

The Official MATLAB Crash Course

Written mostly by Adam Attarian (arattari@unity)

September 14, 2007

As you may have heard, MATLAB is a powerful platform for mathematical and scientific computation. Compared to C or JAVA, it has an easy to use programming interface and many, many built in functions to do tons of different things. If you can learn its syntax and language now, it will pay off greatly for you in the future. We'll take you through an overview of its usage, and then focus on certain areas that you'll be likely to run into sooner rather than later.

General Usage & Overview

Interacting with MATLAB

When you first open MATLAB, notice the command window, workspace, and command history. The command window is where you'll give MATLAB its input and view its output. The workspace shows you all of your current working variables and other objects. Notice the help menu. This will become your favorite menu.

Just some housekeeping commands: to clear your command window, type `clc`. To clear your workspace (this is like resetting everything – use with caution), type `clear`. You can also clear a specific variable: just follow the `clear` command with the variable name.

To move around, use the standard UNIX commands. To change directories, use the `cd` command. To go back up a directory, use `cd ..`. To figure out where you are, type `pwd`. On the Unity Macs, you're automatically placed in the MATLAB Application folder when you use MATLAB. Problem is, you don't have write privileges there, so to get into your home directory, type `cd ~`.

Entering Data

On the surface, MATLAB is a huge, expensive, glorified, calculator. Everything (well, mostly) in MATLAB is created and treated as a matrix (MATLAB is short for MATrix LABoratory). MATLAB has the standard arithmetic operators that you'd expect: $+$, $-$, \div , \times , and exponentiation, \wedge . So entering

```
>> 5+6
```

This returns 11, as you would hopefully expect. To suppress the output (e.g you don't want to see the result), end your command with a semicolon. To enter a vector, put the input in brackets. For instance, to create a row vector in \mathbb{R}^4 , you could enter:

```
>> A=[4,1,5,6];
```

We have just assigned the vector $[4, 1, 4, 6]$ to the variable **A**. MATLAB is case sensitive, so the variable $\mathbf{a} \neq \mathbf{A}$. If you wanted to make that a column vector, just use semicolons instead of colons, or transpose the vector using the `'` operator. To enter a matrix, stick some rows together. For example, a 2×2 matrix **B** could be entered as:

```
>> B=[1 2; 3 4]    This enters the matrix  $B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ 
```

Some vectors would be unpleasant to type in all of their glory, and MATLAB understands that. For example, to obtain the vector $t = [0.5 \ 1 \ 1.5 \ 2 \dots 9 \ 9.5 \ 10]$, you would type `t=[0:.5:10]`, which creates a vector from 0 to 10 in half increments. Or, to get a vector containing equally spaced entries, you can use the `linspace` command. For example, to get 400 linearly spaced entries from 0 to 100, you could type `>> r=linspace(0,100,400)`. Some special cases:

- A scalar does not need brackets. `g=9.81;` will suffice.
- Square brackets with no elements (`X=[]`) creates a null matrix. Can be useful to delete rows/columns of a matrix. Example: if you wanted to delete the 4th row of matrix **A**, you could say `A(4,:)=[];`

Since everything in MATLAB is a vector or a matrix, it makes sense to be able to extract data from certain points of a vector. For instance, to obtain the 5th entry of a vector *r*, type `>> r(5)`. For a range of values, say from *a* to *b*, type `>> r(a:b)`. For a matrix, the syntax is (row, column). So to get the *i, j* entry of the matrix *B*, type `>> B(i,j)`. Similarly, you can pull out ranges of values as you could with a vector. If you wanted the entire range of rows, or columns, use the colon. So the row space of the fourth column of **A** would be `A(:,4)`.

