Matlab Tutorial I

CS 310 — Fall 1999

This handout describes how to work with matrices and vectors in Matlab. Matlab is a very widely used numerical mathematics package.

To start Matlab, go to
`Start -> CAE Applications -> Engineering -> M-S -> Matlab 5.3`

1 Matrices and Vectors

Everything in Matlab is stored using a matrix. (In fact, the name "Matlab" comes from *Mat*rix *lab*oratory.) If a matrix has only one row or only one column, it is often called a vector.

Try the following commands. Be sure to read the accompanying explanation.

Matlab Command	Explanation
`x = [1, 2, 3]`	This sets `x` to be a 1×3 vector, $x = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$.
`y = [1 2 3]`	This sets `y` to the same vector we set to `x`. This shows that the commas are optional.
`z = [1, 2; 3, 4]`	Commas (or blank spaces) are used to separate elements in the same row. Semicolons are used to separate the rows. So, this command sets `z` to be the 2×2 matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.
`a = [1 2 3; 4 5 6]`	This results in $a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$.
`a = [y; 4 5 6]`	This results in $a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ also; the first row of `a` was set to `y`, which we previously set to $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$.

Another way to create the above matrices is to use Matlab's "`:`" operator. The expression `1:5` represents $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$. When you use something like `1:2:9`, Matlab creates a vector starting at 1, incrementing by 2, and ending at 9. So, `1:2:9` represents the vector $\begin{bmatrix} 1 & 3 & 5 & 7 & 9 \end{bmatrix}$. On the next page are some examples using `:`.

Matlab Command	Explanation
`x = 1:3`	This sets `x` to be the same vector as above, $x = \begin{bmatrix}1 & 2 & 3\end{bmatrix}$.
`y = 1:8`	`y` is now $\begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6 & 7 & 8\end{bmatrix}$.
`a = [1:3; 4:6]`	This sets $a = \begin{bmatrix}1 & 2 & 3 \\ 4 & 5 & 6\end{bmatrix}$, the same as above.
`b = 1:3:10`	This sets `b` to be the vector $\begin{bmatrix}1 & 4 & 7 & 10\end{bmatrix}$.
`c = 10:-1:7`	This sets $c = \begin{bmatrix}10 & 9 & 8 & 7\end{bmatrix}$.
`B = [3:5; 3:-1:1; -1:1 ]`	Each row of `B` has 3 elements. Matlab will give you an error if all the rows are not of equal length.
`A = [b; c]`	This creates a matrix with `b` as its first row, and `c` as its second. Both have length 4, so the matrix `A` will be 2×4.

The above examples also show that Matlab is "case-sensitive" — meaning that `A` and `a` are different matrices.

To print the value of a matrix, you can just type its name. To print the value of a certain element of a matrix, you use parentheses, like this: `A(1,4)`. This represents the element of `A` in the first row, fourth column. If the matrix happens to be a vector, you only need to specify one number: `x(2)`. The Matlab command `who` will print all of your currently existing variables.

Matlab Command	Explanation
`x = 8:-2:2`	
`y = 0:3:9`	
`A = [x; y; -x]`	This creates a 3×4 matrix whose first row is `x`, second row is `y` and third row is `-x`.
`who`	List the currently existing varibles.
`x`	Display the vector `x`.
`A(1,3)`	Display the element in `A` in row 1, column 3.
`y(3)`	Display the third element of the vector `y`.
`y(3) = 999`	Change the third element of `y` to 999.

☞ What if, instead of typing `A = [x; y; -x]`, you had typed:

(i) `A = [x; y -x]` ?

(ii) `A = [x; y-x]` ?

Try each of them. Do you understand the difference between these three commands?

To wipe out all of your variables, use the command `clear`. (This is like `restart` in Maple.)

2 Matrix Operations

You can add, subtract, and multiply matrices and vectors together, provided the sizes match up correctly. Remember that to add or subtract two matrices, they must be exactly the same size. To multiply two matrices, the number of columns in the first matrix must be the same as the number of rows in the second.

