

# MATLAB, Statistics, and Linear Regression

Justin Werfel

9.29 Optional Lecture #1, 2/09/04

## 1 MATLAB

MATLAB is a powerful software package for matrix manipulation. It's a very useful language not only for this class, but for a variety of scientific applications, and is used widely throughout industry. Just as when you have a hammer, everything looks like a nail, so when you have MATLAB, everything looks like a matrix. You may learn to approach this Zen-like state by the end of the class.

### 1.1 Free your mind

The basic MATLAB data type is a matrix, an array of values (by default, double-precision floating point numbers), typically arranged as a two-dimensional<sup>1</sup> grid (though arrangements of more or fewer dimensions are possible).

The simplest matrix consists of a single element. We can assign a value to a variable with a statement like<sup>2</sup>

```
i = 4
```

Notice that when you enter this command, MATLAB echoes the result back to you. You can suppress that echo by ending the statement with a semicolon:

```
i = 4;
```

A vector (one-dimensional matrix) can be created with the colon operator. To make a row vector of successive integers, use a statement like  $x=2:9$ . The default increment between successive entries is 1; you can specify a different increment by putting it between two colons, e.g.,  $x=9:-2:3$  creates a row vector with the elements (9, 7, 5, 3).

The transpose operator in MATLAB is the single-quote; the easiest way to create a column vector, then, is to transpose a row vector, e.g.,  $x=(1:8)'$ . Note that parentheses in MATLAB, like in C and other languages, can be used to specify the order in

---

<sup>1</sup>There are at least two ways we can use the word 'dimension' when talking about matrices. One refers to the layout, in the sense that a line is one-dimensional, a square two-dimensional, etc. The other is in the sense of dimensionality of a matrix, e.g.,  $(x_1, x_2, x_3, x_4)$  is four-dimensional (although as a vector, its elements are arranged in a line when it's written; that is, a vector is one-dimensional in the first sense). MATLAB's help system tends to use the first sense, Sebastian tends to use the second. Context should hopefully always make it clear which sense is meant. If you have questions about any specific situation, ask.

<sup>2</sup>I'm not going to display the results of every operation here; try them out for yourself and see what happens.

which you want operations performed; if you type `x=1:8'`, MATLAB first transposes the element 3 (with no effect), then applies the colon operator and gives you a row vector as before.

To enter a two-dimensional matrix, use commas or spaces to separate entries on the same row, and semicolons or carriage returns to separate rows. For example, the statement

```
A = [1, 5, 3 6
     2 7 4, 4; 3, 8 3 8
     ]
```

specifies the matrix

$$\begin{bmatrix} 1 & 5 & 3 & 6 \\ 2 & 7 & 4 & 4 \\ 3 & 8 & 3 & 8 \end{bmatrix}$$

Parentheses are used to index matrices, and commas to separate dimensions; so for instance, with the last definition we used for `x`, `x(5)` is 5, and `A(1,4)` is 6.<sup>3</sup> You can specify ranges of indices with the colon operator; a colon by itself means all entries in that dimension; the keyword `end` specifies the last entry in a dimension. See what these statements give you:

```
A(2:3, 1:3)
A(:, 2)
A(3, 2:end)
A(1:end-1, 3)
```

Some operators have special meanings when applied to matrices (notably multiplication, `*`, and raising to a power, `^`); if you want to apply an operation to every element of a matrix separately, put a period before the operator. For instance, `(1:8)*(1:8)` gives an error because the inner dimensions of the matrices don't match; `(1:8)*(1:8)'` is an inner product, returning a single value; `(1:8).*(1:8)` gives you a row vector of the squares of the first eight counting numbers; so does `(1:8).^2`.

## 1.2 M-files, data sets, looping and why you should never do it

Suppose we've got some set of  $N$  sensors from which we can read measurements (temperature, voltage, whatever) all at the same time; and we sample from them repeatedly  $M$  times, so we have  $M$  sets of  $N$  values. Here's our first look at MATLAB as hammer: one natural way to store these data points is in a matrix  $A$ , say  $M$  rows by  $N$  columns. Then each row is a snapshot of the sensor readings at a given time;  $A(i, j)$  is the value the  $j$ th sensor had at time  $i$ . Let's further suppose that we want to calculate some weighted sum of the sensor readings at each time step; the weights are given by values stored in the column vector  $b$ .

