

Data Analysis in Geophysics

ESCI 7205

Class 16

Bob Smalley

More Matlab.

More on vectorization.

MATLAB is a vectorized high level language

Requires change in programming style
(if one already knows a non-vectorized
programming language such as Fortran, C, Pascal,
Basic, etc.)

Vectorized languages allow operations over
arrays using simple syntax, essentially the same
syntax one would use to operate over scalars.
(looks like math again.)

What is vectorization? (with respect to matlab)

Vectorization is the process of writing code for MATLAB that uses matrix operations or other fast built-in functions instead of using explicit loops.

The benefits of doing this are usually sizeable.

The reason for this is that MATLAB is an interpreted language. Function calls have very high overhead, and indexing operations (inherent in a loop operation) are not particularly fast.

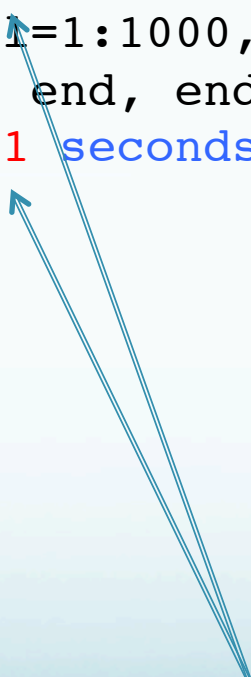
Loop versus vectorized version of same code.
New commands “tic” and “toc” – time the execution of the code between them.

```
>> a=rand(1000);  
>> tic;b=a*a;toc  
Elapsed time is 0.229464 seconds.  
>> tic;for k=1:1000,for l=1:1000,c(k,l)=0;for m=1:1000, c(k,l)=c  
(k,l)+a(k,m)*a(m,l);end, end, end, toc  
Elapsed time is 22.369451 seconds.  
>> whos
```

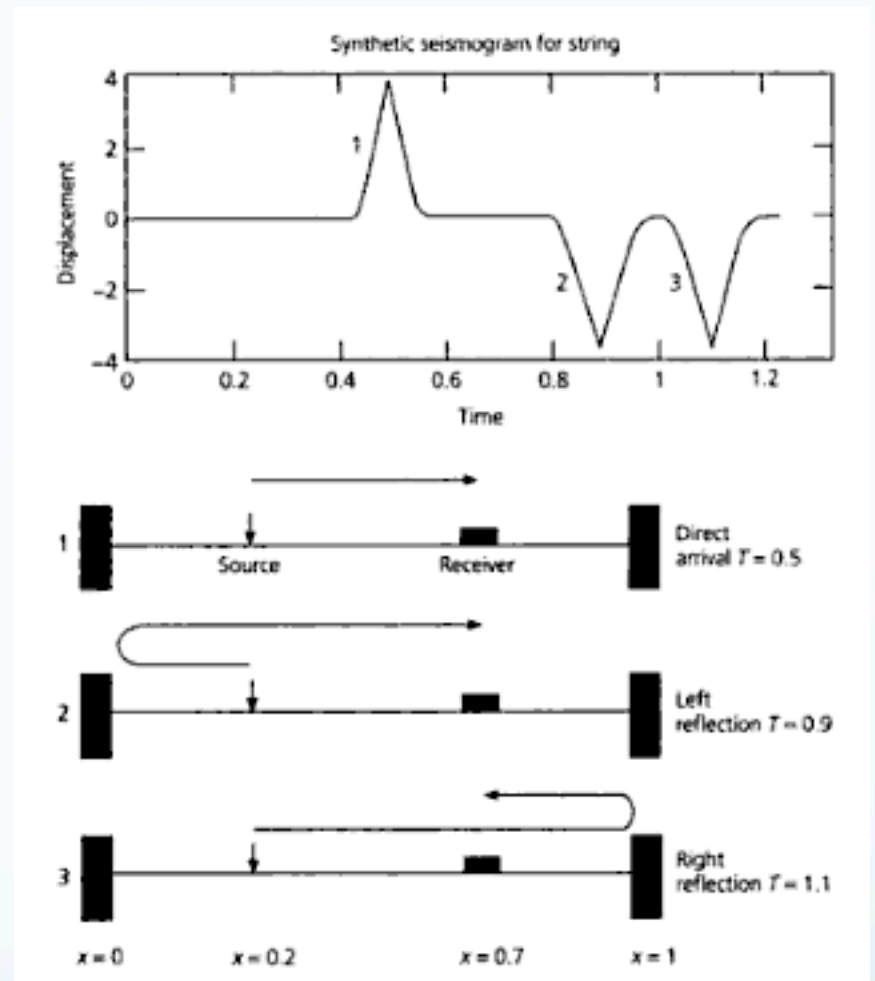
Name	Size	Bytes	Class	Attributes
a	1000x1000	8000000	double	
b	1000x1000	8000000	double	
c	1000x1000	8000000	double	
k	1x1	8	double	
l	1x1	8	double	
m	1x1	8	double	

```
>> max(max(b-c))  
ans =  
9.6634e-13
```

Factor 100 difference in time for multiplication of $10^6 \times 10^6$ matrix!



Vectorization of synthetic seismogram example from Stein and Wyssession, Intro to Seismology and Earth Structure.



$$u(x, t) = \sum_{n=1}^{\infty} \sin(n\pi x / L) \sin(n\pi x_s / L) \cos(\omega_n t) \exp[-(\omega_n \tau / 4)]$$

This is just the Fourier transform for a standing wave

$$u(x,t) = \sum_{n=1}^{\infty} \sin(n\pi x_s / L) \sin(n\pi x / L) \cos(\omega_n t) \exp[-(\omega_n \tau / 4)]$$

(Note: $\omega_n = n * \omega_0$)

$$u(x,t) = \sum_{n=1}^{\infty} \left(\sin(n\pi x_s / L) \exp[-(\omega_n \tau / 4)] \right) \sin(n\pi x / L) \cos(\omega_n t)$$

Weight - no dependence
on x or t

$$u(x,t) = \sum_{n=1}^{\infty} a_n \sin(n\pi x / L) \cos(\omega_n t)$$

Standing wave made from 2 opposite
direction traveling waves. Amplitude
varies with time, but does not "move"

$$u(x,t) = \sum_{n=1}^{\infty} a'_n \left[\cos(n\pi x / L + \omega_n t) + \cos(n\pi x / L - \omega_n t) \right]$$

Normal Mode (and combination of traveling waves to make standing wave) formulation for displacement of a string

$$u(x,t) = A \cos(kx + \omega t) + A \cos(kx - \omega t)$$

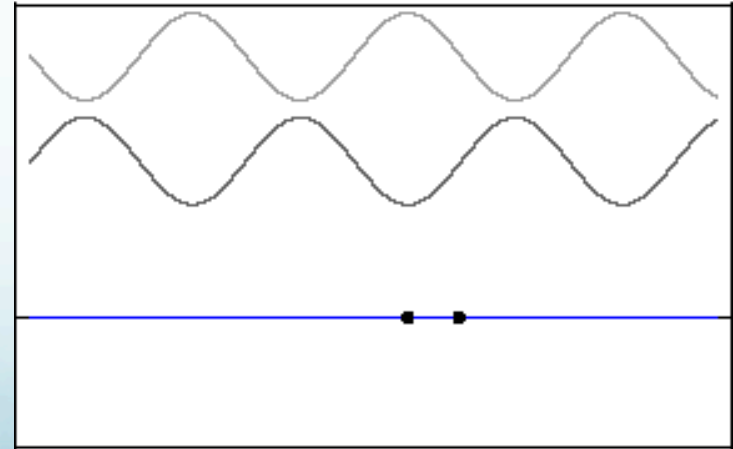
$$u(k,\omega) = A \cos(kx + \omega t) + A \cos(kx - \omega t)$$

$$u(x,t) = u(k,\omega) = 2A \cos(\omega t) \cos(kx)$$

$$u_n(x,t) = \cos(k_n x / L) \cos(\omega_n t)$$

where $\omega_n = v k_n$

This is a sinusoidal wave that is fixed in space, $\cos(kx)$, whose amplitude is modulated harmonically in time, $\cos(\omega t)$



