ENME202 Matlab

LINEAR ALGEBRA

TOPICS:

- vector and matrix formation & manipulation
- matrix slicing
- row & column expansion
- vector and matrix algebra

FUNCTIONS:

- dot() -- dot product
- cross() -- cross product
- norm() -- vector norm (magnitude)
- size() -- find matrix dimensions
- eye() -- identity matrix
- diag() -- create a diagonal matrix / return diagonal
- inv() -- matrix inverse
- det() -- determinant of a square matrix

ARRAYS AS VECTORS

So far we have used arrays for two things:

1. holding a list of numbers
2. representing polynomials

Another key use for Matlab arrays is to represent physical vectors, e.g. the coordinates of a point or the components of a force or velocity. We often express such vectors as column vectors, e.g.:

```
x = [1 0 -1]        % row vector
```

```
x = 1×3
     1     0    -1
```

```
y = [1 2 3]'        % column vector
```

```
y = 3×1
     1
     2
     3
```

Vector arithmetic works as expected:

```
x'+y     % addition (note the transpose to make dimensions match!)
```

```
ans = 3×1
```

```
     2
     2
     2
```

```
x-y'     % subtraction
```

```
ans = 1×3
     0    -2    -4
```

```
2*x      % scalar multiplication
```

```
ans = 1×3
     2     0    -2
```

Norm:

The **norm()** function yields the L2 norm (Euclidean norm), interpreted as the vector length in n-dimensional space. This is not to be confused with length(), which just gives the number of elements in an array!

```
norm(x)
```

```
ans = 1.4142
```

norm() is the same as:

```
sqrt(x*x')
```

```
ans = 1.4142
```

```
sqrt(dot(x,x))
```

```
ans = 1.4142
```

Dot product:

```
dot(x,y)
```

```
ans = -2
```

```
dot(y,x)   % = dot(x,y)
```

```
ans = -2
```

```

dot(x,x)   % = norm(x)^2
```

```
ans = 2
```

Manual calculation of dot(x,y):

```
x(1)*y(1) + x(2)*y(2) + x(3)*y(3)
```

```
ans = -2
```

A physical interpretation of the dot product is that dot(x,y) yields the length of the projection of x onto y, multiplied by the length of y:

dot(x,y) = norm(x)*norm(y)*cos(q)   where q = angle between x and y

This interpretation allows us to determine the angle between two vectors:

```
q = acos(dot(x,y)/(norm(x)*norm(y)))
```

```
q = 1.9584
```

dot() does not care if the arrays are row vectors, column vectors, or a mix of both:

```
dot(x',y)            % ans = -2
```

```
ans = -2
```

Cross product:

```
cross(x,y)
```

```
ans = 1×3
     2    -4     2
```

```
cross(y,x)       % = -cross(x,y)
```

```
ans = 1×3
    -2     4    -2
```

A physical interpretation of the cross product is that the magnitude of cross(x,y) is the area of a parallelogram with sides lxl and lyl, and the direction of cross(x,y) is orthogonal to the plane containing the parallelogram:

cross(x,y) = lxl lyl sin(q)*n_hat

where n_hat is a unit normal vector orthogonal to x and y (using the right hand rule). To find the plane in which x and y lie:

```
n_hat = cross(x,y)/(norm(x)*norm(y)*sin(q))   % q was found via dot(x,y)
```

```
n_hat = 1×3
    0.4082   -0.8165    0.4082
```

Note that n_hat has unit length:

```
norm(n_hat)
```

```
ans = 1
```

Matlab can work with *any* length vector not just 2 or 3. Indeed, you will often need to deal with very large vectors in advanced engineering problems:

```
x=[1 -2 3 4 -5 7 9 -pi]'
```

```
x = 8×1
```

```
    1.0000
   -2.0000
    3.0000
    4.0000
   -5.0000
    7.0000
    9.0000
   -3.1416
```

```
y=(1:8)'
```

```
y = 8×1
     1
     2
     3
     4
     5
     6
     7
     8
```

```
norm(x)
```

```
ans = 13.9596
```

```
dot(x,y)         % length(x) must be same as length(y)
```

```
ans = 76.8673
```

Cross products, however, are ONLY defined between 3 dimensional vectors, so cross(x,y) will not work here! The cross product *can* be extended to n>3 dimensions, but Matlab's cross() does not support this generalization.

