

Data Analysis in Geophysics

ESCI 7205

Class 15

Bob Smalley

More Matlab.

Matlab does all arithmetic in double precision.

Matlab "knows" about other types of entities (single precision, integers of varying lengths, unsigned integers, logicals) but converts them to floating point to use them.

(This is somewhat of a disaster when processing topographic data bases for which one square degree of data can be 13 Mega points (3600×3600 points) each 2 bytes long, that turn into 13 Mega points each 8 bytes long for a total of about 100 Mbytes for one square degree worth of data. Considering that there are about $0.3 \times 360 \times 180 \sim 20,000$ (est 70% earth surface is water) square degrees of land. So if you want to process all the topo data that's 2 Terrabytes as double precision, versus about 500 Gibaybtes in raw format)

This combined with fact that Matlab is in general interpreted means that it is not a speed deamon.

So it is important to do whatever you can to make it as fast as possible when using it for heavily used number crunching.

(hint – Vectorize)

```

>> x=1:3
x =
    1    2    3
>> sum(x)
ans =
    6
>> xt=x'
xt =
    1
    2
    3
>> sum(x)
ans =
    6
>> y=[1 2; 4 4]
y =
     1     2
     4     4
>> sum(y)
ans =
     5     6
>> sum(sum(y))
ans =
    11
>>

```

Review Matlab “sum” command with multiple dimension arrays.

Sums elements in vector (row or column) – result is a scalar.

For a matrix, sums elements by column (the order stored in memory) – result is a row vector of the column sums.

To sum whole matrix, call twice (once to sum columns, then second time to sum resulting row vector) – result is a scalar.


```
>> b=[1:16]
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     3
     2     4
```

```
b4d(:, :, 2, 1) =
```

```
     5     7
     6     8
```

```
b4d(:, :, 1, 2) =
```

```
     9    11
    10    12
```

```
b4d(:, :, 2, 2) =
```

```
    13    15
    14    16
```

```
>> sum(b4d(:, :, 1, 1))
```

```
ans
```

```
     3     7
```

```
>> sum(b4d(:, :, 2, 1))
```

```
ans =
```

```
    11    15
```

```
>>
```

Summing parts of the 4-d matrix.

Same as summing on the 2-d matrices.

```

>> b=[1:8]
b =
     1     2     3     4     5     6     7     8
>> b3d=reshape(b,2,2,2)
b3d(:,:,1) =
     1     3
     2     4
b3d(:,:,2) =
     5     7
     6     8
>> sum(b3d(:,:,1))
ans =
     3     7
>> sum(b3d(:,:,2))
ans =
    11    15
>> sum(sum(b3d(:,:,1)))
ans =
    10
>> sum(sum(b3d(:,:,2)))
ans =
    26
>> sum(sum(sum(b3d)))
ans =
    36
>>

```

Easier to see with 3-D
matrix?

```
>> sum(sum(b3d(1, :, :)))  
ans =  
    16  
>> sum(sum(b3d(2, :, :)))  
ans =  
    20  
>> sum(sum(b3d(:, 1, :)))  
ans =  
    14  
>> sum(sum(b3d(:, 2, :)))  
ans =  
    22  
>>
```

Can slice anyway you want.

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

```
1
```

```
ans(:,:,2) =
```

```
5
```

```
ans(:,:,3) =
```

```
9
```

```
ans(:,:,4) =
```

```
13
```

```
>> sum(b4d(1,1,:))
```

```
ans =
```

```
28
```

```
>>
```

Sum ~ adds them.

Formatting screen output

`format` may be used to affect the spacing in the display of all variables as follows:

`format compact` Suppresses extra line-feeds.

`format loose` Puts the extra line-feeds back in (the default).

```
>> pi
```

```
ans =
```

```
3.1416
```

```
>> format compact
```

```
>> pi
```

```
ans =
```

```
3.1416
```

```
>>
```

Formatting screen output

`format short` fixed point with 4 decimal places (the default)

`format long` fixed point with 14 decimal places

`format short e` scientific notation with 4 decimal places

`format long e` scientific notation with 15 decimal places

Accessing and initializing array values
(will use to do more on vectorizing)

Previously I snuck this in

```
>> a3r1 = a3(:, [2:size(a3,2) 1], :)
```

and there was not much of a-do made of it, but
we will now return to it in detail.

What does this do?

