Data Analysis in Geophysics
ESCI 7205

Bob Smalley
Room 103 in 3892 (long building), x-4929

Tu/Th - 13:00-14:30
CERI MAC (or STUDENT) LAB

Lab – 5, 09/10/13

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
```

The save function

Writes files containing matrices (from memory).
Default format – matlab binary.

```
>> save                                    default name "matlab.mat"
>> myfile='my_file.mat'
>> save(myfile)
>> save('my_file.mat','a','b')
>> save(myfile,'a', '-ascii')

save(FILENAME,VARIABLES)
save(FILENAME, , ...,FORMAT)
save(FILENAME, ..., '-append')

'-mat'                              Binary MAT-file format (default).
'-ascii'                            8-digit ASCII format.
'-ascii', '-tabs'                   Tab-delimited 8-digit ASCII format.
'-ascii', '-double'                 16-digit ASCII format.
'-ascii', '-double', '-tabs'        Tab-delimited 16-digit ASCII format
```

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.

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

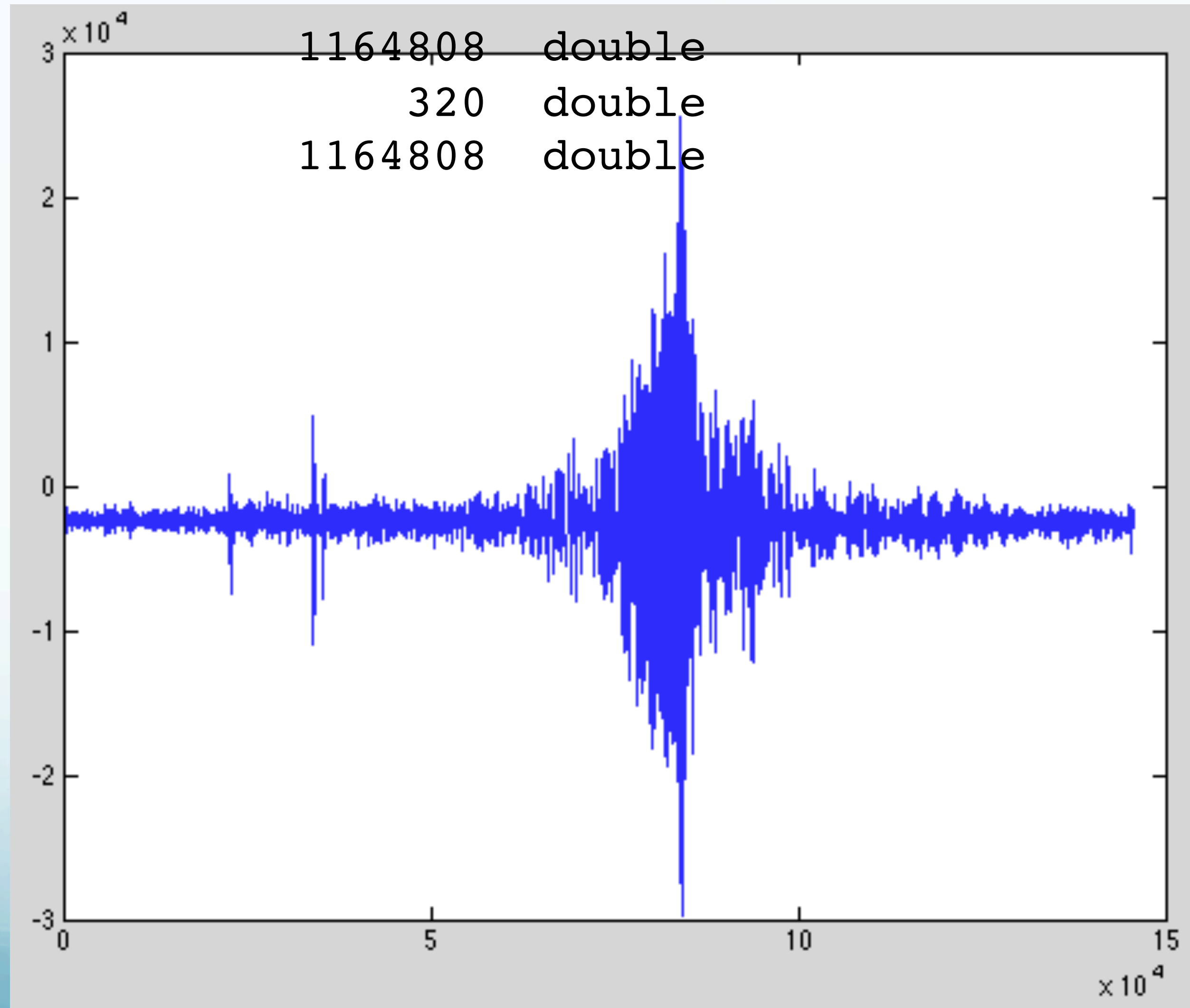
```
>> magik    #places matrix in A
```

Reading SAC files

```
>> [t,a,p]=readsac('ccm_india_.bhz');
>> whos
  Name        Size                       Bytes  Class     Attributes

  a           1x145601                 1164808  double
  p           1x40                         320  double
  t           1x145601                 1164808  double

>> plot(a)
```



Functions

Are also 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 – but...

Encapsulates some calculation, etc., and called by the function name, with defined inputs and outputs (similar to "built-in" functions such as sin, cos, etc.)

Write in any editor, save as .m file. Function name has to be same as file name, one function per file.