Matlab Command	Explanation
`A = [1 2; 3 4]`	
`B = [3 -1; 8 0]`	
`C = A + B`	`C` is the matrix sum of `A` and `B`.
`A-B`	Compute the difference of `A` and `B`. The result is assigned to the variable `ans`.
`P = A*B`	Compute the product `A*B`, and assign the result to `P`.
`P + [1 4; -9 2]`	The result of this computation is assigned to `ans`, since we didn't specify a variable to assign the result to.
`ans^2`	This computes the square of the previous result.
`iA = A^(-1)`	Set `iA` to be the inverse of A.
`iA*A`	Now `ans` should be the 2×2 identity matrix.
`P*P^(-1)`	This should also result in the 2×2 identity matrix.

☞ At this point, you should be able to solve the system

$$\begin{array}{rcrcrcr} 3x_1 & + & 4x_2 & - & x_3 & = & -1 \\ x_1 & + & x_2 & + & x_3 & = & 7 \\ -x_1 & & & + & x_3 & = & 1 \end{array}$$

using Matlab. Use the same technique you learned in Maple Tutorial III. (The correct answers are $x_1 = 4, x_2 = -2, x_3 = 5$.) Remember that the right hand side needs to be entered as a column vector — a matrix with only one column.

3 Using a Diary

You can save a record of your Matlab session by using the `diary` *filename* command. This creates a file called *filename* in which a copy of all your subsequent Matlab commands are kept. The command `diary off` turns off the diary, and `diary on` turns in back on again.

Here is an example:

Matlab Command	Explanation
`cd f:`	Change directory to your network drive
`diary` *filename*	Start recording the Matlab session in the file *filename* .
`a = [1 -2; 3 -7]`	Now we'll enter some commands to put in the diary.
`a4 = a^4`	Calculate `a`×`a`×`a`×`a`.
`a*a4`	This is `a` to the fifth power.
`x = ans^(-1)`	Computes the inverse of the previous result.
`diary off`	Turns the diary off.

After doing this, you will have a file in the `matlab` folder in your `F:\` directory called *filename*, which will contain the commands you typed and what Matlab responded with. You can print this file by opening it with Notepad and printing it from there.

Note: *filename* is an arbitrary file name chosen by you. For example, `diary homework4` would record all of your commands in the file "homework4" until you type `diary off`.

4 More Matrix Operations

To take the transpose of a matrix, you put a `'` after the variable name. The transpose "switches" the rows and the columns. The command `ones(`m`,`n`)` creates a matrix containing all ones with size $m \times n$. Similarly, the command `zeros(`m`,`n`)` creates a matrix containing all zeros with size $m \times n$. Here are some examples of these commands.

Matlab Command	Explanation
`clear`	Wipe out all previous variables.
`x = 1:4`	Create a 1×4 matrix
`y = x'`	`y` is the transpose of `x`.
`x*y`	Since `x` is 1×4, and `y` is 4×1, the result will be a 1×1 matrix, which is just a number. You should get 30.
`y*x`	This results in a 4×4 matrix.

Matlab Command	Explanation
`Z = ones(3,4)`	Create a 3×4 matrix of all 1's.
`W = [ 2:5 ; Z ]`	Look at this example carefully! How is `W` being created?
`K = [ones(2,4); zeros(2,4)]`	Top half of `K` is 1's, bottom half is 0's.
`W * K'`	Multiply `W` by the transpose of `K`.
`W * [ 4 3 2 1; ones(3,4) ]'`	Multiply `W` by the transpose of the 4×4 matrix that is all ones, except the first row, which is `4:-1:1`.
`ans/2`	The previous result is divided by 2. It is impossible to divide a matrix by a matrix, but you can divide a matrix by a number (except 0, of course). Each element of the matrix is divided by the number.
`ans*2`	Now, all of the elements are multiplied by 2. We could have also used `2*ans`. (But, instead of `ans/2` you could *not* have used `2/ans`!)