MATRICES

Matrices are rectangular arrays of numbers, that have their own special arithmetic rules. Note that a vector can be viewed as a special case of a matrix containing only a single row (or column, depending on orientation).

Define a matrix in Matlab by specifying the numbers in each row, separated by spaces or commas. Start a new row with a semicolon

```
A = [1 2 3 -1; 4 5 6 0; 7 8 9 1]
```

```
A = 3×4
     1     2     3    -1
     4     5     6     0
     7     8     9     1
```

Can also put each row on a separate line (semicolons optional):

```
A = [ 1 2 3 -1
      4 5 6 0
      7 8 9 1 ]
```

```
A = 3×4
```

```
     1     2     3    -1
     4     5     6     0
     7     8     9     1
```

This matrix A is 3x4 (3 rows, 4 columns)

When defining a matrix, *all* rows & columns must have the same number of elements

```
% AA = [1 2 3 -1; 4 5 0; 7 8 9 1]      % ERROR
```

Oops; 2nd row only had 3 elements in above, while the other two rows had 4.

Indexing into a matrix to get one of its elements by specifing both the row and column number of the element

Element of A in 1st row, 3rd column:

```
r = 1;
c = 3;
A(r,c)      % ans = 3
```

```
ans = 3
```

Element of A in 2nd row, 4th column:

```
A(2,4)      % ans = 0
```

```
ans = 0
```

If only one argument is given for the index, Matlab will start counting at the upper left corner and work down each *column* in sequence:

```
A(1)        % ans = 1
```

```
ans = 1
```

```
A(4)        % ans = 2
```

```
ans = 2
```

We can change an individual matrix element:

```
A(2,4) = pi
```

```
A = 3×4
    1.0000    2.0000    3.0000   -1.0000
    4.0000    5.0000    6.0000    3.1416
    7.0000    8.0000    9.0000    1.0000
```

size() returns an array containing the number of rows and columns in a matrix:

```
[m,n] = size(A)     % m = 3, n = 4
```

```
m = 3
n = 4
```

We previously used sum() to find the summation of all values in an array (vector). When applied to a matrix, sum() will return the summation of values within each *column* of the matrix:

```
sum(A)
```

```
ans = 1×4
   12.0000   15.0000   18.0000    3.1416
```

Concatenating linear arrays into (longer) vectors or into rectangular arrays (matrices)

```
x = [1 2 3];
y = [-3 -2 -1];

z = [x y]              % z = [1 2 3 -3 -2 -1]
```

```
z = 1×6
     1     2     3    -3    -2    -1
```

```
A1 = [x
      y]               % A1 = [1 2 3; -3 -2 -1]
```

```
A1 = 2×3
     1     2     3
    -3    -2    -1
```

```
A2 = [x' y']           % A2 = [1 -3; 2 -2; 3 -1]
```

```
A2 = 3×2
     1    -3
     2    -2
     3    -1
```

Recall colon notation to slice a range of elements in a linear array:

```
x = [1 0 -1 2];
x(2:3)                 % ans = [0 -1]
```

```
ans = 1×2
     0    -1
```

Same notation can also be used for the row or column index (or both) in a matrix.

