing_init()
{
/* ------------ define the dynamical system in this segment --------------- */

static int     n_varb=2;                 /* dim of phase space           */
static char    *variable_names[]={"u","v"}; /* list of phase varb names   */
static double  variables[]={0.5,0.5};    /* default varb initial values  */
static double  variable_min[]={-2.,-2.}; /* default varb min for display */
static double  variable_max[]={2.,2.};   /* default varb max for display */

static char    *indep_varb_name="time";  /* name of indep variable            */
static double  indep_varb_min=0.;        /* default indep varb min for display */
static double  indep_varb_max=1000;      /* default indep varb max for display */
```

Defining the parameter ranges is somewhat arbitrary. Sometimes it is difficult to tell *a priori* what range of parameters will provide interesting bifurcations. This is not a big problem since it is a trivial matter to change the range upon which a function (or parameter) is plotted once within DsTool. We choose $[0, 1]$ as a range for each parameter. We choose $(\delta_0, \gamma_0, \omega_0) = (0.25, 0.4, 1.0)$:

```
static int    n_param=3;                 /* dim of parameter space        */
static char   *parameter_names[]={"delta","gamma", "w"}; /* list of param names */
static double parameters[]={0.25,0.4,1.}; /* initial parameter values      */
static double parameter_min[]={0.,0.,0.}; /* default param min for display */
static double parameter_max[]={1.,1.,1.}; /* default param max for display */
```

The initialization of our two auxiliary functions is accomplished by the code segment:

```
static int     n_funct=2;              /* number of user-defined functions */
static char    *funct_names[]={"H", "P_Section"}; /* list of funct names; {""} if none */
static double  funct_min[]={-4.,-1.};   /* default funct min for display    */
static double  funct_max[]={4.,1.};     /* default funct max for display    */
```

As in the case with parameters, it is sometimes difficult to choose *a priori* what appropriate ranges for the functions should be.

The manifold type for Duffing's equations is `EUCLIDEAN` since we do not have any periodic spatial variables. Thus we do not need to modify the following segment of code:

```
static int    manifold_type=EUCLIDEAN; /* EUCLIDEAN or PERIODIC                */
static int    periodic_varb[]={FALSE, FALSE}; /* if PERIODIC, which varbs periodic? */
static double period_start[]={0.,0.};  /* if PERIODIC, begin fundamental domain */
static double period_end[]={1., 1.};   /* if PERIODIC, end of fundamental domain*/
```

The last segment of code we need to modify is the segment which tells DsTool which numerical algorithms may be used on the Duffing's equations. We complete the definition of duffing_init() with the code segment

```
static int  mapping_toggle=FALSE;       /* this is a map? TRUE or FALSE         */
static int  inverse_toggle=FALSE;       /* if so, is inverse FALSE, APPROX_INV, */
                                        /* or EXPLICIT_INV? FALSE for vec field */

/*  In this section, input NULL or the name of the function which contains...   */
int     (*def_name)()=duffing;          /* the eqns of motion                   */
int     (*jac_name)()=duffing_jac;      /* the jacobian (deriv w.r.t. space)    */
int     (*aux_func_name)()=duffing_aux; /* the auxiliary functions              */
int     (*inv_name)()=NULL;             /* the inverse or approx inverse        */
int     (*dfdt_name)()=duffing_dfdt;    /* the deriv w.r.t time                 */
int     (*dfdparam_name)()=NULL;        /* the derivs w.r.t. parameters         */
```

As in Section 3.2, we now need to edit two other files found in the models directory. We edit `Makefile` and add `duffing_def.c` to the list of files to compile. We also edit the file `modellib_ds_def.c` and add the lines

```
extern int  duffing_init();
```

and

```
{1, "Duffing's Eqns", duffing_init},
```

to the second and third blocks of code, respectively.

Changing directories to your local DsTool directory and typing `make dstool_u` will create a local executable of DsTool which includes Duffing's equations among its installed dynamical systems.

3.3.1 A Few Remarks on the Definition of Duffing's Equations

Recall that when we installed Duffing's equations as a time-dependent vector field, we defined $\sin\omega t$ as an auxiliary function and claimed that we could use it to study the time-$2\pi/\omega$ stroboscopic map. In theory, there is nothing wrong with this, however in practice we will encounter numerical errors in the evaluation of transcendental functions such as $\sin\omega t$ for large values of t. Since we are often interested in generating Poincaré maps for

extremely long times, and since the function $\cos \omega t$ also appears in the definition of our vector field, the user may want to extend phase space by introducing the variable θ. Then we can rewrite Duffing's equations in the form

$$\begin{aligned} \dot{u} &= v, \\ \dot{v} &= u - u^3 - \delta v - \gamma \cos \omega\theta, \\ \dot{\theta} &= 1, \end{aligned} \tag{3.6}$$

where $(u, v, \theta) \in \mathbb{R}^2 \times S^1$, and S^1 is the circle of length $2\pi/\omega$. That is, θ takes values in $[0, 2\pi/\omega)$. The problem with this formulation is that DsTool *cannot handle periodic variables whose length depends on a parameter!* To overcome this difficulty, we change coordinates via the transformation $\psi = \omega\theta$. Thus we could study Duffing's equations on extended phase space in the form

$$\begin{aligned} \dot{u} &= v, \\ \dot{v} &= u - u^3 - \delta v - \gamma \cos \psi, \\ \dot{\psi} &= \omega, \end{aligned} \tag{3.7}$$

where $(u, v, \psi) \in \mathbb{R}^2 \times S^1$, and S^1 is now the circle of length 2π.

The advantage to an extended phase space such as we have for Equation 3.7 is that it is trivial to plot Poincaré sections for this set of equations because we can make ψ a periodic variable. This allows us to request that DsTool only plot points when $\theta = \theta_0$ for some θ_0. In contrast, DsTool *never treats time as a periodic variable*, so we needed to define the auxiliary function $\sin \omega t$ in order to be able to generate a Poincaré map for Duffing's equations.

3.4 Deleting Dynamical Systems

Suppose that we want to remove the dynamical system entitled "D4 Symmetric System" which is defined in the file `d4_def.c`. To remove this:

1. Edit the `Makefile` in your local models subdirectory. Delete the name of the file which contains the dynamical system from the `SOURCES_A` list. In our example, we would remove the line `d4_def.c\` from the `Makefile`.

2. Edit the file `modellib_ds_def.c` and delete all references to the dynamical system you wish to remove. To remove the D4 Symmetric System, delete the lines

   ```
   extern int d4_init();
   ```

   and

   ```
   {2, "D4 Symmetric System", d4_init},
   ```

   from the file.

3. Remake `modellib.a` and `dstool_u` as explained in previous sections.

Acknowledgements

We would like to thank Swan Kim for writing "kaos," the precursor to DsTool. Without his efforts, DsTool might never have been written. We have profited greatly from our experiences with kaos.

We thank Al Thaler and the computational mathematics section of the NSF for their support. Also the Army Research Office and the Mathematical Sciences Institute financed some of the equipment used for development.

We would also like to thank Allen Back and Toby Driscoll for their work on DsTool. Both Allen and Toby have worked on developing a three-dimensional view window; this work is in progress. Allen has also written the tutorial and has made numerous helpful suggestions.

Helpful comments and suggestions have also come from other students and visitors to the Center for Applied Mathematics. In particular we thank Moshe Braner, Feng-Zhao and Rob Manning for their useful ideas and enthusiasm.