Operations

You'll be pleased to know that the standard operations, `sin`, `cos`, `tan`, `arctan`, `log`, `ln`, and `exp` (all are not listed here) exist in MATLAB. You can apply these to vectors, matrices, whatever. If you wanted the sine of a vector, just do `b=sin(a)`. Some things to know:

- The `log` command is the natural log. For \log_{10} , use `log10`.
- `e` is not a protected operator. If you want e^5 , use `exp(5)`.
- MATLAB recognizes `i` and `j` as the imaginary unit.
- Matrix multiplication works as you'd expect, so long as the dimensions match of course. To multiply 2 matrices *A, B* just do `>> C=A*B`. MATLAB can also do element wise operations on matrices and vectors using the dot notation. For example whereas `>> C=A*B` is standard

matrix multiplication, `>> C=A.*B` will create a matrix C whose elements are defined as $c_{ij} = a_{ij} \cdot b_{ij}$. The dot notation works with multiplication, `*`, division, `/`, and exponentiation. You'll have to use this more often than you think.

Very Useful/Helpful Functions

Since you have the hard way to do things, here are some easy ways:

- `size()` returns the dimension of a matrix/vector. If you just want the length of a vector, use `length()`.
- `eye(n)` creates the $n \times n$ identity matrix.
- `zeros(n,m)` creates an $n \times m$ matrix of zeros. Useful for initializing vectors and matrices. Similarly, `ones(n,m)` creates a matrix of ones.
- `rand(n,m)` creates a $n \times m$ matrix with entries from a uniform distribution on the unit interval.
- `randn(n,m)` same as above, just now from a normal gaussian distribution.

So for example, entering

```
>> B=[ones(3) zeros(3,2); zeros(2,3) 4*eye(2)];
```

Produces the matrix $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 4 \end{pmatrix}$.

Exercise: Create the following matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 4 & 5 & 9 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Try some of these basic operations and see what happens: $A * B$, $A .* B$, $2 + A$, $A + B$. Now create a vector and a matrix with the following commands: `v=0:0.2:12; M=[sin(v); cos(v)];`. Find the sizes of `v` and `M` using the `size` command. Extract the first 10 elements of each row of the matrix and display them as column vectors.

Plotting Data / Making Pretty Pictures

Some say your data is only as good as you can make it look, so it is important to make it look *good*. The most basic way to plot 2D data is using the `plot` command: `>> plot(x, y, options)`, where x is a vector defining the independent vector and y is the data you want to plot and options is where you can specify all kinds of well, options, for your plot (type `>> help plot` for more on that). For example, if you want to plot $\sin(x)$ on the interval $[0, 2\pi]$, you would type

```
>> x=[0:.1:2*pi];
>> y=sin(x);
>> plot(x,y);
```

You can also plot several sets of data on the same figure, assuming each vector is the same length. One way is to use the `plot` command like so: `plot(x1,y2,x2,y2)`. Another (perhaps better) way is to use the `hold` command, which holds the plot on the current figure:

```
>> plot(x1,y1)
>> hold on
>> plot(x2,y2)
```

Since the beginning, we've been taught to label our plots. MATLAB allows you to do this through the command line or through the user interface. We could label *and legend* our sine plot like so:

```
>> xlabel('x');
>> ylabel('sin(x)');
>> title('My First MATLAB Plot');
>> legend('sin(x)', 'Location', 'SouthWest');}
```

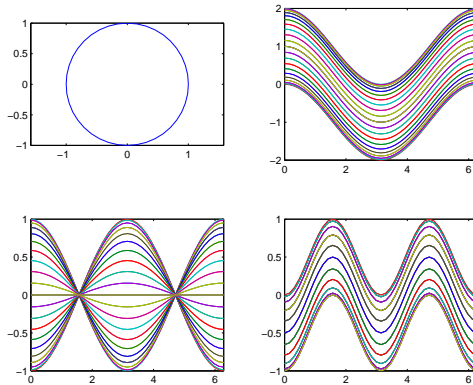
Notice the `location` tag on the legend. This puts the legend in the lower left corner. Do `help legend` for more.

More on Plotting

Lets take a moment to delve a little more deeply into the art of generating plots in MATLAB. We can do a lot with our plots: we can change the linewidth, change the color, change the fonts (and size) of the titles, use \LaTeX notation, and much more. You can either do this via the user interface, or on the command line.