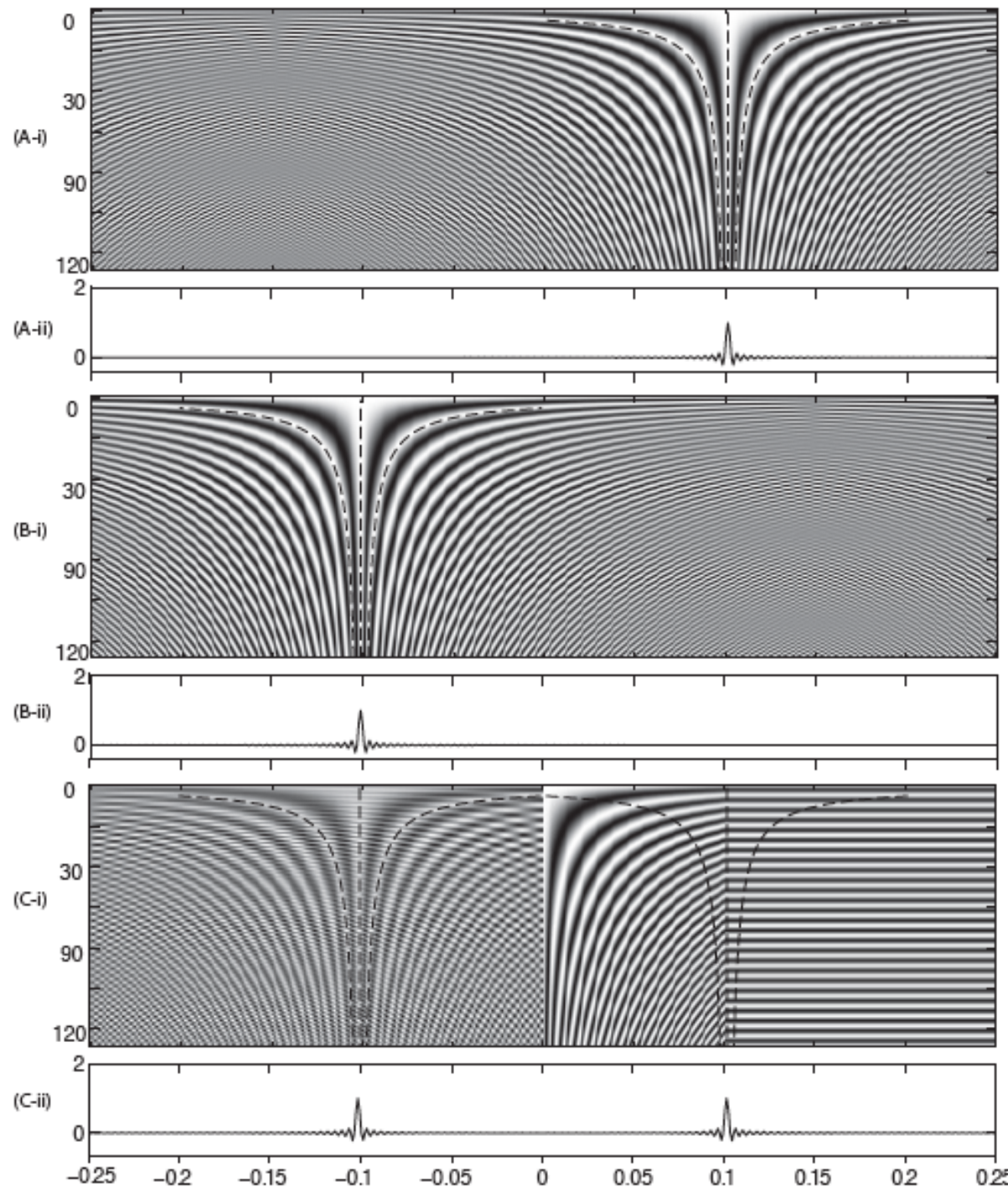
Do over a range
of frequencies.

Delta functions
going right (top)
and left (middle)
and combined
(bottom).

$$u(x, t) = A \cos(kx + \omega t) + A \cos(kx - \omega t)$$

$$u(k, \omega) = A \cos(kx + \omega t) + A \cos(kx - \omega t)$$

$$u(x, t) = u(k, \omega) = 2A \cos(\omega t) \cos(kx)$$



Look at the basic element of Fourier series, weighted sum of sin and cos functions

(look at cos only to see how works).

$$u(t_m) = \frac{a_0}{2} + \sum_{n=1}^N a_n \cos(\omega_n t_m)$$

$$u(t_m) = \frac{a_0}{2} + (a_1 \ a_2 \ a_3 \ \cdots \ a_n) \bullet (\cos(\omega_1 t_m) \ \cos(\omega_2 t_m) \ \cos(\omega_3 t_m) \ \cdots \ \cos(\omega_n t_m)) \leftarrow \text{Dot product}$$

$$u(t_m) = \frac{a_0}{2} + (a_1 \ a_2 \ a_3 \ \cdots \ a_n) \begin{pmatrix} \cos(\omega_1 t_m) \\ \cos(\omega_2 t_m) \\ \cos(\omega_3 t_m) \\ \vdots \\ \cos(\omega_n t_m) \end{pmatrix} \begin{matrix} \leftarrow \text{or matrix multiply} \rightarrow \end{matrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} = \frac{a_0}{2} + (\cos(\omega_1 t_m) \ \cos(\omega_2 t_m) \ \cos(\omega_3 t_m) \ \cdots \ \cos(\omega_n t_m)) \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix}$$

$$\vec{u}(t_m : t_{m+k}) = \frac{a_0}{2} + \begin{pmatrix} \cos(\omega_1 t_m) & \cos(\omega_2 t_m) & \cos(\omega_3 t_m) & \cdots & \cos(\omega_n t_m) \\ \cos(\omega_1 t_{m+1}) & \cos(\omega_2 t_{m+1}) & \cos(\omega_3 t_{m+1}) & \cdots & \cos(\omega_n t_{m+1}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \cos(\omega_1 t_{m+k}) & \cos(\omega_2 t_{m+k}) & \cos(\omega_3 t_{m+k}) & \cdots & \cos(\omega_n t_{m+k}) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix}$$