Pull out elements in 1st and 2nd rows, 3rd and 4th columns:

```
A(1:2, 3:4)            % ans = [3 -1; 6 0]
```

```
ans = 2×2
    3.0000   -1.0000
    6.0000    3.1416
```

Pull out elements in 1st and 2nd rows, 2nd through 4th columns:

```
A(1:2, 2:4)          % ans = [2 3 -1; 5 6 0]
```

```
ans = 2×3
    2.0000    3.0000   -1.0000
    5.0000    6.0000    3.1416
```

We can also assign new values to multiple elements at once using slicing:

```
A(1:2,2) = [-1; -1]
```

```
A = 3×4
    1.0000   -1.0000    3.0000   -1.0000
    4.0000   -1.0000    6.0000    3.1416
    7.0000    8.0000    9.0000    1.0000
```

Alternately, multiple elements can also be changed to a single scalar value:

```
A(2:3,3) = -100
```

```
A = 3×4
    1.0000   -1.0000    3.0000   -1.0000
    4.0000   -1.0000 -100.0000    3.1416
    7.0000    8.0000 -100.0000    1.0000
```

A colon ":" by itself as an index means "all". Pull out all rows of the 2nd column:

```
A(:,2)               % ans = [2; 5; 8]
```

```
ans = 3×1
    -1
    -1
     8
```

Pull out all the columns of the 2nd row:

```
A(2,:)               % ans = [4 5 6 0]
```

```
ans = 1×4
    4.0000   -1.0000 -100.0000    3.1416
```

Matrices may be "partitioned" into a set of rows or columns. Here is an example of column partitioning:

```
C1 = A(:,1)
```

```
C1 = 3×1
     1
     4
     7
```

```
C2 = A(:,2)
```

```
C2 = 3×1
```

```
    -1
    -1
     8
```

```
C3 = A(:,3)
```

```
C3 = 3×1
     3
  -100
  -100
```

```
C4 = A(:,4)
```

```
C4 = 3×1
   -1.0000
    3.1416
    1.0000
```

Now reconstruct the original matrix from the columns:

```
A = [C1 C2 C3 C4]
```

```
A = 3×4
    1.0000   -1.0000    3.0000   -1.0000
    4.0000   -1.0000 -100.0000    3.1416
    7.0000    8.0000 -100.0000    1.0000
```

We could instead use row partitioning to break a matrix into rows (noting there are only n=3 rows vs. m=4 columns for A):

```
R1 = A(1,:)
```

```
R1 = 1×4
     1    -1     3    -1
```

```
R2 = A(2,:)
```

```
R2 = 1×4
    4.0000   -1.0000 -100.0000    3.1416
```

```
R3 = A(3,:)
```

```
R3 = 1×4
     7     8  -100     1
```

```
A = [ R1
      R2
      R3 ]
```

```
A = 3×4
    1.0000   -1.0000    3.0000   -1.0000
    4.0000   -1.0000 -100.0000    3.1416
    7.0000    8.0000 -100.0000    1.0000
```

Matrix partitioning is a key step to understanding how matrix-vector multiplication is defined.

MATRIX-VECTOR MULTIPLICATION

Given a matrix A and column vector x, matrix-vector multiplication (A*x) requires that x has the same number of rows as A has columns, i.e. the inner dimensions of A and x must match.

```
M = [1 2 3 -1; 4 5 6 0; 7 8 9 1];   % M is 3x4
x = [-2; -1; 1];                    % x is 3x1
% y = M*x         % ERROR (4 vs. 3 for inner dimensions)
```

When the dimensional condition is satisfied, the resultant vector y=M*x will be a column vector with the same number of elements as A has rows, i.e. y will have the outer dimensions of M*x:

```
x = [-2; -1; 1; 2];     % x is 4x1
M*x                     % M is 3x4
```

```
ans = 3×1
    -3
    -7
   -11
```

The formal definition of the product can be given expressed as the **weighted sum of the columns of the matrix**

Return to our earlier 3x4 matrix A that was partitioned by column and row, and find A*x:

```
y = A*x
```

```
y = 3×1
         0
 -100.7168
 -120.0000
```

Remember, we defined above C1...C4 to be the respective columns of A, so let's apply the weighted sum concept to find A*x:

```

y = x(1)*C1 + x(2)*C2 + x(3)*C3 + x(4)*C4
```

```
y = 3×1
         0
 -100.7168
 -120.0000
```

A*x can also be written as a vector consisting of the products of the *rows* of A with the vector x:

```
y = [ R1*x
      R2*x
      R3*x ]
```

```
y = 3×1
```

```
         0
 -100.7168
 -120.0000
```

Both row and column expansions of the product are consistent and equivalent, and lead to Matlab's reported result for the product A*x. Matlab applies such rules automatically when the multiplication is valid; you don't need to manually apply the expansion (but be aware of it!)

