Matlab Tutorial

CS2240 Interactive Computer Graphics

Parts of this tutorial were adapted from Professor John Hughes' Matlab Tutorial

Introduction

Matlab is a proprietary numerical computation platform very widely used in computer science, and all engineering disciplines. The name is short for "Matrix Laboratory" and, as you may expect, the platform is particularly good at matrix computations. Matrix equations show up time and time again in computer graphics, whether in the form of complicated fluid dynamic simulations, or simple image processing algorithms. This handout is a crash introduction to Matlab. It provides basic knowledge that you will need in order to complete future assignments for this class.

Getting Started

To start Matlab, log into a lab machine [1] and type

```
$ matlab &
```

If you dislike the fancy (slow) GUI you can run Matlab in your shell by doing the following:

```
$ matlab -nojvm
```

The directory you are in when you run Matlab is important because Matlab will automatically load any .m files that are in your current directory. These are Matlab script files that may contain functions, or a sequence of commands. Still, if you started from the wrong directory, no problem: Matlab will execute Unix commands (like cd or ls).

Matlab is an interpreted language like Python so you can type commands and view the results immediately. Try running the following:

```
3 + (3 - 1) ^ 2
```

Matlab does the calculation and displays the result on the command line. To suppress the printing of results — something that you would want to do when you are executing multiple commands in a function or script — include a semicolon at the end of the command:

```
%nothing printed, result stored in variable called 'ans'
3 + (3 - 1) ^ 2;
```

To quit Matlab, type `quit` into the Matlab command line.

Help!

Always remember your best source for Matlab help is not online tutorials, not the Matlab manuals, but the two following modest scripts at the command line:

```
help <function_name>
```

and

```
lookfor <keyword>
```

Try typing in `help sum` and `lookfor logarithm`. The `lookfor` command may take some time to finish.

[1] If you prefer to run Matlab on a local machine, an academic license is available for free to students at Brown. Visit software.brown.edu

Matrix Operations

This is how you define a matrix:

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

This defines a 2x3 matrix. A semicolon separates rows; space (or a comma) separates elements on the same row. Square brackets go around the matrix. If you read the Eigen Tutorial, you may remember that we discussed how, in Eigen, matrices are defined in *row major* order, but stored in *column major* order. The same is true for Matlab. This fact will be important when we discuss matrix indexing.

Meanwhile, here comes another matrix

```
B = [ 2 3; 3 4]
```

We now have a 2x2, and 2x3 matrix. The inner dimensions match allowing us to multiply them. Let us do so:

```
C = B * A
```

If, ever, the inner dimensions do not match, Matlab will complain.

Heres a good time to see what other operators are available:

```
% addition/subtraction - dimensions must match
D = C + A
M = D - [1 0 0; 0 1 0]

% matrix power - equals B * B
U = B ^ 2

% matrix transposition
Ut = U'

% Solving a system of linear equations: Bu = x
x = [1; 2];
u = B\x

% the above is equivalent to
u = inv(B) * x
```

Notice that * and ^ performs matrix multiplication, and exponentiation respectively. But what if we want to perform these operations element-wise? Matlab has special syntax for that:

```
% a dot before the operator makes it work element-by-element
U = B .^ 3
E = U .* 3

% Element-wise division
U = U ./ B

% Calculate the Euclid-... uh.....um...the Frobe-....er... the
% mmmbmmlbbnl norm of a matrix
sum(sum(U .^ 2))
```

The dot isn't needed for logical operators - they can only be performed element-wise:

```
% The matrix L is a logical matrix of the same size as B and U
L = B == U

% The above is equivalent to - but much more efficient than - the
% following
L = [ B(1, 1) == U(1, 1), B(1, 2) == U(1, 2), B(1, 3) == U(1, 3);
      B(2, 1) == U(2, 1), B(2, 2) == U(2, 2), B(2, 3) == U(2, 3);
      B(3, 1) == U(3, 1), B(3, 2) == U(3, 2), B(3, 3) == U(3, 3)]

% Matlab uses a ~ (tilde) for negation rather than a ! as in C++
L2 = B ~= U
L3 = B <= U
```

The last example illustrated how basic matrix indexing works in Matlab: the row and column number are specified inside parentheses (not square brackets). Again, it can not have escaped the razor-sharp focus, the keen attention to detail, and the elephant-like memory of our astute students that this syntax is similar to Eigen's indexing syntax. Matlab, however, allows a number of other ways to access matrix elements. We will discuss these in more detail presently, but the following examples demonstrate the most commonly used ones:

```
% Select a whole row, or a whole column:
row_1 = A(1,:)
col_2 = A(:,2)

% Select columns 2 through 4:
cols_234 = B(:,2:3)

% Select rows 1, and 3
rows_13 = U([1 3], :)
```

Finally, here is a list of operations that come quite in handy

```
% concatenate (horizontally) C with B (try "help horzcat"):
CB = [C B]
```

```
% concatenate (vertically) A with C (try "help vertcat"):
AC = [A; C]

% Find the dimension of a matrix:
mysize = size(AC)

% make a 3x4 matrix of zeros:
my_zero = zeros(3, 4)

% a matrix of ones, and the identity matrix
my_one = ones(3, 4)
my_id = eye(3)

% Pretty useful: reshaping a matrix into a row vector
my_row_vec = reshape(AC, 1, mysize(1)*mysize(2))

% reshaping it into a column vector
my_col_vec = reshape(AC, mysize(1)*mysize(2), 1)
```

Vectors

> *"What wouldst ye vectore then be, but a matrix with a dimension that were unity."*
>
> —Numair Khan, Musings on the Secrets of the Universe, Vol. XXVII

All operations that work with matrices also work with vectors. In addition, Matlab provides the following convenient syntax for defining vectors:

```
% a vector with elements from 1 to 9
v1 = 1:9;   % v1 = [1 2 3 4 5 6 7 8 9]

% a vector with elements going from 1 to 9, in increments of 2
v2 = 1:2:9; % v2 = [1 3 5 7 9]
```

Indexing

Matrices in Matlab can be indexed using scalars, vectors, or other matrices. Except the special case of logical indices, Matlab expects all indices to be positive numbers: **Matlab indexing starts at one, not zero**.