$$\vec{u}(t_m : t_{m+k}) = \frac{a_0}{2} + \vec{W} \vec{a}$$

matrix multiply, at multiple times to make full seismogram

Look at the basic Fourier series

At constant time, weighted
sum of cosines at different
frequencies at that time

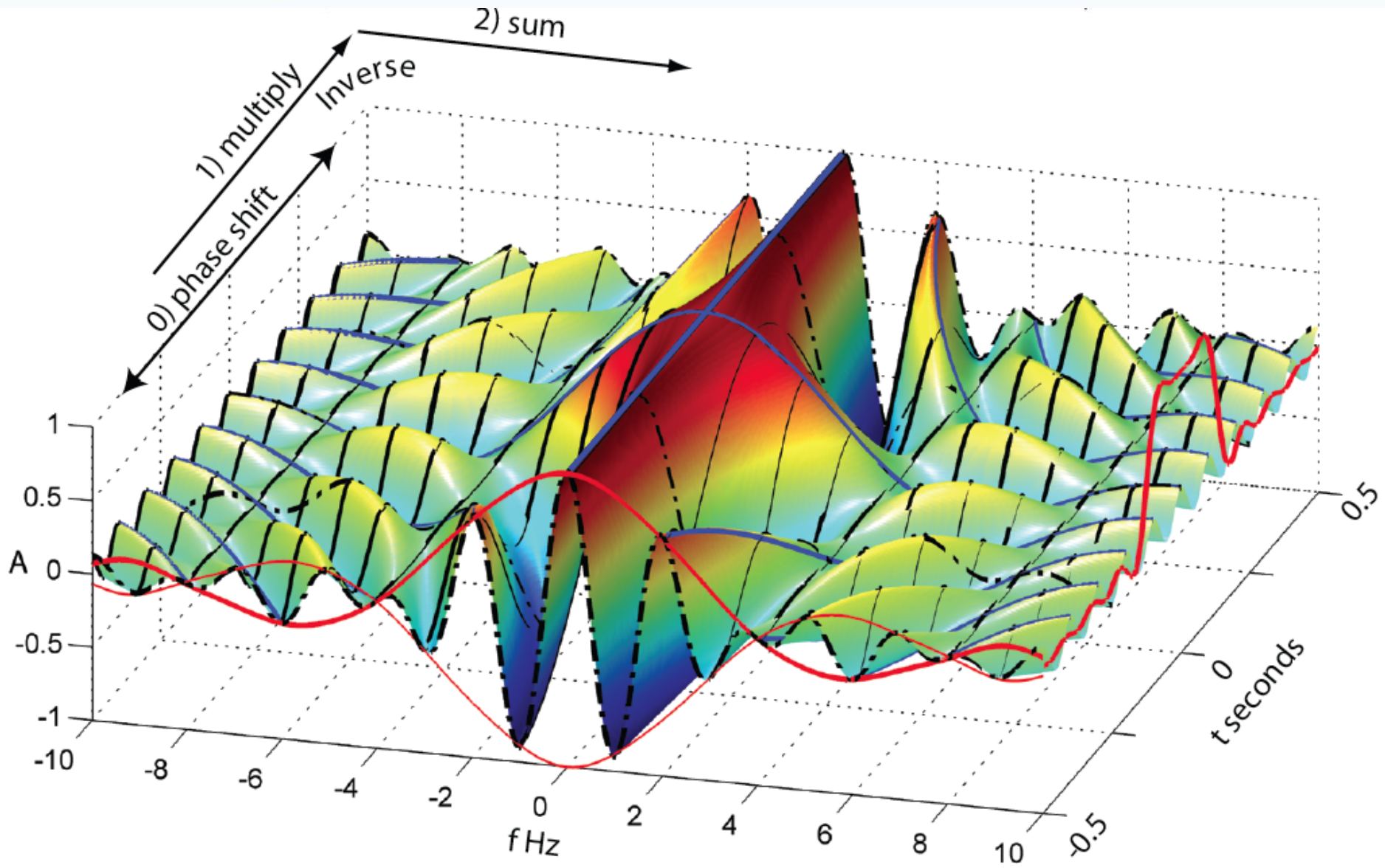
$$\vec{u}(t_m : t_{m+k}) = \frac{a_0}{2} + \begin{pmatrix} \cos(\omega_1 t_m) & \cos(\omega_2 t_m) & \cos(\omega_3 t_m) & \cdots & \cos(\omega_n t_m) \\ \cos(\omega_1 t_{m+1}) & \cos(\omega_2 t_{m+1}) & \cos(\omega_3 t_{m+1}) & \cdots & \cos(\omega_n t_{m+1}) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cos(\omega_1 t_{m+k}) & \cos(\omega_2 t_{m+k}) & \cos(\omega_3 t_{m+k}) & \cdots & \cos(\omega_n t_{m+k}) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \cdots \\ a_n \end{pmatrix}$$

$$\vec{u}(t_m : t_{m+k}) = \frac{a_0}{2} + \vec{W} \vec{a}$$

constant frequency cosine as function of time
(basis functions)

This is multiplication of a matrix (with cosines as
functions of frequency – across – and time –
down) times a vector containing the Fourier series
weights.

We have just vectorized the equations for the
Fourier Transform!



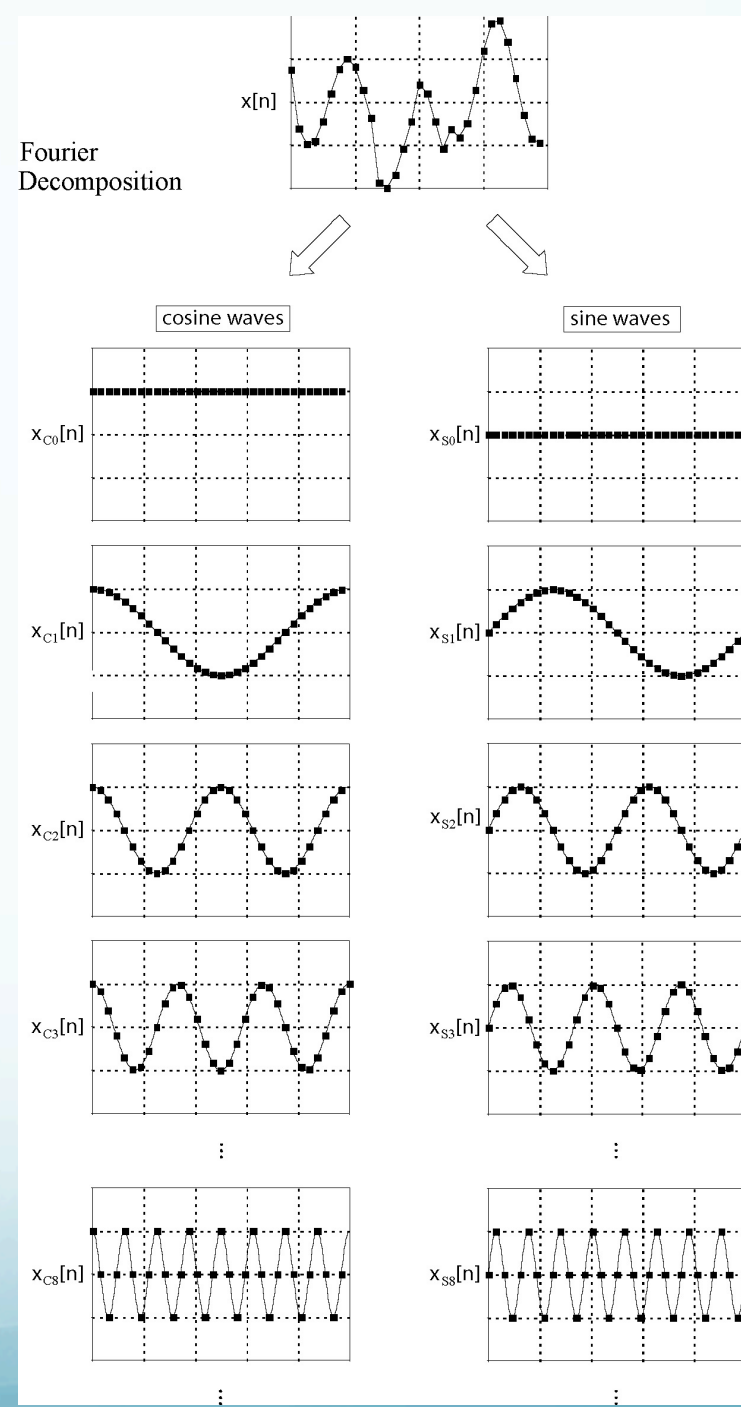
Even though this is a major improvement over doing this with for loops, and is clear conceptually, it is still not "computable" as it takes $O(N^2)$ operations (and therefore time) to do it. This is OK for small N , but quickly gets out of hand.

Fourier analysis is typically done using the Fast Fourier transform (FFT) algorithm – which has $O(N \log N)$ operations and is significantly faster for large N .

Fourier decomposition.

“Basis” functions are the sine and cosine functions.