<sup>3</sup>MATLAB stores matrices in memory such that the order of entries in columns is preserved. Thus `A(2) = 2`, `A(5) = 7`, etc. You can drop the geometry and get all the entries of a matrix as a single column vector with the syntax `A(:)`. Sometimes that knowledge or that operation comes in handy.

The loop-based way to think about doing this would be to loop over all times, and for each time, loop over all sensor values, multiply each by its weight, and add up those products. Because it would be a pain to type in all the separate statements every time we wanted to do this, let's write a program. With your favorite text editor, create the following file, and name it `sensor.m`:

```
M = 5000; N = 500; % Let's choose M and N reasonably large.
A = rand(M,N); b = rand(N,1);
C = zeros(5000,1);
for i=1:5000
    for j=1:500
        C(i) = C(i) + A(i,j) * b(j);
    end
end
```

A few notes about this program:

1. This is a script, a collection of MATLAB commands put into a file with the `.m` extension; typing the filename is then equivalent to typing in all those commands separately at the command line. The other type of M-file is a function. Creating those is slightly more complicated; you can read about it by typing `help function` at the command line. The use of functions is also different from the use of scripts in various subtle ways that I'm not going to get into here, mostly having to do with the scope of variables in memory.
2. In the first line, you'll see that you can put more than one statement on the same line; semicolons separate statements and suppress echoing, commas separate statements and echo the result.
3. Everything after the `%` character on a line is treated as a comment and ignored by the interpreter.
4. The `rand(p,q)` command creates a  $p \times q$  matrix and fills it with random numbers between 0 and 1 drawn from a uniform distribution (every value is equally likely). `randn` has the same syntax, but draws its random numbers from a Gaussian distribution with mean 0 and standard deviation 1. Other commands in the same family include `zeros` and `ones`, which fill the matrices they create with zeros and ones, respectively. MATLAB's intuitive that way. Be warned: if you provide only one argument to any of these commands, it won't give you a vector of that length, as you might expect, but rather a square matrix.
5. `for` looping is done according to the syntax  
`for <index> = <expression>; <statements>; end.`  
`<expression>` is typically a vector; as the loop executes, `<index>` takes on each of the values in that vector in turn. So you can have a loop like `for i=[6 3 12 -1 7]`, if you want; if it's not clear what that does, try it out, having it print the value of `i` on each iteration.

All right, with all that said, we can now run the program. Type `sensor` at the command line. Pay attention to how long it takes to execute.

You may have noticed that this situation was constructed such that we can do exactly the same thing without looping, just by multiplying  $A$  and  $b$  together (compare the definition of matrix multiplication: if  $C = A * b$ , then  $C_{ij} = \sum_k A_{ik} b_{kj}$ . Try  $C=A*b$ , and compare its execution time to the looping case. This is why the problem set stresses so much the need to avoid loops; not only does it make your code much more elegant and readable, but MATLAB is full of optimizations to make these kinds of operations on matrices take place much more quickly.

### 1.3 Plotting and printing

One of the great things about MATLAB is the ease and control with which you can use it to make plots. The command `help plot` will tell you all about it, but I'll give you a few examples to give you an idea of what you can do:

```
t = 0:0.1:10; % say we take samples every .1 seconds from 0 to 10 seconds
y = sin(2*pi*t/5); % make y a sine wave with period 5 seconds
plot(y) % the simplest way to plot; notice that the horizontal axis
% is not 0 to 10 as we would like, but rather corresponds to
% the indices of y, by default
plot(t,y) % specify both the horizontal and vertical quantities to
% plot against one another
plot(t,y,'go--') % change the appearance of the plot with an optional
% third argument, a string that can specify color
% (here, green), point appearance (circles), and
% line style (dashed); look in 'help plot' for more
% options
x = cos(2*PI*x/5);
plot(x,y) % parametric plots are equally possible
plot(t,x,'g-.',t,y,'r**') % more than one plot can be put on the same
% axes, by adding more triplets of arguments
% to the plot command
plot(t,x,'k:') % note that making a new plot by default clears the
% old one
hold on % after this command, future plots will appear on the same
% axes without clearing them
plot(t,y,'m-.')
hold off % and now future plots will clear the axes again
plot(x,y)
semilogx(x+2,y+2) % make the x-axis logarithmic
semilogy(x+2,y+2) % or the y-axis
loglog(x+2,y+2) % or both axes
```

To print out your masterpiece, MATLAB helpfully provides a print command, which has a longer help file than any other command I can think of. Here are the options you're most likely to use:

```

print % send current figure to the default printer
print -Pname % send figure to printer called 'name'
print -dps file.ps % print to a PostScript file called 'file.ps'
print -dpsc file.ps % or color PostScript
print -depsc2 file.eps % or color encapsulated PostScript
print -djpeg file.jpeg % or JPEG

```

## 1.4 How to learn MATLAB

It's actually pretty difficult to go about explaining how to use MATLAB in a methodical way. The best way to learn, and the primary way I learned, is to use its excellent help system, and play around with commands.

