Introduction to Vectors in Matlab

This is the basic introduction to Matlab. Creation of vectors is included with a few basic operations. Matlab is a software package that makes it easier for you to enter matrices and vectors, and manipulate them. Almost all of Matlabs basic commands revolve around the use of vectors. To simplify the creation of vectors, you can define a vector by specifying the first entry, an increment, and the last entry. Matlab will automatically figure out how many entries you need and their values. For example, to create a vector whose entries are 0, 2, 4, 6, and 8, you can type in the following line:

```
>> 0:2:8

ans =

   0    2    4    6    8
```

Matlab also keeps track of the last result. In the previous example, a variable "*ans*" is created. To look at the transpose of the previous result, enter the following:

```
>> ans'

ans =

   0
   2
   4
   6
   8
```

To be able to keep track of the vectors you create, you can give them names. For example, a row vector v can be created:

```
>> v = [0:2:8]

v =

   0    2    4    6    8

>> v

v =

   0    2    4    6    8

>> v'

ans =

   0
   2
   4
   6
   8
```

Note that in the previous example, if you end the line with a semi-colon, the result is not displayed. This will come in handy later when you want to use Matlab to work with very large systems of equations. Matlab will allow you to look at specific parts of the vector. If you want to only look at the first three entries in a vector you can use the same notation you used to create the vector:

```
>> v(1:3)
ans =

    0     2     4

>> v(1:2:4)

ans =

    0     4

>> v(1:2:4)'

ans =

    0
    4
```

Once you master the notation you are free to perform other operations:

```
>> v(1:3)-v(2:4)

ans =

   -2    -2    -2
```

Introduction to Matrices in Matlab

A basic introduction to defining and manipulating matrices is given here. It is assumed that you know the basics on how to define and manipulate vectors using matlab. Defining a matrix is similar to defining a vector. To define a matrix, you can treat it like a column of row vectors (note that the spaces are required!):

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

A =

   1   2   3
   3   4   5
   6   7   8
```

You can also treat it like a row of column vectors:

```
>> B = [ [1 2 3]' [2 4 7]' [3 5 8]']

B =

   1   2   3
   2   4   5
   3   7   8
```

If you have been putting in variables through this and the tutorial on vectors, then you probably have a lot of variables defined. If you lose track of what variables you have defined, the *whos* command will let you know all of the variables you have in your work space.

```
>> whos
        Name      Size            Elements    Bytes  Density  Complex

        A         3 by 3             9         72     Full      No
        B         3 by 3             9         72     Full      No
        ans       1 by 3             3         24     Full      No
        v         1 by 5             5         40     Full      No

Grand total is 26 elements using 208 bytes
```

As mentioned before, the notation used by Matlab is the standard linear algebra notation you should have seen before. Matrix-vector multiplication can be easily done. You have to be careful, though, your matrices and vectors have to have the right size!

```
>> v = [0:2:8]

v =

   0   2   4   6   8

>> A*v(1:3)
??? Error using ==> *
Inner matrix dimensions must agree.

>> A*v(1:3)'

ans =

   16
   28
   46
```

Get used to seeing that particular error message! Once you start throwing matrices and vectors around, it is easy to forget the sizes of the things you have created. You can work with different parts of a matrix, just as you can with vectors. Again, you have to be careful to make sure that the operation is legal.

```
>> A(1:2,3:4)
??? Index exceeds matrix dimensions.

>> A(1:2,2:3)

ans =

    2    3
    4    5

>> A(1:2,2:3)'

ans =

    2    4
    3    5
```

Once you are able to create and manipulate a matrix, you can perform many standard operations on it. For example, you can find the inverse of a matrix. You must be careful, however, since the operations are numerical manipulations done on digital computers. In the example, the matrix A is not a full matrix, but matlab will still return a matrix.

```
>> inv(A)

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
4.565062e-18

ans =

  1.0e+15 *

  -2.7022    4.5036   -1.8014
   5.4043   -9.0072    3.6029
  -2.7022    4.5036   -1.8014
```

Other operations include finding an approximation to the eigen values of a matrix. There are two versions of this routine, one just finds the eigen values, the other finds both the eigen values and the eigen vectors. If you forget which one is which, you can get more information by typing *help eig* at the matlab prompt.