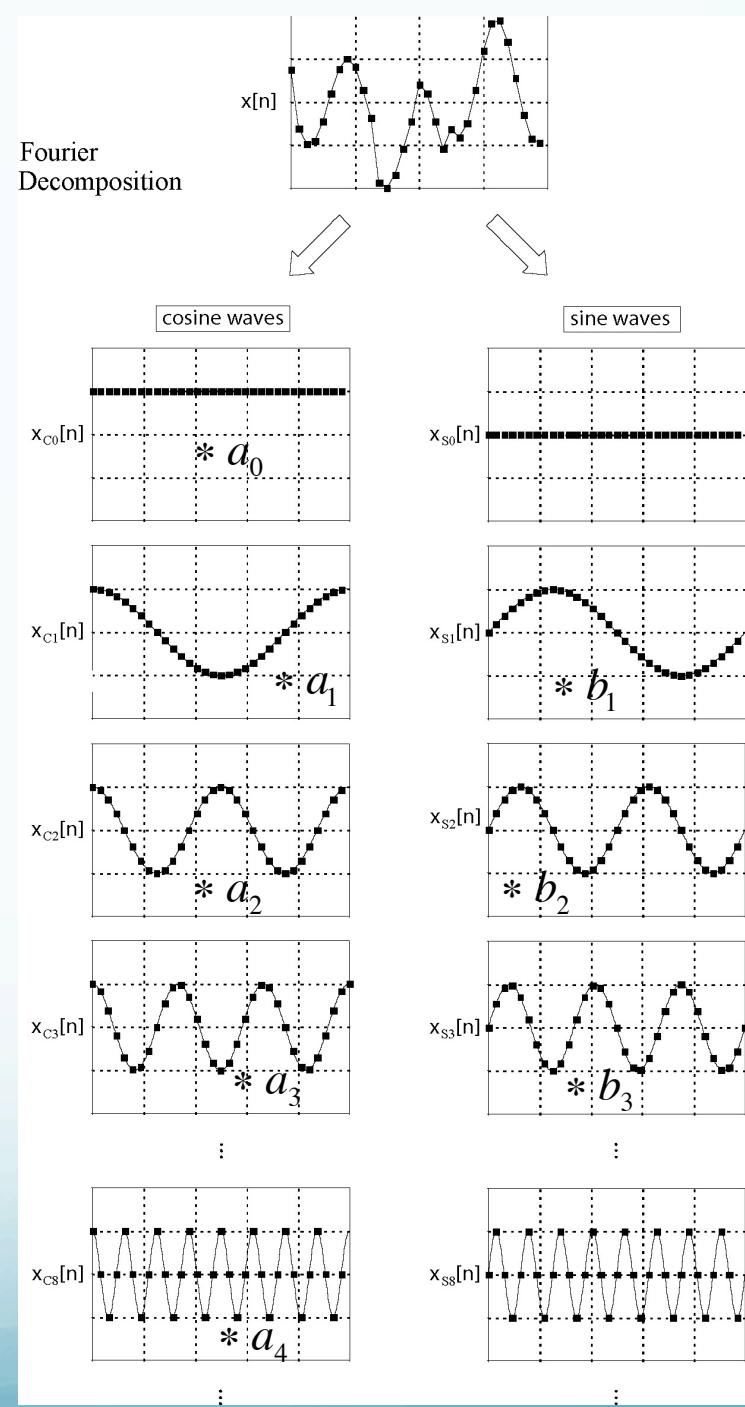
Notice that first sine term is all zeros (so don't really need it) and last sine term (not shown) is same as last cosine term, just shifted one – so will only need one of these).



Fourier transform actually Fourier series

$$u(t_m) = \frac{a_0}{2} + \sum_{n=1}^N a_n \cos(\omega_n t_m) + \sum_{n=1}^N b_n \sin(\omega_n t_m)$$

The Fast Fourier Transform (FFT) depends on noticing that there is a lot of repetition in the calculations – each higher frequency basis function can be made by selecting points from the ω_0 function. The weight value is multiplied by the same basis function value an increasing number of times as ω increases.



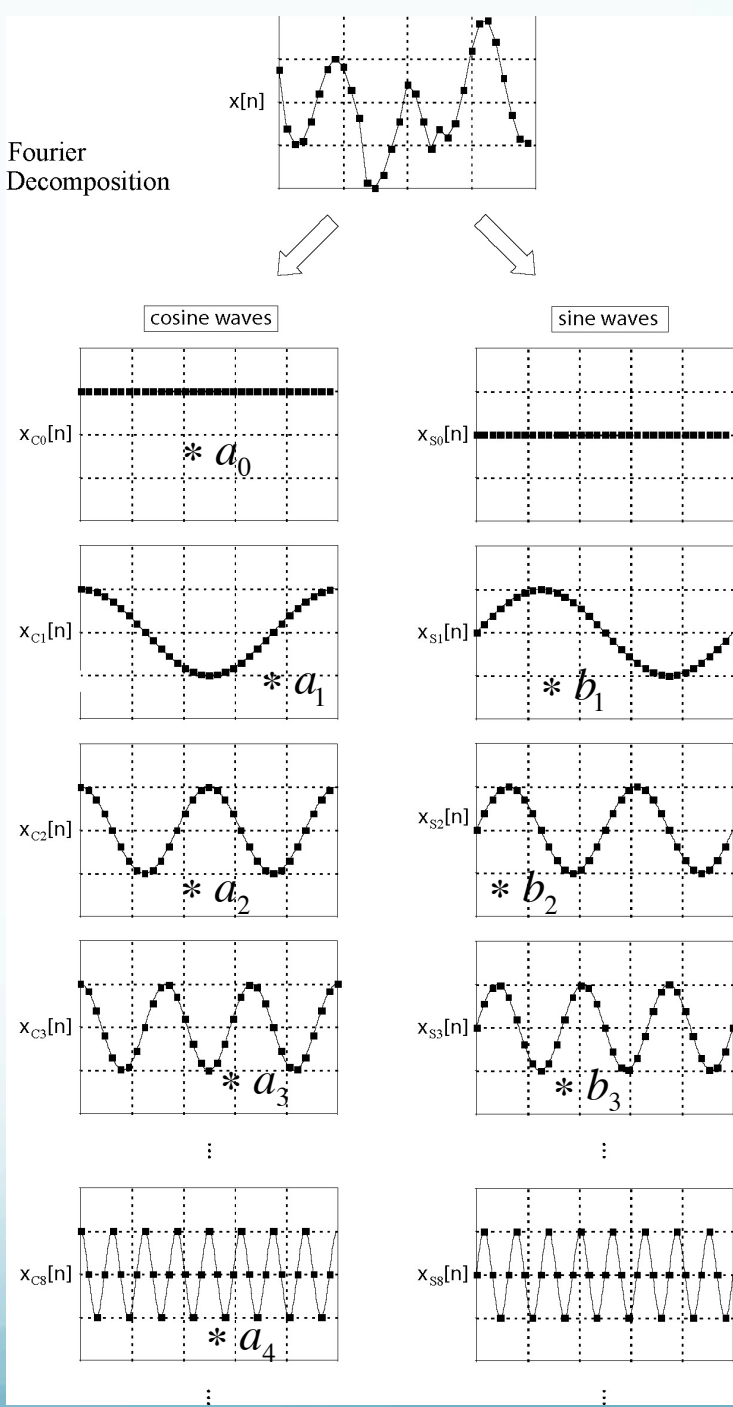
FFT

$$u(t_m) = \frac{a_0}{2} + \sum_{n=1}^N a_n \cos(\omega_n t_m) + \sum_{n=1}^N b_n \sin(\omega_n t_m)$$

The FFT uses regularities in the calculation and basically does each unique multiplication only once, stores it, and then does the bookkeeping to add them all up correctly.

The points in the trace at the top are made from vertical sums of the weighted points at the same time in the **cos** and **sin** traces in the bottom.

Fourier
Decomposition



```

C SYNTHETIC SEISMOGRAM FOR HOMOGENEOUS STRING
C DISPLACEMENT U AS FUNCTION OF TIME T
C CALCULATED BY NORMAL MODE SUMMATION
    DIMENSION U(200)
    PI = 3.1415927

C
C PARAMETERS (NORMALLY WOULD COME FROM INPUT)
C STRING LENGTH (M)
    ALNGTH = 1.0
C VELOCITY (M/S)
    C = 1.0
C NUMBER OF MODES
    NMODE = 200
C SOURCE POSITION (M)
    XSRC = 0.2
C RECEIVER POSITION (M)
    XRCVR = 0.7
C SEISMOGRAM TIME DURATION (S)
    TDURAT = 1.25
C NUMBER TIME STEPS
    NTSTEP = 50
C TIME STEP (S)
    DT = TDURAT/NTSTEP
C SOURCE SHAPE TERM
    TAU = .02

C
C LIST PARAMETERS
    WRITE (6,3000)
3000 FORMAT('SYNTHETIC SEISMOGRAM FOR STRING')
    WRITE (6,3001) NMODE
3001 FORMAT('NUMBER OF MODES', I6)
    WRITE (6,3002) ALNGTH, C
3002 FORMAT('LENGTH (M)' F7.3, 'VELOCITY,
X (M/S)', F7.3)
    WRITE (6,3003) XSRC, XRCVR
3003 FORMAT('POSITION (M): SOURCE', F7.3,
X 'RECEIVER', F7.3)
    WRITE (6,3004) TDURAT, NTSTEP
3004 FORMAT('SEISMOGRAM DURATION (S)', F7.3,
X I6, 'TIME STEPS')
    WRITE (6,3005) TAU
3005 FORMAT('SOURCE SHAPE TERM', F7.3)

C
C INITIALIZE DISPLACEMENT
    DO 5 I = 1, NTSTEP
        U(I) = 0.0
    5    CONTINUE

C
C OUTER LOOP OVER MODES
    DO 10 N = 1, NMODE
        ANPIAL = N*PI/ALNGTH

