Data Analysis in Geophysics
ESCI 7205

Class 14

Bob Smalley

MATLAB

the Matrix

A matrix is a rectangular array of numbers

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Vectors are matrices with only one row or column

16	3	2	13

Scalars can be thought of as 1-by-1 matrices

16

Matlab basically thinks of everything as a matrix.

Handles math operations on

Scalars
Vectors
2-D matricies

With ease

Gets ugly with higher dimension matrices – as there are no mathematical rules to follow.

Entering Matrices

- Enter an explicit list of elements.

- Load from external data files.

- Generate using built-in functions

- Create with your own functions in m-files (matlab's name for a file containing a matlab program. Same as shell script, sac macro, batch file, commnad file, etc. but for matlab.)

Entering a matrix from the command line (method 1):

Separate the elements (columns) of a row with blanks or commas.

Use a semicolon, "**;**" or **<CR>**, to indicate the end of each row.

Surround the entire list of elements with square brackets, **[ ]**.

```
>> A44 = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A44 =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
>> A44 = [
16 3 2 13
5 10 11 8
9 6 7 12
4 15 14 1
]
A44 =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

Looks like mathematical matrix

```
>> A44 = [16 3 2 13
5 10 11 8
9 6 7 12
4 15 14 1]
```

Matrices indexed the same as math (row, column)

```
>> A14 = [16 3 2 13]
A14 =
    16     3     2    13
>> A41 = [16; 5; 9; 4]
A41 =
    16
     5
     9
     4
>> whos
  Name      Size            Bytes  Class     Attributes
  A14       1x4                32  double
  A41       4x1                32  double
  A44       4x4               128  double
>>
```

Row vector/matrix

Column vector/matrix

whos – reports what is in memory

Matrices indexed the same as math (row, column)

Suppressing Output

If you simply type a statement and press Return or Enter, MATLAB automatically displays the results on screen.

If you end the line with a semicolon, MATLAB performs the computation but does not display any output. This is particularly useful when you generate large matrices.

Matlab normally prints out results – to stop printout, end line with semi-colon " **;** " (this is general rule).

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =

    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1];
>>
```

The `load` function

reads binary files containing matrices (generated by earlier MATLAB sessions), or text files containing numeric data.

The text file should be organized as a rectangular table of numbers, separated by blanks, with one row per line, and an equal number of elements in each row.

```
>> cat magik.dat
16.0    3.0     2.0     13.0
5.0     10.0    11.0    8.0
9.0     6.0     7.0     12.0
4.0     15.0    14.0    1.0
>> A=load('magik.dat') #places matrix in variable A
>> load magik.dat   #places matrix in variable magik
```

Matlab is particularly difficult to use if data files do not fit this format (varying number columns for example).

Matlab is also particularly difficult to use for processing character data.

Generate matrices using built-in functions.

Complicated way of saying “run commands” and send output to new matrices.

Matlab also does matrix operations (e.g. - transpose).

```
>>magik'  #transpose matrix magik
ans =
   16  5   9   4
   3   10  6   15
   2   11  7   14
   13  8   12  1
```

m-Files

Text files with MATLAB code (instructions). Use MATLAB Editor (or any text editor) to create files containing the same statements you would type at the MATLAB command line.

Save the file with a name that ends in .m

```
% vim magik.m
i
A = [ 16.0 3.0 2.0 13.0
5.0 10.0 11.0 8.0
9.0 6.0 7.0 12.0
4.0 15.0 14.0 1.0 ];
(esc)wq
```

in matlab, execute the m file magik.m

```
>> magik   #places matrix in A
```

Entering long statements

If a statement does not fit on one line, use an ellipsis (three periods), "...", followed by "Carriage Return" or "Enter" to indicate that the statement continues on the next line.

```
>>s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
          - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

Subscripts

Matrices consists of rows and columns. The element in row `k` and column `l` of `A` is denoted by `A(k,l)` (same as math).

Example: `A(4,2)=15.0`

k \ l	1	2	3	4
1	16.0	3.0	2.0	13.0
2	5.0	10.0	11.0	8.0
3	9.0	6.0	7.0	12.0
4	4.0	15.0	14.0	1.0

4th row, 2nd column.

If you store a value in an element outside of the current size of a matrix, the size increases to accommodate the newcomer:

```
>> A = [ 16.0 3.0 2.0 13.0
5.0 10.0 11.0 8.0
9.0 6.0 7.0 12.0
4.0 15.0 14.0 1.0 ];
>> X = A;
>> X(4,5) = 17
X =
16 3     2    13  0
5  10  11     8   0
9  6     7    12  0
4   15  14    1    17
>>
```

You can also access the element of a matrix by referring to it as a single number.

This is because computer memory is addressed linearly – a single line of bytes (or words).

There are therefore (at least) two ways to organize a two dimensional array in memory – by row or by column (and both are/have been used of course).

MATLAB (and Fortran) store the elements by columns (called column major order).

```
>> A = [ 16.0 3.0 2.0 13.0
5.0 10.0 11.0 8.0
9.0 6.0 7.0 12.0
4.0 15.0 14.0 1.0 ]
A=
16  3    2    13
5   10   11   8
9   6    7    12
4   15   14   1
```

The elements are stored in memory by column.

```
16,  5,  9,  4,  3,  10, 6,  15, 2,  11,   7,  14,  13,   8,  12,   1.
(1) (2) (3) (4) (5)  (6)(7)  (8)(9)(10)(11)(12)(13)(14)(15)(16)
```

So `A(11)=7`.

How stuff stored in memory – column major order

```
>> a=[11 12; 21 22]
a =
    11    12
    21    22
>> a(:)
ans =
    11
    21
    12
    22
>> b=[11 12 21 22]
b =
    11    12    21    22
>> b(:)
ans =
    11
    12
    21
    22
>> whos
  Name      Size            Bytes  Class     Attributes
  a         2x2                32  double
  ans       4x1                32  double
  b         1x4                32  double
>>
```

```
>> a=[1,2,3]
a =

     1     2     3
>> a(:)
ans =
     1
     2
     3
>> b=[1;2;3]
b =
     1
     2
     3
>> b(:)
ans =
     1
     2
     3
>>
```

Same in memory, different "vectors"

`A(k,l)`

`k` varies most rapidly

`l` varies least rapidly

For `4x4` 2-D matrix

```
(1,1), (2,1), (3,1), (4,1), (1,2), (2,2)…(3,4), (4,4,)
 (1)    (2)    (3)    (4)    (5)    (6)   (15)   (16)
```

This may be important when reading and writing very large matrices – one wants the data file to have the same storage order as memory to minimize time lost due to page faulting.

When you go to 3 dimensions, order of subscript variation is maintained (1st to last)

`A(k,l,m)`

`k` varies most rapidly

`l` varies next most rapidly

`m` varies least rapidly

For `3x2x2` matrix

