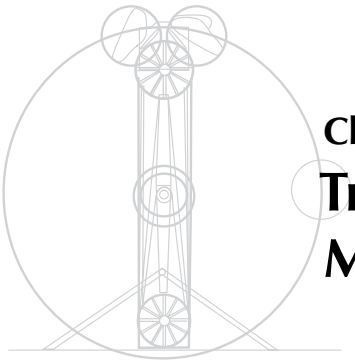

Chapter 9
Tricks and Advanced Methods

9.1 Introduction

In this chapter, I will describe some tricks and other advanced features that are not used that often but can be a great deal of help on those occasions when they are needed.

9.2 Graphics tricks

9.2.1 Better plots

Exporting data

If you want high quality graphics, say, for publication, with multiple graphs and figures within figures, then you should probably export the data to an ASCII file and use a plotting program such as Xmgr or Excel. Most of these programs will read a series of (x,y) values that are in space-delimited format in a file. For example, if you have an ODE file with two curves plotted on it, say $V(t)$, $w(t)$ as functions of t, then click on `Graphic stuff Export` and give a filename. The resulting file will have three columns t, V, w separated by spaces. You can read these in and plot them in your favorite graphing program.

You can export the entire simulation data set by clicking the `Write` button in the **Data Viewer**.

In the **AUTO Window** you can write data from the current view by clicking on `File Write pts`. *XPPAUT* can read this data and interpret it so that you can place a color bifurcation diagram into the *XPPAUT* main window for labeling, etc. The file consists of five columns of data `x,y1,y2,n,b`. The first three coordinates are the x coordinate of the current view, and the maximum and minimum y values, respectively. $n = 1, 2, 3, 4$ for stable fixed points, unstable fixed points, stable periodic orbits, or unstable periodic orbits, respectively. b is the branch number. More complete information about a bifurcation diagram is obtained by using the `File All info` command. The result is a series of rows of numbers with the following information: $(n, b, p_1, p_2, T, y_1^{hi}, \ldots, y_N^{hi}, y_1^{lo}, \ldots, y_N^{lo}, r_1, m_1, \ldots, r_N, m_N)$, where n is the type of point, b is the branch number, p_1, p_2 are the

active parameters, T is the period, y^{hi}, y^{lo} are all the max and min values of the coordinates of the system, and (r_j, m_j) are the real and imaginary parts of the eigenvalues at that point. So, for example, if you had a three-dimensional system and you wanted to plot the real part of the eigenvalues as a function of the main parameter, you would want to plot the third column p_1 and the 12th, 14th, and 16th columns r_1, r_2, and r_3.

You can also import data to *XPPAUT* using the `Read` button in the **Data Viewer**. Each column goes into the associated column in the **Data Viewer**. If there are more columns in the data file than in the **Data Viewer**, the first k columns will be read in, where k is the number of columns in the **Data Viewer**. I often read in data from PDE simulations so that I can use the animation features of *XPPAUT*.

Text, etc.

XPPAUT has some limited capabilities for adding text, lines, and symbols to your graph. These are found under the menu item `Text etc`. You can add text of five different sizes and two different fonts to your graph. The fonts are Times Roman and Symbol. The latter allows you to add Greek letters, but you have to know the corresponding Latin letters, e.g., $a = \alpha$ is obvious, but $q = \theta$ is not so obvious! You can mix fonts and have sub- and superscripts in the text that you write. To do this, use the following five escape sequences:

0 Use the normal Times Roman font

1 Use the Symbol font

s Subscript

S Superscript

n Normal (no sub- or superscripts)

{ `expr` **}** evaluate the expression `expr` before rendering the text.

Thus if you type in `\1q\0\sj\n = a+b\S2\n+\1b`, then the output will be

$$\theta_j = a + b^2 + \beta$$

since "q" becomes θ in the Symbol font and "b" becomes β.

If you want a text expression to be evaluated and permanently set so that when the quantity `expr` changes, the text will not, then preface the string with the % symbol. For example, consider the two text strings `%I=\{iapp}` and `I=\{iapp}`. Both will appear the same. However, if you change the parameter `iapp`, then the second one will reflect this change while the first remains frozen with the original value.

An arrow is drawn by clicking the mouse at a point, moving the mouse with the button held down, and then releasing the button. The first point is the tip of the arrow and the second determines the direction and size. You will have to play around with this until you are happy with it, as the relationship between the `size` and what is drawn is obscure. The `Pointer` option lets you draw straight lines with arrowheads. Choose size 0 for no arrowheads. The `Marker` option lets you draw a variety of symbols all centered at the position of the mouse when you click it. The `markerS` option lets you mark points along

the trajectory you have drawn. In addition to size, shape, and color, you should choose the starting row (row 0 starts at the beginning of the trajectory, row 1 starts at the next output point, etc.), the number of markers, and the number of rows to skip between each marker. You can alter the little graphics objects or delete them altogether from this menu item.

Linetype

In the `Graphic stuff` `Add curve` or `Edit curve` dialog box, you can choose both a color and a linetype. Linetype 1, the default, is a solid line. Linetype 0 draws a single point instead of a line. However, a negative line type, say, -3, will draw a small circle of radius 3 points at each point in the trajectory. Negative linetypes are useful for displaying periodic orbits of maps since they are easier to see on the screen than single points.

Problems with three-dimensional plots

Sometimes, three-dimensional plots seem to cut off the axes. You can rectify this by clicking on the `Window` `Window` command and making the window for `xlo,xhi,ylo,yhi` a bit bigger.

9.2.2 Plotting results from range integration

XPPAUT is a computer program. It is thus pretty dumb and can be tricked into doing what you want it to even though it was not meant to. Suppose you have a system of equations and you want to keep the results from a range integration (`Initialconds` `Range`). If you have fewer than 26 different curves, then you can click on `Graphic` `stuff` `Freeze` `On` `freeze` and then do the range integration. Each curve will be kept and can be plotted or output via PostScript. This is probably the best way to keep the results.

There is another way which doesn't limit the number of curves and does not use up the frozen curves. Here is the trick. In the ODE file, make the maximum amount of storage large, e.g., in the ODE file, add the line `@ maxstor=20000` so that 20,000 points will be stored rather than the default 4000. Run the file. In the `Initialconds` `Range` dialog box, type in `No` in the `Reset` `storage` box and run the range integration. All the data from each of the runs is concatenated into one long stream. The problem is that when you plot the results, a line from the end of one run is drawn to the beginning of the next. This can look quite ugly. There are two ways to avoid this. The first is to make the linetype 0 or negative. But this is not completely satisfactory since it generates just points instead of nice solid lines. The second is to take advantage of a plotting trick that *XPPAUT* uses. When *XPPAUT* plots solutions to equations defined on a torus, it "knows" that there is a jump from T to 0, where T is the period of the torus. (See section 9.5.4.) As a consequence, it looks for jumps and will not plot lines joining points that are more than some fraction of this apart in distance. Thus, look at your graph and estimate how big the jump is from the end of one run to the beginning of the next. Say, for example, you are plotting many runs against time and the time interval is 20. **AFTER** you have done your integration click on `phAsespace` and choose `All`. For the `Period` choose a number slightly less than this jump. *XPPAUT* will see that the distance between these two points is larger than the "period" and will not plot a line between them. Furthermore, the PostScript plots will also not have this jump.

9.3 Fitting a simulation to external data

XPPAUT is able to import external data, solve an initial value problem, and use a least-squares approach to find the best values of initial data and parameters to match the data. It does all this by using the Marquardt–Levenberg algorithm (see Press et al. [28]). Note that the implementation I have used here is somewhat restrictive and every parameter and initial condition is given equal weight. Furthermore, there are no constraints on the parameters or initial data. I will give an example of how to use this on some model data from an enzymatic reaction. The differential equations are

$$c' = k_1(a_0 - c - 2d)(b_0 - c - 2d) - k_2 c - 2(k_3 c^2 - k_4 d),$$

$$d' = k_3 c^2 - k_4 d,$$

and the data present 11 equally spaced values of (c, d) at $t = 0, \ldots, 70$. The data file is called `mod.dat` and here it is:

```
0     0         0
7     1.065     0.0058
14    1.383     0.2203
21    0.9793    0.4019
28    1.107     0.3638
35    0.7289    0.456
42    0.7236    0.5014
49    0.4674    0.715
56    0.6031    0.4723
63    0.6149    0.7219
70    0.3369    0.7294
```

Suppose that you have a vague idea of what the parameters are but want to do better. Here is the ODE file for this problem, `kinetics.ode`:

```
# kinetics.ode
# simple enzyme model to fit some data
c'=k1*(a0-c-2*d)*(b0-c-2*d)-k2*c-2*(k3*c*c-k4*d)
d'=k3*c*c-k4*d
init c=0,d=0
par a0=2,b0=3,k1=.02,k2=.002,k3=.02,k4=.004
@ total=70
done
```