- **LineWidth.** To increase the thickness of the traces on the plot, add in the `LineWidth` option: `plot(x,y,'LineWidth',2)`. This will plot with a line width of 2.

- Changing Fonts. You can probably tell what is going on by this example:
`title('My Awesome Plot','FontName','courier','FontSize',20,'FontStyle','bold');`
 The font will be Courier, at size 20, bolded. Substitute your favorite in as you so desire.
- \LaTeX Inclusion. MATLAB, by default uses \TeX to render the axes and titles, which can be inadequate when we're used to using \LaTeX characters. To get around this, we need to specify the \LaTeX interpreter. If I wanted to have $\frac{\partial u}{\partial t} = e^{-t}$ as the title to my plot, I would use
`title('$\frac{\partial u}{\partial t}=e^{-t}$','Interpreter','Latex');`
- Axes. MATLAB gives you a few ways to modify your axes. `xlim` and `ylim` take a range vector as an input and will adjust your axes accordingly. To do it all at once, you can do `axis([xlow xhigh ylow yhigh])`.
- Grids. Type `grid on` (and `off`) to get a grid on your plot. Useful for journal quality plots.
- Subplots. Use these. You can display multiple plots in the same figure window and print them on the same piece of paper with this function. Check out this plot:



This is the code that was used to generate the plot:

```
t=0:pi/20:2*pi;
[x,y]=meshgrid(t);

subplot(2,2,1)
plot(sin(t),cos(t)); axis equal;

subplot(2,2,2)
```

```

z=sin(x)+cos(y);
plot(t,z);
axis([0 2*pi -2 2]);

```

```

subplot(2,2,3)
z=sin(x).*cos(y);
plot(t,z)
axis([0 2*pi -1 1]);

```

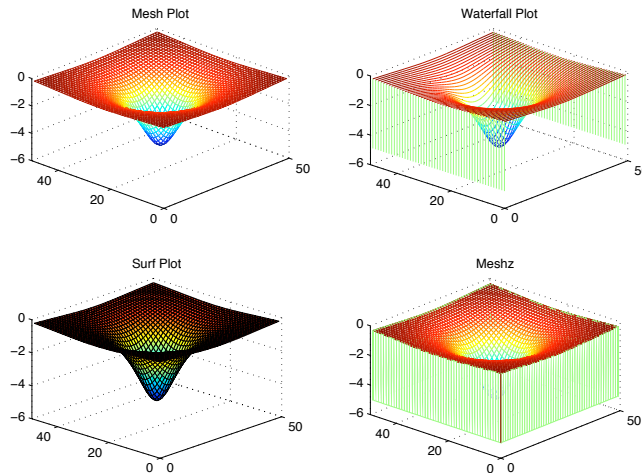
```

subplot(2,2,4)
z=(sin(x).^2)-(cos(y).^2);
plot(t,z);
axis([0 2*pi -1 1])

```

`subplot(m,n,i)` creates an $m \times n$ array of plots, and then you count through them with `i`, counting from left to right, top to bottom. Each subplot is rendered independently of the rest.

- Surface and 3D plots. There are two types of \mathbb{R}^3 plots in MATLAB, surface plots, and line plots. An example of a line plot is given in the next exercise where you plot a helix. Some examples of surface plots are `mesh`, `surf`, `surf1`, `waterfall`, and there are more. To really use these plotting tools, you'll need to know about `meshgrid`. `meshgrid` transforms vectors into arrays so that you can evaluate functions in \mathbb{R}^2 and plot them with these commands. Compare the plots below:



And the code used to generate:

```

x=linspace(-3,3,50);
y=x;
[x,y]=meshgrid(x,y);
z=-5./(1+x.^2+y.^2);

subplot(2,2,1)
mesh(z); title('Mesh Plot')

subplot(2,2,2);
waterfall(z); title('Waterfall Plot');
hidden off;

subplot(2,2,3);
surf(z); title('Surf Plot');

subplot(2,2,4);
meshz(z); title('Meshz');

```

Lastly, to export your figures for use in L^AT_EX and other nice programs, be sure to save the figures as EPS format. This way, you can use them in your T_EX documents with no problem at all.

