Lecture 2

Arrays

1 Introduction

As the name Matlab is a contraction of "matrix laboratory," you would be correct in assuming that Scilab/Matlab have a particular emphasis on matrices, or more generally, arrays. Indeed, the manner in which Scilab/Matlab handles arrays is one of its great strengths. Matrices, vectors and the operations of linear algebra are of tremendous importance in engineering, so this material is quite foundational to much of what we will do in this course. In this lesson we are going to focus on generating and manipulating arrays. In later lessons we will use them extensively to solve problems. Consider the following session.

```
-->A = [1,2;3,4]
 A  =
    1.    2.
    3.    4.

-->x = [1;2]
 x  =
    1.
    2.

-->y=A*x
 y  =
    5.
    11.
```

Here we defined a 2-by-2 matrix `A` and a 2-by-1 vector `x`. We then computed the matrix-vector product `y=A*x` which is a 2-by-1 vector.

There are various ways to create an array. In general the elements of an array are entered between the symbols `[...]`. A space or a comma between numbers moves you to the next column while a carriage return ("enter" key on the keyboard) or a semi-colon moves you to the next row. For example

```
-->B = [ 1 2
-->      3 4]
 B  =
    1.    2.
    3.    4.
```

creates a 2-by-2 array using spaces and a carriage return. The following example shows how either commas or spaces can be used to separate columns.

```
-->u = [1,2,3]
 u  =
    1.    2.    3.

-->v = [1 2 3]
 v  =
    1.    2.    3.
```

It's a matter of preference, but I find that using commas increases readability, especially when

entering more complicated expressions. The comma clearly delimits the column entries. Now we demonstrate how either semi-colons, carriage returns or both can be used to separate rows.

```
-->x = [1;2;3]
 x  =
    1.
    2.
    3.

-->y = [1
-->     2
-->     3]
 y  =
    1.
    2.
    3.

-->z = [1;
-->     2;
-->     3]
 z  =
    1.
    2.
    3.
```

Again, my preference is for a visible delimiter.

Exercise 1: Input the following arrays in the console:

$$z=\begin{bmatrix} 7 \\ -2 \\ 3 \end{bmatrix}, \quad B=\begin{bmatrix} -2 & 1 & 5 \\ 1 & -3 & 4 \end{bmatrix}$$

2 Initializing an array

As shown above, arrays can be entered directly at the command line (or within a program). There are some special arrays that are used frequently and can be created using built-in functions. An array of all zeros can be created using the zeros(m,n)command

```
-->A = zeros(2,3)
 A  =
    0.    0.    0.
    0.    0.    0.
```

The command `ones(m,n)` creates an array of all 1's.

```
-->B = ones(3,2)
 B  =
    1.    1.
    1.    1.
    1.    1.
```

An array with 1's on the diagonal and 0's elsewhere is created using the `eye(m,n)` command ("eye" as in "i"dentity matrix).

```
-->C = eye(3,3)
 C  =
    1.    0.    0.
    0.    1.    0.
    0.    0.    1.
```

Here this creates a 3-by-3 identity matrix. However, the matrix need not be square, as in

```
-->D = eye(2,3)
 D  =
    1.    0.    0.
    0.    1.    0.
```

To create a square matrix of all 0's except for specified values along the diagonal we use the `diag([...])` command

```
-->D = diag([1,2,3])
 D  =
    1.    0.    0.
    0.    2.    0.
    0.    0.    3.
```

The diag command also allows you to extract the diagonal elements of a matrix

```
-->A = [1,2;3,4]
 A  =
    1.    2.
    3.    4.

-->diag(A)
 ans  =
    1.
    4.
```

The function `size(A)` returns the dimensions of the array A, as in

```
-->size(A)
 ans  =
    2.    2.
```

If you only want the number or rows or columns of a matrix you can specify that

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

    1.    2.    3.
    4.    5.    6.

-->size(A,'r') //or size(A,1) which is how Matlab does it
 ans  =

    2.

-->size(A,'c') //or size(A,2) which is how Matlab does it
 ans  =

    3.
```

Vectors are one-dimensional arrays. The `size()` command works on vectors, but the `length()` command can be more convenient in that it returns a single number which is the number of elements in the vector. Consider the following.

```
-->v = [1;2;3]
 v  =
    1.
    2.
    3.

-->size(v)
 ans  =
    3.    1.

-->length(v)
 ans  =
    3.
```

An array operator that can be useful in entering array values is the transpose operator. In Scilab/Matlab this is the single quote sign. Consider the following

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

Sometimes it is convenient to produce arrays with random values. The `rand(m,n)` command does this.