Let's first see how the data and the model compare before running the least squares. Run *XPPAUT* on this file. Don't integrate the equations yet. Click on `Load` in the **Data Viewer** and load the data `mod.dat` into *XPPAUT*. Plot d versus time and freeze this in color 1. Plot c vs time and freeze this in color 0. Now integrate the equations. Use `Graphic stuff Edit curve`, and change the color to 8, and click `Ok`. In the dialog box, click on `Graphic stuff Add Crv` and make `Y-axis:d` and `Color:9`. Set window coordinates to $[0, 70] \times [0, 1.5]$ so that all curves are visible. You will see four curves; two are the data

curves and two are the components of the simulation. This is the **before** picture. Now we will run the curve fit to see if we can do better. Click on `nUmerics stochastic fIt data` and you will get a large dialog box. Fill it in as follows:

`File:` mod.dat	`Ncols:` 3
`Fitvar:` c,d	`To Col:` 2,3
`Params:` a0,b0,k1,k2,k3,k4	`Params:`
`Tolerance:` 0.001	`Epsilon:` 1e-5
`Npts:` 11	`Max iter:` 20

This tells *XPPAUT* the name of the file, how many columns the file has, and the names of the variables you want to fit to their respective columns. (Thus `c` is fit to column 2 and `d` to column 3.) This also tells *XPPAUT* the names of the parameters that will be varied. The `tolerance` is used to determine when further changes are no longer necessary. Let χ_1, χ_2 be two successive least-squares values. If either $|\chi_1 - \chi_2|$ or $|\chi_1 - \chi_2|/\chi_2$ are less than the tolerance, then the program assumes success. Don't make this too small or you will be wasting time. The parameter `Epsilon` is used in numerical derivatives. The parameter `Max iter` sets the maximum number of iterates to try to achieve the desired tolerance. Finally `Npts` is the number of data points (rows) that you want to use. *XPPAUT* uses the first column of data in the data file to determine the time points of the simulation. These should be set in increasing order, but are not required to be equally spaced.

Once you have put the values into the dialog box, click `Ok` and a bunch of stuff will flash by in the terminal window. After a few seconds, a message box should pop up that says `Success`. This means convergence was achieved and a minimum was found. However, notice that in the **Parameter Window**, two of the kinetic constants, k_2, k_4, are negative. This is nonphysical. So, what can one do about this? The easiest strategy is to set both of them to zero or some very small positive number. Then redo the fit, only this time do not include them as parameters in the fitting algorithm—they will be left alone. Click on `stochastic fIt data` again and edit the `Params` entry by deleting `k2,k4` from the list and clicking `Ok`. Again, the fit will run successfully. This time no physical constraints are violated. `Escape` from the numerics menu and click on `Initialconds Go` and look at the new solutions you have computed. They are much better. Figure 9.1 shows the results of the initial and final choices of parameters.

9.4 The data browser as a spreadsheet

The **Data Viewer** has a number of interesting features that allow you to manipulate data in the columns much like you would in a spreadsheet. For example, sometimes you might want to numerically differentiate or integrate data in a column or look at a logarithmic plot. This is all possible by manipulating columns in the **Data Viewer**. So, let's look at some of the functions of the data browser. Run the Morris–Lecar equation model, `ml.ode`. In the **Data Viewer**, click on `Add col`. For the name, type in `il`, an abbreviation for the leak current I_L. For the formula, type in `gl*(v-el)` and then click on `Add it`. A new column will appear in the data browser. You can plot this new quantity and, when you integrate the equations, this new quantity changes appropriately. You can add more quantities as well. The `Del Col` does not work properly and so has been deactivated. If you make a mistake

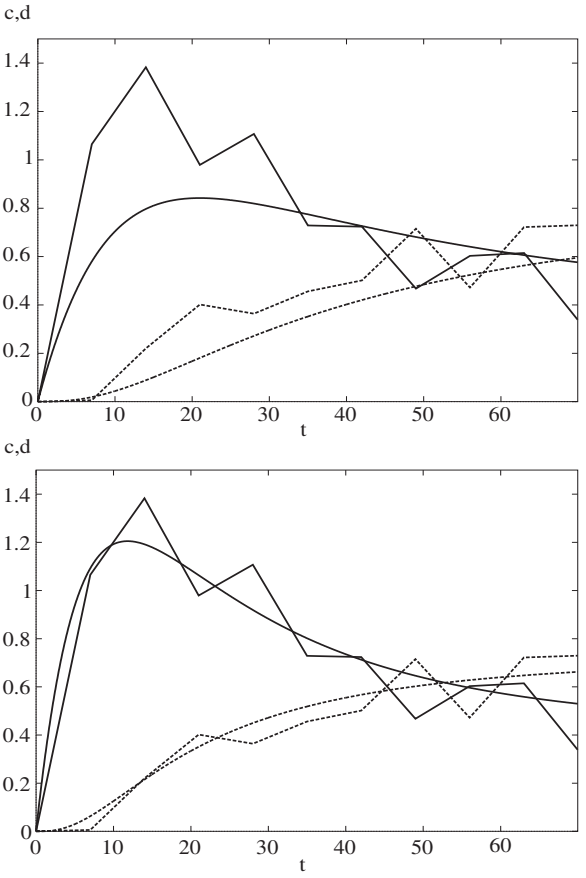

Figure 9.1. *Curve fit to experimental kinetics model. Top: The initial guess. Bottom: The parameters found by curve fitting.*

in the definition of the formula for the new quantity, you can always edit it by using `File Edit RHS's` in the **Main Window**.

If you don't want to add a new column but instead just want to look at, say, the log of a quantity for that particular run, you can always click the `Replace` button in the **Data Viewer**. This prompts you for the name of the column and then the formula. The `Unreplace` button undoes the most recent replacement.

You can numerically integrate or differentiate a column of data using the symbols `&` and `@` respectively. Thus, in the Morris–Lecar file, suppose you wanted the derivative of the voltage. Then replace the voltage by first choosing `v` as the column to replace and then typing in `@v` for the formula. Then you can plot the derivative of the voltage. Note that differentiation and integration are **post integration** commands only. That is, you cannot use these in a new column formula or right-hand side.

Occasionally, you need a sequential list of numbers. You can replace a column with such a list. The formula `a:b` creates a list of numbers the same length as the number of

rows in the **Data Viewer**, starting with `a` and ending with `b`. The formula `a;b` creates a list starting with `a` and incrementing by `b` for each row. For example, replace `t` with `0:6.283` and then replace `V` with `sin(t)`. Plot `V` versus t and you will see a nice sine wave! Pretty lame, huh?

One trick is to read in data from a file and manipulate it with the data browser and use *XPPAUT* as a fancy graphing program.

9.5 Oscillators, phase models, and averaging

One of the main areas of my own research concerns the behavior of coupled nonlinear oscillators. In particular, I have been interested in the behavior of weakly coupled oscillators. Before describing the features of *XPPAUT* that make such analyses easy, I will discuss a bit of the theory. Consider an autonomous differential equation

$$X' = F(X)$$

which admits as a solution an asymptotically stable periodic solution $X_0(t) = X_0(t + P)$, where P is the period. Now suppose that we couple two such oscillators, say, X_1 and X_2:

$$X_1' = F(X_1) + \epsilon G_1(X_2, X_1),$$

$$X_2' = F(X_2) + \epsilon G_2(X_1, X_2),$$

where G_1, G_2 are possibly different coupling functions and ϵ is a small positive number. One can apply successive changes of variables and use the method of averaging to show that, for ϵ sufficiently small,

$$X_j(t) = X_0(\theta_j) + O(\epsilon),$$

with

$$\theta_1' = 1 + \epsilon H_1(\theta_2 - \theta_1), \qquad \theta_2' = 1 + \epsilon H_2(\theta_1 - \theta_2),$$

and H_j are P-periodic functions of their arguments. This type of model is called a **phase model** and has been the subject of a great deal of research. One of the key questions is, What is the form of the functions H_j and how can one compute them? This is where *XPPAUT* can be used. The formula for H_j is

$$H_j(\phi) \equiv \frac{1}{P} \int_0^P X^*(t) \cdot G_j[X_0(t + \phi), X_0(t)]\, dt \tag{9.1}$$

which is the average of the coupling with a certain P-periodic function X^* called the **adjoint** solution. This equation satisfies the linear differential equation and normalization:

$$\frac{dX^*(t)}{dt} = -[D_X F(X_0(t))]^T X^*(t), \quad X^*(t) \cdot X_0'(t) = 1.$$

Here, $D_X F$ is the derivative matrix of F with respect to X and A^T is the transpose of the matrix A.

Thus, to compute the functions H_j, the adjoint must be computed along with the integral. *XPPAUT* implements a numerical method invented by Graham Bowtell for computing the adjoint $X^*(t)$. The method only works for oscillations which are asymptotically stable. Here is the idea. Let $B(t) = (D_X F(X_0(t)))$. Since the limit cycle is asymptotically stable, if we integrate the equation

$$Y' = B(t)Y$$

forward in time, it will converge to a periodic orbit proportional to $X_0'(t)$ as a consequence of the stability of the limit cycle and the translation invariance. Similarly, the solution to

$$Z' = -B(t)^T Z$$

will converge to a periodic orbit if we integrate it *backward* in time. As you can see, this is the desired adjoint solution and is how *XPPAUT* computes it. Once $X^*(t)$ is computed, it is trivial to compute the integral and thus compute the interaction function.

