Instructions for the Matlab-ODEs Files

This collection of simple Matlab scripts and m-files provides a simple system for plotting direction fields and numerically generated solutions of systems of ODEs in 1, 2, and 3 dimensions. I have used it for teaching both undergraduate and graduate courses. In addition to classroom demonstrations I've had students use it for assigned homework.

Getting Started. Create a folder on your computer in which to keep working copies of the files.

Start Matlab and set its working directory to the folder where you have placed your copies of the files. The easiest way to do that is click the small "..." icon near the upper right corner of the Matlab window and navigate to the desired folder in the usual way. To confirm that the working directory is set correctly, give Matlab the command **`what`**. It should respond by listing all the files you have placed there.

Next type **`setup`** at the command prompt. This will clear away all pre-existing Matlab variables, and create some new ones which are essential to the plotting commands.

Basic Design. The commands use the following essential variables.

`dimension` - the dimension (1,2, or 3) of the example under consideration.
`eqn_name` - the name of the m-file defining the right side of the example under consideration.
`parameter` - a (global) variable used to pass various parameter values to the m-file defining the right side of the differential equation.
`t0, x0` - initial conditions for the solution to be computed.
`Tinc` - length of the time interval over which solution is to be computed.

You can enter values for them directly, but in many cases there are *.mat files which you can load to set all of them to preselected values. (See below.)

Once the above variables are specified, you can use the following commands:

`point` - in dimensions 1 and 2 this sets the initial conditions (**`t0, x0`**) to the point you click on in the graphics window
`sol` - solves from the current initial conditions and plots the solution
`dirf` - plots a standard direction field (in 1 or 2 dimensions)
`level` - plots level sets in dimensions 1 or 2 (for isoclines, Lyapunov functions, etc...)
`wash` - wipes the graphics window clean (but saves settings)

How these work is described in more detail below.

Dimension 1: Scalar Equations: $x'=f(t,x)$.

To work with first order equations, after you enter **`setup`** give Matlab the command

```
load scalar
```

This will set **`dimension=1`** and **`eqn_name='eqn1d'`**. The function **`eqn1d.m`** evaluates the right side based on a string expression specified in the common variable **`parameter`**. For example if we are going to work with the differential equation $x'=2t-x$ then we would make the following assignment:

```
parameter='2*t - x'
```

Any valid Matlab expression can make up the string, as long as the only two variables used are **`t`** and **`x`**. For instance **`'exp(t^3 -1)/sqrt(log(x^2+1))'`** would be fine. It is important that the expression be inside single quote. The string variable must be named **`parameter`** and **`t`** and **`x`** can be the only variables involved. (The **`setup`** command made sure that it was declared to be a common variable.)

Next you need to specify the portion of the t,x plane that will be displayed . To work in the rectangle $a<t<b$, $c<x<d$, give Matlab the command **`axis([a,b,c,d])`**. The t-axis is horizontal, the x-axis is vertical. (It you later decide you want a different section of the plane displayed, just reissue the **`axis`** command with your new values. You would then **`wash`** the graphics window (see below) and replot the slope field, solutions, or whatever you want to display.)

`dirf(n)` This plots the slope field for the differential equation $x'=f(t,x)$ based on the expression in **`parameter`**. It will use a grid of $n \times n$ points filling the window specified by the most recent **`axis`** command. I find n=15 or 20 pleasing, but you pick a value to suit yourself.

The variables **`t0`**, **`x0`** specify initial conditions for a solution. You can enter values for them directly at the command line, or you can type the command **`point`**. If you type **`point`** then move the mouse over the graphics window. Crosshairs will appear and when you click on a point; it's coordinates will be placed in **`t0`**, **`x0.`**

To calculate and plot the solution with the specified initial conditions use the command **`sol`**. For 1-dimensional examples it will compute the solution both forward and backward in time to cover the range of t-values corresponding to the current graphics window. If the solution isn't defined for that range of t-values the ODE solver may get bogged down. (If this happens to you the *<control>-C* keystroke combination will make MATLAB give up its current calculation and come back to the command prompt.) To plot numerous solutions to fill up a phase portrait, you can just use **`point, sol`** repeatedly.

If you want to plot isoclines use **`level(parameter,c)`** This will plot the isocline $f(t,x)=c$. You can give it a list of values to get several isoclines at once: **`level(ftx,[c1,c2,c3])`**. You can also use this command to add other curves to your plot by giving it a different string than **`parameter`**. For instance if you want the circles of radii 1 and 2 to appear in your picture you could type **`level('t^2+x^2',[1,4])`**. You could plot level sets of a Lyapunov function in the same way.