MATRIX-MATRIX MULTIPLICATION

Define 2 matrices A and B

```
A = [1 2 3 -1
     4 5 6 0
     7 8 9 1];

B = [-1 0 1
     1 2 3];
```

Note that A is 3x4, B is 2x3

```
% A*B         % ERROR (inner dimensions are different)
```

Product A*B is not defined since inner dimensions don't match (4 vs 2)

```
B*A         % yields a 2x4 matrix result
```

```
ans = 2×4
     6     6     6     2
    30    36    42     2
```

Product BA is defined since inner dimensions match (both are 3), and result matrix is 2x4 (outer dimensions)

```
A = [1 2
     3 4
     5 6];   % 3x2 matrix

A*B                   % 3x2 * 2x3 --> 3x3
```

```
ans = 3×3
     1     4     7
     1     8    15
     1    12    23
```

Product is defined since inner dimensions match (both are 2). Result is 3x3 (outer dimensions)

```
B*A         % 2x3 * 3x2 --> 2x2
```

```
ans = 2×2
     4     4
    22    28
```

Product is also defined since inner dimensions also match, and result is 2x2.

Note from the three examples above that matrix multiplication is NOT COMMUTATIVE: generally A*B is different from B*A.

The row,column value resulting from matrix multiplication is given by the product of the row from the 1st matrix and the column of the 2nd matrix:

```
AB = A*B;

AB(2,3)                % ans = 15
```

```
ans = 15
```

```
A(2,:) * B(:,3)        % ans = 15
```

```
ans = 15
```

Similarly, ith column of C_AB = A*B is given by A * (ith column of B)

```
AB(:,2)
```

```
ans = 3×1
     4
     8
    12
```

```
A * B(:,2)
```

```
ans = 3×1
     4
     8
    12
```

Both these equivalent ways of viewing the matrix-matrix product give the same results, and agree with Matlab's reported result for A*B

Matrix transpose:

A' turns the rows of A into columns, and vice-versa (but remember that if your matrix contains imaginary numbers you will need to use .' instead of ' for the non-conjugate transpose!)

```
A
```

```
A = 3×2
     1     2
     3     4
     5     6
```

```
A'
```

```
ans = 2×3
     1     3     5
     2     4     6
```

Identity matrix:

The identity matrix is a square matrix (same number of rows and columns), with "1" on the diagonal and "0" everywhere else:

```
I = [ 1 0 0
      0 1 0
      0 0 1 ]
```

```
I = 3×3
     1     0     0
     0     1     0
     0     0     1
```

Identity matrices are equivalent to the number 1 in ordinary scalar arithmetic. However, keep in mind that dimensions matter:

```
I*A          % 3x3 * 3x2 --> 2x2
```

```
ans = 3×2
     1     2
     3     4
     5     6
```

```
% A*I          % 3x2 * 3x3 --> ERROR

% I*B          % 2x3 * 3x3 --> ERROR
B*I          % 2x3 * 2x3 --> 2x3
```

```
ans = 2×3
    -1     0     1
     1     2     3
```

Also true for vectors, assuming they are of correct size for multiplication

```
A2 = A(:,2)     % Second column of A
```

```
A2 = 3×1
     2
     4
     6
```

```


I*A2            % 3x3 * 3x1 --> 3x1
```

```
ans = 3×1
     2
     4
     6
```

We can easily generate identity matrices of any size using Matlab's **eye()** function

```
I = eye(4)      % 4x4 identity matrix
```

```
I = 4×4
```

```
1     0     0     0
0     1     0     0
0     0     1     0
0     0     0     1
```