9.5.1 Computing a limit cycle and the adjoint

To use *XPPAUT* to compute the adjoint, as a necessary first step to computing the interaction function H we have to compute exactly one full cycle of the oscillation. Let's use as our example the Morris–Lecar equation:

$$v' = I + g_l(e_l - v) + g_k w(e_k - w) + g_{ca} m_\infty(v)(e_{ca} - v),$$

$$w' = (w_\infty(v) - w)\lambda_w(v).$$

We will look at

$$v_1' = f(v_1, w_1) + \epsilon(v_2 - v_1),$$

$$w_1' = g(v_1, w_1),$$

$$v_2' = f(v_2, w_2) + \epsilon(v_1 - v_2),$$

$$w_2' = g(v_2, w_2).$$

Since the method of averaging depends on the two oscillators being identical, except possibly for the coupling, all you need to do is integrate the isolated oscillator. Use the file `ml.ode`. Change the parameters `I=.09, phi=0.5`. Integrate the equations and click on `Initialconds Last` a few times to make sure you have gotten rid of all transients. Now you are pretty much on the limit cycle. To compute the adjoint, you need one full period. In many neural systems, coupling between oscillators occurs only through the voltage and thus the integral (9.1) involves only one component of the adjoint, the voltage component. For whatever reason, the numerical algorithm for computing the adjoint for a given component converges best if you start the oscillation at the *peak* of that component. Thus, since we will only couple through the potential in this example, we should start the oscillation at the peak of the voltage. Here is a good **trick** for finding that maximum. In the **Data Viewer** click on `Home` which makes sure that the first entry of the data is at the

top of the **Data Viewer**. Click on `Find` and, in the dialog box, choose the variable `v` and, for a value, choose `1000` and then click `Ok`. *XPPAUT* will find the closest value of `v` to `1000` which is obviously the maximum value of `v`. Now click on `Get` in the **Data Viewer** which grabs this as initial conditions. In the **Main Window**, click on `Initialconds Go` to get a new solution. Plot the voltage versus time. Use the mouse to find the time of the next peak. (With the mouse in the graphics window, hold down the left button and move the mouse around. At the bottom of the windows, read off the values.) The next peak is at about 22.2. In the **Datad Viewer** scroll down to this time and find exactly where `v` reaches its next maximum. Note that, as we suspected, it is at $t = 22.2$. In the **Main Window**, click on the `nUmerics` menu and then `Total` to set the total integration time. Choose `22.21` (it is always best to go a little bit over—but just a little). Escape to the main menu and click on `Initialconds Go`. Now we have one full cycle of the oscillation! The rest is easy.

Computing the adjoint

Click on `nUmerics Averaging New adjoint` and after a brief moment, *XPPAUT* will beep. (Sometimes, when computing the adjoint, you will encounter the `Out of Bounds` message. In this case, just increase the bounds and it recomputes the adjoint.) Click on `Escape` and plot v versus time. This time, the adjoint of the voltage is in the **Data Viewer** under the voltage component. (Note that the adjoint is almost strictly positive and looks nearly like $1 + \cos t$. This is no accident and has been explained theoretically.)

9.5.2 Averaging

Now we can compute the average. Recall that the integral depends on the adjoint, the original limit cycles, and a phase-shifted version of the limit cycle:

$$X^*(t) \cdot G(X_0(t+\phi), X_0(t)).$$

In *XPPAUT*, you will be asked for each component of the function G. For unshifted parts, use the original variable names, e.g., `x,y,z` and, for the shifted parts, use primed versions, `x',y',z'`. The coupling vector in our example is

$$(V(t+\phi) - V(t), 0).$$

Thus, for our model, the two components for the coupling are `(v'-v, 0)`. This says that we take the phase-shifted version of the variable `v`, called `v'` by *XPPAUT*, and subtract from it the unshifted version of `v`. With these preliminaries, it is a snap to compute the average. Click on `nUmerics Averaging Make H`. Then type in the first component of the coupling, `v'-v` and the second `0`. Then let it rip. In a few moments, the calculation will be done. If your system has more than two columns in addition to time `t` (as this example does with - `v, w, ica, ik`), then the first column contains the average function $H(\phi)$, the second column contains the odd part of the interaction function, and the third column contains the even part. Plot the function H by escaping back to the main menu and plotting `v` versus time—remember `v` is the first column in the browser after the `t` column. This is a periodic function. For later purposes, we want to approximate this periodic function. Click

on nUmerics stocHastic Fourier and choose v as the column to transform. Look at the data browser and observe the first three cosine terms (column 1 after the t column) and the first three sine terms (column 2). They are (3.34, −3.05, −.29) and (0,3.61,−0.33), respectively. Thus to this approximation,

$$H(\phi) = 3.34 - 3.05\cos\phi - 0.29\cos 2\phi + 3.61\sin\phi - 0.33\sin 2\phi. \qquad (9.2)$$

To get the interaction function, adjoint, or original orbit back into the data browser, click on Averaging Hfun, etc. from the nUmerics menu.

Summary

Here is how to average weakly coupled oscillators:

1. Compute exactly one period of the oscillation—you may want to phase-shift it to the peak of the dominant coupling component.
2. Compute the adjoint from the nUmeric Averaging menu.
3. Compute the interaction function using the original variable names for the unshifted terms and primed names for the shifted terms.

9.5.3 Phase response curves

One of the most useful techniques available for the study of nonlinear oscillators is the **phase response curve** (PRC). The PRC is readily computed experimentally in real biological and physical systems and is defined as follows. Suppose that there is a stable oscillation with period T. Suppose that at $t = 0$ one of the variables reaches its peak. (This can always be done by translating time.) At a time τ after the peak, one of the state variables is given a brief perturbation that takes it off the limit cycle. This will generally cause the next peak to occur at a time $T'(\tau)$ that is different from the time it would have peaked in absence of the perturbation. The PRC $\Delta(\tau)$ is defined as

$$\Delta(\tau) \equiv 1 - T'(\tau)/T.$$

If $\Delta(\tau) > 0$ for some τ, we say that the perturbation advances the phase since the time of the next maximum is shortened by the stimulus. Delaying the phase occurs when $\Delta(\tau) < 0$. Experimental biologists have computed PRCs for a variety of systems such as cardiac cells, neurons, and even the flashing of fireflies. We will use *XPPAUT* to compute the PRC for the van der Pol oscillator

$$\ddot{x} = -x + \dot{x}(1 - x^2)$$

subjected to a square-wave pulse of width σ and amplitude a.

Here is how we will set up the problem. We will let τ be a variable rather than a parameter so that we can range through it and keep a record—it is like a bifurcation parameter. We will define a parameter T_0 which is the unperturbed period. We will use the ability of *XPPAUT* to find the maxima of solutions to ODEs and have the computation stop when the maximum of x is reached. The time at which this occurs is the time $T'(\tau)$ and so we will also track an auxiliary variable $1 - t/T_0$ which is the PRC. Here is the ODE file:

```
# vdpprc.ode
# PRC of the van der Pol oscillator
init x=2
x'=y
y'=-x+y*(1-x^2)+a*pulse(t-tau)
tau'=0
pulse(t)=heav(t)*heav(sigma-t)
par sigma=.2,a=0
par t0=6.65
aux prc=1-t/t0
@ dt=.01
done
```

Note that the function `pulse(t)` produces a small square-wave pulse. To compute the PRC, you want to integrate the equations for a bit to get a good limit cycle with no perturbation ($a = 0$). Then start at the maximum value of x and integrate until the next maximum. This will tell you the base period. Then you want to apply the perturbation at different times and compute the time of the next maximum and then from this get the PRC. Fire up *XPPAUT* with this file. Follow these simple steps:

1. Integrate it and then integrate again using the `Initialconds Last` (**I L**) command to be sure you have integrated out transients.

2. Now find the variable of interest, in this case, x. To do this, in the **Data Viewer** click on `Home` to go to the top of the data window. Click on `Find` and type in `x` for the variable and `100` for the value. *XPPAUT* will try to find the value of `x` closest to 100. This will be the maximum. Click on `Get` to load this as an initial condition.

3. Figure out the unperturbed period. One way is to integrate the equations and estimate the period. A better way is to let *XPPAUT* do it for you. In the **Main Window**, click on `nUmerics Poincare map Max/Min` and fill in the dialog box as follows:

Variable: x
Section: 0
Direction: 1
Stop on sect: Y

and then click on `Ok`. You have told *XPPAUT* to integrate, plotting out only the maxima (Direction=1) of x. The `Stop on section` ends the calculation when the section is crossed. Click on `Transient` and choose 4 for the value. This is so we don't stop at the initial maximum. `Transient` allows the integrator to proceed for a while before looking at and storing values. Now exit the numerics menu by clicking `Esc`. Click on `Initialconds Go` and the program will integrate until x reaches a maximum. In the **Data Viewer**, click on `Home` and you should see that `Time` has a value of around 6.6647. This is the unperturbed period. Set the parameter `t0` to the value in the **Data Viewer**.