Exercise: Plot the sine and cosine functions on the interval $[0, 2\pi]$ on the same graph. Use different colors and line styles for each trace, and label all aspects of the plot. Investigate and use the `legend` command. Also, MATLAB can also plot 3D data with the `plot3(x,y,z)` command. Try and use this command to plot the standard helix, where $x(t) = \sin(t)$, $y(t) = \cos(t)$, $z(t) = t$ on the interval $t \in [0, 20]$.

M-Files, Scripts, and other files

Up until now, we've been entering commands in the command window, one line at a time. While this may work for a few commands or if you're testing something, this definitely won't work for longer programs or anything remotely complicated.

To get around the command window, we can type things up in m-files, and then call and run the m-file from the command line. There are two main types of m-files that we'll be using: script files, and function files. The differences are big:

- Script files can use any and all variables in the workspace when the script is run, and any results of the script are leftover in the workspace when complete.

- Functions can take dynamic inputs, but all variables are local to the function and are not explicitly saved in the workspace when complete.

Why is this important? We'll talk about it. To write an m-file, you can either use your favorite text editor on your computer or (even better) the builtin MATLAB file editor. To get started, you can either click the new file button in the main MATLAB window, or type `>> edit myscript`, (myscript being whatever you want to call your new m-file) and then you can go from there. Also, anything behind an `%` is a comment and is thusly ignored by MATLAB. **Comment your code as much as you can!** Here is an example of a script that plots the unit circle:

```
% CIRCLE - A script to draw the unit circle
% -----
theta=linspace(0,2*pi,100); % create a vector
x=cos(theta); % generate x-coords
y=sin(theta); % generate y-coords
plot(x,y,0,0,'+'); %plot data and put a + at the origin
axis('equal'); % set equal scale on axes
title('Unit Circle Plot');}
```

This is a script file. To run it, simply type `circle` at the command line. Note that you have to be in the same directory as the file in order for it to run. A **function file** is also an m-file, except that all the variables are local. A function file begins with a *function definition line*, which as a well-defined list of inputs and outputs. Without this line, the file is simply a script. The syntax is:

```
function [output variables] = function_name(input variables)
```

where the *function_name* must be the same as the filename (without the .m extension). For example, the function definition `function [rho,H,F] = motion(x,y,t)` takes (x,y,t) as inputs and can return the variables (ρ, H, F) to the user. To execute this example function, you would type:

```
>> [r, angmom, force] = motion(xt,yt,time);
```

The input variables `xt`, `yt`, `time` must be defined beforehand, and the output variables will be saved in `r`, `angmom`, `force`.

Exercise: Turn the `circle.m` script file above into a function that accepts an arbitrary radius r , and plots the resulting circle.

Basic MATLAB Programming

Things can get complicated quick in MATLAB. Knowing some basic programming techniques can really speed things along. We'll go over loops, global variables, and conditional boolean algebra (what?).

For Loops

There are two kinds of loops we'll use: `for` and `while` loops. A `for` loop is used to repeat a statement of a group of statements for a *fixed* number of times. An example:

```
for m=1:100
    num = 1/(m+1)
end
```

This will print out the value of $\frac{1}{m+1}$ for $1 \leq m \leq 100$. The counter in the loop can also be given by an explicit increment, like `for i=m:k:n`, to advance the counter `i` by `k` each time. You can have nested `for`-loops, but each must match with its own `end`.

While Loops

A `while` loop is used to execute a statement or a group of statements for an indefinite number of times until the condition specified by `while` is no longer satisfied. An example!

```
% find all the powers of 2 below 10000
v=1; num=1; i=1;
while num < 10000
    num = 2^i;
    v = [v; num];
    i = i+1;
end
```

Once again, a `while` loop needs a matching `end` statement.

If Statements

Frequently you'll have to write statements that will occur only *if* some condition is true. This is called a branch statement. Typically you'll have an `if` statement, and if it *isn't* satisfied, a corresponding `else` or `elseif` statement.