```

```

C SPACE TERMS: SOURCE AND RECEIVER
        SXS = SIN(ANPIAL*XSRC)
        SXR = SIN(ANPIAL*XRCVR)

C MODE FREQUENCY
        WN = N*PI*C/ALNGTH

C TIME INDEPENDENT TERMS
        DMP = (TAU*WN)**2
        SCALE = EXP(-DMP/4.)
        SPACE = SXS*SXR*SCALE

C
C INNER LOOP OVER TIME STEPS
        DO 15 J = 1, NTSTEP
            T = DT*(J - 1)
            CWT = COS(WN*T)

C COMPUTE DISPLACEMENT
            U(J) = U(J) + CWT*SPACE

15    CONTINUE
10    CONTINUE

C
C OUTPUT SEISMOGRAM FOR LATER PLOTTING
    WRITE (6, 3101) (U(J), J = 1, NTSTEP)
3101 FORMAT (7F10.4)

    STOP
    END

```

Traditional
programming with
nested loops (left
Fortran from Stein's
book, right "translated"
to Matlab).

Related to the
details of the math
(as if you were doing
it by hand).

```

%synthetic seismogram for homogeneous string, u(t)
%calculated by normal mode summation
%string length
alngth=1;
%velocity m/sec
c=1.0;
%number modes
nmode=200;
%source position
xsrc=0.2;
%receiver position
xrcvr=0.7;
%seismogram time duration
tdurat=1.25;
%number time steps
nstep=50;
%time step
dt=tdurat/nstep;
%source shape term
tau=0.02;
fprintf('%s\n','synthetic seismogram for string')
fprintf('%s %0.5g\n','number modes', nmode)
fprintf('%s %0.5g %0.5g\n','length and velocity', alngth, c)
fprintf('%s %0.5g %0.5g %0.5g\n','posn src and rcvr',xsrc,xrcvr)
fprintf('%s %0.5g %0.5g %0.5g\n','durn, time steps, del
t',tdurat,nstep,dt)
fprintf('%s %0.5g\n','source shape', tau)
%initialize displacement
for cnt=1:nstep
    u(cnt)=0;
end
for k=1:nstep
    t(k)=dt*(k-1);
end
%outer loop over modes
for n=1:nmode
    anpial=n*pi/alngth;
    %space terms - src & rcvr
    sxs=sin(anpial*xsrc);
    sxr=sin(anpial*xrcvr);
    %mode freq
    wn=n*pi*c/alngth;
    %time indep terms
    dmp=(tau*wn)^2;
    scale=exp(-dmp/4);
    space=sxs*sxr*scale;
    %inner loop over time steps
    for k=1:nstep
        % t=dt*(k-1);
        % cwt=cos(wn*t);
        cwt=cos(wn*t(k));
        %compute disp
        u(k)=u(k)+cwt*space;
    end
end
plot(t,u)

```

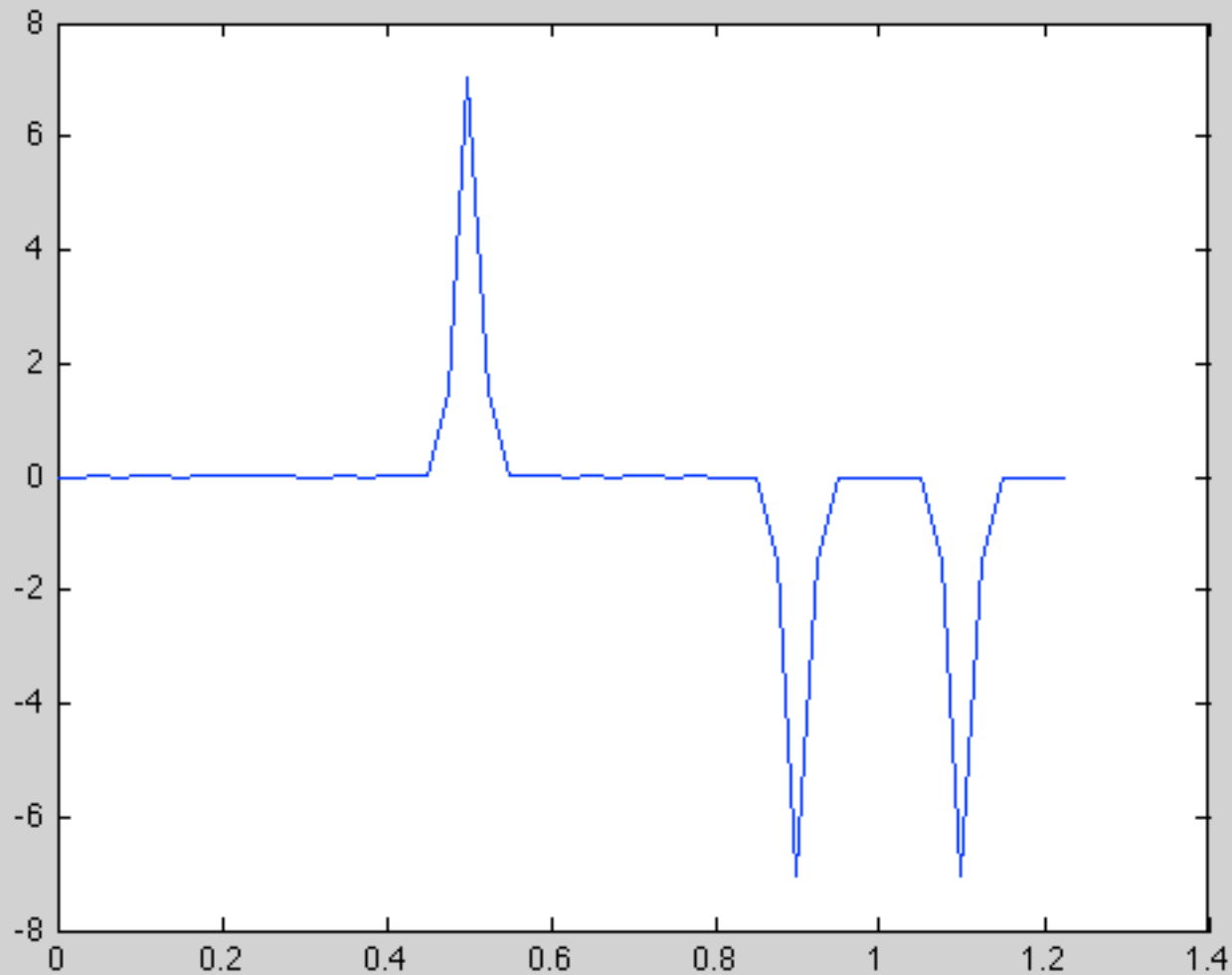

<pre> %synthetic seismogram for homogeneous string, u(t) %calculated by normal mode summation %string length alngth=1; %velocity m/sec c=1.0; %number modes nmode=200; %source position xsrc=0.2; %receiver position xrcvr=0.7; %seismogram time duration tdurat=1.25; %number time steps nstep=50; %time step dt=tdurat/nstep; %source shape term tau=0.02; fprintf('%s\n','synthetic seismogram for string') fprintf('%s %0.5g\n','number modes', nmode) fprintf('%s %0.5g %0.5g\n','length and velocity', alngth, c) fprintf('%s %0.5g %0.5g\n','posn src and rcvr',xsrc,xrcvr) fprintf('%s %0.5g %0.5g %0.5g\n','durn, time steps, del t',tdurat,nstep,dt) </pre>	<pre> fprintf('%s %0.5g\n','source shape', tau) %initialize displacement for cnt=1:nstep u(cnt)=0; end for k=1:nstep t(k)=dt*(k-1); end %outer loop over modes for n=1:nmode anpial=n*pi/alngth; %space terms - src & rcvr sxs=sin(anpial*xsrc); sxr=sin(anpial*xrcvr); %mode freq wn=n*pi*c/alngth; %time indep terms dmp=(tau*wn)^2; scale=exp(-dmp/4); space=sxs*sxr*scale; %inner loop oner time steps for k=1:nstep % t=dt*(k-1); % cwt=cos(wn*t); cwt=cos(wn*t(k)); %compute disp u(k)=u(k)+cwt*space; end end plot(t,u) </pre>	<p style="color: red;">Slightly cleaned up version of Fortran program in Stein and Wyssession “translated” to Matlab.</p>
---	---	---