4. Now we are all set to compute the PRC. Change the perturbation amplitude from 0 to 3. Click on `Initialconds Range` and fill in the dialog box as follows:

`Range over: tau`
`Steps: 100`
`Start: 0`
`End: 6.6647`
`Reset storage: N`

and click `Ok`. It should take a second or two.

5. Plot the auxiliary quantity `PRC` against the variable `tau`. (Click on `Viewaxes 2D` and put `tau` on the `X-axis` and `PRC` on the `Y-axis`. Click `OK` and then `Window Fit` to let *XPPAUT* figure out the window.) You should see something like the top curve in Figure 9.2.

Freeze this curve and try using a different value of the amplitude `a`.

Exercise. Compute the PRC for the Morris–Lecar oscillator by adding the required parts to the file `ml.ode`. Define a square-wave pulse just like above with a width of 0.25 and a magnitude of 0.1. If you are stuck, download the file `mlprc.ode` which sets up the ODE for you. You should get something like the bottom plot of Figure 9.2.

9.5.4 Phase models

In the previous section we saw how to reduce a pair of coupled oscillators to a pair of scalar models in which each variable lies on a circle. *XPPAUT* has a way of dealing with flows on systems of scalar variables each of which lie on a circle. The phase-space of a system of, say, two such variables, is the two-torus. Let's start with the simplest phase model:

$$\theta_1' = \omega_1 + a\sin(\theta_2 - \theta_1),$$

$$\theta_2' = \omega_2 + a\sin(\theta_1 - \theta_2)$$

representing a pair of sinusoidally coupled oscillators with different uncoupled frequencies, ω_1, ω_2 and coupling strength, a. The ODE file for this is straightforward:

```
# phase2.ode
# phase model for two coupled oscillators
th1'=w1+a*sin(th2-th1)
th2'=w2+a*sin(th1-th2)
par w1=1,w2=1.2,a=.15
@ total=100
done
```

Fire this up and integrate the equations. You will get the `Out of bounds` message at $t = 90$ or so. That is because the variables θ_j are actually defined only modulo 2π and

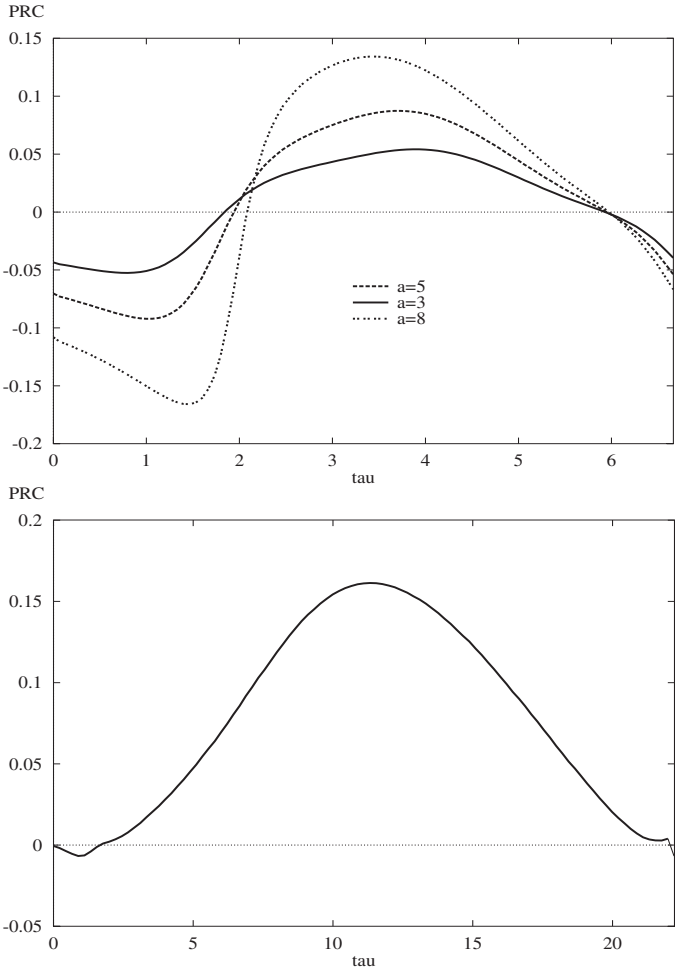

Figure 9.2. *Top: The PRC for the van der Pol oscillator with different amplitudes. Bottom: The PRC for the Morris–Lecar model.*

are not really going out of bounds but instead just wrapping around the circle. Thus, it is more proper to look at θ_j modulo 2π. If you define an auxiliary variable which is $\theta \bmod 2\pi$ and then plot it, you will get a series of ugly lines that cross from one part of the screen to the other. This is because *XPPAUT* does not know that this particular variable is defined on the circle. There is a simple way to tell *XPPAUT* which of the state variables lie on the circle. Click on `phAsespace Choose` (**A C**) and when prompted for the period, choose the default which is approximately 2π. A little window will appear with all your variables included. Move the cursor to the left of each of them and click the mouse to see a little `X` next to the variable. Put this mark next to both variables and click on `done`. This tells *XPPAUT* that these are both considered "folded" variables and they will be folded mod 2π. Now reintegrate the equations. This time you get no such out of bounds message—instead

you will see that the plots are all modulo 2π. Switch the view to the phase-plane for the two variables, θ_1, θ_2. (In the **Initial Data Window** click on the box to the left of each variable and then on the `xvsy` button.) You will see that the trajectories seem to converge on a diagonal line which is shifted slightly upward. Try integrating using the `Initialconds mIce` (**I I**) command to choose a variety of initial conditions. They should converge to the same line. This is an example of a phase-locked solution—an invariant circle on the torus. Change a from 0.15 to 0.08 and integrate the equations again. Notice how the trajectories do not converge to an attractor. Instead, the whole torus will gradually fill up. Phase-locking no longer occurs.

A derived example

Now let's turn to the example that we derived in the previous section and see if a pair of oscillators will phase-lock when we use the interaction function defined in (9.2), `phase_app.ode`:

```
# phase_app.ode
# phase model for two coupled oscillators
# using numerically computed H
h(x)=3.34-3.05*cos(x)-.29*cos(2*x)+3.61*sin(x)-.33*sin(2*x)
th1'=w1+a*h(th2-th1)
th2'=w2+a*h(th1-th2)
par w1=1,w2=1,a=.1
@ total=100
@ fold=th1,fold=th2
@ xlo=0,ylo=0,xhi=6.3,yhi=6.3,xp=th1,yp=th2
done
```

I have added a few new `@` commands. The `fold=th1` directive tells *XPPAUT* to make a circle out of the variable `th1`, that is, to mod it out. All variables that you want to mod out can be set by the `fold=name` directive. This automatically tells *XPPAUT* to look for modded variables. The default period is 2π so we don't have to change it. If you want to change the period to, say, 3, set it with the command `@ tor_period=3`. Run *XPPAUT* on this, using the mouse to set some initial conditions. Note how all initial data synchronize along the diagonal, implying $\theta_1 = \theta_2$. Change the intrinsic frequency `w2` and see how big you can make it before phase-locking is lost.

Pulsatile coupling

Winfree [42] introduced a version for coupling oscillators using the PRC of the oscillator, assuming that the interaction between oscillators occurred only through the phase and took the form of a product. Ermentrout and Kopell [11] proved that this was a reasonable model when certain assumptions were made concerning the attractivity of the limit cycle. Here is a pair of pulse-coupled phase models:

$$\frac{d\theta_1}{dt} = \omega_1 + P(\theta_2)R(\theta_1),$$

$$\frac{d\theta_2}{dt} = \omega_2 + P(\theta_1)R(\theta_2).$$

Think of $R(\theta)$ as the PRC of the oscillator and $P(\theta)$ as the pulse coupling. For example, if $P(\theta)$ is a Dirac delta-function, then this model reduces to a one-dimensional map. This instance of coupling is simulated in section 9.6.2. The following ODE is a case in which the coupling occurs through a smooth function:

```
# phasepul.ode
# pulsatile phase model
r(x)=-a*sin(x)
p(x)=exp(-beta*(1-cos(x)))
par a=3,beta=5
x1'=w1+p(x2)*r(x1)
x2'=w2+p(x1)*r(x2)
par w1=1,w2=3
@ total=100
@ fold=x1,fold=x2
@ xp=x1,yp=x2,xlo=0,ylo=0,xhi=6.3,yhi=6.3
done
```

As in the previous example, I have set this up so that the projection is onto the torus phase-plane. Here are some things to do to explore the behavior of this system.