```
-->B = rand(2,3)
 B  =
    0.2113249    0.0002211    0.6653811
    0.7560439    0.3303271    0.6283918
```

The random numbers are uniformly distributed between 0 and 1. You can use commands like `zeros, ones` and `rand` to create an array with the same dimensions as an existing array. For example, in Scilab

```
-->A = eye(3,2)
 A  =
    1.    0.
    0.    1.
    0.    0.

-->ones(A)   //Scilab specific
 ans  =
    1.    1.
    1.    1.
    1.    1.
```

The ones(A) command uses the dimensions of A to form the array of ones. Matlab is slightly different. In Matlab the corresponding command would be

```
>>A = eye(3,2)
 A  =
    1.    0.
    0.    1.
    0.    0.

>>ones(size(A)) %Matlab specific
 ans  =
    1.    1.
    1.    1.
    1.    1.
```

In Matlab you need to use the `size()` function to pass the dimensions of the array A. In Scilab you do not.

In many applications you want to create a vector of equally spaced values. For example

```
-->t = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]
 t  =
    0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8
```

Instead of entering all the values directly you can use the following short cut

```
-->t = 0:0.1:0.8
 t  =
    0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8
```

This "colon" notation tells Scilab/Matlab to create a vector named `t`, starting at 0 and incrementing by 0.1 up to the value 1. This approach allows you to define the increment (0.1 in this case). The default increment is 1 as shown here

```
-->x = 1:5
 x  =
    1.    2.    3.    4.    5.
```

Sometimes you are more concerned about the total number of elements in the vector rather than a specific increment. The `linspace()` command allows you to specify the start and end values and the total number of elements. Consider the following.

```
-->x = linspace(0,1,6)
 x  =
    0.    0.2    0.4    0.6    0.8    1.
```

This creates a vector with values ranging from 0 to 1 and a total of 6 elements. It is sometimes convenient to be able to "reshape" an array. In Scilab the `matrix()` command allows you to do this.

```
-->A = [1,2,3;4,5,6]
 A  =
    1.    2.    3.
    4.    5.    6.

-->matrix(A,3,2) //Scilab specific
 ans  =
    1.    5.
    4.    3.
    2.    6.
```

In Matlab the `reshape()` command does this.

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

>>reshape(A,3,2) %Matlab specific
 ans  =
    1.    5.
    4.    3.
    2.    6.
```

Finally, the elements of an array are not limited to numerical values but can consist of defined

constants, variables or functions. For example,

```
-->x = 0.2
 x  =
0.2


-->C = [cos(x),sin(x);-sin(x),cos(x)]
 C  =
    0.9800666    0.1986693
  - 0.1986693    0.9800666
```

Exercise 2: Use the colon notation to generate the array 0, 0.25, 0.5, 0.75, 1.

Exercise 3: Enter the array $B = \begin{bmatrix} -2 & 1 & 5 \\ 1 & -3 & 4 \end{bmatrix}$ (already defined in Ex 1). Use the ones function to create an array of 1 of the same size as B.

3 Combining arrays

Two or more existing arrays can be combined into a single array by using the existing arrays as elements in the new array. Consider the following.

```
-->v = [1,2,3]
 v  =
    1.    2.    3.

-->A = [v;v]
 A  =
    1.    2.    3.
    1.    2.    3.

-->B = [v,v]
 B  =
    1.    2.    3.    1.    2.    3.
```

The arrays being combined must “fit” together by having compatible dimensions, otherwise you receive an error.

```
-->u = [4,5]
 u  =
    4.    5.

-->C = [u;v]
         !--error 6
Inconsistent row/column dimensions.
```

However consider

```
-->[u,0;v]
 ans  =
    4.    5.    0.
    1.    2.    3.

-->D = [u,v]
 D  =
    4.    5.    1.    2.    3.
```

Here are two more examples.

```
-->A = [1,2;3,4];

-->B = [5,6;7,8];
-->[A,B]
 ans  =
    1.    2.    5.    6.
    3.    4.    7.    8.

-->[A;B]
 ans  =
    1.    2.
    3.    4.
    5.    6.
    7.    8.
```

These kinds of operations are very useful in many applications where large matrices are constructed by stacking sub-matrices together. The submatrices might represent specific pieces of a system, and the stacking operation corresponds to "assembling" the system.