We will start by looking at ways to access array elements.

```
>> a=10
```

```
a =  
    10
```

```
>> a
```

```
a =  
    10
```

```
>> whos
```

Name	Size	Bytes	Class	Attributes
a	1x1	8	double	

```
>>
```

So a is a scalar

But everything in Matlab is really a matrix so -

```
>> a(:)
ans =
    10
```

We can list all the elements of a
(there is only 1)

```
>> a(1)
ans =
    10
```

We can address a as a 1-d vector.

```
>> a(1,1)
ans =
    10
```

We can address a as a 2-d (or
higher d) vector as (1, 1) in 2-d is
same memory location as (1) in 1-d,
which is the memory location as the
single element.

```
>> a(1,1,1,1)
ans =
    10
```

```
>>
```

```
>> el=1  
el =
```

```
1  
>> a(el) ← We can also use a variable for the index  
ans =
```

```
10  
>> a([1]) ← Or an array (explicitly) or as a variable  
ans =
```

```
10  
>> array=[1]
```

```
array =
```

```
1  
>> a(array) ←  
ans =
```

```
10
```

```
>>
```

But everything in Matlab is really a matrix so -

```
>> a(1,2)
Index exceeds matrix dimensions.
>> a(2)
Index exceeds matrix dimensions.
>>
```

If we try to address
beyond one element
we get an error
message.

These methods work in general

```
>> a=1:27
```

```
a =
```

```
Columns 1 through 21
```

```
1 2 3 4 5 6 7 8 9 10
```

```
11 12 13 14 15 16 17 18 19 20 21
```

```
Columns 22 through 27
```

```
22 23 24 25 26 27
```

```
>> a3d=reshape(a,3,3,3)
```

```
a3d(:,:,1) =
```

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

```
...
```

```
>> a3d(:,:,1)
```

```
ans =
```

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

```
>> a3d(:,1:2,1)
```

```
ans =
```

```
1 4
2 5
3 6
```

```
>> a3d(:,[1 3],1)
```

```
ans =
```

```
1 7
2 8
3 9
```

```
>> a3d(:,[1 3:-1:2],1)
```

```
ans =
```

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

Specify ranges with :
operator, use arrays w/ and
w/o colon operator.

Look at reshape again

reshape does not change order of elements in memory, just gives another way to index to elements.

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

More on vectorizing

Say we want to take the cross product of each of the columns of a matrix a with the column vector b .

```
>> a=[1 2; 3 4 ; 5 6]
```

```
a =
```

```
1     2
3     4
5     6
```

```
>> b=[1;2;3]
```

```
b =
```

```
1
2
3
```

Another way
to make b

```
>> b=[1:3]'
```

```
b =
```

```
1
2
3
```

More on vectorizing

We could do a loop over the columns of **a**, crossing each with the vector **b**, putting the answer in a new matrix.

(but we don't know how to do loops yet – so we can't do this.)

```
>> a=[1 2; 3 4 ; 5 6]
```

```
a =
```

```
1     2
3     4
5     6
```

```
>> b=[1;2;3]
```

```
b =
```

```
1
2
3
```

Another way to
make **b**

```
>> b=[1:3]'
```

```
b =
```

```
1
2
3
```

Vectorizing

Find a way to do with out a loop.

Can make a matrix `bb` with a copy of `b` in each column, such that we can now do all the cross products with just one call to the cross product.

One way to make the matrix `bb`.

Post multiply column vector by row vector of all ones ($3 \times 1 * 1 \times 2 = 3 \times 2$).

Then do cross product of all pairs of columns with one call.

```
>> a=[1 2; 3 4 ; 5 6]
```

```
a =  
     1     2  
     3     4  
     5     6
```

```
>> b=[1;2;3]
```

```
b =  
     1  
     2  
     3
```

```
>> o=ones(1,2)
```

```
o =  
     1     1
```

```
>> bb=b*o
```

```
bb =  
     1     1  
     2     2  
     3     3
```

```
>> cross(a,bb)
```

```
ans =  
    -1     0  
     2     0  
    -1     0
```

```
>>
```


More on vectorizing

What we've done is correct/OK,
but it is slow due to the
multiplying.

Turns out it is much faster to
simply copy the vector **b** multiple
times, rather than doing the
multiply.

In addition the multiply solution
does not always work (if can't
make the result by multiplication
of a vector/matrix and a matrix)

```
>> a=[1 2; 3 4 ; 5 6]
```

```
a =
```

```
1 2  
3 4  
5 6
```

```
>> b=[1;2;3]
```

```
b =
```

```
1  
2  
3
```

```
>> bb=[b b]
```

```
bb =
```

```
1 1  
2 2  
3 3
```

```
>> cross(a,bb)
```

```
ans =
```

```
-1 0  
2 0  
-1 0
```

```
>>
```

More on vectorizing

The problem now is that this is not a very convenient way (you have to hard code it) to make the matrix.