- Integrate the equations and note how solutions tend to the diagonal. There is a stable phase-locked synchronous solution.
- Change the integration time step to `-.05` and integrate the equations again. Note that there is another phase-locked solution that is out of phase. Change the integration time step back to 0.05.
- Change the coupling strength a from 3 to -3 and repeat the first two exercises. Note how synchrony is unstable when $a < 0$.
- Change a to 3 again. Draw the nullclines. They don't intersect. Change `w1`, the frequency of the first oscillator, to 0.1. Redraw the nullclines. Determine the stability of the fixed points and draw all the invariant manifolds for the saddle-point. Where do the stable manifolds go? They tend to "vertical" lines in the middle of the phase-plane. That is, there is a repelling invariant circle in which oscillator 2 fires repeatedly and oscillator 1 just sort of wobbles around near π. All stable solutions end up at a fixed point. This is called **phase-death**.
- Now change `w1` back to 1 and change `w2` to 3. Integrate the equations. Note that for every three cycles of oscillator 2, there is one cycle of oscillator 1. This is an example of 3:1 phase-locking. Open another graphics window by clicking on `Makewindow New` and in this window graph both `x1` and `x2` against `t`. Change the window for $80 < t < 100$ so you can see it more clearly.
- Gradually decrease `w2` toward 1 and see what other kinds of mode-locking are possible. (`w1=1.9` is interesting.)

9.6 Arcana

Here I describe a number of useful tricks that belong to no special category.

9.6.1 Iterating with fixed variables

Because "fixed" variables in *XPPAUT* are evaluated in succession, one can actually compute iterative procedures with them which will be used in the right-hand sides of the equations. In chapter 4, we saw an example of this in the Taylor series plots. Here, I present a more sophisticated example of this trick. I want to create and analyze the Morse–Thule sequence, M_n. This is a sequence of 0's and 1's. The nth number in the sequence is the number of 1's in the binary expression of n modulo 2. For example, the 23rd element is found by writing $23 = 10111$. This has four 1's and $4 = 0$ modulo 2, so $M_{23} = 0$. Suppose that $n < 2^p$. Then the number of 1's in the binary expansion of n is found from the iteration

```
s=0;
x=n;
for(i=0;i<p;i++){
s=s+mod(x,2);
x=flr(x/2);
}
```

Here `flr(x)` is the integer part of `x`. Thus, I just keep dividing x by 2 and checking if the result is odd. (This can be done much more quickly in C than I have indicated here, but this is less obscure.) After the iteration, s contains the number of 1's. We then just get M_n by modding by 2. Here is a fragment of *XPPAUT* code that implements this iteration:

```
s0=0
x0=n
%[1..16]
s[j]=s[j-1]+mod(x[j],2)
x[j]=flr(x[j-1]/2)
%
```

These are all "fixed" variables so they are evaluated at every time step. The block delimited by the `%` does the iteration 16 times so that this will work as long as $n < 2^{16} = 65536$. The quantity `s16` has the desired sum. We have used the fact that the fixed variables are evaluated in the order in which they are defined. Here we have defined 34 fixed variables. With this bit of code, the rest of the ODE file is easy. Here is `mt.ode`:

```
# mt.ode
# the morse-thule sequence of 0's and 1's
# the sum mod 2 of the 1's in the binary expansion
# of the integers
# look at the power spectrum for z
#
init n=0
s0=0
```

```
x0=n
%[1..16]
s[j]=s[j-1]+mod(x[j],2)
x[j]=flr(x[j-1]/2)
%
n'=n+1
z'=mod(s16,2)
aux ss=s16
@ meth=discrete,total=10000,maxstor=68000
@ bound=100000
@ xlo=0,xhi=10000,ylo=0,yhi=16,yp=ss
done
```

Run this ODE file and look at the nice self-similar nature of the sums (which are plotted.) The sequence itself is just the auxiliary variable, `ss`. Now, the really cool part of this is to look at the power spectrum of the sequence, `ss`. Click on `nUmerics stocHast Power` and choose `ss` as the variable to transform. Click on `Escape` to get back to the main menu. Click on `Xi vs t` and plot `N`. You will see a very strange power spectrum which is self-similar. That is, if you blow up a region, it looks like the full spectrum. Try this with 64,000 iterations and see that it looks essentially the same.

9.6.2 Timers

There are often situations where you may want a switch to turn on after a prescribed amount of time. For example, if you have a model of a synapse that has the form

$$ds/dt = A(t)(1 - s) - s/\tau,$$

where $A(t) = A_0$ is $T < t < T + h$ and zero otherwise. T is the firing time for the cell that corresponds to the synapse, s. Here is a way to do this:

```
init tf=-1000
s'=a0*heav(tf+h-t)*(1-s)-s/tau
tf'=0
global 1 v-vt {tf=t}
```

I have used the global flag to determine when the cell's voltage `v` crosses some threshold `vt`. At this point the variable `tf` is set to the current time `t`. The Heaviside step function switches to 1 and remains as long as `t<tf+h` and thus turns off `h` time units later.

Coupled PRCs

Related to this issue of tracking time is keeping a record of interspike intervals and/or relative firing times in a system of neural oscillators. For simplicity, consider two coupled neural oscillators, say, coupled by a PRC (see section 9.5.3)

$$x_1' = \omega_1 + \delta(x_2)P_{21}(x_1), \qquad x_2' = \omega_2 + \delta(x_1)P_{12}(x_2).$$

We assume that whenever x_j crosses 1, it is reset to zero and $x_k = x_k + P(x_k) \equiv F(x_k)$. We also assume that P is 1-periodic and vanishes at $x = 0$. Then, we can write the ODE file for this as

```
# map2.ode
# pair of coupled maps
x1'=w1
x2'=w2
p(x)=-a*sin(2*pi*x)
f(x)=mod(x+p(x),1)
par a=.05
par w1=1,w2=.9
global 1 x1-1 {x1=0;x2=f(x2)}
global 1 x2-1 {x2=0;x1=f(x1)}
@ dt=.0101
done
```

The strange value of `dt` is due to a problem with the `global` command which occasionally fails if the test condition is satisfied exactly, i.e., `x1=1` at a particular time step.

Now, this particular ODE file gives the values of the state variables, but not the timing difference between the two cells nor the firing time interval between them. Let ϕ be the time difference between the time that cell 1 fires and cell 2 fires. Let T_j be the interval between successive firings of the respective cells. Then we can track these, keeping in mind that the `global` command only allows you to modify state variables. We introduce five new variables, `t1f`, `t2f`, `t1`, `t2`, `phi`, all of which satisfy $u' = 0$. Here is the new ODE file:

```
# map2.ode
# pair of coupled maps
# and timing info
x1'=w1
x2'=w2
t1f'=0
t2f'=0
t1'=0
t2'=0
phi'=0
p(x)=-a*sin(2*pi*x)
f(x)=mod(x+p(x),1)
par a=.05
par w1=1,w2=.9
global 1 x1-1 {x1=0;x2=f(x2);t1=t-t1f;t1f=t}
global 1 x2-1 {x2=0;x1=f(x1);phi=t-t1f;t2=t-t2f;t2f=t}
@ dt=.0101,total=50, bound=10000
done
```

Each time x_1 fires, I update the interval between the last firing, `t1=t-t1f`, and then update the firing time of 1, `t1f=t`. Each time that x_2 fires, I update similar intervals but also

the timing difference, `phi=t-t1f`. Try this file with the parameters as in the file. Plot `t1,t2` on the same plot. Plot `phi`. Increase `a` to 0.06. Try this again. Change `a` to -0.06 and solve it again. Finally, set `a=0.1` and `w2=0.53` and integrate the equations. Look at `t1,t2` and interpret what this means.

9.6.3 Initial data depending on parameters

One unfortunate shortcoming in *XPPAUT* is the inability of the program to set initial data that depend on parameters. Parameters can be defined in terms of other parameters with the

`!<name>=<formula>`

declaration. However, there is no easy way to define initial data in terms of a parameter. In spite of this shortcoming, there is a little trick that you can use. This trick exploits the fact that `global` flag conditions can set the state variables to anything. Let's consider the classic problem of firing a cannonball. The parameter that you want to vary is the angle of the cannon, θ. Thus, the differential equation is

$$m\ddot{x} = -f\dot{x}, \qquad m\ddot{y} = -mg - f\dot{y}$$

with

$$x(0) = y(0) = 0, \quad \dot{x}(0) = v_0 \cos\theta, \quad \dot{y}(0) = v_0 \sin\theta.$$

Here is the trick. We will flag $t = 0$ and immediately switch initial data. The key here is to use the global sign `0` instead of `1,-1`. The difference with using `0` is that the flag is set only when equality occurs exactly. This essentially never happens except at the start of a problem. Thus, it is a good way to set initial data that depend on a parameter. Here is the ODE file:

```
# cannon.ode
# with linear friction
x'=vx
vx'=-f*vx
y'=vy
vy'=-f*vy-g
global 0 t {vx=v0*cos(pi*theta/180);vy=v0*sin(pi*theta/180)}
par theta=30
par f=.01,g=1
par v0=2.5
init x=0,y=0
@ xlo=0,xhi=5
@ ylo=0,yhi=3
@ xp=x,yp=y,bound=100000
done
```

I have set up the problem in the (x, y)-plane so that you can see the trajectory of the cannon. I have also scaled the angle θ so that it is in degrees rather than radians. Try to hit the number 4 on the x-axis!