```
(1,1,1), (2,1,1), (3,1,1),
(1,2,1), (2,2,1), (3,2,1),
(1,1,2), (2,1,2), (3,1,2),
(1,2,2), (2,2,2,), (3,2,2),
...
```

C uses row major order (stores by row).

If mixing Matlab and Fortran there is no problem as both use column major order.

If mixing Matlab or Fortran and C – one has to take the array storage order into account.

If mixing Matlab or Fortran and C – one has to take the array storage order into account.

one also has to deal with how information is passed

- by reference [the address of the information in memory – Fortran]

- or value [a copy of the information – C].)

Although all three pass arrays by reference (can't always copy big arrays)

`0-d` scaler

`1-d` vector

`2-d` matrix

`3-d` think of as a stack of `2-d` matrices

`>3-d` something hard to visualize – but fine mathematically (`4-d` is `2-d` matrix with each element itself a `2-d` matrix)

The Colon Operator

The colon, " : ", is one of the most important (and sometimes seemingly bizarre) MATLAB operators

It can be used to

- Create a list of numbers
- Work with all entries in specified dimensions
- Collapse trailing dimensions (right- or left-hand side)
- Create a column vector (right-hand side behavior related to reshape)
- Retain an array shape during assignment (left-hand side behavior)

Creating a List of Numbers

You can use the ":" operator to create a 1-d vector of evenly-spaced numbers.

Here are the integers from -3 to 3.

```
>> list1=-3:3
list1 =
    -3    -2    -1     0     1     2     3
```

Don't need the braces (are optional)

Creating a List of Numbers

Here are the first few odd positive integers.

```
>>list2 = 1:2:10
list2 =
     1     3     5     7     9
```

Can use negative increments

```
>>100:-7:51
ans =
     100 93 86 79   72  65  58  51
```

syntax for this use of colon operator –

start:[increment if ≠1:]end

(default increment = 1)

Creating a List of Numbers

Here's how to divide the interval between 0 and **pi** (Matlab knows about π) into equally spaced samples (increment does not have to be whole #).

```
>>nsamp = 5;
>>sliceOfPi = (0:1/(nsamp-1):1)*pi)
sliceOfPi =
         0    0.7854    1.5708    2.3562    3.1416
```

(Note – can also define single dimension row matrix with `()`'s. Does not work when try to use "`;`" for another row.)

```
a=(1:3)
a =
     1     2     3
```

Things that don't work

```
>> a=(1:3;4:6)
 a=(1:3;4:6)
        |
Error: Unbalanced or unexpected parenthesis or bracket.

>> a=1:3;4:6

ans =

     4     5     6

>>
```

Second one (no error reported) creates array named `a = 1 2 3` and then and array named `ans = 4 5 6`. It uses the `;` as a line separator, not a row separator (when outside `[ ]`).

Aside – for languages that, unlike Matlab, don't have π predefined, how can one get the "best" representation of pi (most precise on that computer)?

Aside to the aside – Matlab also knows about the imaginary numbers `i`, and `j`
(so don't use them for loop indices [you should not be using loops if you can help it in Matlab anyway!]).

Working with all the Entries in Specified Dimensions

To manipulate values in some specific dimensions, use the " : " operator to specify the dimensions.

A " : " by itself indicates all elements of that index position (usually rows or columns)

```
>>a(:,1)
```

Means "all rows, in column 1"

```
>>a(1,:)
```

Means "all columns, in row 1"

Suppose we have the 4-d matrix below

```
>> b=[1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
>> b4d=reshape(b,2,2,2,2)
b4d(:,:,1,1) =
     1     9
     5    13
b4d(:,:,2,1) =
     2    10
     6    14
b4d(:,:,1,2) =
     3    11
     7    15
b4d(:,:,2,2) =
     4    12
     8    16
```

In the print out of the array the colon represents the full range of the index/indices represented by the colon and not shown explicitly

Replacing single index with a colon– runs over that index.

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
>> reshape(b,2,2,2,2)
ans(:,:,1,1) =
     1     9
     5    13
ans(:,:,2,1) =
     2    10
     6    14
ans(:,:,1,2) =
     3    11
     7    15
ans(:,:,2,2) =
     4    12
     8    16
```

```
>> b4d(:,1,1,1)
ans =
     1
     5
>> b4d(1,:,1,1)
ans =
     1     9
>>
```

```
>> b(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b4d(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

How the indices vary in the 4-d array.

>> b(:)	>> b4d(:)	
ans =	ans =	
1	1	b4d(1,1,1,1)
5	5	b4d(2,1,1,1)
9	9	b4d(1,2,1,1)
13	13	b4d(2,2,1,1)
2	2	b4d(1,1,2,1)
6	6	b4d(2,1,2,1)
10	10	b4d(1,2,2,1)
14	14	b4d(2,2,2,1)
3	3	b4d(1,1,1,2)
7	7	b4d(2,1,1,2)
11	11	b4d(1,2,1,2)
15	15	b4d(2,2,1,2)
4	4	b4d(1,1,2,2)
8	8	b4d(2,1,2,2)
12	12	b4d(1,2,2,2)
16	16	b4d(2,2,2,2)

`k:l` - Refers to range of values for indices (portions) of a matrix

```
>> k=2;
>> l=3;
>> a(k:l,1)
```

'rows 2 through 3, in column 1'

Same as

```
>> a(2:3,1)
```

`k:l:n` - range of values, in steps of index. Can also do over multiple indices

```
>> a=1:64;
>> a=reshape(a,4,4,4)
a(:,:,1) =
     1     5     9    13
     2     6    10    14
     3     7    11    15
     4     8    12    16
a(:,:,2) =
    17    21    25    29
    18    22    26    30
    19    23    27    31
    20    24    28    32
a(:,:,3) =
    33    37    41    45
    34    38    42    46
    35    39    43    47
    36    40    44    48
a(:,:,4) =
    49    53    57    61
    50    54    58    62
    51    55    59    63
    52    56    60    64
>> a(2:3,3:4,1,1)
ans =
    10    14
    11    15
>> a(2:3,4:-1:3,1,1)
ans =
    14    10
    15    11
```

Can be pretty tricky.

For example, suppose I want to perform a left shift on the values in the second dimension of my 3-D array.

Let me first create an array for illustration.

```
>> a3 = zeros(2,3,2);
>> a3(:) = 1:numel(a3)
a3(:,:,1) =
     1     3     5
     2     4     6
a3(:,:,2) =
     7     9    11
     8    10    12
```

```
>> a3 = zeros(2,3,2);
>> a3(:) = 1:numel(a3)
a3(:,:,1) =
     1     3     5
     2     4     6
a3(:,:,2) =
     7     9    11
     8    10    12
```

Now shift all columns one to the left, and have the `left`-most column "wrap" to become the right most column. Columns are index 2. Here's a way (there are others) to do it.