To get help on the syntax for any command, type `help <command>`. A lot of commands have intuitive names; if you want to know how to find the mean of a vector, for instance, try `help mean`, and there it is. If you want a command that performs some function but you can't think of the name of the command, use `lookfor <keyword>`; for instance, try `lookfor variance`.

Here's a set of commands that you might find especially useful; I'll let you read about how to use them. As always, ask any questions you like.

```

while: the other kind of loop
if/elseif/else: the other use of conditional statements
size(A) : returns M and N for an M-by-N matrix A
repmat(a,b,c) : tiles matrix A, repeating it b times vertically and c
    times horizontally; try 'repmat(rand(2,2),2,3)'
max, min : note that 'm = max(x)' returns in m the maximum value in
    the vector x; '[m,i]' = max(x)' returns the same thing, plus the
    index of that value in i!
mean, sum
find : very useful; read the help file
who : returns the names of all the variables in memory
whos : like who, but returns more detailed information
clear : clears all variables from memory; 'clear x y z' clears just
    variables x, y, and z
why : answers any question (try it the next time you get an error!)

```

## 2 Variance and covariance

In class, Sebastian explained that variance is a measure of how much a function or time series fluctuates about its mean. It differs from second moment,  $\langle x^2 \rangle$ , in that the mean is subtracted; thus with  $x(t) = 10 + \sin(t)/10$  and  $y(t) = 2 * \sin(t)$ , the second moment of  $x$  will be much greater than that of  $y$ , but the variance of  $y$  will be much greater—which fits with our intuitive notion of what variance should be.

Sebastian gave the definition  $\text{Var}[x] \equiv \langle x^2 \rangle - \langle x \rangle^2$ . I find it easier to see how this corresponds to subtracting the mean by starting from a formulation where the mean is

subtracted more explicitly:

$$\begin{aligned}
 \text{Var}(x) &= \langle (x - \langle x \rangle)^2 \rangle \\
 &= \langle (x^2 - 2x\langle x \rangle + \langle x \rangle^2) \rangle \\
 &= \frac{1}{N} \sum_i x_i^2 - 2\frac{1}{N} \sum_i x_i \langle x \rangle + \frac{1}{N} \sum_i \langle x \rangle^2 \\
 &= \langle x^2 \rangle - 2\langle x \rangle \frac{1}{N} \sum_i x_i + \langle x \rangle^2 \frac{1}{N} \sum_i 1 \\
 &= \langle x^2 \rangle - 2\langle x \rangle^2 + \langle x \rangle^2 \\
 &= \langle x^2 \rangle - \langle x \rangle^2.
 \end{aligned}$$

Here we used the definition  $\langle x \rangle = \frac{1}{N} \sum_i x_i$ , and relied on the fact that  $\langle x \rangle$  is a fixed quantity which does not depend on the index  $i$  and thus can be taken outside the sum.

Covariance is a measure of the extent to which two variables fluctuate together; if both tend to increase at the same time and decrease at the same time, the covariance will be high. You can see that if  $x$  and  $y$  are each large and positive at the same time as one another, and large and negative at the same time as one another, the term  $\langle xy \rangle$  will be large and positive, reflecting the high covariance. However, this term is not enough on its own; for the same reasons as with variance, to avoid getting artificially high or low values for the result, we want to subtract the mean from each variable. Again starting from an expression where we do so explicitly,

$$\begin{aligned}
 \text{Cov}(x, y) &= \langle (x - \langle x \rangle)(y - \langle y \rangle) \rangle \\
 &= \langle xy \rangle - \langle x \rangle \langle y \rangle - \langle x \rangle \langle y \rangle + \langle x \rangle \langle y \rangle \\
 &= \langle xy \rangle - \frac{1}{N} \sum_i x_i \langle y \rangle - \frac{1}{N} \sum_i \langle x \rangle y_i + \frac{1}{N} \sum_i \langle x \rangle \langle y \rangle \\
 &= \langle xy \rangle - \langle y \rangle \frac{1}{N} \sum_i x_i - \langle x \rangle \frac{1}{N} \sum_i y_i + \langle x \rangle \langle y \rangle \frac{1}{N} \sum_i 1 \\
 &= \langle xy \rangle - \langle y \rangle \langle x \rangle - \langle x \rangle \langle y \rangle + \langle x \rangle \langle y \rangle \\
 &= \langle xy \rangle - \langle x \rangle \langle y \rangle
 \end{aligned}$$

which is the definition Sebastian gave in class.