Computing basin boundaries

The ability to parametrically control initial data allows us to perform some interesting computations—we can compute the basins of attraction of systems with multiple attractors. Let's first consider a classic example, the complex cube roots of 1:

$$z^3 = 1.$$

If we rewrite this in terms of the real and imaginary parts of $z = x + iy$, then we want to solve

$$x^3 - 3xy^2 = 1, \qquad 3x^2y - y^3 = 0$$

for (x, y). This represents two nonlinear equations in two unknowns. The way to solve these is to use Newton's method which we write as an iteration:

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} - \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ 6xy & -3y^2 \end{bmatrix}^{-1} \begin{bmatrix} x^3 - 3xy^2 - 1 \\ 3x^2y - y^3 \end{bmatrix}.$$

Here (x_n, y_n) is the nth approximation to the root. The matrix above is the Jacobi matrix for the two functions. The point is that Newton's method produces a discrete dynamical system. There are three cube roots to 1 in the complex plane, the real root, $(x, y) = (1, 0)$, and the complex conjugate roots, $(x, y) = (1/2, \pm\sqrt{3}/2)$. Thus, we can ask the following question: Given an initial guess (x_0, y_0), which of these roots will the guess converge to? If we color code the point (x_0, y_0) according to the root that it is attracted to, the resulting picture looks like that in Figure 9.3.

This picture can be made with *XPPAUT* by taking advantage of the ability to vary initial data parametrically. Here is the ODE file used to produce the figure:

```
# cube.ode
# newtons method in the complex plane to find cube roots of 1
# z^3=1
# x^3-3*xy^2 = 1 , 3x^2y-y^3=0
# draws fractal basin boundaries
#
# tolerances
par eps=.001
# discretization of the phase-space
par dx=0,dy=0
# the three roots
par r1=1,s1=0
par r2=-.5,s2=-.8660254
par r3=-.5,s3=.8660254
# the functions
f=x^3-3*x*y^2-1
g=3*x^2*y-y^3
# the derivatives of the functions
fx=3*(x^2-y^2)
fy=-6*x*y
```

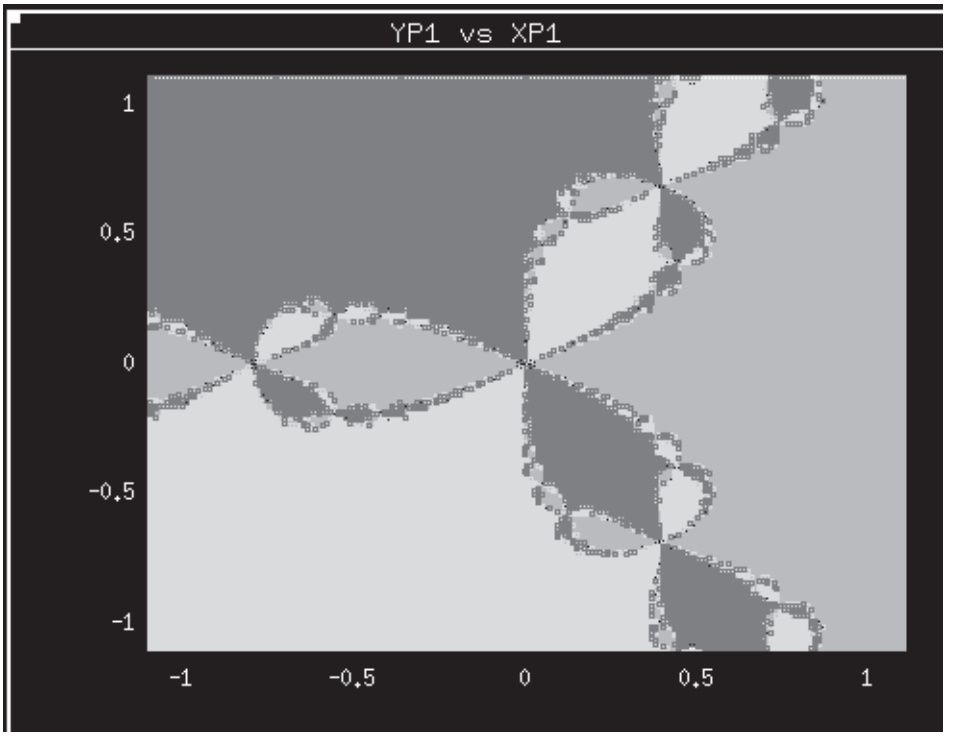

Figure 9.3. *The basin boundaries for the iteration which arises from the application of Newton's method to the equation $z^3 = 1$ in the complex plane. Each point in the plane is colored according to the root that it converges to.*

```
gx=6*x*y
gy=3*(x^2-y^2)
# the Jacobian -- used in the inverse
det=fx*gy-fy*gx
# a euclidean distance function
dd(x,y)=x*x+y*y
# if close to the root then plot otherwise out of bounds for
# each root
aux xp[1..3]=if(dd(x-r[j],y-s[j])<eps)then(x0)else(-100)
aux yp[1..3]=if(dd(x-r[j],y-s[j])<eps)then(y0)else(-100)
#
# iteration
x'=x-(f*gy-g*fy)/det
y'=y-(g*fx-f*gx)/det
# initial data
x0'=x0
y0'=y0
```

```
# set initial data as parameters
glob 0 t {x0=-1.1+dx*2.2;y0=-1.1+dy*2.2;x=-1.1+dx*2.2;
      y=-1.1+dy*2.2}
#
# plot all three sets of points in lousy colors
#
@ meth=discrete,total=20,bound=1000,maxstor=100000,trans=20
@ xp=xp1,yp=yp1,xp2=xp2,yp2=yp2,xp3=xp3,yp3=yp3,nplot=3,lt=-1
@ xlo=-1.1,ylo=-1.1,xhi=1.1,yhi=1.1
done
```

This rather lengthy ODE file is actually pretty simple and works as follows. We draw three different sets of data corresponding to the three different roots. We compute 20 iterations and keep only the last iterate by setting the total and the transient to 20. If (x_{20}, y_{20}) is close to the real root, then we plot the initial conditions (x_0, y_0); otherwise we plot $(-100, -100)$ which is out of the plotting range. Similarly, if the 20th iterate is close to the first complex root, then we plot the initial condition; otherwise we plot the out-of-bounds point. We treat the data for each root as three different sets of points. (I hope the large number of comments makes it self-explanatory.) I point out the use of the `global` statement to set the initial conditions according to the values of the parameters `dx,dy`. In the options statements, I tell *XPPAUT* to plot all three pairs of points `xp1,yp1`, `xp2,yp2`, and `xp3,yp3` which are either the initial conditions of the iteration or the points $(-100, -100)$. By choosing linetype `-1`, a series of small circles will be drawn instead of dots or lines. Run this file. Click on `Initialcond 2 par range` (**I 2**) and fill in the dialog box as follows:

`Vary1: dx`	`Reset storage: N`
`Start1: 0`	`Use old ic's: Y`
`End1: 1`	`Cycle color: N`
`Vary2: dy`	`Movie: N`
`Start2: 0`	`Crv(1) Array(2): 2`
`End2: 1`	`Steps2: 101`
`Steps: 101`	`:`

and click on `Ok`. A series of colored points will be drawn. To better distinguish the points, you should click on `Graphics Edit curve`, select curve `2`, and change the color from 2, which is red-orange, to 7 which is green.

Exercises

- Compute the basin boundaries for the complex roots of $z^2 = 1$ and $z^4 = 1$. Note that the first is equivalent to $x^2 - y^2 = 1, 2xy = 0$ and the second is equivalent to $x^4 - 6x^2y^2 + y^4 = 1, 4(x^3y - xy^3) = 0$. For the fourth roots problem you should set the total number of iterates to 50 as well as the transient.

- Consider the discrete bistable system

$$x_{n+1} = y_n \quad y_{n+1} = ay_b + bx_n(c^2 - x_n^2)$$

in the same square as the above examples. There are two stable fixed points for this system $(\pm c, 0)$. Choose $a = .6, b = .3, c = .15$ and compute the basins of attraction of the two roots. You should set the total time and the transient to 100.