Variables

```
>> whos
```

Name	Size	Bytes	Class	Attributes
alngth	1x1	8	double	
anpial	1x1	8	double	
c	1x1	8	double	
cnt	1x1	8	double	
cwt	1x1	8	double	
dmp	1x1	8	double	
dt	1x1	8	double	
k	1x1	8	double	
n	1x1	8	double	
nmode	1x1	8	double	
nstep	1x1	8	double	
scale	1x1	8	double	
space	1x1	8	double	
sxr	1x1	8	double	
sxs	1x1	8	double	
t	1x1	8	double	
tau	1x1	8	double	
tdurat	1x1	8	double	
u	1x50	400	double	
wn	1x1	8	double	
xrcvr	1x1	8	double	
xsrc	1x1	8	double	

Synthetic seismogram produced by Matlab code on previous slide.



```

% number of time samples M
% points
% source position xs (meters)
% speed c (meters/sec)
% length L (meters)
% number of modes N
% source pulse duration Tau
% (sec)
% length of seismogram T (sec)

M=50;
xs=0.25;
c=1;
L=1;
N=200;
Tau=0.02;
T=1.25;

%time vector, 1 row by M
% columns
%start, step, stop
dt=T/M;
t=0:dt:T-dt;

% receiver posn

xr = 0.7;

%stein actually starts at mode
% 1
%freq vector from 0 to n*pi*c/L
%, 1 row by N columns
wn= linspace(1,N,N);
wn=wn*pi*c/L;

%time independent terms - modes
%- 1xN vector (row vector)
timeindep=sin(wn*xr).*sin
(wn*xs).*exp(-(wn*Tau).^2/4);

%time dependent terms -
%time*freqs = MxN matrix
timedep=cos(t'*wn);

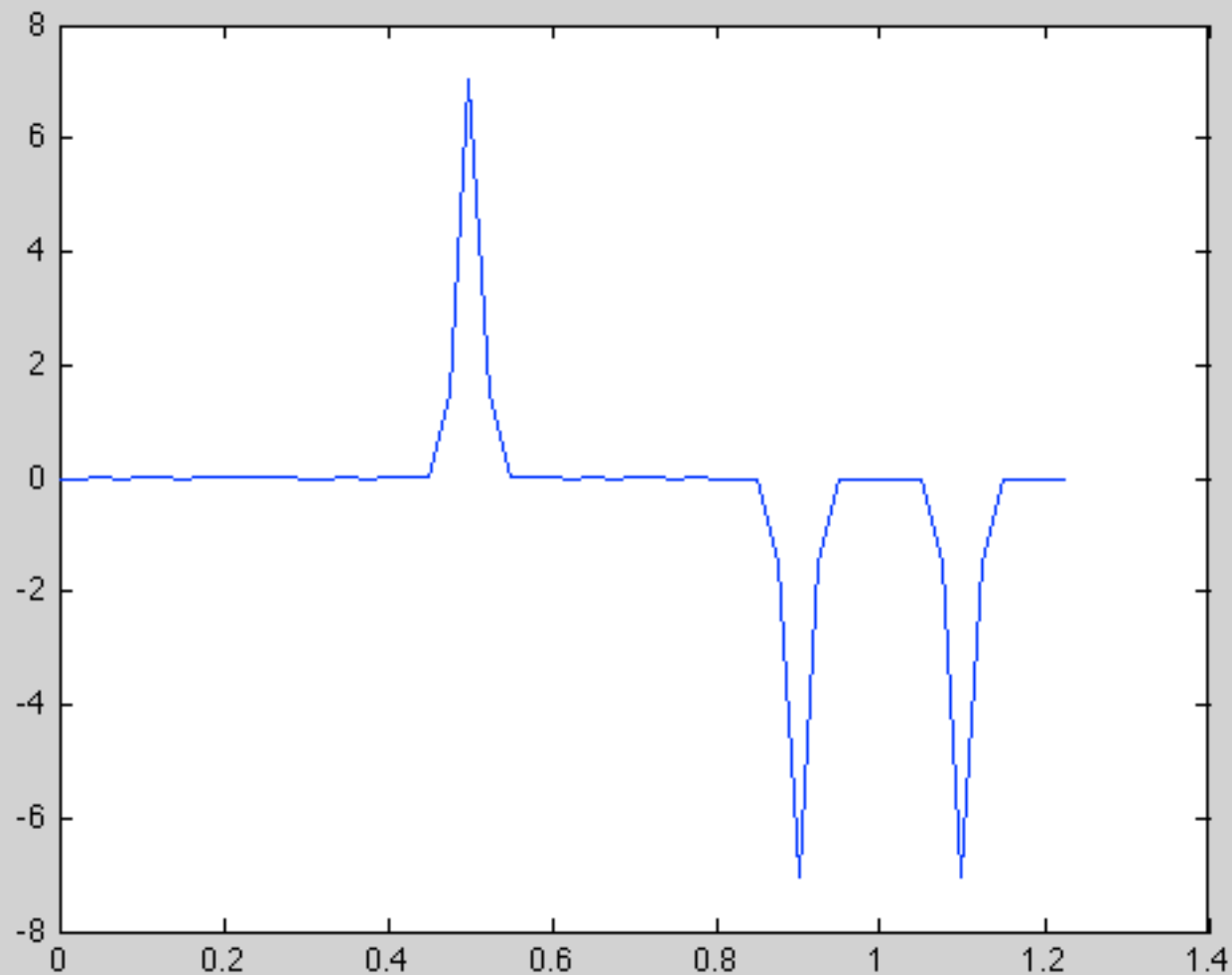
%use matrix * vector multiply
%to do "loop"
%(MxN)times(Nx1)=(Mx1) (column
%vector)
seism=timedep*timeindep';

plot(t,seism)

```

Same program in Matlab after vectorization (is mostly comments!)

Get same figure as before.



m-files

As we have seen before, it is generally convenient to save programs in some sort of file (script, macro, batch, etc.) for program development and reuse.

Matlab offers this feature through m-files, which are ascii text files containing a set of Matlab commands.

m-files

There are two kinds of m-files:

Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.

Functions, which can accept input arguments and return output arguments. Functions have internal variables that are local to the function.

The filename has to end in “.m”

You have been using scripts already in your previous matlab homework.