```
>> a3r1 = a3(:,[2:size(a3,2) 1],:)
a3r1(:,:,1) =
     3     5     1
     4     6     2
a3r1(:,:,2) =
     9    11     7
    10    12     8
```

For all rows, put columns 2 to end (get from 2nd element of size – the middle dimension), then column 1, for all "planes" (2-d matrices in 3rd dimension).

```
>> a3r1 = a3(:,[2:size(a3,2) 1],:)
a3r1(:,:,1) =
     3     5     1
     4     6     2
a3r1(:,:,2) =
     9    11     7
    10    12     8
```

Notice new way to refer to values of an index – using an array.

Collapsing Trailing Dimensions

Suppose we have the following 4-d array.

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
>> b(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

The reshape does not change the order of things in memory – just renames them (actually copies in same order)

```
>> b4d=reshape(b,2,2,2,2)
b4d(:,:,1,1) =
     1     9
     5    13
b4d(:,:,2,1) =
     2    10
     6    14
b4d(:,:,1,2) =
     3    11
     7    15
b4d(:,:,2,2) =
     4    12
     8    16
```

```
>> b4d(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b=[1:4; 5:8; 9:12; 13:16]
…
>> b4d=reshape(b,2,2,2,2)
b4d(:,:,1,1) =
     1     9
     5    13
…
>> b4d(1,1,1,1)
ans =
     1
>> b4d(1,2,1,1)
ans =
     9
>> b4d(2,1,1,1)
ans =
     5
>> b4d(2,2,1,1)
ans =
    13
>>
```

Match up elements here with those in `b4d(:,:,1,1)` above

```
>> b(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b4d(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

Colon

- When used at the end of a list it "compresses" all the remaining indices into a single index (indexed linearly as in memory – by a single subscript). This is called "collapsing" trailing dimensions.

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16

>> b4d(1,1,1,:)
ans(:,:,1,1) =
     1
ans(:,:,1,2) =
     3

>> b4d(1,1,:)
ans(:,:,1) =
     1
ans(:,:,2) =
     2
ans(:,:,3) =
     3
ans(:,:,4) =
    4

>> b4d(1,:)
ans =
     1     3     5     7     9    11    13    15

>> b4d(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
>> b4d(1,1,1,:)
ans(1,1,1,1) =
     1
ans(1,1,1,2) =
     9
>> b4d(1,1,:)
ans(1,1,1) =
     1
ans(1,1,2) =
     5
ans(1,1,3) =
     9
ans(1,1,4) =
    13
>> b4d(1,:)
ans =
     1     3     5     7     9    11    13    15
```

```
>> b4d(1,1,1,:)
ans(:,:,1,1) =
     1
ans(:,:,1,2) =
     9
>> b4d(1,1,:)
ans(:,:,1) =
     1
ans(:,:,2) =
     5
ans(:,:,3) =
     9
ans(:,:,4) =
    13
```

Colons in print out here represent the 1,1. Not sure why Matlab does not print out values of index.

```
>> b4(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
>> b4d(1,1,:)
ans(:,:,1) =
     1
ans(:,:,2) =
     2
ans(:,:,3) =
     3
ans(:,:,4) =
     4
>> b4d(2,1,:)
ans(:,:,1) =
     5
ans(:,:,2) =
     6
ans(:,:,3) =
     7
ans(:,:,4) =
     8
>>
```

```
>> b4(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

(these are the elements

1,1,1,1 1,1,2,1

1,1,1,2 1,1,2,2

a total of 4 elements)

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
>> b4d(1,1,:)
ans(1,1,1) =
     1
ans(1,1,2) =
     2
ans(1,1,3) =
     3
ans(1,1,4) =
     4
>> b4d(2,1,:)
ans(2,1,1) =
     5
ans(2,1,2) =
     6
ans(2,1,3) =
     7
ans(2,1,4) =
     8
>>
```

```
>> b4(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b4d(1,1,1,1)
ans =
     1
>> b4d(1,1,2,1)
ans
     2
>> b4d(1,1,1,2)
ans =
     3
>> b4d(1,1,2,2)
ans =
     4
>>
```

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
>> b4d(1,:)
ans =
     1     9     2    10     3    11     4    12
>> b4d(2,:)
ans =
     5    13     6    14     7    15     8    16
>> b4d(:,:)
ans =
     1     9     2    10     3    11     4    12
     5    13     6    14     7    15     8    16
```

```
>> b=[1:4; 5:8; 9:12; 13:16]
b =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
>> b4d(:,:,:)
ans(:,:,1) =
     1     9
     5    13
ans(:,:,2) =
     2    10
     6    14
ans(:,:,3) =
     3    11
     7    15
ans(:,:,4) =
     4    12
     8    16
>>
```

Same output as b4d.

```
>> b=[1:4; 5:8; 9:12; 13:16]
>> reshape(b,2,2,2,2)
ans(:,:,1,1) =
     1     9
     5    13
ans(:,:,2,1) =
     2    10
     6    14
ans(:,:,1,2) =
     3    11
     7    15
ans(:,:,2,2) =
     4    12
     8    16
```

```
>> b4d(:,:,:,1)
ans(:,:,1) =
     1     9
     5    13
ans(:,:,2) =
     2    10
     6    14
>> b4d(:,:,1)
ans =
     1     9
     5    13
>> b4d(:,1)
ans =
     1
     5
>>
```

When compress in front – works differently - but still iterates over all values of some index.

```
>> b(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b4d(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b=[1:4; 5:8; 9:12; 13:16]
>> reshape(b,2,2,2,2)
ans(:,:,1,1) =
     1     9
     5    13
ans(:,:,2,1) =
     2    10
     6    14
ans(:,:,1,2) =
     3    11
     7    15
ans(:,:,2,2) =
     4    12
     8    16
```

```
>> b4d(:,:,:,1)
ans(:,:,1) =
     1     9
     5    13
ans(:,:,2) =
     2    10
     6    14
>> b4d(:,:,1)
ans =
     1     9
     5    13
>> b4d(:,1)
ans =
     1
     5
>>
```

Takes some head scratching to figure out. (the final index with a number value compresses missing indices.

In first example final index can be 1 or 2, in second can be 1-4, in third can be 1-8.

Probably dangerous.

```
>> b(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b4d(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b=[1:4; 5:8; 9:12; 13:16]
>> reshape(b,2,2,2,2)
ans(:,:,1,1) =
     1     9
     5    13
ans(:,:,2,1) =
     2    10
     6    14
ans(:,:,1,2) =
     3    11
     7    15
ans(:,:,2,2) =
     4    12
     8    16
```

```
>> b4d(1,:,1)
ans =
     1     9
>> b4d(1,1,:,1)
ans(:,:,1) =
     1
ans(:,:,2) =
     2
>> b4d(1,:,1,1)
ans =
     1     3
>>
```

When compress in middle – works differently - but still iterates over all values of some index.