Along with the standard greater than, and less than conditions (`a>b`, `b<a`, `a>=b`, `b<=a`), there are also:

- **and:** `a & b`
- **or:** `a | b`
- **not-equal:** `a ~= b`

- **equal** (notice the two equal signs!): `a==b`

Observe:

```
i=6; j=21;
if i >5
    k=i;
elseif (i>1) & (j==20)
    k=5*i+j;
else
    k=1;
end
```

Of course you can (and probably will) nest `if` statements, as long as you have matching `end` statements. Lastly, you can define variables to be global. This is handy if you want one function to be able to access another variable, and vice-versa. To declare a variable, just say `>> global variable-name`. You'll have to make this declaration wherever you want access to the variable.

Debugging Output

Sooner or later, you'll have to debug your code. One thing that may help in this endeavor is the `disp()` command. `disp()` merely displays its input to the command window. Try `>> disp('Hello World!')`; Sticking a `disp` on key lines of your program can help you trace where MATLAB is when it gets stuck. Other useful commands in this department:

- `num2str()`: converts a double number to a string. If you want to display a number in a string, you'll need this function.
- `str2num()`: goes the other way. Useful for when you import strings of number from a file. In order to do anything with them, they have to be converted to doubles first.

Exercise: Create a 10×10 random matrix. Then, have some fun by:

- Multiply all elements by 100 and then round off all elements of the matrix to integers using the `fix` command.
- Replace all elements of $A < 10$ with zeros.
- Replace all elements of $A > 90$ with infinity (`inf`).
- Extract all $30 \leq a_{ij} \leq 50$ in a vector `b`, that is, find all elements of A that are between 30 and 50 and put them in a new vector.

Next, simulate a random walk. Say someone can walk right or left. They flip a coin. If heads, they take a single step right, tails they go left. The individual does this 5000 times. Can you plot the path? What if the coin were weighted to one side? Welcome to Monte-Carlo simulations.

Linear Algebra

Since everything in MATLAB is a matrix, it makes sense that MATLAB would be well-suited to linear algebra problems. For example, to solve the common problem $\mathbf{Ax} = \mathbf{b}$, where $\dim(A) = n \times m$, and x, b are vectors m long, you can simply type `>> x=A\b`. Consider the system:

$$\begin{aligned}5x - 3y + 2z &= 10 \\ -3x + 8y + 4z &= 20 \\ 2x + 4y - 9z &= 9\end{aligned}$$

Its coefficient matrix is

$$\mathbf{A} = \begin{pmatrix} 5 & -3 & 2 \\ -3 & 8 & 4 \\ 2 & 4 & -9 \end{pmatrix}$$

And the known constant vector is $\mathbf{b} = [10 \ 20 \ 9]^T$. To solve this system in MATLAB:

```
>> A=[5 -3 2; -3 8 4; 2 4 -9];
>> b=[10; 20; 9];
>> x=A\b
x =
    3.4442
    3.1982
    1.1868
```

Note that the backwards division operator is equivalent in usage to `inv(A)*b`. Some other useful linear algebra related commands:

- `rref(C)`: finds the reduced row echelon form of the matrix C
- `inv(A)`: computes the inverse of the matrix A , assuming it exists.
- `det(A)`: finds the determinant of an $n \times n$ matrix A .
- `[V,d]=eig(A)`: finds the $n \times n$ matrix V whose columns are eigenvectors and D is an $n \times n$ matrix with the eigenvalues of A on the diagonal. Solves the problem $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$.
- Built-in matrix factorizations: `[L,U]=lu(A)`, `[Q,R]=qr(A)`.
- `diag(v)`: generates a diagonal matrix with vector v on the diagonal. `diag(A)` extracts the diagonal of matrix A as a vector.

