Arithmetic operations

MATLAB

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

```
>> x=[1 2];
>> y=[1 3];
>> whos
  Name      Size            Bytes  Class     Attributes
  x         1x2                16  double
  y         1x2                16  double
>> z=x+y
z =
     2     5
>> whos
  Name      Size            Bytes  Class     Attributes
  x         1x2                16  double
  y         1x2                16  double
  z         1x2                16  double
>> x=1+i;
>> y=2+2i;
>> z=x+y
z =
   3.0000 + 3.0000i
>>
```

But - be careful if not same length will still give result.

```
>> 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
>>
```

Multiply
Vector * Vector

Now have some choices

```
>> 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

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

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

Apostrophe is transpose.

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 – 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 columns. 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
>>
```

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

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

Element by element divide.

Right <u>array</u> divide.

Left <u>matrix</u> divide

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 <u>matrix</u> 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 matrix 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 equations *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 — that is, 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 3
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 3

```
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 multiplication

Element by element.

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    48a
```

(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
>>
```

Matrix Maniputlation

MATLAB

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 created 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)

Can do by individual dimensions

Sizes of matrices:

```
>> length(a)
ans =
     4
>> length(a(:))
ans =
     8
>>
```

Linear size (as vector - amount memory

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 -

How they work is context sensitive.

```
max
min
sum
mean
```

These functions work on vectors, or columns for matrix input (matrix is treated like group of column vectors)

Functions:

Work element by element when appropriate

```
sin
cos
(Other trig fns)
exp
log
abs
…
```

Perform matrix operations

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

```
inv
eig
triu
tril
…
```

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: (is element by element)

```
>> 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 determines if a matrix has at least one nonzero entry.

all determines if all the entries are nonzero,.

Programming

MATLAB

Relational Operators

Returns 1 if true and 0 if false.
(opposite of shell)

All relational operators are left to right associative.

Make element-by-element comparisons.

Relational Operators (review)

< : test for less than

<= : test for less than or equal to

>: test for greater than

>= : test for greater than or equal to

== : test for equal to

~= : test for not equal

Relational Operators with matrices

Relational operators may not behave like you think with matrices, so be careful.

Some useful relational operators for matrices include the following commands:

isequal : tests for equality
isempty: tests if an array is empty
all : tests if all elements are nonzero
any: tests if any elements are nonzero; ignores NANs

These return 1 if true and 0 if false

Logical Operators

Logical array operators return 1 for true and 0 for false

Work element-by-element

& : logical AND; tests that both expressions are true

| : logical OR ; tests that one or both of the expressions are true

~ : logical NOT; tests that expression is true

Logical Operators w/ Short-circuiting

If the first tested expression will automatically cause the logical operator to fail, the remainder of the expression is not evaluated.

&& : short-circuit logical AND

|| : short-circuit logical OR

Logical Operators w/ Short-circuiting

(b ~= 0) && (a/b > 18.5)

if (b ~= 0) evaluates to false, MATLAB assumes the entire expression to be false and terminates its evaluation of the expression early.

This avoids the warning that would be generated if MATLAB were to evaluate the operand on the right.

if/elseif/else/end

If expression is true, then run the first set of commands. Else if a second expression is true, run the second set of commands. Else if neither is true, run a third set of commands. End the if command

```
if rem(n,2) ~= 0          %calculates remainder of n./2
   M = odd_magic(n)
elseif rem(n,4) ~= 0    % ~= is 'not equal to' test
   M = single_even_magic(n)
else
   M = double_even_magic(n)
end
```

Often indented for readability.

switch, case, and otherwise

switch executes the statements associated with the first case where *switch_expr* == *case_expr*

If no case expression matches the switch expression, then control passes to the otherwise case (if it exists).

```
switch switch_expr
case case_expr
    statement, ..., statement
otherwise
    statement, ..., statement
end
```

Often indented for readability.

For

one of the most common loop structures is the for loop, which iterates over an array of objects

for x values in array, do this

```
for  m = 1:m
    for n = 1:n
       h(i,j) = 1/(i+j);
    end
end
```

Often indented for readability.

Try to avoid using i and j as loop counters

(matlab uses them for sqrt(-1))

while/end

<u>while:</u> continues to loop as long as condition exited successfully

```
n= 1;
while (1+n) > 1, n= n/2;,  end
n= n*2
```

Note the use of the "," rather than a newline (carriage return) to separate the parts of this loop

(the semicolon ";" is for "silence" - else it prints out n/2 each time through).

This can be done with any type of loop structure.

Break

<u>break</u>: allows you to break out of a for or while loop

exits only from the loop in which it occurs

```
while condition1          # Outer loop
    while condition2      # Inner loop
       break                 # Break out of inner loop only
    end
    …                    # Execution continues here after break
end
```

Often indented for readability.

Continue

continue: pass control to next iteration of for or while loop

passes to the next iteration of the loop in which it occurs

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line,'%',1)
      continue
    end
    count = count + 1;
end
disp(sprintf('%d lines',count));
```

Often indented for readability.

Return

return: returns to invoking function

allows for termination of program before it runs to completion

```
%det(magic)
function d = det(A)
%DET det(A) is the determinant of A.
if isempty(A)
    d = 1;
    return        %exit the function det at this point
else
    …
end
```