Again, takes some head scratching to figure out. (last index takes 4 values in first ex, 2 in third and fourth ex)

Probably dangerous.

```
>> b(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b4d(:)
ans =
     1
     5
     9
    13
     2
     6
    10
    14
     3
     7
    11
    15
     4
     8
    12
    16
```

```
>> b4d=reshape(b,2,2,2,2)
b4d(:,:,1,1) =
     1     9
     5    13
b4d(:,:,2,1) =
     2    10
     6    14
b4d(:,:,1,2) =
     3    11
     7    15
b4d(:,:,2,2) =
     4    12
     8    16
```

```
>> b4d(1,1,:,:)
ans(:,:,1,1) =
     1
ans(:,:,2,1) =
     5
ans(:,:,1,2) =
     9
ans(:,:,2,2) =
    13
```

```
>> b4d(1,1,:)
ans(:,:,1) =
     1
ans(:,:,2) =
     5
ans(:,:,3) =
     9
ans(:,:,4) =
    13
>>
```

These colons mean/do different things – one set of them is output by matlab (blue) and you tuype the other set (black)

These compressions are equivalent

```
>> b4d=reshape(b,2,2,2,2)
b4d(:,:,1,1) =
     1     9
     5    13
b4d(:,:,2,1) =
     2    10
     6    14
b4d(:,:,1,2) =
     3    11
     7    15
b4d(:,:,2,2) =
     4    12
     8    16
```

```
>> b4d(1,:,:,1)
ans(:,:,1) =
     1     9
ans(:,:,2) =
     2    10
>> b4d(:,1,1,:)
ans(:,:,1,1) =
     1
     5
ans(:,:,1,2) =
     3
     7
>> b4d(1,:,1,:)
ans(:,:,1,1) =
     1     9
ans(:,:,1,2) =
     3    11
>> b4d(:,1,:,1)
ans(:,:,1) =
     1
     5
ans(:,:,2) =
     2
     6
>>
```

Get 4 elements back on each reference with two colons. Two, two element, row or column vectors.

Creating a column vector from another vector or matrix. (note first example would usually be done using transpose operator `at=a'`, but not second since start with matrix and end up with vector).

```
>> a=[1 2 3 4]
a =
     1     2     3     4
>> at=a(:)
at =
     1
     2
     3
     4
```

```
>> a22=[1 2; 3 4]
a22 =
     1     2
     3     4
>> a22c=a22(:)
a22c =
     1
     3
     2
     4
>>
```

Retaining Array Shape During Assignment – colon operator on left side of the equals sign "pours" value on RHS into elements defined on LHS.

```
>> b4d
b4d(:,:,1,1) =
     1     9
     5    13
b4d(:,:,2,1) =
     2    10
     6    14
b4d(:,:,1,2) =
     3    11
     7    15
b4d(:,:,2,2) =
     4    12
     8    16
>> b4d(2,:,:,2)
ans(:,:,1) =
     7    15
ans(:,:,2) =
     8    16
```

```
>> b4d(2,:,:,2)=21
b4d(:,:,1,1) =
     1     9
     5    13
b4d(:,:,2,1) =
     2    10
     6    14
b4d(:,:,1,2) =
     3    11
    21    21
b4d(:,:,2,2) =
     4    12
    21    21
>>
```

Concatenation

You can concatenate using the square brackets, [ ] (same as making a matrix, but using other matrices as the elements)

```
>>B = [A   A+32;   A+48          A+16]
B =
16  3   2   13  48  35  34  45
5   10  11  8   37  42  43  40
9   6   7   12  41  38  39  44
4   15  14  1   36  47  46  33
64  51  50  61  32  19  18  29
53  58  59  56  21  26  27  24
57  54  55  60  25  22  23  28
52  63  62  49  20  31  30  17
```

Deleting rows and columns

You can also combine **:** with **[ ]** to remove rows, columns, or elements (again – variation on theme of assigning elements in a matrix – have a syntax rule and read it like a lawyer for all possible interpretations and implications.)

e.g. Remove the second column

```
>>X=A;
>>X(:,2) = [];
```

Create vector from X; removes every 2^{nd} element from 2 to 10

```
>>X(2:2:10) = []
X =
16  9   2   7   13  12  1
```

Done with the colon operator for now.

But will continue to show up in examples.

Variables

MATLAB does not require any type declarations

(actually all variables are double precision floating point – you can declare them to be other things if needed – however many/most Matlab routines [such at FFT, filtering, etc.] will not work with anything other than double precision floating point data)

or dimension statements.

Variables

When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage.

If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage.

MATLAB is case sensitive. ("A" is not the same as "a")

Matlab

Arithmetic operations

Add/Subtract: Adds/subtracts vectors (element by element) (=> the two vectors have to be the same length).

```
>> x=[1 2];
>> y=[1 3];
>> z=x+y
z =
     2     5
>> whos
  Name      Size            Bytes  Class     Attributes
  x         1x2                16  double
  y         1x2                16  double
  z         1x2                16  double
```

Knows about complex numbers.

```
>> x=1+i;
>> y=2+2i;
>> z=x+y
z =
   3.0000 + 3.0000i
>> whos
  Name      Size            Bytes  Class     Attributes
  x         1x1                16  double    complex
  y         1x1                16  double    complex
  z         1x1                16  double    complex
>>
```

But - can add a scalar (1x1) array to every element of matrix.

```
>> x=[1 2];
>> y=1;
>> x+y
ans =
     2     3
>> whos
  Name      Size            Bytes  Class     Attributes
  ans       1x2                16  double
  x         1x2                16  double
  y         1x1                 8  double
>>
```

Multiply
Now things get interesting

Scalar*vector

```
>> x=[1 2];
>> y=3;
>> z=y*x
z =
     3     6
>> x=[1+i 2-i];
>> y=1-i;
>> z=y*x
z =
   2.0000                  1.0000 - 3.0000i
>>
```

Matlab knows how to do it. You just write what looks like math. (No looping to do element by element multiplies, does complex math)

Multiply
Vector * Vector

Now have some choices

Apostrophe is transpose if needed to get sizes correct.

```
>> x=[1 2];
>> y=[3 4];
>> z=x*y'
z =
   11
```

Regular matrix multiplication – in this case with vectors 1x2 * 2x1 = 1x1 => dot product

```
>> w=x.*y
w =
    3     8
```

Element by element multiplication (the ".")

```
>> z=x'*y
z =
    3     4
    6     8
>>
```