Solving ODEs

One of the more prolific problems that we'll deal with in MATLAB is numerically solving and plotting systems of ordinary differential equations (ODEs). MATLAB can only solve first order ODEs, so if you have higher order systems you'll have to devolve them to first order equations; we'll cover this below. The basic syntax for solving an ODE in MATLAB is:

```
[time, solution] = ode45(@odefun, [tspan], [ic], options, parameters)
```

where `ode45` is one available solver (a 4-5 order Runge-Kutta Method), `odefun` is your ODE file (more on this shortly), `tspan` is either an interval or a vector where the ODE will be solved, `ic` is an initial condition vector, and `options` are ODE specific options, such as tolerances. Solving most ODEs in MATLAB requires you to:

1. If the order is greater than 2, write the differential equations as a set of first order ODEs. This just involves introducing some new variables.
2. Write a function to compute the state derivative. This function needs to return out the state derivative $\dot{\mathbf{x}}$.
3. Call the solver, extract the desired variables, use knowledge to understand what is happening.

Example 1: A Very Simple Case

Consider the most fundamental differential equation out there, written in all our favorite notations:

$$\frac{dy}{dt} = y(t) \iff \dot{y} = y \iff y' = y$$

An introductory calculus class tells us that the solution to this is e^x . Lets code this up and compare it to the analytical solution. First thing first, we need to create our ode file. Here is one possible example:

```
function dy = simpleode(t,y)
dy = y;
```

To solve this, call at the command line `[t y]=ode45(@simpleode,[0,2],[1])`. This solves the function on the domain $[0, 2]$ with an initial condition of $y(0) = 1$. To plot, call the plot command: `plot(t,y)`. Now plot e^x along the same time steps and plot this on the same axis – should be the same thing. It may be worthwhile to compute the error between the ODE solution and the analytical solution to see how accurate the solver is.

Example 2: A System of Equations

Solve this system of equations:

$$\begin{aligned}\dot{x} &= 2x - y + 3(x^2 - y^2) + 2xy \\ \dot{y} &= x - 3y - 3(x^2 - y^2) + 3xy\end{aligned}$$

First thing, make the odefile. Here is mine:

```
function xdot = aode(t,y);
% y(1) = x
% y(2) = y
xdot = zeros(2,1);
xdot = [2*y(1) - y(2) + 3*(y(1)^2-y(2)^2) + 2*y(1)*y(2);
        y(1)-3*y(2) - 3*(y(1)^2-y(2)^2) + 3*y(1)*y(2)];
```

Some things to notice from this example: the input y is a vector, and we assume that $y(1)$ is the x variable and $y(2)$ is the y variable. Also, note how we had to initialize the `xdot` vector. Next, call the ODE solver on the domain $[0, \frac{1}{2}]$ with the initial conditions $y(0) = 3, x(0) = 5$:

```
>> [t, y]=ode45(@aode,[0,.5],[3; 5]); plot(t,y).
```

For something more interesting, plot the *phase portrait*, that is, plot y against x :

```
>> plot(y(:,1),y(:,2))
```

Changing the initial conditions and dramatically change the phase portrait.

Example 3: Second Order Systems

Consider the equation of motion of a simple, non-damped, nonlinear pendulum:

$$\ddot{\theta} + \omega^2 \sin \theta = 0 \quad \Rightarrow \quad \ddot{\theta} = -\omega^2 \sin \theta, \quad \theta(0) = 1, \dot{\theta}(0) = 0.$$

To recast this equation as a system of two first-order equations that MATLAB can solve, we perform a variable substitution:

Let $u_1 = \theta$ and $u_2 = \dot{\theta}$. Then, $\dot{u}_1 = \dot{\theta} = u_2$ and $\dot{u}_2 = \ddot{\theta} = -\omega^2 \sin(u_1)$. To write the ODE file, it may help to see what this now looks like in vector form:

$$\begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \end{bmatrix} = \begin{bmatrix} u_2 \\ -\omega^2 \sin(u_1) \end{bmatrix}$$

Before we write the file, note that ω will have to be coded into the odefile or passed in as a parameter. The former is easiest, so we'll code it so we can pass it in as a parameter:

```
function udot = pend(t,u,omega)
udot = zeros(2,1);
udot = [u(2); -omega^2*sin(u(1))];
```

To call and pass the parameter $\omega = 1.56$, call like so:

```
>> [t, y]=ode45(@pend,[0 20],[1; 0],[],1.56);
```

The empty `[]` is a place-holder for ODE options. We're not using any here, hence the empty brackets. Go ahead and plot the displacement and velocity vectors, and then construct a phase portrait. As always, label and title your plots.