Invariant sets revisited

Recall that the periodically forced Duffing equation is chaotic and there can be transverse intersections of the invariant manifolds emerging from the saddle-point. The dynamics under the flow of the equation is such that points get all mixed up like cake batter being stirred. We can see this happen as follows. Choose a region in the phase-plane. Color each point in the phase-plane black (white) if the solution with those initial data is in the $x > 0$ ($x < 0$) half-plane at some fixed time t_0. Do this for a variety of different times to see how the flow mixes everything up. This is like the basin boundaries we computed above but only for finite times. The *XPPAUT* file is like the one in the previous section but simpler:

```
# duffbas.ode
# a trick to compute the basin boundaries of the
# duffing equation
# better be patient it takes awhile
# here is the ODE
x'=y
y'=-.15*y+.5*x*(1-x^2)+f*cos(.8333*t)
#  dx,dy are the increment sizes, f is the forcing amplitude
par dx,dy,f=.1
# here are the initial data evaluated once per cycle
# initial data lie in the square [-2.4,2.4]x[-1.2,1.2]
!x0=-2.4+dx*4.8
!y0=-1.2+dy*2.4
# save the initial data at different time slices
aux xp[1..10]=if((x>0)&(abs(t-2*[j]))<.1)then(x0)else(-100)
aux yp[1..10]=if((x>0)&(abs(t-2*[j]))<.1)then(y0)else(-100)
# set initial data with parameter
glob 0 t {x=x0;y=y0}
# set some options
@ total=22
@ xp=xp5,yp=yp5,xlo=-2.4,ylo=-1.2,xhi=2.4,yhi=1.2,lt=-1
@ maxstor=500000
@ trans=1,dt=2,meth=8
done
```

The lines that start with `!` are treated like invisible parameters that are evaluated anytime the other parameters are changed. Unlike **fixed variables**, these are only evaluated once per integration of the equations. The 10 auxiliary variables contain the initial data if they correspond to the time slice and the x coordinate is positive; otherwise they are set to $(-100, -100)$. I use the Dormand–Prince 8(3) integrator, as it is very fast especially when the output times are large (e.g., `dt=1`).

Run this file, click on `Initialconds 2 par range`, and fill in the resulting dialog box as follows:

`Vary1: dx`	`Reset storage: N`
`Start1: 0`	`Use old ic's: Y`
`End1: 1`	`Cycle color: N`
`Vary2: dy`	`Movie: N`
`Start2: 0`	`Crv(1) Array(2): 2`
`End2: 1`	`Steps2: 101`
`Steps: 101`	`:`

and click on `Ok`. A series of small circles will be drawn at different points. Since `xp5,yp5` are in the plot window and the plots are at even times 2, 4, etc., this graph shows all the initial conditions which end up in the right half-plane at $t = 10$. Look at a plot of `xp10,yp10` to see a big difference.

Make a movie

You can make a quick flip-book of the plotted sets at the different time slices as follows. Click on `Viewaxes 2D` and change the plotted variables to `xp1,yp1`, the first in the set. Then click on `Kinescope Capture` to grab the screen image. Next change the view to `xp2,yp2` and capture the screen again with the (**K C**) commands. Repeat this until you have plotted all 10 sections. Now they are stored in memory. Click on `Kinescope Playback` (**K P**) and click the left-hand mouse button to cycle through them. Press `Esc` when you are tired of doing this.

As an exercise, you could try this method with some other chaotic system like the Lorenz equations or the Rossler attractor.

9.6.4 Poincaré maps revisited

The use of adaptive integration methods can make it easy to compute Poincaré sections for periodically driven systems. The idea is to just set the output time stem `Dt` to the period of the forcing and then just plot the results. That is, we can ignore the `Poincare` option completely. This only works for systems where the Poincaré map is with respect to the time variable. Recall the forced Duffing equation from Chapter 3,

$$x' = v, \qquad v' = x - x^3 - fv + c\cos(\omega t)$$

and the ODE file for it, `duffing.ode`. Run *XPPAUT* with this file. Change `c=.7`, `f=.3`, `omega=1.25`. In the `nUmerics` menu, change the `Method` to `DorPri(8)3` with a relative and absolute tolerance of 1e-5. Now click on `Dt` and type the following string: `%2*Pi/omega`. *XPPAUT* evaluates command-line numbers that start with the % sign and converts them to floats. Thus, output will happen only at multiples of $2\pi/\omega$, the period. Now change `Total` to `%2*Pi*4000/omega` to get 4000 points! Make a two-dimensional window of (x, v) in the plane, $[-2, 2] \times [-2, 2]$. Finally, edit the graph type (**G E**) and choose `0`. Change the `Linetype` to `-1` for large dots. Now integrate the equations and you will see the usual attractor.

Flagged output

Newer versions of *XPPAUT* incorporate a feature that allows you to make Poincaré maps (and many other calculations) very rapidly. The disadvantage of the following method is that you need to add a special line in your ODE file and this cannot be done within the program. Recall from section 3.5 in Chapter 3, that *XPPAUT* allows you to catch crossings of variables and, based on this, take some action. One such action is to **output** a point to the data browser. Just include the action `out_put=1` within the list of actions and when the event occurs. *XPPAUT* will send output to the data browser. Since you generally want to suppress all other output, you should, for example, set `Transient` to some large number from the numerics menu. This guarantees that *XPPAUT* won't print out any other data. The advantage of this over a standard Poincaré map is that there can be many possible events that are flagged. It is generally faster than the Poincaré map feature since graphic output is suppressed.

9.7 Don't forget...

1. Initial conditions and parameters can be typed in at the command line as formulae. Use the `Parameter` command and then type the name of the parameter. When asked for a value, type in the % sign first and then the formula. Use the `Initialconds New` command to type in the initial value in the same manner.

2. There is a stupid little calculator built into *XPPAUT*. Click on `File Calculator` to get it. You can type in the usual math-type expressions to get an answer. Even sums can be evaluated such as `sum(0,100)of(exp(-i'))`. You can set any variable or parameter to an expression by typing, e.g., `x:1/(1+pi^2)` which would set `x` to 0.091999 . . .. Press `Esc` to exit the calculator.

3. You can run *XPPAUT* over a modem line in `-silent` mode. That is, set up your ODE using the `@` options. Then, type `xpp file.ode -silent` and it will run without ever invoking the interface. The data will be saved in a file `output.dat`. This is also a great way to program in batch mode.

4. Parameter sets are a very useful way to set up problems so that different groups of parameters are associated with nice names that can be invoked by using the `File Get par set` command. For example, consider the three different ways to integrate the harmonic oscillator:

```
# harmonic oscillator
x'=y
y'=-x
@ dt=.2,total=10
set euler {meth=euler}
set backeul {meth=backeul}
set rk4 {meth=rk4}
done
```

Use it with the `-silent` mode of *XPPAUT* and *XPPAUT* will dump out three data files called `euler.dat`, `backeul.dat`, `rk4.dat`.

To go one step better, use the parameter sets with the `array` trick to go through a range of sets. Here is a stupid example where I integrate $x' = -x$ for 11 different initial conditions. I use many of the tricks discussed earlier:

```
# integrate over a range of initial conditions
# silently
x'=-x
global 0 t {x=a}
!a=f(index)
par index=0
f(x)=.1*x
set x[0..10] {index=[j]}
@ total=5
done
```

Run this in silent mode and you will get 11 data files that contain the outputs from the solutions to $x' = -x$ with $x(0) = .1 * j$ and $j = 0, \ldots, 10$.

Parameter sets can contain any of the declarations used in option files as well as initial data and parameter declarations.

5. Take advantage of the one-parameter range integration as well. If all you are varying is one parameter or initial condition, then you can set the ODE file for a range integration in silent mode. Here are two examples. In the first, one file is produced which contains all the runs:

```
#ex1.ode
x'=y
y'=-x
@ rangeover=y,rangestep=10,rangelow=-1,rangehigh=1
@ range=1,rangereset=no
done
```

Here the initial value of y varies between -1 and 1. If you type `xpp ex1.ode -silent` one output file will be produced that has all the integrations together. There are occasions when you want to do this; I doubt this is such an occasion. In the second example, 11 different files are produced, one for each run:

```
#ex1.ode
x'=y
y'=-x
@ rangeover=y,rangestep=10,rangelow=-1,rangehigh=1
@ range=1,rangereset=yes
done
```

The only difference is `rangereset=yes` which tells *XPPAUT* to reset the storage at every run. In `silent` mode, *XPPAUT* produces files with names, `output.dat.0`, `output.dat.1` , etc.

6. If you have many sets of many parameters and want to run these in a single run in batch mode, keeping each run in a separate file, then here is how to do it. I will use an example with three parameters a, b, c taking on 4 sets of values (1, 2, 3), (1, 2, 4), $(-1, 0, 2)$, and $(2, -2, 6)$. I first make a table with 12 entries; the first 3 are for set #1, and so on. Here is the transposed table called `pars.tab`:

```
12 0 11 1 2 3 1 2 4 -1 0 2 2 -2 6
```

I make an ODE file with a parameter for the run number, use range integration, and define parameters to complete the file. Here is my example:

```
# par.ode
#
table pp pars.tab
par n
!a=pp(3*n)
!b=pp(3*n+1)
!c=pp(3*n+2)
x'=-x+a
y'=-y+b
z'=-z+c
aux my_a=a
aux my_b=b
aux my_c=c
@ total=10
@ range=1,rangeover=n,rangelow=0,rangehigh=3,rangestep=3
@ rangereset=yes
done
```

I have added auxiliary variables as a "hardcopy" of the actual parameter values used. The `n` parameter runs from 0 to 3 in increments of 1. The table `pp` has a domain of 0 to 11 with the first 3 integers corresponding to the first 3 parameter values, and so on. That is, `pp(2*3+1)` is the value of `b` for the third run. Run this with the line `xpp par.ode -silent` and you will get 4 files; one for each run.