Enter the `repmat` command.

This lets you program up the construction of the new matrix.

```
>> a=[1 2; 3 4 ; 5 6]
```

```
a =
```

```
1 2
3 4
5 6
```

```
>> b=[1;2;3]
```

```
b =
```

```
1
2
3
```

```
>> bb=repmat(b,1,2)
```

```
bb =
```

```
1 1
2 2
3 3
```

```
>> cross(a,bb)
```

```
ans =
```

```
-1 0
2 0
-1 0
```

```
>>
```

New routine repmat

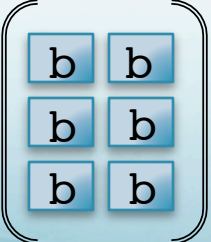
`repmat(b,n,m)` repeats the "input" matrix, `b`, `n` times in row dimension and `m` times in column dimension.

```
>> bb=repmat(b,1,2)
```

```
bb =
```

1	1
2	2
3	3

So this will take the 3×1 vector `b` and repeat it twice columnwise to produce a 3×2 matrix `bb`.

 `repmat(b,3,2)` would repeat `b` this way `[b b; b b; b b]`

That said – most people will not do it this way
either!

The astute reader will notice that we can simply
use the array addressing tools introduced earlier
to also produce the desired result.

(the astute reader will also get this technique named after them after sending it to
Matlab's discussion groups – it is known as Tony's trick after Tony Booer of
Schlumberger. Will see lots more of it later.)

```
>> bb=b(:, [1 1])
```

```
bb =
```

```
1    1  
2    2  
3    3
```

```
>>
```

This will return the 1st column
twice as two column vectors.

Lets look at this in a little more detail.

```
>> a=1:9
a =
     1     2     3     4     5     6     7     8     9
>> a2d=reshape(a,3,3)
a2d =
     1     4     7
     2     5     8
     3     6     9
>> a(:) '
ans =
     1     2     3     4     5     6     7     8     9
>>
```

What is `a2d(2,3)`?

What is `a2d(8)`?

```
>> a=1:9
a =
     1     2     3     4     5     6     7     8     9
>> a2d=reshape(a,3,3)
a2d =
     1     4     7
     2     5     8
     3     6     9
>> a(:) '
ans =
     1     2     3     4     5     6     7     8     9
>>
```

What is `a2d([8])`?

How about `a2d([8 8])`?

And `a2d([8 8]')`?

And `a2d([8;8])`?

```
>> a2d([8 8])
ans =
     8     8
```

And `a2d([8 1])`?

```
>> a2d([8 1])  
ans =  
      8      1  
>>
```

And `a2d([1 8]')`?

And `a2d([1;8])`?

The thing to notice about using the array as an index is that the result takes the shape of the array used to index the values you are accessing.

Here we are accessing the elements linearly and getting an array out based on the array used for the addressing.

But a2d is a 2 d matrix.

So what does this do?

```
>> a2d(:, [3 2 1])  
ans =  
    7     4     1  
    8     5     2  
    9     6     3
```

Get the UNIX/computer thinking cap out.

The `:` runs over all values of the first index (rows).

The array `[3 2 1]` says use these as the values for the second index (columns)

So it pulls out columns 3, 2 and 1 and makes a new array that is composed of all the rows (from the : for the first index) with the columns in this order.

Compare

```
>> a2d
```

```
a2d =
```

1	4	7
2	5	8
3	6	9

```
>> a2d(:, [3 2 1])
```

```
ans =
```

7	4	1
8	5	2
9	6	3

So you should now be able to figure out what this does

```
>> a2d([3 2 1], :)
```

So now we can answer

```
>> a3r1 = a3(:, [2:size(a3,2) - 1], :)
```

What does this do?

From Drea Thomas at the MathWorks (the company that produced Matlab)

Ask any crusty MATLAB programmer how to speed up *your* code and they'll tell you "Vectorize!".

"Vectorize!".

OK, you say. How?

This is a hard question to answer generally because:

- There are different techniques for different problems.
- There are different techniques for the same problem.
- Different techniques are better or worse depending on the matrix size.
- There is no end to the clever and obscure ways to vectorize in MATLAB.

Vectorizing is not algorithmic, there is no "recipe" that will result in (either well, or,) vectorized code.

You have to learn the already discovered tricks or invent your own.

Matlab

Programming – relational operators

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.

Some useful relational operators for whole 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

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

$\sim=$: test for not equal

Relational Operators with matrices.