Increasing Resolution

By now you may have noticed that when you specify a time span in the solver, you get a seemingly arbitrary number of data points in your solution vector. The number of points is far from arbitrary, but rather is a result of the way the ODE solver works and the absolute and relative tolerances in the solver's algorithm from the n^{th} time step to the $(n + 1)^{th}$ time step. In fact, even if you specify a time vector the ODE solver will still solve the ODE where it wants to, and then linearly interpolate to your points. The only way to increase accuracy is to decrease the tolerance through the ODE options.

The default tolerances are 1E-3 and 1E-6 for the relative and absolute tolerances. The fastest way to get more data points is to decrease the relative tolerance to something smaller, like 1E-6:

```
>> options=odeset('RelTol', 1E-6);
```

You just now need to specify your `options` in your solver call (continuing previous example):

```
>> [t, y]=ode45(@pend,[0 20],[1; 0],options,1.56);
```

There are lot of other ODE options that will affect the performance, accuracy, and overall fidelity of the solver. Type `help odeset` to see what all is available. For casual usage of `ode45`, the default options are typically sufficient. By the way, if you don't assign any output variables to the solver call, the solver will automatically display a plot with the solutions.

Exercise: Consider the differential equation

$$y'' - 2y' + 10y = te^t \sin(3t), \quad y'(0) = 0, y(0) = 1$$

which could model the transient response of an RLC circuit. Solve and plot the resultant functions. If you want, compare the numerical approximation with an analytical solution (Maple may be helpful in that endeavor), or even the accuracy of `ode23` versus `ode45`.

Optimization Problems

One of the big problems we'll be doing this week is an optimization problem, specifically fitting data to a model. We'll go through a different problem, but use the same procedure so that the coding won't be as difficult as it could be.

The basic, most general minimization problem is

$$\min_{x \in \Omega} f(x)$$

where Ω is some domain where the solution lies. How to achieve this is the question. In many cases, you want to optimize the problem subject to constraints, be they linear, non-linear, or even PDE constrained. In this example, we'll just consider the unconstrained case.

A classic test example is the Rosenbrock banana function, $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$. To see what this surface looks like, try this:

```
>> [x,y]=meshgrid(-3:.1:3,-3:.1:3); z=100*(y-x.^2).^2+(1-x).^2; surf(x,y,z);
```

The minimum for the banana function is at (1, 1) and has value 0. We want MATLAB to automagically find it. First, write a function file that takes a vector input and outputs the function value that you wish to minimize:

```
function fx = banana(x)
fx = 100*(x(2)-x(1)^2).^2+(1-x(1)).^2;
```

To find the minimum in this unconstrained case, we'll use `fminsearch`, which attempts to find the minimum of a function, be it scalar, vector, or matrix valued. To use `fminsearch`, it needs only the function name and an initial iterate – where it should start to look. For this problem, let's start looking in the neighborhood of (2, 3):

```
>> [x,fvalue]=fminsearch(@banana, [2 3]);
```

This returns an answer that says it found a minimum at (1, 1) with a function value of $2.718E-10$, which is more or less zero. `fminsearch` is an algorithm that is *derivative free*. It doesn't need gradient information about the function in order to march along. Other algorithms, such as those implemented in `fmincon` (constrained optimization) and `fminunc` either need or approximate the gradient of the objective function. One last note: `fminsearch` and other algorithms can be very sensitive to the initial iterate. It's possible to become stuck in a local minimum and get an answer that is non-sensible. If this happens, just start looking somewhere else.

Curve Fitting and Least-Squares

Tomes have been written on the topic, and it is only fair that you have some sense of how to do it in MATLAB. This comes up very frequently in parameter estimation problems. To understand what we're doing, we'll motivate the topic with an example.