### 3 Linear regression

You should be familiar with the idea of solving systems of linear equations. For instance, if you have the equations

$$\begin{aligned}
 3x + 2y &= 8 \quad \text{and} \\
 x - 4y &= -3,
 \end{aligned}$$

and you want to solve for  $x$  and  $y$ , you can write this in matrix form:

$$\begin{bmatrix} 3 & 2 \\ 1 & -4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ -3 \end{bmatrix}$$

If we define

$$A = \begin{bmatrix} 3 & 2 \\ 1 & -4 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, d = \begin{bmatrix} 8 \\ -3 \end{bmatrix},$$

the equation becomes  $A\mathbf{x} = d$ , which has solution  $\mathbf{x} = A^{-1}d$ . In MATLAB, you can find this solution with the command `x = inv(A)*d` or `x = A\d`.

Geometrically, if you think of  $x$  and  $y$  as coordinates, we're finding the location where these two lines meet. Or the same approach can describe a different situation: with the familiar equation for a line  $y = mx + b$ , the equations

$$\begin{aligned} 3 &= 7m + b & \text{and} \\ 1 &= 3m + b \end{aligned}$$

give the matrix equation

$$\begin{bmatrix} 7 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

The interpretation suggested here is that we're finding the line connecting the two points  $(7, 3)$  and  $(3, 1)$ .

If you know much about matrix algebra, you know that things aren't always this simple. This is a case with a unique solution, one and only one. Sometimes the matrix isn't invertible, e.g.,

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix}$$

This is the case where the two lines are identical, or the two points are the same; you don't have enough information to lock the solution down exactly. In this case the problem is called underconstrained. If your matrix  $A$  has fewer rows than columns, it'll necessarily be underconstrained. If you've got two equations and four unknowns, you're stuck; there's no unique solution, there's an infinity of solutions.

The opposite situation is when you're overconstrained. Here you also have no unique solution, but in this case it's not that you have infinitely many solutions, but rather that you have none; there are too many constraints and they can't all be satisfied at once. Your lines don't intersect in a single point, or (and this is clearly the case I'm working up to) your points don't all lie on a single line. In that case, the best you can do in trying to fit a line to those points is to minimize the total error, for some convenient measure of error. The one that's always used, and this is the one Sebastian talked about in class, is the total squared error, the sum of the squares of the vertical distances from each point to the line:

$$E = \frac{1}{2} \sum_i (y_i - (mx_i + b))^2$$

Minimizing this squared error gives rise to the familiar term 'least-squares'.

For the problem to be overconstrained,  $A$  must have more rows than columns.<sup>4</sup> Since  $A$  isn't square, you can't simply take its inverse. But what you can do is multiply both sides of the equation on the left by its transpose:

$$A^T A x = A^T d$$

---

<sup>4</sup>More rows than columns doesn't necessarily mean the problem is overconstrained; it's a necessary but not sufficient condition. The data set could have a unique solution or be underconstrained.

This is called the normal equation. And this is the equation that, for a given  $A$  and  $d$ , minimizes that squared error. In that sense, it makes the left and right sides of the equation as close as they can get.

Now look at this matrix  $A^T A$ . If  $A$  is  $M \times N$ , then  $A^T A$  is  $(N \times M)(M \times N) = N \times N$ ; so it's square, which means we can at least try to take an inverse, and in almost all real-world cases an inverse exists. Then the least-squares solution is

$$x = (A^T A)^{-1} A^T d.$$

This  $(A^T A)^{-1} A^T$  is called the pseudoinverse.

You can find this solution in MATLAB by typing in the expression  $x = \text{inv}(A' * A) * A' * d$ , by using `pinv`, or by writing `A\d`. I think all of these are actually calculated differently, but in real-world cases of real-valued data they should all give the same result.