What do you think they do?

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

```
a =
```

1

2

3

```
>> b=[ 1  1  3 ]
```

```
b =
```

1

1

3

```
>> a==b
```

```
ans =
```

1

0

1

```
>>
```

These return a matrix with the results of element by element testing,

return 1 if true and 0 if false.

Can use the result matrix as a mask for further processing.

Logical Operators

Logical array operators return 1 for true and 0 for false

As you might expect, 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; inverts logical value

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 the first test `(b ~= 0)` evaluates to false then MATLAB already knows the entire expression will 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 (due to a divide by zero).

Matlab

Programming – control structures

if/elseif/else/end

if expression is true, run this set of commands.
elseif if another expression is true, run this set of commands (can repeat).
else if nothing true so far, run this set of commands.
end the if block.

```
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/end
switch executes the statements associated with
the first case where

switch_expr == case_expr

If no case expression, you can have multiple
cases, 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/end

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(M,N) = 1/(m+n);
    end
end
```

Often indented for readability.

Try to avoid using **i** and **j** as loop counters
(matlab uses them for `sqrt(-1)`)

while/end

while: 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 when written on one line

(the semicolon “;” is for “silence” – else it prints out $n/2$ each time through, you need the “,” to separate the statement $n=n/2$ from the end statement).

This can be done with any type of loop structure.

break

`break`: 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 (skips remaining body of 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) | strcmp(line,'% ',1)  
        continue  
    end  
    count = count + 1;  
end  
disp(sprintf('%d lines',count));
```

Often indented for readability.

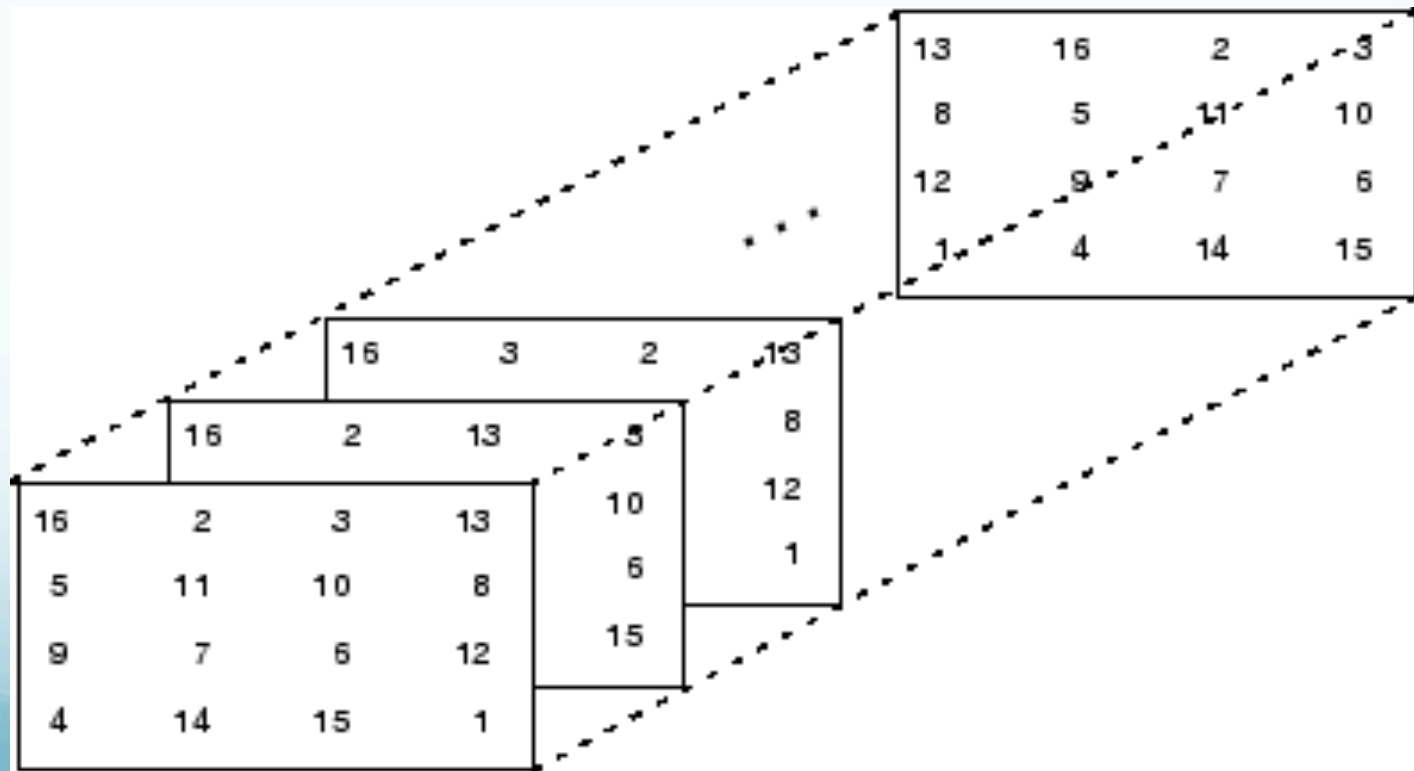
Matlab

Multi-dimensional arrays

Multidimensional Arrays

Arrays with more than two subscripts