Suppose you've observed a rocket trajectory in flight and took position measurements from the ground. You know the flight path is in the general shape of a parabola, $ax^2 + bx + c$, and that your measurements are corrupted by noise. You want to determine the values a, b, c from the data. That is, you want to minimize the sum-of-square error between the data and the model. Or, minimize

$$J = \sum (y_{\text{model}} - y_{\text{data}})^2$$

Suppose the data that you collect is the following:

$$\text{data} = \begin{bmatrix} -0.44 & 3.09 \\ 2.38 & -22.82 \\ -3.67 & -75.35 \\ 3.19 & -44.67 \\ -2.46 & -33.55 \\ 0.767 & 2.57 \end{bmatrix}$$

where the first column is your x data, and the second column is the y data. Here is one way we can obtain the best-parameter values (that is, the best a, b, c to minimize J): Write a cost function that computes the error between the data and the model for a given $q = [a \ b \ c]^T$, and then minimize the cost using one of the algorithms. Here is my program that computes the cost:


```

function J = parab(q);

a=q(1); b=q(2); c=q(3);

data=[-.44 3.09;
      2.38 -22.82;
      -3.67 -75.35;
      3.19 -44.67;
      -2.46 -33.55;
      .767 2.57];

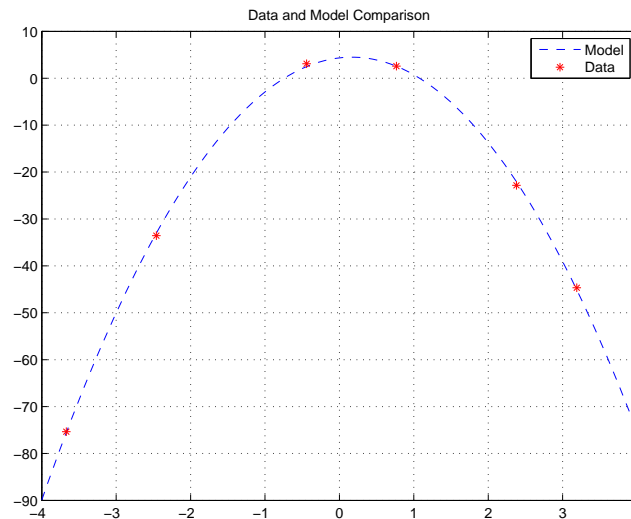
xdata=data(:,1);
ydata=data(:,2);

ymodel=a*xdata.^2 + b*xdata + c;

J=sum((ymodel-ydata).^2)/2;

```

Now, we just minimize J to obtain the parameters. I minimized with `fminsearch`, though other non-sampling algorithms will work here too. My result was a parameter vector of $\hat{q} = [-5.43 \ 1.83 \ 4.35]^T$ and a final cost function value of 0.7055. When I plot the data with the parabola generated by \hat{q} , here is what it looks like:



Lastly: Saving Your Work

Obviously, this is something that you should do often. To save your workspace and all of its variables, type `save workspacename`. This will save all of your variables and your current state to a `.mat` file that can then be loaded back into your workspace via the `load workspacename` command. You can also save and load specific variables contained in your workspace: `save workspacename variablename`.

To save figures and plots, you can either save them through the figure window (File → Save As...), or from the command line. If you are going to be including your figures in L^AT_EX documents, you need to save your plots as EPS or PDF format. To save a *single* figure (as in you only have one figure open) as a color EPS figure from the command, try using `>> print filename.eps -depsc`. If you're using PowerPoint, save in PNG format.

A few final thoughts: If there is a function in MATLAB that you feel should exist, then it probably does. MATLAB does exactly what you tell it to do, for better or worse. Good luck, and happy coding.

Original inspiration for this came from John David's MATLAB Tutorial from last year. Some of the examples came from the very good book "Getting Started with MATLAB", by Rudra Pratap, the MATLAB help files, and my old 341 book. Comments? email me: arattari@unity.ncsu.edu.