```
>> eig(A)

ans =

  14.0664
  -1.0664
   0.0000

>> [v,e] = eig(A)

v =
  -0.2656    0.7444   -0.4082
  -0.4912    0.1907    0.8165
  -0.8295   -0.6399   -0.4082
```

```
e =

   14.0664         0         0
         0   -1.0664         0
         0         0    0.0000

>> diag(e)

ans =

   14.0664
   -1.0664
    0.0000
```

There are also routines that let you find solutions to equations. For example, if Ax=b and you want to find x, a slow way to find x is to simply invert A and perform a left multiply on both sides (more on that later). It turns out that there are more efficient and more stable methods to do this (L/U decomposition with pivoting, for example). Matlab has special commands that will do this for you. Before finding the approximations to linear systems, it is important to remember that if A and B are both matrices, then AB is not necessarily equal to BA. To distinguish the difference between solving systems that have a right or left multiply, Matlab uses two different operators, "/" and "\". Examples of their use are given below. It is left as an exercise for you to figure out which one is doing what.

```
>> v = [1 3 5]'

v =

     1
     3
     5

>> x = A\v

Warning: Matrix is close to singular or badly scaled.  Results may be inaccurate. RCOND =
4.565062e-18

x =

   1.0e+15 *

    1.8014
   -3.6029
    1.8014

>> x = B\v

x =

     2
     1
    -1

>> B*x

ans =

     1
     3
     5
```

```
>> x1 = v'/B

x1 =

   4.0000   -3.0000    1.0000

>> x1*B

ans =

   1.0000    3.0000    5.0000
```

Finally, sometimes you would like to clear all of your data and start over. You do this with the "clear" command. Be careful though, it does not ask you for a second opinion and its results are **final**.

```
>> clear

>> whos
```

Vector Functions

Matlab makes it easy to create vectors and matrices. The real power of Matlab is the ease in which you can manipulate your vectors and matrices. Here we assume that you know the basics of defining and manipulating vectors and matrices. In particular we assume that you know how to create vectors and matrices and know how to index into them. In this tutorial we will first demonstrate simple manipulations such as addition, subtraction, and multiplication. Following this basic "element-wise" operations are discussed. Once these operations are shown, they are put together to demonstrate how relatively complex operations can be defined with little effort.

First, we will look at simple addition and subtraction of vectors. The notation is the same as found in most linear algebra texts. We will define two vectors and add and subtract them:

```
>> v = [1 2 3]'

v =
    1
    2
    3

>> b = [2 4 6]'

b =

    2
    4
    6

>> v+b

ans =

    3
    6
    9

>> v-b

ans =

   -1
   -2
   -3
```

Multiplication of vectors and matrices must follow strict rules. Actually, so must addition. In the example above, the vectors are both column vectors with three entries. You cannot add a row vector to a column vector. Multiplication, though, can be a bit trickier. The number of columns of the thing on the left must be equal to the number of rows of the thing on the right of the multiplication symbol:

```
>> v*b
Error using ==> *
Inner matrix dimensions must agree.

>> v*b'
ans =

    2    4    6
    4    8   12
    6   12   18
```

```
>> v'*b

ans =

   28
```

There are many times where we want to do an operation to every entry in a vector or matrix. Matlab will allow you to do this with "element-wise" operations. For example, suppose you want to multiply each entry in vector *v* with its corresponding entry in vector *b*. In other words, suppose you want to find *v(1)*b(1), v(2)*b(2), and v(3)*b(3)*. It would be nice to use the "*" symbol since you are doing some sort of multiplication, but since it already has a definition, we have to come up with something else. The programmers who came up with Matlab decided to use the symbols ".*" to do this. In fact, you can put a period in front of any math symbol to tell Matlab that you want the operation to take place on each entry of the vector.

```
>> v.*b

ans =

    2
    8
   18

>> v./b

ans =

   0.5000
   0.5000
   0.5000
```

Since we have opened the door to non-linear operations, why not go all the way? If you pass a vector to a predefined math function, it will return a vector of the same size, and each entry is found by performing the specified operation on the corresponding entry of the original vector:

```
>> sin(v)

ans =

   0.8415
   0.9093
   0.1411

>> log(v)

ans =

        0
   0.6931
   1.0986
```

The ability to work with these vector functions is one of the advantages of Matlab. Now complex operations can be defined that can be done quickly and easily. In the following example a very large vector is defined and can be easily manipulated. (Notice that the second command has a ";" at the end of the line. This tells Matlab that it should not print out the result.)