Exercise 4: Enter $u=\begin{bmatrix}1 & 2 & 3\end{bmatrix}$ and $v=\begin{bmatrix}4\\5\end{bmatrix}$. Combine these to form the array $C=\begin{bmatrix}1 & 2 & 3 & 4\\1 & 2 & 3 & 5\end{bmatrix}$.

4 Operating on array elements

If A is a two-dimensional array, then `A(m,n)` refers to the element in the mth row and nth column. Consider the following.

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

-->A(3,2)
 ans  =
    8.

-->A(3,2) = 10
 A  =
    1.    2.     3.
    4.    5.     6.
    7.    10.    9.
```

We can both access and assign the value of `A(3,2)` directly. Often you want to extract a row or column of a matrix. The "colon" notation allows you to do this. For example,

```
-->A(:,1)
 ans  =
    1.
    4.
    7.
```

```
-->v = A(2,:)
 v  =
    4.    5.    6.
```

In the first case we extract the 1st column of `A`. In the second case we extract the 2nd row of `A` and assign it to the variable `v`. In general the semicolon tells Scilab/Matlab to run through all values of the corresponding subscript or index. More generally you can extract a subrange of values. Consider

```
-->A(1:2,2:3)
 ans  =
    2.    3.
    5.    6.
```

This extracts rows 1 through 2 and columns 2 through 3 to create a new 2-by-2 matrix from the elements of the original 3-by-3 matrix. Elements of a vector can be deleted in the following manner.

```
-->t = 0:0.2:1
 t  =
    0.    0.2    0.4    0.6    0.8    1.

-->t(2) = []
 t  =
    0.    0.4    0.6    0.8    1.


-->t(3:5) = []
 t  =
    0.    0.4
```

In the first case we delete the second element of *t*. In the second case we delete elements 3 through 5. This technique applies to deleting rows or columns of a matrix.

```
 A  =
    1.    2.    3.
    4.    5.    6.
    7.    10.   9.

-->A(1,:) = []
 A  =
    4.    5.    6.
    7.    10.   9.


-->A(:,2) = []
 A  =
    4.    6.
    7.    9.
```

Exercise 5: Use the eye command to generate the array $A=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}$. Change a single element of A to get $A=\begin{bmatrix}1&0&0\\0&1&2\\0&0&1\end{bmatrix}$.

Exercise 6: Starting with $A=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 2\\ 0 & 0 & 1\end{bmatrix}$, extract the 2nd row to get $u=\begin{bmatrix}0 & 1 & 2\end{bmatrix}$. Extract the 3rd column to get $v=\begin{bmatrix}0\\ 2\\ 1\end{bmatrix}$.

5 Arithmetic operations on arrays

Arrays can be multiplied or divided by a scalar (single number)

```
-->1.5*[1,2,3]
 ans  =
    1.5    3.    4.5

-->[1,2,3]/2
 ans  =
    0.5    1.    1.5
```

An operation such as $A+1$ where A is a matrix makes no sense algebraically, but Scilab/Matlab interprets this as a shorthand way of saying you want to add 1 to each element of A

```
-->[1,2;3,4]+1
 ans  =
    2.    3.
    4.    5.
```

Addition, subtract and multiplication of two arrays follow the rules of linear algebra. To add or subtract arrays they must be of the same size. If they are not you get an error.

```
-->A = [1,2;3,4]
 A  =
    1.    2.
    3.    4.


-->B = [1,2,3;4,5,6]
 B  =
    1.    2.    3.
    4.    5.    6.

-->A+B
    !--error 8
Inconsistent addition.
```

Otherwise each element of the resulting array is the sum or difference of the corresponding elements in the two arrays.

```
-->C = [2,1;4,3]
 C  =
    2.    1.
    4.    3.

-->A+C
 ans  =
```

```
    3.    3.
7.    7.

-->A-C
 ans  =
  - 1.    1.
  - 1.    1.
```

To multiply two arrays as in `A*B` the number of columns of `A` must equal the number of rows of `B`. If `A` is m-by-n then `B` must be n-by-p. The product `A*B` will be m-by-p.

```
-->A
 A  =
    1.    2.
    3.    4.

-->B
 B  =
    1.    2.    3.
    4.    5.    6.

-->A*B
 ans  =
    9.     12.    15.
    19.    26.    33.

-->B*A
    !--error 10
Inconsistent multiplication.
```

If `C=A*B` the elements of `C` are computed as $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$.

The inner product, or dot product of two vectors can be calculated by transposing one and performing an array multiplication.

```
-->u = [1;2;3]
 u  =
    1.
    2.
    3.