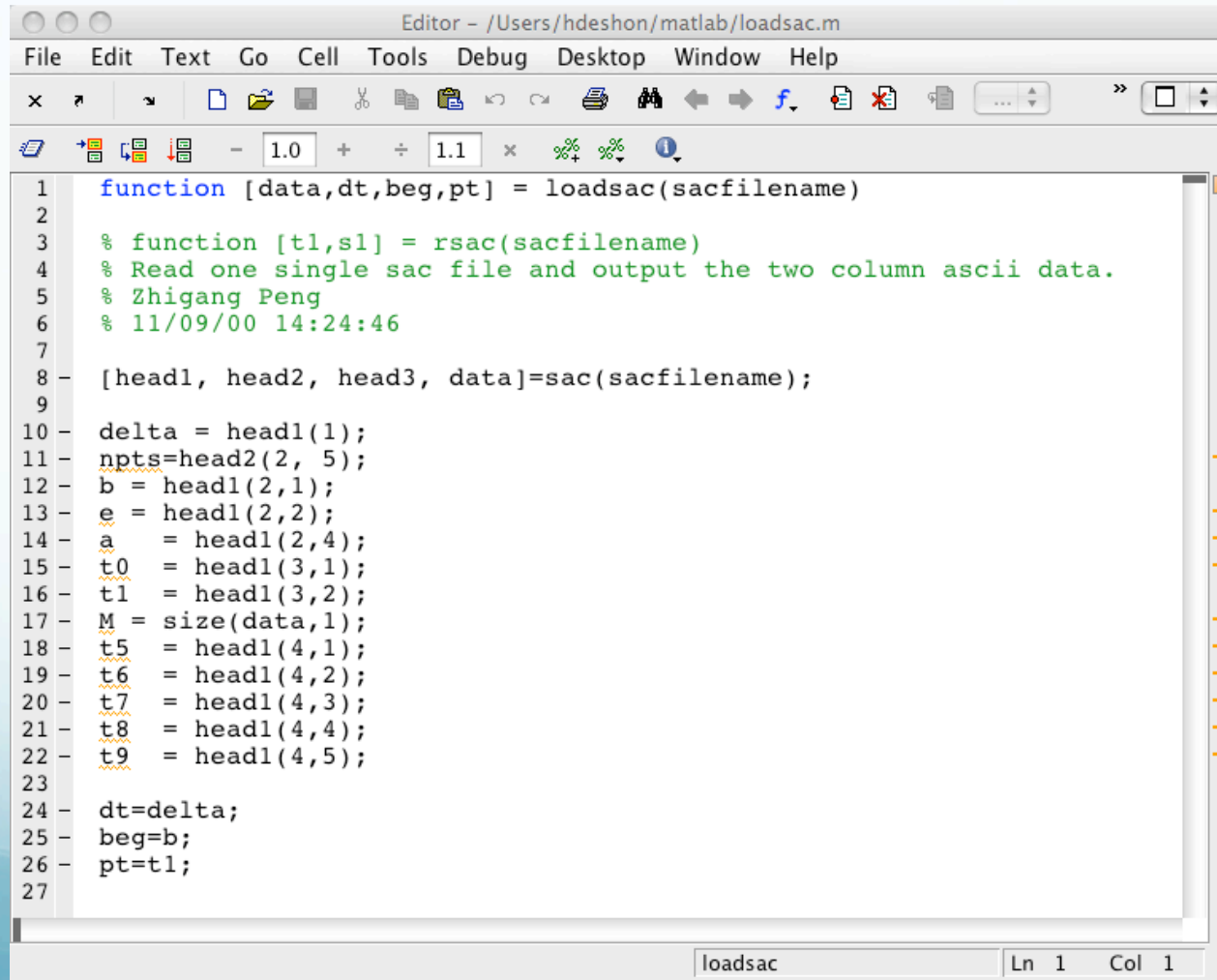
These files are the same things that you would type in when running interactively.

They can have `for` and `while` loops, `if-elseif-else-end`.

They are executed by entering the file name in the matlab command window.

Functions

Functions are M-files that can accept input arguments and return output arguments.
(Comments in Matlab are denoted using the % symbol.)



```
Editor - /Users/hdeshon/matlab/loadsac.m
File Edit Text Go Cell Tools Debug Desktop Window Help
x [Icons] - 1.0 + ÷ 1.1 x % % ?
1 function [data,dt,beg,pt] = loadsac(sacfilename)
2
3 % function [t1,s1] = rsac(sacfilename)
4 % Read one single sac file and output the two column ascii data.
5 % Zhigang Peng
6 % 11/09/00 14:24:46
7
8 [head1, head2, head3, data]=sac(sacfilename);
9
10 delta = head1(1);
11 npts=head2(2, 5);
12 b = head1(2,1);
13 e = head1(2,2);
14 a = head1(2,4);
15 t0 = head1(3,1);
16 t1 = head1(3,2);
17 M = size(data,1);
18 t5 = head1(4,1);
19 t6 = head1(4,2);
20 t7 = head1(4,3);
21 t8 = head1(4,4);
22 t9 = head1(4,5);
23
24 dt=delta;
25 beg=b;
26 pt=t1;
27
```

loadsac Ln 1 Col 1

This function, `loadsac`, calls another function, `sac`, with the filename to read. It then works with the 3 matrices returned by `sac`, returning a data matrix, and 3 scalars `dt`, `beg`, `pt`.

We also see here
that functions
can call other
functions.

But Matlab is not
recursive, so
functions cannot
call themselves.

The screenshot shows the MATLAB Editor window titled "Editor - /Users/hdeshon/matlab/loadSac.m". The menu bar includes File, Edit, Text, Go, Cell, Tools, Debug, Desktop, Window, and Help. Below the menu bar is a toolbar with various icons for file operations, editing, and execution. The main workspace contains the following code:

```

1 function [data,dt,beg,pt] = loadsac(sacfilename)
2
3 % function [t1,s1] = rsac(sacfilename)
4 % Read one single sac file and output the two column ascii data.
5 % Zhigang Peng
6 % 11/09/00 14:24:46
7
8 [head1, head2, head3, data]=sac(sacfilename);
9
10 delta = head1(1);
11 npts=head2(2, 5);
12 b = head1(2,1);
13 e = head1(2,2);
14 a = head1(2,4);
15 t0 = head1(3,1);
16 t1 = head1(3,2);
17 M = size(data,1);
18 t5 = head1(4,1);
19 t6 = head1(4,2);
20 t7 = head1(4,3);
21 t8 = head1(4,4);
22 t9 = head1(4,5);
23
24 dt=delta;
25 beg=b;
26 pt=t1;
27

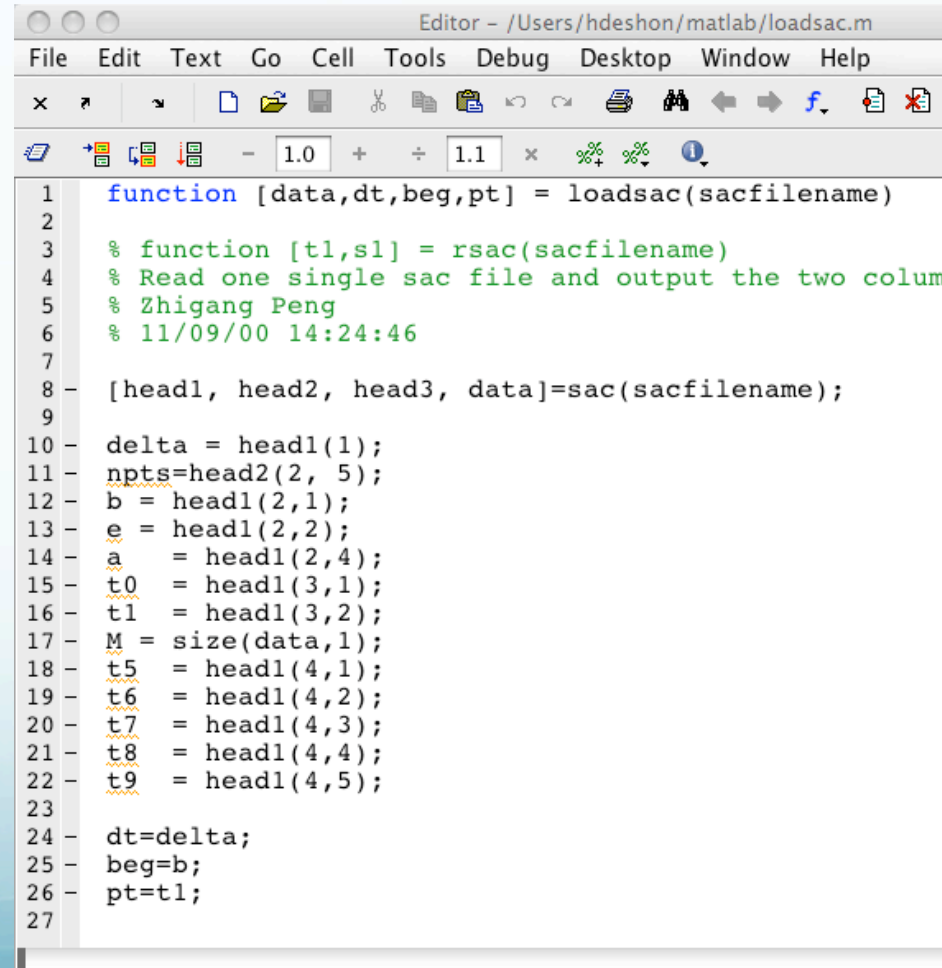
```

The status bar at the bottom indicates the current position as "loadsac" and "Ln 1 Col 1".

One of the biggest differences between a function and a "regular" m-file is that the set of variables available to the function and routine that called the function are independent.

loadsac gets a variable with the file name of the file we want to process from the calling program.

loadsac does not "see" or know about any other variables in the calling program.