Regular matrix multiplication – in this case with vectors 2x1 * 1x2 = 2x2 matrix

A little more complicated for complex valued matrices.

```
>> a=[1-i 2-i;3-i 4-i]
a =
   1.0000 - 1.0000i   2.0000 - 1.0000i
   3.0000 - 1.0000i   4.0000 - 1.0000i
>> a'
ans =
   1.0000 + 1.0000i   3.0000 + 1.0000i
   2.0000 + 1.0000i   4.0000 + 1.0000i
```

Complex conjugate transpose (Hermitian)

```
>> a.'
ans =
   1.0000 - 1.0000i   3.0000 - 1.0000i
   2.0000 - 1.0000i   4.0000 - 1.0000i
```

Non-complex conjugate transpose

```
>> ctranspose(a)
ans =
   1.0000 + 1.0000i   3.0000 + 1.0000i
   2.0000 + 1.0000i   4.0000 + 1.0000i
>>
```

Dot and Cross products

(using this form – built in functions - don't have to match dimensions of vectors in the sense that you can mix column and row vectors – although they have to be the same length)

```
>> a=[1 2 3];
>> b=[4 5 6];
>> c=dot(a,b)
c =
    32
>> d=dot(a,b')
d =
    32
>> e=cross(a,b)
e =
   -3     6    -3
>> f=cross(a,b')
f =
    -3     6    -3
>> g=cross(b,a)
g =
     3    -6     3
>>
```

Dot products

For matrices – does dot product of each column.
The matrices have to be the same size.

```
>> a=[1 2;3 4]
a =
     1     2
     3     4
>> b=[5 6;7 8]
b =
     5     6
     7     8
>> dot(a,b)
ans =
    26    44
>>
```

Should try to write functions so they behave like this – if give it "vector" does it to every element. Here the 2-d matrix is a vector of vectors.

Cross products

For matrix – does cross product of columns. (one of the dimensions has to be 3 and takes other dimension as additional vectors)

```
>> a=[1 2;3 4;5 6]
a =
     1     2
     3     4
     5     6
>> b=[7 8;9 10;11 12]
b =
     7     8
     9    10
    11    12
>> cross(a,b)
ans =
   -12   -12
    24    24
   -12   -12
```

Cross products

```
>> a=[1 3 5]
>> b=[7 9 11]
>> cross(a,b)
ans =
   -12    24   -12
>> a=[2 4 6]
>> b=[8 10 12]
>> cross(a,b)
ans =
   -12    24   -12
>> cross(a',b')
ans =
   -12
    24
   -12
>> cross(a',b)
ans =
   -12    24   -12
>> cross(a,b’)
ans =
   -12    24   -12
>>
```

Output can be row or column vector

Array and Matrix divide
Even more fun

Element by element divide (the ".").

```
>> a=[1 2;3 4]
a =
     1     2
     3     4
>> b=[2 4;6 8]
b=
     2     4
     6     8
>> a./b
ans =
    0.5000    0.5000
    0.5000    0.5000
```

Right <u>array</u> divide.

```
>> a.\b
ans =
     2     2
     2     2
```

Left <u>matrix</u> divide

```
>> b./a
ans =
     2     2
     2     2
>> b.\a
ans =
    0.5000    0.5000
    0.5000    0.5000
>>
```

Matrix on top is dividend.
Matrix on bottom is divisor.

Array and Matrix divide

```
>> a=[1 2;3 4]
a =
     1     2
     3     4
>> det(a)
ans =

    -2
>> b=[5 6]
b =
     5     6
>> c=a*b'
c =
    17
    39
>> d=a\c
d =
    5.0000
    6.0000
>>
```

Left matrix division.

Dividing a into c.

This is equivalent to inv(a)*c=b.

Note this is the solution to a*b=c.

Sizes have to be appropriate.

With a matrix for b, get solutions for each column b'.

(we needed the b' when b was a vector to get things to multiply correctly – to get the same values we have to transpose b also)

```
>> b=[5 6;7 8]
b =
    5     6
    7     8
>> c=a*b'
c =
   17    23
   39    53
>> d=a\c
d =
   5.0000    7.0000
   6.0000    8.0000
>>
```

`mldivide(A,B)` and the equivalent `A\B` perform matrix left division (back slash).

`A` and `B` must be matrices that have the same number of rows, unless `A` is a scalar, in which case `A\B` performs element-wise division — that is, `A\B = A.\B`.

`mldivide(A,B)` and the equivalent `A\B` perform matrix left division (back slash).

If `A` is a square matrix, `A\B` is roughly the same as `inv(A)*B`, except it is computed in a different way.

If `A` is an `n-by-n` matrix and `B` is a column vector with `n` elements, or a matrix with several such columns, then

```
X = A\B
```

is the solution to the equation `AX = B`.

A warning message is displayed if `A` is badly scaled or nearly singular.

`mldivide(A,B)` and the equivalent `A\B` perform <u>matrix</u> left division (back slash).

If `A` is an `m-by-n` matrix with `m ~= n` and `B` is a column vector with `m` components, or a matrix with several such columns, then

`X = A\B`

is the solution in the least squares sense to the under- or overdetermined system of equations `AX = B`.

`mldivide(A,B)` and the equivalent `A\B` perform <u>matrix</u> left division (back slash).

In other words, `X` minimizes

`norm(A*X - B)`,

the length of the vector `AX - B`.

The rank `k` of `A` is determined from the QR decomposition with column pivoting.
The computed solution `X` has at most `k` nonzero elements per column. If `k < n`, this is usually <u>not</u> the same solution as

`x = pinv(A)*B`,

which returns a least squares solution.

`mrdivide(B,A)` and the equivalent `B/A` perform <u>matrix</u> right division (forward slash).

`B` and `A` must have the same number of columns.

`mrdivide(B,A)` and the equivalent `B/A` perform matrix right division (forward slash).

If `A` is a square matrix, `B/A` is roughly the same as `B*inv(A)`.

If `A` is an `n-by-n` matrix and `B` is a row vector with n elements, or a matrix with several such rows, then

`X = B/A`

is the solution to the equation `XA = B` computed by Gaussian elimination with partial pivoting.

`mrdivide(B,A)` and the equivalent `B/A` perform matrix right division (forward slash).

A warning message is displayed if `A` is badly scaled or nearly singular.

`mrdivide(B,A)` and the equivalent `B/A` perform matrix right division (forward slash).

If `B` is an `m-by-n matrix` with `m ~= n` and `A` is a column vector with `m` components, or a matrix with several such columns, then

`X = B/A`

is the solution in the least squares sense to the under- or overdetermined system of equation

`XA = B`.