-->v = [4;5;6]
 v  =
    4.
    5.
    6.

-->u'*v
 ans  =
    32.

-->1*4+2*5+3*6
 ans  =

    32.
```

Arrays cannot be divided, but we do have the concept of "inverting" a matrix to solve a linear equation. If $Ax = b$ and A is a square, non-singular matrix, then $x = A^{-1} b$ is the solution to

this system of linear equations. This is sort of like "dividing A into b." In Scilab/Matlab we use the notation `A\b` to represent this. You can also think of the \ operator as representing the inverse operation, so that `A\` functions as A^{-1}. Consider the following.

```
-->A = [1,2;3,4]
 A  =
    1.    2.
3.    4.

-->b = [5;6]
 b  =
    5.
    6.

-->x = A\b
 x  =
  - 4.
    4.5

-->A*x
 ans  =
    5.
    6.
```

If the matrix `A` is singular (has no inverse) you'll get an error. We'll talk more about solving systems of linear equations later.

Exercise 7: If $A=\begin{bmatrix} 2 & 1 \\ -3 & 4 \end{bmatrix}$ and $b=\begin{bmatrix} 1 \\ -1 \end{bmatrix}$ find x such that $Ax=b$.

6 Vectorized functions

A very powerful feature of Scilab/Matlab is that functions can be "vectorized." In a language such as C, if I have an array `x` and I want to calculate the `sin` of each element of `x`, I need to use a for loop, as in

```
for (i=1;i<=n;i++) {
  y(i) = sin(x(i));
}
```

In Scilab/Matlab we simply write

```
y = sin(x)
```

This creates a new array `y` of the same size as `x`. Each element of `y` is the `sin` of the corresponding element of `x`. For example

```
-->x = [0,0.1,0.2,0.3]
 x  =
    0.    0.1    0.2    0.3

-->y = sin(x)
 y  =
    0.    0.0998334    0.1986693    0.2955202
```

Exercise 8: Set `x = 0:0.2:1` and calulate `y = exp(x)` then `z = log(y)`

7 Array operators

In linear algebra, arrays (e.g., vectors and matrices) are considered entities in their own right and there are rules for operating on them, such as matrix multiplication and the inverse. In practice, sometimes an array is just a convenient collection of numbers. In this case you might want to perform operations on the elements of the array independent of the rules of linear algebra. For example, suppose $u=[1\ \ 2\ \ 3]$ and $v=[4\ \ 5\ \ 6]$ and you want to multiply each element of u by the corresponding element of v $w=[1\cdot 4\ \ 2\cdot 5\ \ 3\cdot 6]=[4\ \ 10\ \ 18]$. This is not a standard operation in linear algebra. To perform component-by-component operations (or "array operations") you prefix the operator with a period. For example,

```
-->u = [1,2,3]
 u  =
    1.    2.    3.

-->v = [4,5,6]
 v  =
    4.    5.    6.

-->u.*v
 ans  =
    4.    10.    18.

-->u.^2
 ans  =
    1.    4.    9.
```

This brings up a subtle point. Consider the following.

```
-->A = [1,2;3,4]
 A  =
    1.    2.
    3.    4.

-->A^2
 ans  =
    7.     10.
    15.    22.

-->A.^2
 ans  =
    1.    4.
    9.    16.
```

Notice the very different results. The operation `A^2` tells Scilab/Matlab to use the rules of matrix multiplication to calculate `A*A`. The operation `A.^2` tells Scilab/Matlab to square each element of `A`.

When an array represents a collection of values, say measurements, we often want to look at properties such as the sum or average of the values. Consider the following.

```
-->x = 0:0.1:1
 x  =
    0.  0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9    1.

-->sum(x)
 ans  =
    5.5
```

```
-->mean(x)
 ans  =
    0.5
```

The function `sum()` adds all the elements of an array while `mean()` calculates the average value of the elements. If we wanted to find the mean-square value (average of the squared values) we could use the command

```
-->mean(x.^2)
 ans  =
    0.35
```

This first squares each element of `x` and computes the mean of those squared values. Here's an important point. Consider the following

```
-->x = [1,2];
-->1./x
 ans  =
    0.2
    0.4

-->(1)./x
 ans  =
    1.    0.5
```

In the first instance Scilab interpreted the dot as a decimal point and returned

```
-->(1.)/x
 ans  =
    0.2
    0.4
```

which is something called the “pseudo inverse” and not what we were after. We need to use parentheses to avoid this interpretation