As you experiment with an example, you will often plot things that you later wish you could remove from your picture. There is no way to remove some parts of the figure but not others, but the command **`wash`** will wipe your graphics window clean while saving all the variables and settings. You can then replot just the things you want.

Dimension 2: Planar (Autonomous) Systems: $x'=f_1(x,y)$, $y'=f_2(x,y)$.

Examples in 2 or 3 dimensions work a little differently. The definition of the right sides of the equations is no longer handled by a string variable, but by specifying an *.m file that will evaluate the right side. *.m files for several different examples are provided for you. (You can write your own if you want to try other

examples.) For example, consider the pendulum equations:

$x'=y$
$y'=-a\,y-b\,\sin(x)$,

where *a* and *b* are parameters. The right side is provided in **pendulum.m**, which looks for the global varable **parameter** to determine the parameter values: **parameter=[a,b]**. The name of the *.m file you want to use should be specified as a string in the variable **eqn_name**. So for this example we want **eqn_name='pendulum'**. But you don't need to know all these details. To work with the pendulum example simply load the file **pendulum.mat** by entering the command

load pendulum

When Matlab reads the file **pendulum.mat** it will find values specified there for **eqn_name='pendulum', dimension=2, parameter=[0,1],** and **Tinc=3.** Note that by using **[0,1]** for **parameter** means it will evaluate the pendulum equations with *a=0* and *b=1*, giving the standard simplified pendulm equations. **Tinc** specifies the length of time over which **sol** will calculate and plot the solution. You can change any of these variables from the command line if you want to modify the example (and save them in a new *.mat file of your own if you want to work with that example again in the future).

Before plotting solutions you will want to specify the size of the graphics window with an **axis([a,b,c,d])** command. For the pendulum example I find **[0,10,-5,5]** suitable.

dirf(n) works the same as before.

The initial conditions are again **t0** and **x0**, except that now **x0** is a vector of 2 coordinates. You can specify them from the command line, or use the **point** command to read **x0** from the point you click on in the graphics window. (That will also set **t0=0**.) When you enter **sol** it will compute and plot the solution for $t_0<t<t_0+Tinc$. Unlike one dimension it does not solve backwards in time. When **sol** is finished it will leave the final values of the solution in the variables **t0** and **x0**, so that simply typing **sol** a second time will calculate and plot more of the same solution, simply picking up where it left off the last time. If you find yourself typing **sol, sol, sol** repeatedly to see as much of a solution as you want, then you probably should increase **Tinc**.

level(*'string expression'***,c)** still works if you want to see contours of a Lyapunov function for instance. Now, however, you should use expressions involving variables **x** and **y**.

wash works the same as before.

Here is a list of the 2-dimensional examples for which *.m files are provided. For each of them you can ask for help on the filename to see what it expects to find in parameter (e.g. **help pendlm**). The *.mat files associated with each of these are listed in parentheses. You can simply load them to see what parameter values they contain.

cclin.m (**node1.mat, node2.mat, spiral.mat, center.mat**) Constant coefficient linear systems: y'=Ay (A is **parameter**).
species2.m (**compspec.mat, predprey.mat**) The basic model of two interacting species model (with 6 parameters).
fish.m (**fish.mat**)
Hahn.m (**Hanh.mat**) A modified example of Hahn in which all solutions converge to 0 but 0 is unstable.
pendulum.m (**pendulum.mat**) The pendulum example
vanderPol.m (**vanderPol.mat**) The van der Pol example exhibiting a limit cycle.

Dimension 3

In three dimensions the commands work essentially the same as for two dimensions, with two exceptions. There is no way way to click on a point in the graphics window to read three coordinates. If you try to use **point** it will just tell you to enter the initial conditions at the command line. And **dirf** would be useless in three dimensions, so it does nothing. The axis command now requires 6 entries: **axis([a,b,c,d,e,f])**. You my want to just leave the **axis** set to **auto** and let Matlab handle it.

There are three 3-dimanesional examples provided are as follows. For the significance of these, see the text by Hale and Kocak.

Lrnz.m (**Lrnz.mat**) The famous Lorenz system. (Don't overlook Matlab's built-in simulator for this.)
Rossler.m (**Rossler.mat**)
torex.m (**torex.mat**)