☞ Before entering the following commands, see if you can determine what the value of `A` will be.

```
z = -1:1
A = 2*ones(3,2) + [ z', ones(3,1) ]
```

☞ Create the following 10×10 matrix in Matlab. You are not allowed to just type the whole thing in; you should be more clever.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 0 & 0 & 0 & 0 & 1 & 2 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 3 & 2 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 4 & 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

If you don't want to see the result of a Matlab calculation, you can suppress the printing to the screen by ending the command with a semicolon. For example, if you type `A = ones(100,100)` then Matlab will display a 100×100 matrix of ones. If you type `A =ones(100,100);` then Matlab will still create the 100×100 matrix but won't display it on the screen. In Maple, the semicolon was required to end a command; in Matlab it is not required. It means the the output of the command should not be displayed.

5 Gaussian elimination

You have learned how to solve linear systems of equations using the matrix inverse. A better way is to use Gaussian elimination, since computing the inverse of a matrix can be very time consuming. As easy as it was to find the solution with Matlab by computing the inverse of a matrix, using Gaussian elimination is even easier. Here is an example. Suppose we have the following linear system:

$$\begin{array}{rcrcrcr} x & + & y & + & z & = & 3 \\ x & - & y & + & 2z & = & -3 \\ -2x & + & y & - & 3z & = & 4 \end{array}$$

In Matlab, we would enter this sytem like this:

Matlab Command	Explanation
`clear`	Wipe out all previous variables.
`A = [1 1 1; 1 -1 2; -2 1 -3]`	
`b = [3 -3 4]'`	Be sure you notice the use of the transpose operation here. We're converting a row vector to a column vector.

To solve it by taking the matrix inverse, we would use the command `A^(-1)*b`. In the resulting vector, the first entry would be the value of x, the second the value of y, and the third the value of z. To solve it using Gaussian elimination, the proper command is `A\b`. Note that this is *not* `A/b`! You have to use the backslash, "\". For example,

(... previous example continued)

Matlab Command	Explanation
`x1 = A\b`	`x1` is the solution using Gaussian elimination.
`x2 = A^(-1)*b`	`x2` is the solution using the matrix inverse.

You should get the same values for `x1` and `x2`. You can check to see if your answer is correct by computing `b-A*x1` and `b-A*x2`. Typing these commands amounts to substituting the answers into the equations and seeing if they work. Both should be very close to zero.

☞ Use Gaussian elimination to solve the following matrix equation.

$$\begin{bmatrix} 1 & 2 & 3 & 5 \\ -1 & 2 & 5 & 8 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 10 \\ 3 \\ 2 \\ 4 \end{bmatrix}$$

It might help to notice that the first matrix is

```
[ 1:3 5 ; -1:3:8 ; 1 0 0 1 ; ones(1,4) ]
```

The answers are $x_1 = -\frac{5}{3}, x_2 = \frac{38}{3}, x_3 = -\frac{32}{3}, x_4 = \frac{11}{3}$. Could you have done it by hand?

Which is faster?

You have learned two methods for determining the solution to a linear system: Gaussian elimination, and using the matrix inverse. Which is better? The answer is Gaussian elimination, and this section will illustrate why: Gaussian elimination uses fewer floating point operations (flops).

To show this, we'll use Matlab's command `flops` which simply counts the number of floating point operations performed thus far. We'll compare the number of flops required to solve a big linear system using each of the two methods.

Which linear system should we solve? All we want is the flops required to solve it; we don't care what the actual linear equations are. So, we'll just create a random linear system, with 100 variables and 100 equations. To do this, we'll use Matlab's `rand` command.

Matlab Command	Explanation
`clear`	Wipe out all previous variables.
`A = rand(100,100);`	Create a 100×100 random matrix. This is the coefficient matrix for our linear system.
`b = rand(100,1);`	Create a vector for the right hand sides of the 100 equations.
`flops(0);`	Reset the flops counter to zero.
`A\b; flops`	Solve the problem with Gaussian elimation, and print the flops required. Note that we aren't printing the solution to the system, since we don't care about that.
`flops(0);`	Reset the flops counter to zero.
`inv(A)*b; flops`	Solve the problem by taking the inverse of `A` and multiplying by `b`. Don't print the results, only the number of flops.

You should see that Gaussian elimination requires about a third as many flops, which in general means that it is three times faster.