Note: matrix right division and matrix left division are related by the equation

```
B/A = (A'\B')'.
```

Example 1- Suppose A and B are -

```
>> A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
>> b = [1;2;3]
b =
     1
     2
     3
```

To solve the matrix equation **Ax = b,** enter

```
>> x=A\b
x =
    0.0500
    0.3000
    0.0500
```

You can verify **x** is the solution to the equation as follows.

```
>> A*x
ans =
    1.0000
    2.0000
    3.0000
```

Magic matrix – square matrix with property that column, row and diagonal sums add to same value.

```
>> tst=magic(3)
tst =
     8     1     6
     3     5     7
     4     9     2
>> sum(tst)
ans =
    15    15    15
>> sum(tst’)
ans =
    15    15    15
>> sum(sum(tst.*eye(3)))
ans =
    15
>> sum(sum(tst'.*eye(3)))
ans =
    15
>>
```

Example 2 — `A` Singular

If `A` is singular, `A\b` returns the following warning.

Warning: Matrix is singular to working precision.

In this case, `Ax = b` might not have a solution.

Example 2 — A Singular

```
A = magic(5);
A(:,1) = zeros(1,5); % Set column 1 of A to zeros
b = [1;2;5;7;7];
x = A\b
Warning: Matrix is singular to working precision.
ans =
    NaN
    NaN
    NaN
    NaN
    NaN
```

If you get this warning, you can still attempt to solve `Ax = b` using the pseudoinverse function `pinv`.

Example 2 — A Singular

If you get this warning, you can still attempt to solve `Ax = b` using the pseudoinverse function pinv.

```
x = pinv(A)*b
x =
0 0.0209
0.2717
0.0808
-0.0321
```

The result x is least squares solution to `Ax = b.`

Example 2 — A Singular

To determine whether x is a exact solution

— i.e., a solution for which Ax - b = 0 —

simply compute

```
A*x-b
ans =
   -0.0603
    0.6246
   -0.4320
    0.0141
    0.0415
```

The answer is not the zero vector, so x is not an exact solution.

Example

Suppose that

```
A = [1 0 0;1 0 0];
b = [1; 2];
```

Note **Ax = b** cannot have a solution, because **A*x** has equal entries for any **x**. Entering

```
x = A\b
```

returns the least squares solution

```
x =
1.5000
0
0
```

along with a warning that **A** is rank deficient.

Example

```
A = [1 0 0;1 0 0];

b = [1; 2];

x = A\b

x =
1.5000
0
0
```

Note that **x** is not an exact solution:

```
A*x-b
ans =
0.5000
-0.500
```

Raising array to power

```
>> a=[1 2;3 4]
a =
    1    2
    3    4
>> a^2
ans =
    7   10
   15   22
>> a*a
ans =
    7   10
   15   22
>> a.^2
ans =
    1    4
    9   16
>>
```

Array exponentiation and multiplication

Element by element exponentiation.

Operators

Arithmetic operators.

plus	- Plus	+
uplus	- Unary plus	+
minus	- Minus	-
uminus	- Unary minus	-
mtimes	- Matrix multiply	*
times	- Array (element by element) multiply)	.*
mpower	- Matrix power	^
power	- Array (element by element) power	.^
mldivide	- Backslash or left matrix divide	\
mrdivide	- Slash or right matrix divide	/
ldivide	- Left array (element by element) divide	.\
rdivide	- Right array (element by element) divide	./
kron	- Kronecker tensor product	kron

```
>> help kron
 KRON   Kronecker tensor product.
    KRON(X,Y) is the Kronecker tensor product of X and Y.
    The result is a large matrix formed by taking all possible
    products between the elements of X and those of Y.   For
    example, if X is 2 by 3, then KRON(X,Y) is

       [ X(1,1)*Y  X(1,2)*Y  X(1,3)*Y
         X(2,1)*Y  X(2,2)*Y  X(2,3)*Y ]

    If either X or Y is sparse, only nonzero elements are
multiplied
    in the computation, and the result is sparse.

    Class support for inputs X,Y:
       float: double, single

    Reference page in Help browser
       doc kron
```

```
>> x=[1 2 3;4 5 6]
x =
     1     2     3
     4     5     6
>> y=[7 8;9 10]
>> y=[7 8]
y =
     7     8
>> kron(x,y')
ans =
     7    14    21          = ( 1 2 3)*7
     8    16    24          = ( 1 2 3)*8
    28    35    42          = ( 4 5 6)*7
    32    40    48          = ( 4 5 6)*8
>>
>> kron(x,y)
ans =
     7     8    14    16    21    24
    28    32    35    40    42    48
```

(1 4)*7
(1 4)*8
(2 5)*7
(2 5)*8
(3 6)*7
(3 6)*8

Operators

Relational operators.

eq - Equal ==

ne - Not equal ~=

lt - Less than <

gt - Greater than >

le - Less than or equal <=

ge - Greater than or equal >=

Logical operators.

and - Logical AND &

or - Logical OR |

not - Logical NOT ~

xor - Logical EXCLUSIVE OR

any - True if any element of vector is nonzero

all - True if all elements of vector are nonzero

Exclusive or

```
>> a=[0 0 1 1]
>> b=[0 1 0 1]
>> xor(a,b)
ans =
     0     1     1     0
>>
```

Matlab

Matrix Maniputlation

A few things to remember:

- Cannot use spaces in names of matrices (variables, everything in matlab is a matrix)

```
~~cool x = [1 2 3 4 5]~~
```

- Cannot use the dash sign (-) because it represents a subtraction.

```
~~cool-x = [1 2 3 4 5]~~
```

- Don't use a period (.) unless you want to create something call a *structure*.

```
~~cool.x = [1 2 3 4 5]~~
```

A few things to remember:

- Your best option, is to use the underscore (_) if you need to assign a long name to a matrix

```
my_cool_x = [1 2 3 4 5]
```

Changing and adding elements in existing matrix:

```
>> a=[1 2 3]
a =
     1     2     3
>> a(1,2)=4
a =
     1     4     3
>> a(2,4)=5
a =
     1     4     3     0
     0     0     0     5
>>
```

Sizes of matrices:

```
a =
     1     4     3     0
     0     0     0     5
>> size(a)
ans =
     2     4
>> sizea=size(a);
>> whos
  Name          Size              Bytes  Class     Attributes

  a             2x4                  64  double
  ans           1x2                  16  double
  sizea         1x2                  16  double
>> sizea
sizea =
     2     4
>> size(a,1)
ans =
     2
>> size(a,2)
ans =
     4
```

Dimension of matrix (mathematically) – rows, columns

Can do by individual dimensions

Sizes of matrices:

```
>> length(a(:))
ans =
     8
>> x=[1 2; 3 4; 5 6; 7 8]
x =
     1     2
     3     4
     5     6
     7     8
>> length(x)
ans =
     4
>> x=[1 2 3 4;5 6 7 8];
>> length(x)
ans =
     4
>>
```

Linear size (as vector – total number elements)

Length of matrix gives the max dimension)

Building matrices from other matrices: (have to match dimensions)

```
>> a=[1 2; 3 4]
a =
     1     2
     3     4
>> b=[1 2]
b =
     1     2
>> c=[a b']
c =
     1     2     1
     3     4     2
>> d=[a;b]
d =
     1     2
     3     4
     1     2
>>
```