```
>>p = perms(1:4);  
>>A = magic(4);  
>>M = zeros(4,4,24);  
>>for k = 1:24  
M(:, :, k) = A(:, p(k, :));
```



Create multidimensional arrays using `reshape` or `repmat` command (we have already seen these)

Use to change the shape of matrices
(does not change order of elements in memory,
only how we refer to them).

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

```
x =
```

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

```
>> x3d=reshape(x,2,2,2)
```

```
x3d(:,:,1) =
```

```
1 3
```

```
2 4
```

```
x3d(:,:,2) =
```

```
5 7
```

```
6 8
```

```
>> x3d(:)
```

```
x =
```

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

reshape command

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

```
x =
```

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

```
>> x3d=reshape(x,2,2,2)
```

```
x3d(:,:,1) =
```

```
1     5
3     7
```

```
x3d(:,:,2) =
```

```
2     6
4     8
```

```
>> x=reshape(x,2,4)
```

```
x =
```

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

```
>> x(:)
```

```
x =
```

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


reshape can figure out (one) dimension (of any of them).

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

```
x =
```

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

```
>> x3d=reshape(x, 2, [],2)
```

```
x3d(:, :, 1) =
```

```
1     5
3     7
```

```
x3d(:, :, 2) =
```

```
2     6
4     8
```

The dimensions specified have to be compatible with the number of elements in the matrix.

Building matrices by repeating parts

`repmat` command
(we have already seen this command)

```
>> x=[1 2;3 4]
```

```
x =  
    1    2  
    3    4
```

```
>> xr=repmat(x,2,1)
```

```
xr =  
    1    2  
    3    4  
    1    2  
    3    4
```

```
>> xr=repmat(x,1,2)
```

```
xr =  
    1    2    1    2  
    3    4    3    4
```

```
>>
```

Create constant matrix
This is something that shows up a lot.

```
>> val=pi
val =
    3.1416
>> siz=[2 2 2]
siz =
     2     2     2
>> x=repmat(val,siz)
x(:, :, 1) =
    3.1416    3.1416
    3.1416    3.1416
x(:, :, 2) =
    3.1416    3.1416
    3.1416    3.1416
>>
```

Another way (seems more roundabout, showing for completeness)

```
>> xx(prod(siz))=val
```

```
xx =  
      0      0      0      0      0      0  
0      3.1416
```

```
>> xx(:)=xx(end)
```

```
xx =  
      3.1416      3.1416      3.1416      3.1416      3.1416      3.1416  
3.1416      3.1416
```

```
>> xx=reshape(xx,siz)
```

```
xx(:, :, 1) =  
      3.1416      3.1416  
      3.1416      3.1416
```

```
xx(:, :, 2) =  
      3.1416      3.1416  
      3.1416      3.1416
```

Another way (m, n and o have to be scalar variables, again for completeness)

```
>> m=2
m =
     2
>> n=2
n =
     2
>> o=2
o =
     2
>> x(1:m,1:n,1:o)=val
x(:, :, 1) =
     3.1416     3.1416
     3.1416     3.1416
x(:, :, 2) =
     3.1416     3.1416
     3.1416     3.1416
>> x(1:m*n*o)=val
```

Also works using single dimension addressing

Another way (actually the most popular, this is Tony's trick again!) (`val` has to be a scalar variable, this syntax populates the array with `val`)

```
>> x=val(ones(siz))  
x(:, :, 1) =  
    3.1416    3.1416  
    3.1416    3.1416  
x(:, :, 2) =  
    3.1416    3.1416  
    3.1416    3.1416  
>>
```

Again - avoid using

```
X = val * ones(siz);
```

since it does unnecessary multiplications (versus just storing, above) and only works for classes for which the multiplication operator is defined.

Tony's trick does not work for NaN's (since NaN is not a variable or array, it is the same as a number, so Tony's trick does not work directly with it)

Below does not work (NaN not scalar variable, same with Inf)