7. You can make little inline tutorials by using quoted comments that have an action associated with them. The user invokes these by clicking on `File PrtInfo` which creates a window with the ODE source code. Clicking on `Action` in this window produces special comments from the ODE file and certain associated actions. For example, here is a linear planar system `planar.ode` with a built-in tutorial about the nature of the fixed points:

```
# the linear planar systems
# planar.ode
x'=a*x+b*y
y'=c*x+d*y
par a=-1,b=0,c=0,d=-2
init x=2,y=0
@ xp=x,yp=y,xlo=-5,ylo=-5,xhi=5,yhi=5
# here is a tutorial
"       Linear planar systems
" There are several different behaviors for the phase-plane\
 of a linear 2D system
" To see these click on the (*) and then DirField Flow 5
" {a=-1,b=0,c=0,d=-2} 1. Stable node - two real negative
     eigenvalues
" {a=0,b=1,c=3,d=0} 2. Saddle point - a positive and negative
     eigenvalue
" {a=-1,b=3,c=-2,d=0} 3. Stable vortex - complex eigenvalues \
with negative real parts
" {a=.5,b=2,c=-2,d=-.25} 4. Unstable vortex - complex with \
positive real parts
" {a=.5,b=0,c=-.5,d=.5} 5. Unstable node - two positive \
     eigenvalues
" {a=.5,b=2,c=-2,d=-.5} 6. Center - imaginary eigenvalues
" {a=-1,b=1,c=2,d=-2} 7. Zero eigenvalue
"
" Try your own values and classify them!
done
```

Note the lines that have the { } within them will be seen by the user as

```
* 7. Zero eigenvalue
```

so that what is being done is "invisible." Clicking on the * will result in everything within the curly brackets being done.

9.8 Dynamic linking with external C routines

XPPAUT has a way of allowing you to define the right-hand sides of your ODE directly in C code. Why would you want to do this? Occasionally, you may have a tremendously complex right-hand side that is the output from some computer algebra system. Most of these systems, (e.g., Maple or MATHEMATICA) have a command that allows you to output the algebra in C or FORTRAN code. It is then a simple matter to edit this code, compile it, write an ODE file which exploits this and then run *XPPAUT*. Note that you do not have to recompile *XPPAUT* at all. The dynamic linking is done inside *XPPAUT*. You should make sure that you are running a version of *XPPAUT* with dynamic linking enabled. The default is to not use it. Another example is when you just cannot figure out how to implement the right-hand side in *XPPAUT*.

This example is taken directly out of the users' manual for *XPPAUT*. I will write a C file that has three different right-hand sides. I will then switch between them on the fly while in *XPPAUT*. In order to allow *XPPAUT* to communicate directly with the C file, you have to include a line in the ODE file of the form

```
export {x,y,a,b,t} {xp,yp}
```

where the first group of variables and parameters are values that you want to pass to the external routine. The second group usually consists of `FIXED` variables that you want the routine to pass back to you. So, here is an ODE file for a two-dimensional system with a bunch of parameters:

```
# tstdll.ode
# test of dll
#
# In XPP click on File-Edit-Load Library
# and choose libexample.so
# Then pick either lv, vdp, duff as the function.
x'=xp
y'=yp
xp=0
yp=0
export {x,y,a,b,c,d,t} {xp,yp}
par a=1,b=1,c=1,d=1
init x=.1,y=.2
done
```

I have exported both the state variables as well as all four parameters and the independent variable `t`, so the C program will set the fixed variables `xp,yp` to the desired values depending on the state variables, etc. These are then the true right-hand sides of the system.

Next, you must write some C code to communicate with *XPPAUT*. Here is the three-right-hand example.

```
#include <math.h>
/*
 some example functions
*/

lv(double *in,double *out,int nin,int nout,double *var,
     double *con)
{
  double x=in[0],y=in[1];
  double a=in[2],b=in[3],c=in[4],d=in[5];
   double t=in[6];
  out[0]=a*x*(b-y);
  out[1]=c*y*(-d+x);
}
```

```
vdp(double *in,double *out,int nin,int nout,double *var,
     double *con)
{
  double x=in[0],y=in[1];
  double a=in[2],b=in[3],c=in[4],d=in[5];
   double t=in[6];
  out[0]=y;
  out[1]=-x+a*y*(1-x*x);

}

duff(double *in,double *out,int nin,int nout,double *var,
     double *con)
{
  double x=in[0],y=in[1];
  double a=in[2],b=in[3],c=in[4],d=in[5];
 double t=in[6];
  out[0]=y;
  out[1]=x*(1-x*x)+a*sin(b*t)-c*y;
}
```

Each right-hand side has the form

```
rhs( double *in, double *out, int nin, int nout, double *var,
     double *con)
```

The double array `in` contains all the exported variables, etc. in the order you exported them. So `in[0]` contains the value of `x`. The double array `out` should be set to the values that you want to send back to the fixed variables; `xp` will have the value of `out[0]`, etc. The integers, `nin,nout` are just the dimensions of the arrays. Finally, two more arrays are passed which you can generally ignore. These contain the complete list of all state and fixed variables as well as all parameters. They are organized as follows for the above example:

```
con[2]=a,con[3]=b,con[4]=c,con[5]=d
v[0]=t,v[1]=x,v[2]=y,v[3]=xp,v[4]=yp
```

That is, the array `con` contains all your parameters in order, starting with `con[2]`, and the array `var` contains the time variable, followed by the state variables, and then the fixed variables. Thus, you could directly communicate with all the variables or parameters.

Once you have written your C file, you should compile it and create a library. Here is how to do that:

```
gcc -shared -fpic -o libexample.so funexample.c
```

This just compiles it as a relocatable object file and creates a shared library.

Now run *XPPAUT* with the file `tstdll.ode`. Click on `File Edit Load DLL`. Choose the library you have created (`libexample.so`) and then choose one of the three

functions, `lv`, `vdp`, `duff` which are, respectively, the Lotka–Volterra model, the van der Pol oscillator, and the forced Duffing equation. Now integrate the equations and you should see the Lotka–Volterra (or whatever) solutions. Change parameters, etc., and these will be reflected in the solutions to the ODE.

9.8.1 An array example

The `export` directive is useful for small numbers of variables or parameters but is not so good if you are exporting a big array of values. In this example, I consider an array of 51 coupled oscillators with nearest neighbor coupling. I only need equations for 50 oscillators since the relevant variables are the relative phases. The differential equations are

$$x_j' = H(x_{j+1} - x_j) + H(x_{j-1} - x_j) - x_0',$$

where

$$H(u) = a_0 + a_1 \cos u + a_2 \cos 2u + b_1 \sin u + b_2 \sin 2u + b_3 \sin 3u.$$

I have subtracted x_0' so that this represents the relative phases. The key point to remember is that the ordering for the storage of the time variable, the dependent variables, and the fixed variables is `t,v1,v2,...,f1,f2,...,` where `v` are the variables and `f` are the fixed variables. Here is the ODE file, `chain.ode`:

```
# this is a chain of 50 oscillators using
# dll's to speed up the right-hand sides
x[1..50]'=xp[j]
xp[1..50]=0
par n=50,a0=0,a1=.25,a2=0,b1=1,b2=0,b3=0
export {a0,a1,a2,b1,b2,b3,n}
@ total=100
done
```

Notice that I have defined 50 fixed variables which will contain the right-hand sides. They are set to 0 but will be altered by the external library routines. I pass the number of oscillators and the parameters for the interaction function. I do not need to import anything since this will all be done via the fixed variables. Here is the C code for the right-hand sides:

```
#include <math.h>
#define H(u) a0+a1*cos(u)+a2*cos(2*u)+b1*sin(u)+b2*sin(2*u)
     +b3*sin(3*u)
f(double *in,double *out,int nin,int nout,double *var,
     double *con)
{
 int i;
 double a0=in[0],a1=in[1],a2=in[2];
 double b1=in[3],b2=in[4],b3=in[5];
 int n=(int)in[6];
 double *x=var+1;
```

```
 double *xdot=x+n;
 double x0dot;
 x0dot=H(x[0]);
 xdot[0]=H(-x[0])+H(x[1]-x[0])-x0dot;
 for(i=1;i<(n-1);i++)
 xdot[i]=H(x[i+1]-x[i])+H(x[i-1]-x[i])-x0dot;
 xdot[n-1]=H(x[n-2]-x[n-1])-x0dot;
}
```

This code is very flexible in that I could use it for any size array since the number of oscillators is exported. The time variable, dependent variables, and fixed variables are sent in the array `var`. Thus, `var[0]=t`, `var[1]=x1`, and so on. The fixed variables are `var[1+50]=xp1`, etc. These are sent back to XPP to use as the right-hand sides. I define some pointers and some macros to make the file easier to write and read. The pointer `*x=var+1` points to the first variable and `*xp=x+n=var+n+1` points to the first fixed variable.

To run this, fire up *XPPAUT* on the file `chain.ode`. Compile the C file with the line

```
cc -shared -o chain.so -fpic chain.c
```

where I have called it `chain.c`. In *XPPAUT*, click on `File Edit Load DLL`. Choose `chain.so` from the list and for the function name type in `f`. Now let it rip. Change parameters and see what happens.