Some predefined matrix making tools:

```
>> rand(3)
ans =
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
>> rand(1,3)
ans =
    0.9649    0.1576    0.9706
>> rand(3,1)
ans =
    0.9572
    0.4854
    0.8003
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
>>
```

Also – `ones, zeros, magic, hilb`

Aside:

Some predefined values:

`pi`

`i, j`

`eps`

To see what variables are defined

who, who vari_name

To clear variables

clear vari_name, clear (does all of them)

Functions:

Many of them.
Here are a few -

In general these functions work on vectors (for vectors does not matter if row or column), or columns for matrix input (matrix treated as group column vectors)

```
max
min
sum
cumsum
mean
abs
```

Functions:

Work element by element on vector/matrix (if appropriate)

```
sin
cos
(Other trig and inverse fns)
exp
log
abs
...
```

Functions:

Perform matrix operations

(output can be same size matrix, different size matrix or matrices, scalar, other.)

```
inv
eig
triu
tril
...
```

(not "vectorized")

Round/truncate

```
round(f)
fix(f)
ceil(f)
floor(f)

>> help round
 ROUND  Round towards nearest integer.
    ROUND(X) rounds the elements of X to the nearest integers.
>> help fix
 FIX    Round towards zero.
    FIX(X) rounds the elements of X to the nearest integers
    towards zero.
>> help ceil
 CEIL   Round towards plus infinity.
    CEIL(X) rounds the elements of X to the nearest integers
    towards infinity.
>> help floor
 FLOOR  Round towards minus infinity.
    FLOOR(X) rounds the elements of X to the nearest integers
    towards minus infinity.
>>
```

Logical operations on matrix: (test is element by element) Returns a logical matrix

```
>> a=[1 2 3 4 5]
a =
     1     2     3     4     5
>> b=[5 4 3 2 1]
b =
     5     4     3     2     1
>> a==b
ans =
     0     0     1     0     0
>>
```

`==,  >,  >=,  <,  <=,  ~,  &,  |`

`any(a)` determines if matrix `a` has at least one nonzero entry.

`all(a)` determines if all the entries of matrix `a` are nonzero,.

"Vectorizing"

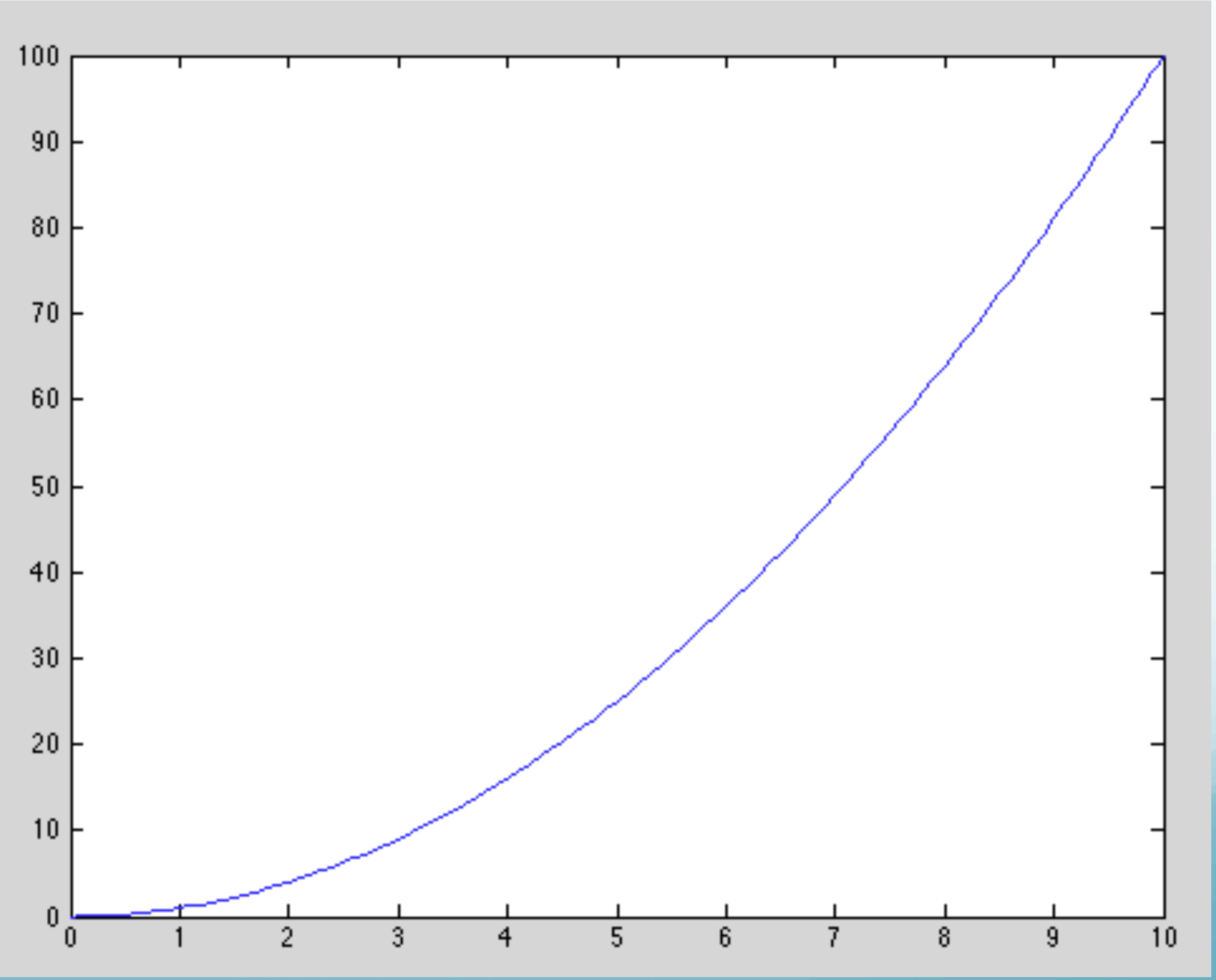
Putting what we have together and doing things without loops.

Say I want to plot the function x^2

The traditional way is to use a loop to generate a sequence of values for x and x^2.