```
-->(1)./x
 ans  =
    1.    0.5
```

Exercise 9: Set `x = 0:0.2:1` and calculate `tan(x)`. Then calculate `sin(x)./cos(x)` and compare the results.

8 Complex array elements

All of the preceding applies to complex array elements. For example, in Scilab (for Matlab use `1i` instead of `%i`)

```
-->A = [1,%i;2,-%i]
 A  =
    1.    i
    2.  - i

-->A^2
 ans  =
    1. + 2.i    1. + i
    2. - 2.i  - 1. + 2.i
```

The output can be a bit difficult to read with complex arrays. `A^2` produces a 2-by-2 matrix of

complex values each expressed as real and imaginary parts. It can be helpful to view these separately. The `real` and `imag` commands allow you to do this.

```
-->B = A^2
 B  =
    1. + 2.i    1. + i
    2. - 2.i  - 1. + 2.i

-->real(B)
 ans  =
    1.    1.
    2.  - 1.

-->imag(B)
 ans  =
    2.    1.
  - 2.    2.
```

One subtle point is that the single-quote operator is actually the "transpose-conjugate" operator. It takes the transpose of a matrix followed by its complex conjugate. For a real matrix this is simply the transpose. But for a complex matrix you need to remember that there is also a conjugation operation.

```
-->A
 A  =
    1.    i
    2.  - i

-->A'
 ans  =
    1.     2.
  - i      i
```

If you just want the transpose of a complex matrix use the `.'` operator.

```
-->A.'
 ans  =

    1.     2.
    i    - i
```

> **Exercise** 10: Set `x = 0:0.2:1` and calculate `w = exp(%i*x)`. (Use `i` instead of `%i` in Matlab.) Calculate `real(w)` and compare to `cos(x)`. Calculate `imag(w)` and compare to `sin(x)`.

9 Structures

Arrays are convenient ways to organize data of a common type. Often you want to organize different types of data as a single entity. Many programing languages allow you to do this using “structures.” In Scilab/Matlab a structure is define as follows.

```
St = struct(field1,value1,field2,value2...);
```

Here's an example

```
-->Object = struct('name','ball joint','mass',2.7)
 Object  =
   name: "ball joint"
   mass: 2.7
```

This creates a structure named “object” having two fields: name and mass. Fields are accessed using the syntax `struct.field`. Here we change the mass

```
-->Object.mass = 3.2
 Object  =
   name: "ball joint"
   mass: 3.2
```

Fields can be strings, numbers or even arrays. For example, say we have a rigid body. The orientation of this body in space is specified by six numbers: three coordinates of its center of mass, and three rotation angles. We might define a structure such as

```
-->Body = struct('pos',[-2,0,3],'ang',[12,-20,28])
 Body  =
   pos: [-2,0,3]
   ang: [12,-20,28]
```

Elements of the arrays are accessed as follows

```
-->Body.ang(2) = -22
 Body  =
   pos: [-2,0,3]
   ang: [12,-22,28]
```

The use of structures can greatly streamline complicated programing projects by uniting various data into a single logical entity. Arrays of structures are possible, as in

```
-->Person = struct('name','','age',0);
-->Member = [person,person,person];
-->Member(2).name = 'Tom';
-->Member(2).age = 32;
-->Member(2)
 ans  =
   name: "Tom"
   age: 32
```

This defines a structure person and then an array of three of these structures named member. Each element could refer to a member of a three-member team. Fields are accessed as shown. You can even have structures serve as fields of other structures. Here's an example.

```
-->Book = struct('title','','author','');
-->Class = struct('name','','text',Book);
-->Class.name = "Engl 101";
-->Class.text.title = "How to Write";
-->Class.text.author = "Kay Smith";
-->Class.text
 ans  =
   title: "How to Write"
   author: "Kay Smith"
```

You can skip the “initializing” of the structure and just start assigning values to the fields. For example.

```
-->clear
-->Class.name = 'Intro Psych';
-->Class.enrollment = 24;
```

10 Three and higher dimensional arrays

Arrays can have as many dimensions as desired. Of course arrays with more than two dimensions are cumbersome to display on the screen or page. Consider

```
-->A = zeros(2,2,2)
 A  =

(:,:,1)

    0.    0.
    0.    0.
(:,:,2)

    0.    0.
    0.    0.
```

This creates a three-dimensional array. Elements are accessed in the usual manner

```
-->A(2,1,2) = 3
 A  =

(:,:,1)

    0.    0.
    0.    0.
(:,:,2)

    0.    0.
    3.    0.
```