```
function TensorOut = TensorRotate( TensorIn, RotAng )
%does tensor rotation through angle RotAng

RotMat=[cosd(RotAng) sind(RotAng); -sind(RotAng) cosd(RotAng)];
TensorOut = RotMat*TensorIn*RotMat';

End
```

To use

```
>> a=[1 0; 0 -1];
>> arot=TensorRotate(a,45)
ans =
                   0  -1.000000000000000
  -1.000000000000000                   0
>> b=[0 1; 1 0];
>> TensorRotate(b,45)
ans =
   1.000000000000000                   0
                   0  -1.000000000000000
>> help TensorRotate
 does tensor rotation through angle RotAng
```

Output can be single variable as in last or multiple items (if only give one output, get first).

```
function [TensorOut RotMat] = TensorRotate( TensorIn, RotAng )
%does tensor rotation through angle RotAng

RotMat=[cosd(RotAng) sind(RotAng); -sind(RotAng) cosd(RotAng)];
TensorOut = RotMat*TensorIn*RotMat';
End
```

To use

```
>> TensorRotate(a,30)
ans =
   0.500000000000000  -0.866025403784439
  -0.866025403784439  -0.500000000000000
>> [arot rm]=TensorRotate(a,30)
arot =
   0.500000000000000  -0.866025403784439
  -0.866025403784439  -0.500000000000000
rm =
   0.866025403784439   0.500000000000000
  -0.500000000000000   0.866025403784439
>>
```

`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)

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 because no errors) creates array named `a = 1 2 3`, and then an array named `ans = 4 5 6`. It uses the `;` as a line separator (and suppresses output, not a row separator (when outside `[ ]`).

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"

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 2nd 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.

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
>> 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
>>
```

Right array divide.

Left matrix divide

Matrix on top is dividend.

Matrix on bottom is divisor.

Array and Matrix divide (Matlab inventions)

```
>> 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.

Check a is invertible

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

Note this is the solution to a*b=c when you know a and c and want b.

d=b above.

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 =
  4.999999999999999   5.999999999999997
  7.000000000000001   8.000000000000002
>>
```

`mldivide(A,B)` and the equivalent `A\B` perform <u>matrix</u> 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 <u>matrix</u> 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 over-determined system of equations

`AX = B`.

`mldivide(A,B)` and the equivalent `A\B` perform matrix 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 not 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 <u>matrix</u> 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 over-determined 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
```

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`

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	

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
>> c=a==b
c =
     0     0     1     0     0
>>
```

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

Exclusive or

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

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]
```

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 of matrix gives the max dimension

```
>> x=[1 2 3; 4 5 6; 7 8 9; 10 11 12];
>> length(x)
ans =
     4
```

(# rows here)

```
>> x=[1 2 3 45 6; 7 8 9 10 11 12];
>> length(x)
ans =
     6
```

(# columns here)

```
>> length(x(:))
ans =
     12
```

Linear size (as vector – total number elements)

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 (3600x3600 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*360*180~20,000 (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)

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

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

We can address **a** as a 1-d vector.

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.

```
>> el=1
el =
     1
>> a(el)
ans =
    10
>> a([1])
ans =
    10
>> arry=[1]
arry =
     1
>> a(arry)
ans =
    10
>>
```

We can also use a variable for the index

Or an array (explicitly) or as a variable

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.

```
>> [1 3:-1:2]
ans =
     1     3     2
>> [1 5:-1:2]
ans =
     1     5     4     3     2
>> [1 5:-1:3]
ans =
     1     5     4     3
>>
```

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 want to do loops - SLOW.)

```
>> 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

```
>> 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
>>
```

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 (`3x1*1x2=3x2`).

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

More on vectorizing

```
>> 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
>>
```

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 multply 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=repmat(b,1,2)
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.

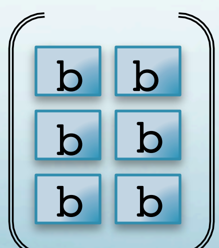
This lets you program up the construction of the new matrix.

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 3x1 vector b and repeat it twice columnwise to produce a 3x2 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],:)
```

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 alogrithmic, 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

~= : 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.
else 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) | strncmp(line,'%',1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines',count));
```

Often indented for readability.