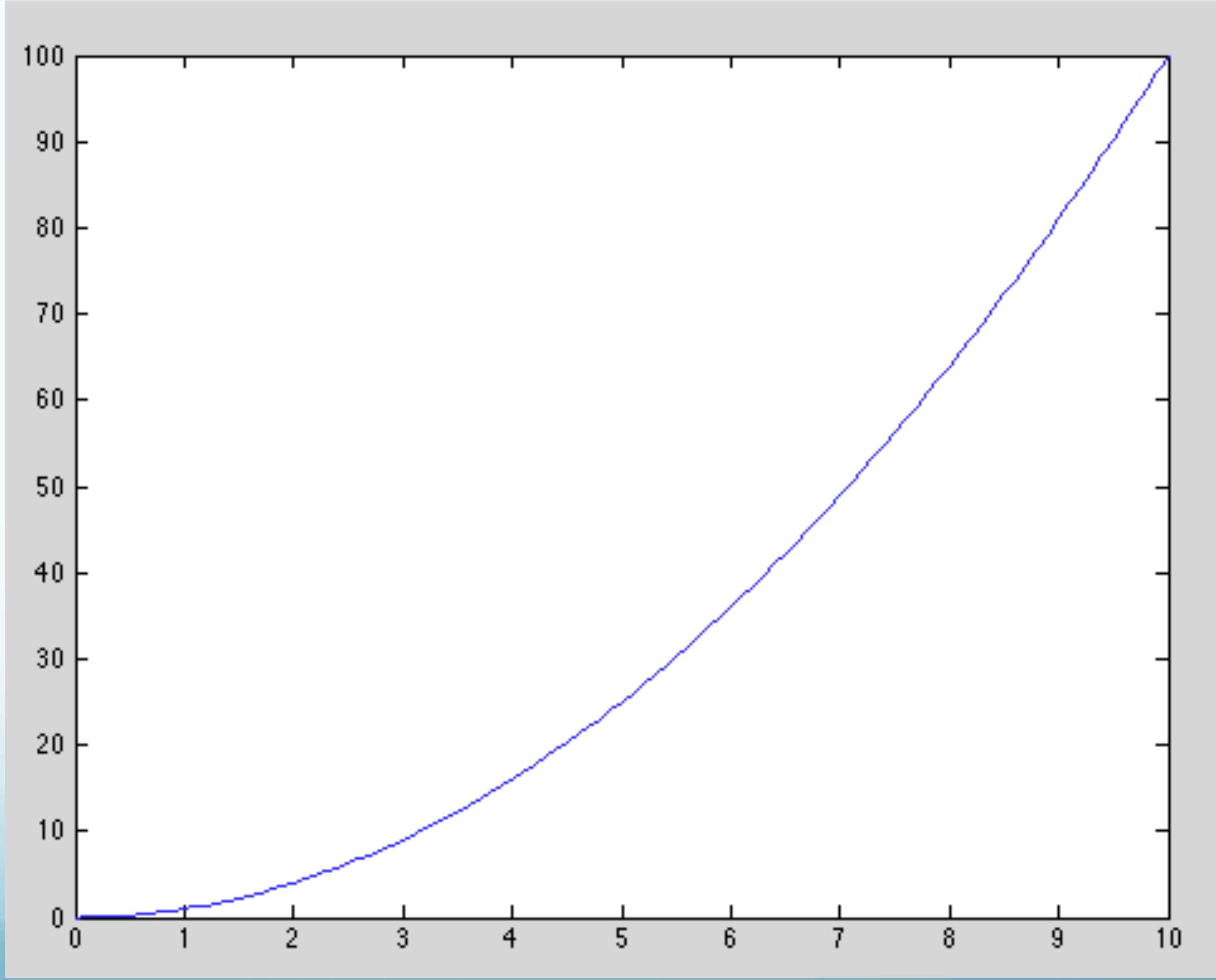
But Matlab gives us an easier way to calculate the whole shooting match in one statement.

```
>> x=1:.1:10;
>> y=x.*x;
>> plot(x,y)
```



Matlab uses geometrical view of complex numbers (x = real axis, y = imaginary axis) ~ `z=x+iy`.

```
>> x=0:.1:1;
>> z=x+i*x.^2;
>> plot(z)
```

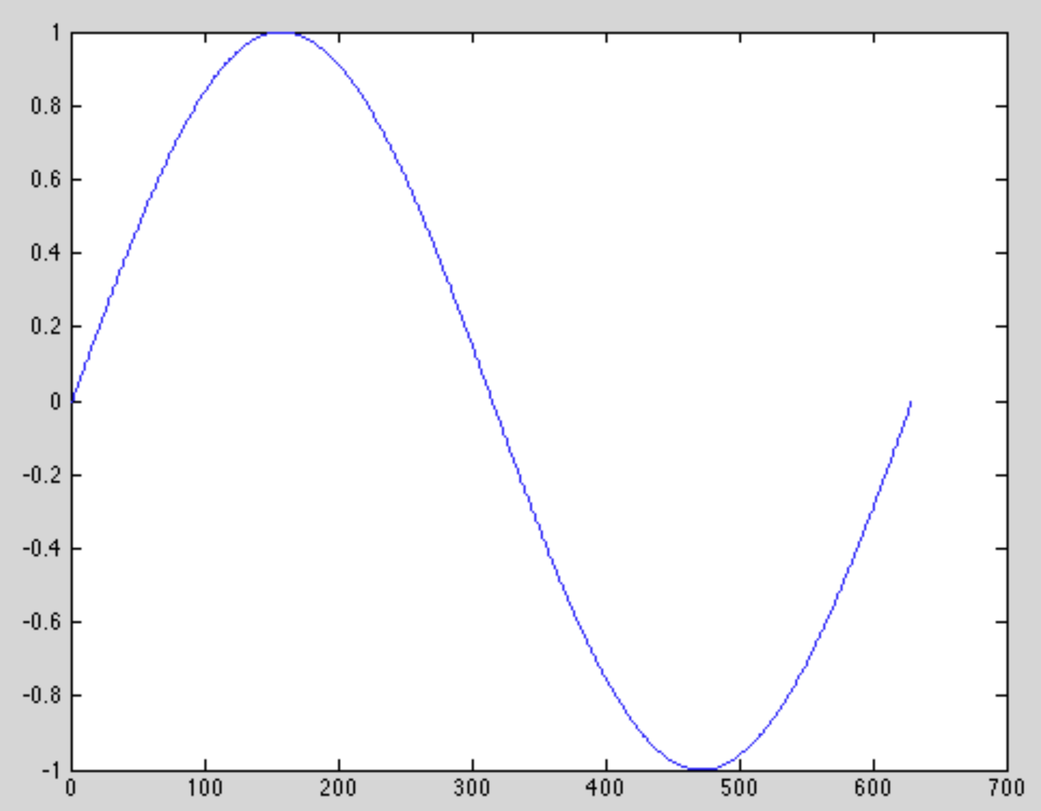


More examples.

```
>> y=sin(0:.01:2*pi);
>> plot(y)
```

Could also do in one line

```
>> plot(sin(0:.01:2*pi))
```



More examples.

```
>> x=0:.01:2*pi;
>> y=sin(x);
>> whos
  Name          Size            Bytes  Class     Attributes
  x             1x629            5032  double
  y             1x629            5032  double
>> plot(x,y)
```

If want actual **x** argument values (radians)