```
x = NaN(ones(siz));
```

But the following do work (back to `repmat`)
(also works for scalar variable or function)

```
>> X = repmat(NaN, siz)
```

```
X(:, :, 1) =  
    NaN    NaN  
    NaN    NaN
```

```
X(:, :, 2) =  
    NaN    NaN  
    NaN    NaN
```

Tony's
trick
version

```
>> val=NaN
```

```
val =  
    NaN
```

```
>> x=val(ones(2,2))
```

```
x =  
    NaN    NaN  
    NaN    NaN
```

```
>>
```

For lots more of this – see Peter Acklam's tutorial
(on class web site)

Flipping vectors or matrices (not the same as the transpose).

```
>> a=[1 2;3 4]
```

```
a =
```

```
1 2
```

```
3 4
```

```
>> fliplr(a)
```

```
ans =
```

```
2 1
```

```
4 3
```

```
>> flipud(a)
```

```
ans =
```

```
3 4
```

```
1 2
```

```
>> a=[1:3;4:6]
```

```
a =
```

```
1 2 3
```

```
4 5 6
```

```
>> rot90(a)
```

```
ans =
```

```
3 6
```

```
2 5
```

```
1 4
```

```
>> a'
```

```
ans =
```

```
1 4
```

```
2 5
```

```
3 6
```

```
>> flipdim(a,1)
```

```
ans =
```

```
4 5 6
```

```
1 2 3
```



How to represent “nothing”

Empty array or string

Array = []

String = ' '

Useful for defining a name to be used on LHS.

Size and length are zero.

Beyond simple array variables

Structures are variables that contain other variables, called fields. They are a very powerful way to organize data in your program.

The different fields of a structure can contain variables of different types, so if one gives the fields a meaningful name this becomes a great way to keep track of the data.

In MATLAB one can define a structure (as any other variable) as one goes, it adds memory as it needs it.

Structures

Like `nawk`, Matlab allows you create structures so that you may refer to elements of an array using *textual field* designators

The format is `structure_name.field_name`

```
S.name = 'Ed Plum';  
S.score = 83;  
S.grade = 'B+'
```

creates a scalar structure with three fields:

```
S =  
name: 'Ed Plum'  
score: 83  
grade: 'B+'
```

Fields can be added one at a time

(producing a vector of the structure elements, note where the array indexing – the parens with the index – is right after the structure name and before the ".")

```
S(2).name = 'Toni Miller';  
S(2).score = 91;  
S(2).grade = 'A-';
```

Or an entire element added in single statement

```
>> S(3) = struct('name','Jerry Garcia','score',70,'grade','C')  
S =  
1x3 struct array with fields:  
name  
score  
grade  
>> scores = [S.score]  
scores =  
83 91 70  
>> avg_score = sum(scores)/length(scores)  
avg_score =  
81.3333
```

Unfortunately structure arrays don't behave as one might expect (hope?)

The following does not work (produces an error message).

```
>> avg_score = sum(S.score)/length(S.score)
```

You have to pull the vector you want to process out of the structure to use it (and make it a vector with the `[]`).

```
>> scores = [S.score]
```

```
scores =
```

```
83 91 70
```

```
>> avg_score = sum(scores)/length(scores)
```

```
avg_score =
```

```
81.3333
```

```
>> avg_score = mean([S.score])
```

```
avg_score =
```

```
81.3333
```

Example of structure and its use.

```
image.data=[1 2 3; 4 5 6; 7 8 9];  
image.date='13-Jan-2008';  
image.blank=NaN;  
image.ra=13.3212;  
image.dec=43.3455;
```

Address element of structure using structure name, decimal point, and element name.

```
image.date
```

Operate on the fields as you would with any variable of that particular type. Ex., to invert the data matrix (reference works with out [] since is scalar structure, problem is when a vector)

```
inv(image.data).
```

Example of structure and its use.