```
>> x = [0:0.1:100]

x =

  Columns 1 through 7

         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000

    (stuff deleted)

  Columns 995 through 1001

   99.4000   99.5000   99.6000   99.7000   99.8000   99.9000  100.0000

>> y = sin(x).*x./(1+cos(x));
```

Through this simple manipulation of vectors, Matlab will also let you graph the results.

```
>> plot(x,y)
>> plot(x,y,'rx')
```

Loops

In this tutorial we will demonstrate how the "for" and the "while" loop are used. First, the "for" loop is discussed with examples for row operations on matrices and for Euler's Method to approximate an ODE. Following the "for" loop, a demonstration of the "while" loop is given. In this tutorial we will assume that you know how to create vectors and matrices and know how to index into them.

The "*for*" loop allows us to repeat certain commands. If you want to repeat some action in a predetermined way, you can use the "for" loop. All of the loop structures in matlab are started with a keyword such as "for", or "while" and they all end with the word "end". Another deep thought, eh. The "for" loop will loop around some statement, and you must tell Matlab where to start and where to end. Basically, you give a vector in the "for" statement, and Matlab will loop through for each value in the vector:

For example, a simple loop will go around four times:

```
>> for j=1:4,
  j
end

j =

    1


j =

    2


j =

    3


j =

    4
```

Once Matlab reads the "end" statement, it will loop through and print out j each time. For another example, if we define a vector and later want to change the entries, we can step though and change each individual entry:

```
>> v = [1:3:10]

v =

    1    4    7   10

>> for j=1:4,
    v(j) = j;
    end

>> v

v =

    1    2    3    4
```

Note, that this is a simple example and is a nice demonstration to show you how a "for" loop works. However, **DO NOT DO THIS IN PRACTICE!!!!** Matlab is an interpreted language and looping through a vector like this is the slowest possible way to change a vector. The notation used in the first statement is much faster than the loop.

A better example, is one in which we want to perform operations on the rows of a matrix. If you want to start at the second row of a matrix and subtract the previous row of the matrix and then repeat this operation on the following rows, a "for" loop can do this in short order:

```
>> A = [ [1 2 3]' [3 2 1]' [2 1 3]']

A =

    1    3    2
    2    2    1
    3    1    3

>> B = A;

>> for j=2:3,
       A(j,:) = A(j,:) - A(j-1,:)
   end

A =

    1    3    2
    1   -1   -1
    3    1    3

A =

    1    3    2
    1   -1   -1
    2    2    4
```

For a more realistic example, since we can now use loops and perform row operations on a matrix, Gaussian Elimination can be performed using only two loops and one statement:

```
>> for j=2:3,
       for i=j:3,
           B(i,:) = B(i,:) - B(j-1,:)*B(i,j-1)/B(j-1,j-1)
       end
   end

B =
    1    3    2
    0   -4   -3
    3    1    3

B =
    1    3    2
    0   -4   -3
    0   -8   -3

B =
    1    3    2
    0   -4   -3
    0    0    3
```

Another example where loops come in handy is the approximation of differential equations. The following example approximates the D.E. y'=x^2-y^2, y(0)=1, using Euler's Method. First, the step size, h, is defined. Once done, the grid points are found, and an approximation is found. The approximation is simply a vector, y, in which the entry y(j) is the approximation at x(j).

```
>> h = 0.1;
>> x = [0:h:2];
>> y = 0*x;
>> y(1) = 1;
>> size(x)

ans =

    1    21

>> for i=2:21,
       y(i) = y(i-1) + h*(x(i-1)^2 - y(i-1)^2);
   end

>> plot(x,y)
>> plot(x,y,'go')
>> plot(x,y,'go',x,y)
```

If you don't like the "for" loop, you can also use a "while" loop. The "while" loop repeats a sequence of commands as long as some condition is met. In this example the D.E. y'=x-|y|, y(0)=1, is approximated using Euler's Method:

```
>> h = 0.001;
>> x = [0:h:2];
>> y = 0*x;
>> y(1) = 1;
>> i = 1;
>> size(x)

ans =
        1        2001

>> max(size(x))

ans =
       2001

>> while(i<max(size(x)))
            y(i+1) = y(i) + h*(x(i)-abs(y(i)));
            i = i + 1;
   end

>> plot(x,y,'go')
>> plot(x,y)
```