Input to eye() is the desired sqaure matrix size (#rows = #cols)

If given a matrix as its input, the Matlab **diag()** function returns a column array with only the diagonal elements of that matrix

```
diag(I)     % ans = [1; 1; 1; 1]
```

```
ans = 4×1
     1
     1
     1
     1
```

```
C = [1 0 1; 2 2 0; 3 1 5]
```

```
C = 3×3
     1     0     1
     2     2     0
     3     1     5
```

```
diag(C)     % ans = [1; 2; 5]
```

```
ans = 3×1
     1
     2
     5
```

diag() can also be used to generate a square matrix If diag is given a (linear) array as its input, it creates a square matrix with the elements of the array on the diagonal, and zeros elsewhere.

This is an easy way to "strip" all but the diagonal elements from a matrix (setting the others to 0):

```
C1 = diag(diag(C))
```

```
C1 = 3×3
     1     0     0
     0     2     0
     0     0     5
```

As another example, use diag() to generate an identity matrix:

```
diag([1 1 1])
```

```
ans = 3×3
     1     0     0
     0     1     0
     0     0     1
```

MATRIX INVERSION

Inverting a 2x2 matrix by hand is easy (assuming you've taken a Linear Algebra course, that is). Inverting a 3x3 is not too bad. Inverting a 4x4, 5x5, ... by hand is a nightmare! Matlab makes this easy using the inv() function:

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

```
A = 2×2
     1     2
     3     4
```

```
inv(A)
```

```
ans = 2×2
   -2.0000    1.0000
    1.5000   -0.5000
```

Product of matrix and its inverse is always the identity matrix:

```
A * inv(A)          % ans = [1 0; 0 1]
```

```
ans = 2×2
    1.0000         0
    0.0000    1.0000
```

```
inv(A) * A          % ans = [1 0; 0 1]
```

```
ans = 2×2
    1.0000         0
    0.0000    1.0000
```

Let's look at a larger matrix

```
A2 = [ 1  2  3
       4 -5  6
       9  8  7 ]
```

```
A2 = 3×3
     1     2     3
     4    -5     6
     9     8     7
```

```
inv(A2)
```

```
ans = 3×3
   -0.4150    0.0500    0.1350
    0.1300   -0.1000    0.0300
    0.3850    0.0500   -0.0650
```

The **determinant** of a matrix can be found using the **det()** function. Note that if the determinant of a matrix is zero, it is "singular" and its inverse does not exist.

det() calculates the determinant of any square matrix:

```
A = [ 1 1
      3 3 ]
```

```
A = 2×2
     1     1
     3     3
```

```
det(A)      % close to zero!
```

```
ans = 0
```

```
inv(A)      % {Warning: Matrix is singular to working precision.}
```

```
Warning: Matrix is singular to working precision.
ans = 2×2
   Inf   Inf
   Inf   Inf
```

Determinant of A is zero here, so its inverse is not defined.

Singluar matrices are the like the number zero in scalar arithmetic; they do not have a defined inverse. More precisely, in scalar arithmetic, there is no meaningful solution for x in the equation a*x = b when a=0 and b<>0 (and if b=0, *all* values of x are valid solutions)

We can encounter, and solve, similar problems in matrix arithmetic, provided the matrix is square and nonsingular.

Solve A2*x = b for x:

```
b = [-1;0;1];
x = inv(A2)*b
```

```
x = 3×1
    0.5500
   -0.1000
   -0.4500
```

```
A2*x      % same as b since x is a solution of A2*x=b
```

```
ans = 3×1
   -1.0000
         0
    1.0000
```

Multiplication by the inverse of a matrix can be simplified in Matlab using the backslash (\) operator. For example, instead of "inv(A2) * b" we could use the following syntax:

```
A2 \ b   % Note the order -- the backslash tells us to read from right to left!
```

```
ans = 3×1
    0.5500
   -0.1000
   -0.4500
```