Scalar Indices

Students are most likely already familiar with the use of scalar indices:

```
A = rand(12, 12) * 100;

% Access elements by specifying (row number, column number)
A(5, 4) = A(2, 2) + 3;
```

Remember when we briefly talked about the order in which Matlab stored matrices in memory — column-major versus row-major? Knowing that is helpful when accessing matrices using a linear index:

```
% Which is true and which false?
A(17) == A(5, 2);
A(17) == A(2, 5);
```

Vector Indices

Using vectors to index a matrix allows the selection of multiple rows/columns in one go. The selected indices are returned as a vector.

```
v = [1 3 5 9 11];

% Then

u = A(v, 3);

% is equivalent to

w = [ A(1, 3); A(3, 3); A(5, 3); A(9, 3); A(11, 3) ];

% which is, in turn, equivalent to

x = A(1:2:12, 3);

% and also equivalent to

y = A(1:2:end, 3);  % ("end" is a Matlab keyword)

% which equals

z = A(1:2:size(A, 1), 3);
```

A colon `:` in place of the row or column number tells Matlab to select all indices for that dimension:

```
A(:, 1) == A(1:end, 1);
A(2, :) == A(2, [1 2 3 4 5 6 7 8 9 10 11 12]);
A(4, :) == A(4, 1:12);
```

What happens when both the column and row are specified as vectors?

```
v = [1 2 5 6];
w = [3 4 5];
R = A(v, w);

% Which says, for each row index specified in v, select all the
% column indices specified in w.
% The above is equivalent to:
R = [ A( v(1), w(1) ) A( v(1), w(2) ) A( v(1), w(3));
```

```
      A( v(2), w(1) ) A( v(2), w(2) ) A( v(2), w(3));
      A( v(3), w(1) ) A( v(3), w(2) ) A( v(3), w(3));
      A( v(4), w(1) ) A( v(4), w(2) ) A( v(4), w(3)); ];
```

A single vector of linear indices can also be used, as long as you remember that matrices are stored in column-major order in Matlab:

```
v = [1:5:22];

M = A(v)

% is the same as ...
M = A([1 5 10 15 20]);

% and the same as ...
M = [ A(1, 1), A(5, 1), A(10, 1), A(3, 2), A(8, 2) ];
```

Matrix Indices

Matlab allows other matrices to be used for indexing. The entries of the indexing matrix act as linear indices into the original matrix. The result is a matrix of the same size as the indexing matrix.

```
A = rand(15, 15) * 100;
B = [5 12 33;
     2 20 17;
     9 13 44];

C = A(B);

% equivelent to...
C = [ A( B(1, 1) ) A( B(1, 2) ) A( B(1, 3) );
      A( B(2, 1) ) A( B(2, 2) ) A( B(2, 3) );
      A( B(3, 1) ) A( B(3, 2) ) A( B(3, 3) ) ];

% C is the same size as B
```

Matlab's behavior would have been slightly different if the indexing matrix (`B` in the example above) had been made up of boolean or, as Matlab calls them, *logical* values. Try running the following:

```
A = rand(15, 15) * 100;
B = A <= 50
C = A(B)
```

Recall that logical operators work element-wise so that `B` is a matrix of the same size as `A` with entry `B(i, j) = 1` if `A(i, j) <= 50`, and `B(i, j) = 0` otherwise. Now when `B` is used for indexing, the Boolean value of its elements act as a mask for deciding whether the corresponding element in `A` will be included in the result or not. The result is a *vector* of values that are selected by the mask.

Code Vectorization

Logical indexing provides one way of making Matlab programs run significantly faster by allowing the code to be *vectorized*. The idea behind vectorization is to structure our code as a series of vector, or matrix operations. Matlab performs vector and matrix operations extremely efficiently. Vectorized code also utilizes the parallel hardware resources of the processor better. The following code snippet provides an example of the speed benefits of vectorization:

```
% 5000x5000 matrix of random integers between 1 and 100
A = randi(100, 5000, 5000);

% Calculate the mean of all elements in A that are greater than
% 50. The code prints the value calculated and the running time
% for non-vectorized and vectorized code variants

%
% Non-vectorized code using for loops
tic;
R = [];
k = 1;
for i = 1:5000
  for j = 1:5000
    if A(j, i) > 50
      R( k ) = A(j, i);
      k = k + 1;
    end
  end
end

mu1 = sum(R) / length(R)
toc

%
% Vectorized code cast as matrix operations
tic;
mu2 = sum( A( A > 50) ) ./ sum(sum(A > 50))
toc
```

Further Reading

The following provide a more thorough overview of the topics we discussed:

- Matlab Documentation
- Matlab Vectorization Guide
- Matlab Primer

In most cases, you can Bing the answers to any questions you have about Matlab.

StackOverflow is also a good resource.