

An Introduction to MATLAB

Michael Vitus

January 13, 2007

1 Introduction

MATLAB is a "programming" application which allows its users to perform a variety of complex scientific calculations and graphical visualization. MATLABs primary use is with numerical calculations, but it does have the ability to perform symbolic manipulation through a Maple kernel (although a little awkward at times). In this class, we will be using the numerical packages, ode integration, and graphical capabilities mostly.

MATLAB is available on campus with all of the required toolboxes at various locations. Specifically, in the Terman Engineering Center with the elaine computer cluster and the Gates Computer Science Building with the myth computer cluster. It is also available on most of the Macintoshes on campus, including those in Green library, Meyer library and the Tressider Union cluster.

2 Getting Started

MATLAB can be started by double clicking on the program icon on a Windows/MAC machine or by executing the command **matlab** in a unix/linux environment. After MATLAB has started up, you should see the command window open up. The command window is used to enter single commands.

One of the reasons that MATLAB is very easy to learn is because of its extensive help system. To access the help system just type the command **help** in the command window and a list of help topics will be displayed. If you know the name of the command you want help on, use **help command** to get information about it. For example **help sqrt** would display:

```
SQRT   Square root.
      SQRT(X) is the square root of the elements of X. Complex
      results are produced if X is not positive.
```

To perform calculations MATLAB uses the standard mathematical operators (+,-,*,/) and ^ for exponents. The common commands for trigonometric functions are **sin**, **cos**, **tan**, and **asin**, **acos**, **atan** for the inverse trigonometric functions.

3 Arrays

MATLAB is an acronym for Matrix Laboratory, and therefore you can assume that it is very efficient operating with matrices and vectors. In fact, MATLAB is optimized for operating with matrices. To construct a matrix in MATLAB you list its elements row by row separated by spaces or commas and each row separated by a semicolon. For example:

```
>> A = [1 2 3 4 5; 6 7 8 9 10]
A =
     1     2     3     4     5
     6     7     8     9    10
```

There are several built-in functions that generate useful matrices:

- **eye(m,n)** generates an m x n matrix with ones on the main diagonal and zeros elsewhere. If m = n, **eye(n)** can be used instead.
- **zeros(m,n)** is an m x n matrix whose elements are all zeros
- **ones(m,n)** is an m x n matrix whose elements are all ones

The element in the i-th row and j-th column of a matrix A can be accessed by **A(i,j)**. Unlike C/C++ programming, the first element of an array is index by **1**. MATLAB also has a specialized command to access the last element of the row/column with the **A(1,end)**, which will access the element in the first row and last column of the array. Examples are shown below:

```
>> A = [1 2 3 4 5; 6 7 8 9 10];
>> A(2,2)
ans =
     7
>> A(1,end)
ans =
     5
>> x = [1 2 3 4 5]
x =
     1     2     3     4     5
>> x(1)
ans =
     1
>> x(end)
ans =
     5
```

The standard mathematical operators will work on arrays, assuming that the dimensions of the arrays are correct. Some useful matrix commands are:

- **inv(A)** - Inverse of the array
- **A'** - Transpose of the array
- **size(A)** - Size of array.
- **length(x)** - Length of a vector

The help topics that refer to operations with arrays are **help elmat** and **help matfun**.

4 MATLAB Scripts

MATLAB also has the functionality of instead of entering individual commands at the commandline, you can create a script and then execute them all at once. MATLAB provides an editor to write these scripts. To create a new script, go to **File**→**New**→**M-file**, and this will bring up the MATLAB Editor where you can start typing your commands. Once you are done entering commands in the script, you can either go to **Debug**→**Run** or hit **F5** to execute the script. Also, the reason that they are called 'M-files' is because they have a '.m' file-extension.

5 Functions

MATLAB also allows you the functionality of creating a user-defined function. To create your own function, you need to save it as a new M-file. The syntax for creating a user-defined function is:

```
function returnVariables = functionName(inputVariables)
    users commands
```

The keyword 'function' signals to MATLAB that this is a function definition. The variable **returnVariables** is the name of the variable in which the value will be returned. You can return multiple variables as well by replacing **returnVariables** with [**returnVariable1**, **returnVariable2**]. An example of a function definition is shown below:

```
function var = addTwo(a,b)

var = a + b;
```

An example of using this function:

```
>> addTwo(1,2)
ans =
     3
```

For more information on functions type **help function** at the command prompt.

6 ODE Solvers

MATLAB provides a number of ordinary differential equation (ODE) solvers. The ODE solvers will numerically integrate the system from a start to final time. Since MATLAB is numerically integrating the equations, there will be some numerical error due to the nature of the methods. The different ODE solvers that MATLAB has are: ode23, ode113, ode15s, ode23s, ode23t, ode23tb, ode45. The function definition for the ODE solvers is:

```
[tout,yout] = ode*(odeRatesFunction, [t0 tFinal],y0)
```

where '*' would be replaced with the specific ODE solver that you wanted to call, **tout** is the time output in column vector format, **yout** is the state output with each state as a different column, **odeRatesFunction** is the function that defines the derivative of the state, **t0** is the initial time, **tFinal** is the final time of the numerical integration, and **y0** is the initial condition of the state.

The function prototype for the derivative of the state is **odeRatesFunction(t,y)**. The first argument of the function is the current time and the second argument is the current value of the state. An example of a function which defines the derivative of the state is in **rates.m** which is shown below:

```
function ydot = rates(t,y)
ydot(1,1) = y(3);
ydot(2,1) = y(4);
ydot(3,1) = -0.1;
ydot(4,1) = 0.2;
```

An example of using this function and calling an ODE solver is shown below:

```
y0 = [0;0;0;0]; %initial conditions
tInitial = 0; %initial time
tFinal = 10; %final time
```

```
[tout, yout] = ode23('rates',[tInitial tFinal],y0); % call to the ode solver
```

The help topic that refers to the ode solvers is **help funfun**.

7 PPlane by Professor Polking at Rice University

Professor Polking has developed a MATLAB and even a stand-alone Java version of an extremely using tool for creating phase portraits. The homepage is: <http://math.rice.edu/~dfield/> . From that page there are links to download pplane* for MATLAB, depending on your version, or to run the Java stand-alone version which can be useful if you don't have MATLAB on your home machine.

If you are using MATLAB, be sure that the path is set to the folder containing the pplane*.m script, which can be done by going to **File**→**set path** and clicking **Add Folder** and browsing to the folder that contains the script. By doing this, you can just type pplane*, which corresponds to the version that you downloaded (ex. pplane7), and the pplane software will load.

Once the software has started. The first two lines are used to enter the two differential equations. You can use any name for the two variables, but just make sure that you are consistent in defining the differential equations. The software also has the ability of defining constants to use in the differential equations so that it makes it easier to change parameters. You can do this by entering variables into the *Parameters or expressions* section. Once you have the differential equations defined, adjust the values of the display window to include the area of interest. To draw the phase portrait click **Proceed**, and this will bring up another window with the plot.

From here, you can click on the plot and it will draw the trajectory of the solution that goes through that point. This is good for getting a better visualization of the orbits. Sometimes when you try to add a trajectory to the plot the odesolver will keep integrating the trajectory forever due to a nearly closed orbit, etc.... Luckily, you can stop the integration with a button that pops up in the upper-righthand corner which says **stop**. By pressing **stop**, it will force the integration to stop, but will keep all of the results up to that point.