```
>> image(1).data=rand(3)
>> image.date='13-Jan-2008';
>> image.blank=NaN;
>> image.ra=13.3212;
>> image.dec=43.3455;
>> image(2).data=2*image.data
image =
1x2 struct array with fields:
    data
    blank
    dec
    date
>> image.data
ans =
    0.3922    0.7060    0.0462
    0.6555    0.0318    0.0971
    0.1712    0.2769    0.8235
ans =
    0.7845    1.4121    0.0923
    1.3110    0.0637    0.1943
    0.3424    0.5538    1.6469
>> image(1).data
ans =
    0.3922    0.7060    0.0462
    0.6555    0.0318    0.0971
    0.1712    0.2769    0.8235
>> image(2).data
ans =
    0.7845    1.4121    0.0923
    1.3110    0.0637    0.1943
    0.3424    0.5538    1.6469
```

```
>> whos
      Name      Size      Bytes  Class      Attributes

      image      1x2           1022  struct

>> inv(image(1).data)
ans =
    0.0019    1.5730   -0.1856
    1.4471   -0.8716    0.0217
   -0.4871   -0.0339    1.2457
>> inv(image(2).data)
ans =
   -0.4740    3.7530   -3.3939
   -1.3356    0.8875    0.3148
    2.0968   -4.4272    3.9447
>> sum(image(1).data)
ans =
    1.2189    1.0148    0.9668
>> sum(image(2).data)
ans =
    2.4378    2.0296    1.9335
>> sum(image.data)
Error using sum
Dimension argument must be a positive integer scalar within
indexing range.
>> sum([image.data])
ans =
    1.2189    1.0148    0.9668    2.4378    2.0296    1.9335
>>
```


Example for earthquake data

```
stn.name='mem';  
stn.lat=34.5'  
stn.lon=-89.5  
stn.elev=70;  
stn.inst='guralp cmg3'  
stn.p=15.673
```

Pass structure by name of structure. Sends it all along as a package.

```
some_fun(stn)
```

etc.

array of structures (and structure elements can be arrays – lots of parentheses).

Can be multidimensional.

```
stn(1).name='mem';  
stn(1).lat=34.5'  
stn(1).lon=-89.5  
stn(1).elev=70;  
stn(1).inst='guralp cmg3'  
stn(1).arrival(1)=15.673  
stn(1).arrival(2)=17.274  
stn(2).name='ceri';  
stn(2).lat=34.53'  
stn(2).lon=-89.57  
stn(2).elev=79;  
stn(2).inst='guralp cmg3'  
stn(2).arrival(1)=16.189  
stn(2).arrival(2)=19.923  
... .
```

Example - Create constant matrix with non-numeric data.

```
>> siz=[2 2 2];
>> s.x=1
s =
    x: 1
>> s.n='ceri'
s =
    x: 1
    n: 'ceri'
>> x=s(ones(siz))
x =
2x2x2 struct array with fields:
    x
    n
>> x
x =
2x2x2 struct array with fields:
    x
    n
```

Tony's trick

```
>> x.x
ans =
    1
    1
. . . 7 more times . . .
>> x.n
ans =
ceri
. . . 7 more times . . .
>> x(2,2,2)
ans =
    x: 1
    n: 'ceri'
```

Cell Arrays

multidimensional arrays whose elements are copies of other arrays.

cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{ }`.

The curly braces are also used with subscripts to access the contents of various cell elements.

```
>>C = {A      sum(A)      prod(prod(A)) }  
[4x4 double] [1x4 double] [20922789888000]
```

to retrieve a cell from a cell array

C{1} -> A, the magic square
C{2} -> row vector of the sum of the columns of A
C{3} -> prod(prod(A))

Important distinction with respect to other programming languages –

cell arrays contain copies of other arrays, not pointers to those arrays.

Cell Arrays vs Multidimensional Arrays

You can use three-dimensional arrays to store a sequence of matrices of the *same size*.

Cell arrays can be used to store a sequence of matrices of *different sizes*.

Characters and Text

Matlab treats text like a character vector

Enter text into MATLAB using single quotes.

```
>> s = 'Hello'
```

essentially, `s` is now a 1 x 5 array with each element equal to a character: H, e, l, l, o

Characters are stored as numbers using ASCII coding with the type *char*

```
a = double(s)
a =
72 101 108 108 111
```

Because characters are stored as numbers, you can convert numeric vectors to their ASCII characters, if the character exists

```
s=char(a)
```

Printable ASCII characters go from 32 to 127

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
<hr/>															
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	\$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c		124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

To manipulate a body of text with lines of different lengths, you have two choices

- a padded character array
- a cell array of strings.

When creating a character array, each row of the array must be the same length.

The `char` function pads with spaces to create equal rows

```
S = char('A','rolling','stone','gathers','momentum.')
```

produces a 5-by-9 character array:

```
S =  
A_____  
rolling_____  
stone_____  
gathers_____  
momentum.
```

You don't have to worry about this with a cell array

```
C = {'A'; 'rolling'; 'stone'; 'gathers'; 'momentum.'}
```

You can convert a padded character array to a cell array of strings with

```
C = cellstr(S)
```

and reverse the process with

```
S = char(C)
```

To create a character array from one of the text fields in a structure (name, for example), call the `char` function on the comma-separated list produced by `S.name`:

```
>>names = char(S.name)
names =
Ed Plum
Toni Miller
Jerry Garcia
```

Checking for special elements (NaN, Inf)

`isnan(a)` Returns 1 for every NaN in array `a`.

`isinf(a)` Returns 1 for every Inf in array `a`.

`isfinite(a)` Returns 1 for every finite number (not a (NaN or Inf)) in array `a`.

`isreal(a)` Returns 1 for every non-complex number array `a`.

Using special elements to your advantage.

Since NaNs propagate through calculations (answer is NaN if there is a NaN somewhere in the calculation), it is sometimes useful to throw NaNs out of operations like taking the mean.
(A handy trick to ignore stuff you don't want while you continue calculating.)

Example of NaNs propagating through calculation (answer is NaN if there is a NaN somewhere in the calculation)

```
>> a=1:4
```

```
a =
```

```
     1     2     3     4
```

```
>> b=10:-1:7
```

```
b =
```

```
    10     9     8     7
```

```
>> a(2)=NaN
```

```
a =
```

```
     1   NaN     3     4
```

```
>> a+b
```

```
ans =
```

```
    11   NaN    11    11
```

```
>>
```

It is sometimes useful to be able to throw NaNs out of operations like taking the mean.

(A handy trick to ignore stuff you don't want while you continue calculating.)

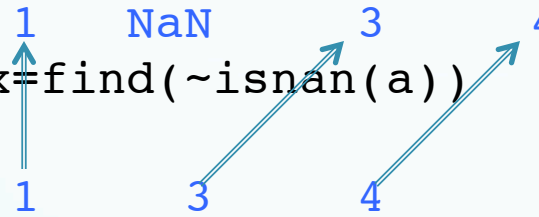
So the function that identifies NaNs can be very useful:

```
>> a
a =
    1    NaN    3    4

>> ix=find(~isnan(a))
ix =
    1    3    4

>> m=mean(a(ix))
m =
    2.6667

>>
```



finds all values of `a` that are not NaNs and averages them (denominator is number of elements averaged, not total number of elements).

help

Built into matlab

help "command"

To get help on the command "command"

Problem when you don't know the name of the command

Just type “help”

```
>> help  
HELP topics:
```

Documents/MATLAB	- (No table of contents file)
matlab/general	- General purpose commands.
matlab/ops	- Operators and special characters.
matlab/lang	- Programming language constructs.
matlab/elmat	- Elementary matrices and matrix manipulation.
matlab/randfun	- Random matrices and random streams.

Lists topics of help available

Then to get contents of topics type help “topic”

```
>> help elmat
```

Elementary matrices and matrix manipulation.

Elementary matrices.

zeros	- Zeros array.
ones	- Ones array.
eye	- Identity matrix.
repmat	- Replicate and tile array.
linspace	- Linearly spaced vector.
logspace	- Logarithmically spaced vector.
freqspace	- Frequency spacing for frequency response.
meshgrid	- X and Y arrays for 3-D plots.
accumarray	- Construct an array with accumulation.
:	- Regularly spaced vector and index into matrix.

Basic array information.

size	- Size of array.
------	------------------

Help on individual command

```
>> help zeros
ZEROS  Zeros array.
ZEROS(N) is an N-by-N matrix of zeros.
ZEROS(M,N) or ZEROS([M,N]) is an M-by-N matrix of zeros.
ZEROS(M,N,P,...) or ZEROS([M N P ...]) is an M-by-N-by-P-
by-... array of
zeros.
ZEROS(SIZE(A)) is the same size as A and all zeros.
ZEROS with no arguments is the scalar 0.
ZEROS(M,N,...,CLASSNAME) or ZEROS([M,N,...],CLASSNAME) is an
M-by-N-by-... array of zeros of class CLASSNAME.
Note: The size inputs M, N, and P... should be nonnegative
integers.
Negative integers are treated as 0.
Example:
    x = zeros(2,3,'int8');
See also eye, ones.
Reference page in Help browser
    doc zeros
```

Matlab file exchange

<http://www.mathworks.com/matlabcentral/fileexchange/>

Or
Google on what you want/need.

Some unix commands (`pwd`, `ls`, `???`) “work” in matlab (they are actually matlab commands)

```
a=pwd;  
b=ls;
```

Some Matlab commands have the same names as UNIX commands, but are not the same

“**cat**” is a matlab command that concatenates matrices (not files)

Matlab does not pass things it does not understand to the OS to see if they are OS commands.