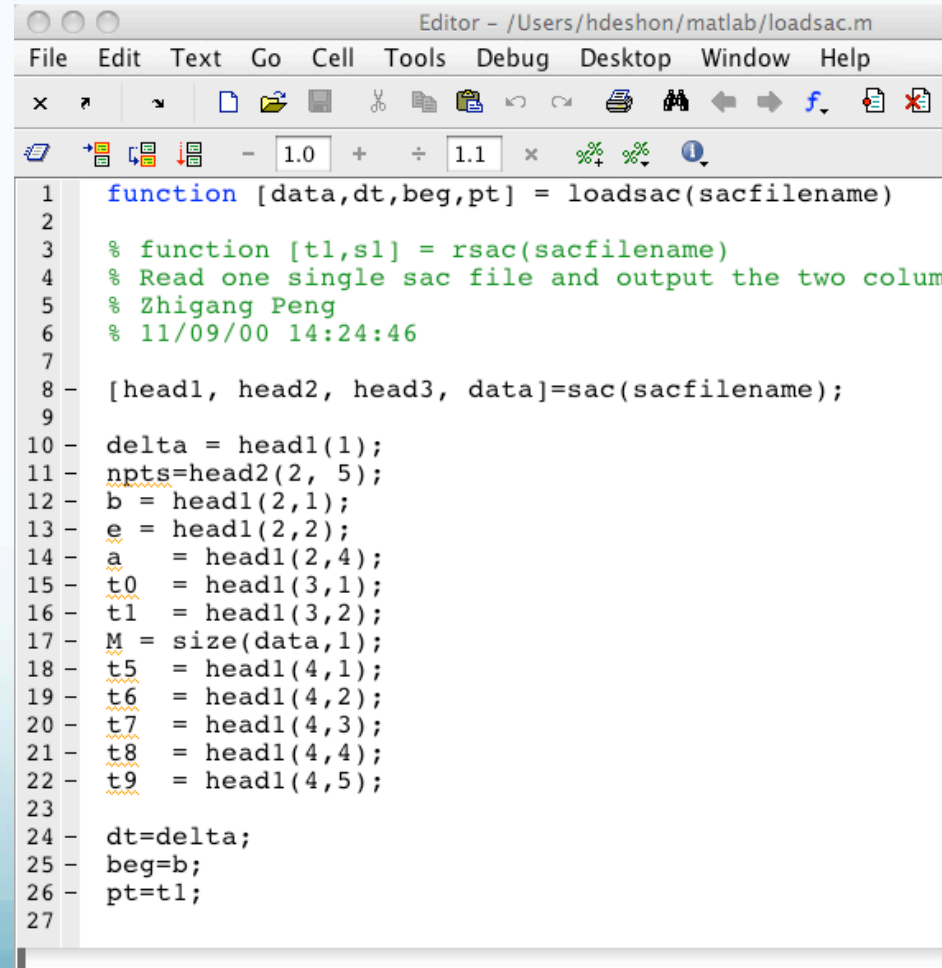


```
Editor - /Users/hdeshon/matlab/loadsac.m
File Edit Text Go Cell Tools Debug Desktop Window Help
x [Icons] - 1.0 + ÷ 1.1 x [Icons]
1 function [data,dt,beg,pt] = loadsac(sacfilename)
2
3 % function [t1,s1] = rsac(sacfilename)
4 % Read one single sac file and output the two column
5 % Zhigang Peng
6 % 11/09/00 14:24:46
7
8 [head1, head2, head3, data]=sac(sacfilename);
9
10 delta = head1(1);
11 npts=head2(2, 5);
12 b = head1(2,1);
13 e = head1(2,2);
14 a = head1(2,4);
15 t0 = head1(3,1);
16 t1 = head1(3,2);
17 M = size(data,1);
18 t5 = head1(4,1);
19 t6 = head1(4,2);
20 t7 = head1(4,3);
21 t8 = head1(4,4);
22 t9 = head1(4,5);
23
24 dt=delta;
25 beg=b;
26 pt=t1;
27
```

loadsac returns four variables to the calling program with the data and some metadata (time between samples, start time, number samples) it got from the SAC file header.

The other variables in loadsac are invisible "outside" loadsac, the calling program does not know about or see them.

The calling program only sees the variables returned (inside the [])



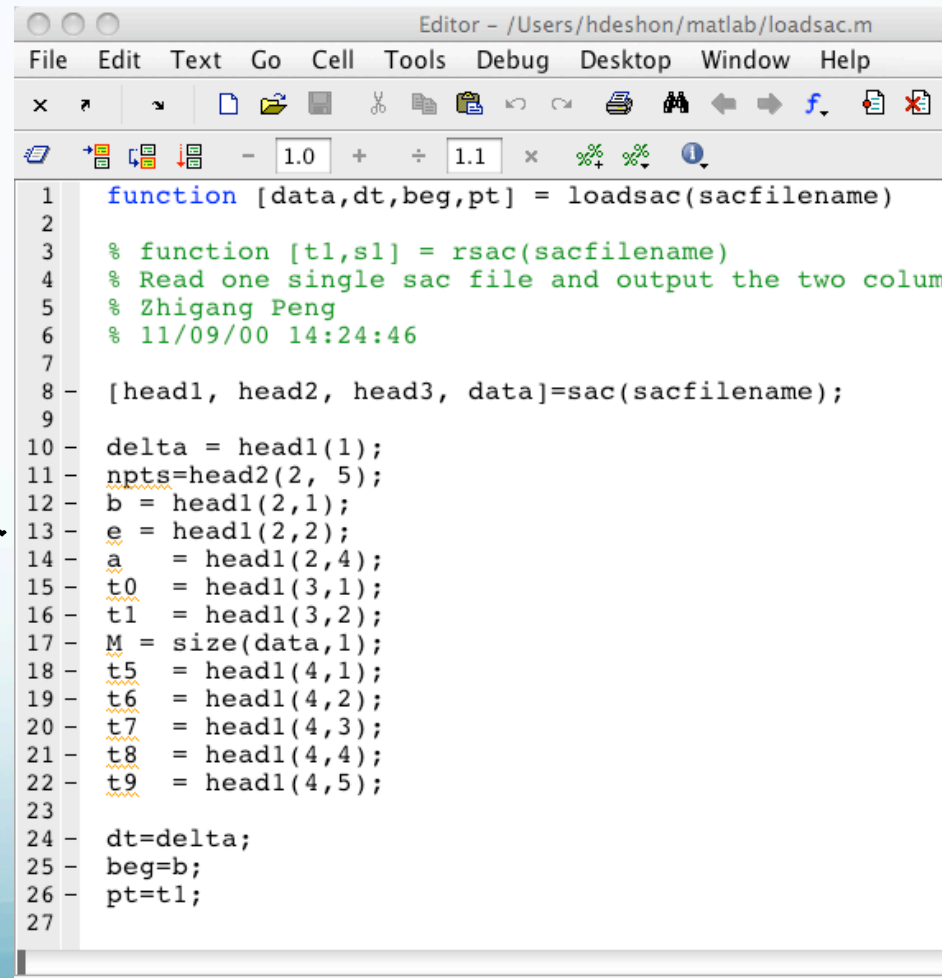
```
Editor - /Users/hdeshon/matlab/loadsac.m
File Edit Text Go Cell Tools Debug Desktop Window Help
x [Icons] - 1.0 + ÷ 1.1 x [Icons]
1 function [data,dt,beg,pt] = loadsac(sacfilename)
2
3 % function [t1,s1] = rsac(sacfilename)
4 % Read one single sac file and output the two column
5 % Zhigang Peng
6 % 11/09/00 14:24:46
7
8 [head1, head2, head3, data]=sac(sacfilename);
9
10 delta = head1(1);
11 npts=head2(2, 5);
12 b = head1(2,1);
13 e = head1(2,2);
14 a = head1(2,4);
15 t0 = head1(3,1);
16 t1 = head1(3,2);
17 M = size(data,1);
18 t5 = head1(4,1);
19 t6 = head1(4,2);
20 t7 = head1(4,3);
21 t8 = head1(4,4);
22 t9 = head1(4,5);
23
24 dt=delta;
25 beg=b;
26 pt=t1;
27
```

The calling program has to know what kind of variables are required in the "call" and what kinds of variables are returned (and possibly if the variables are passed by value or reference [address]).

So the calling program might contain

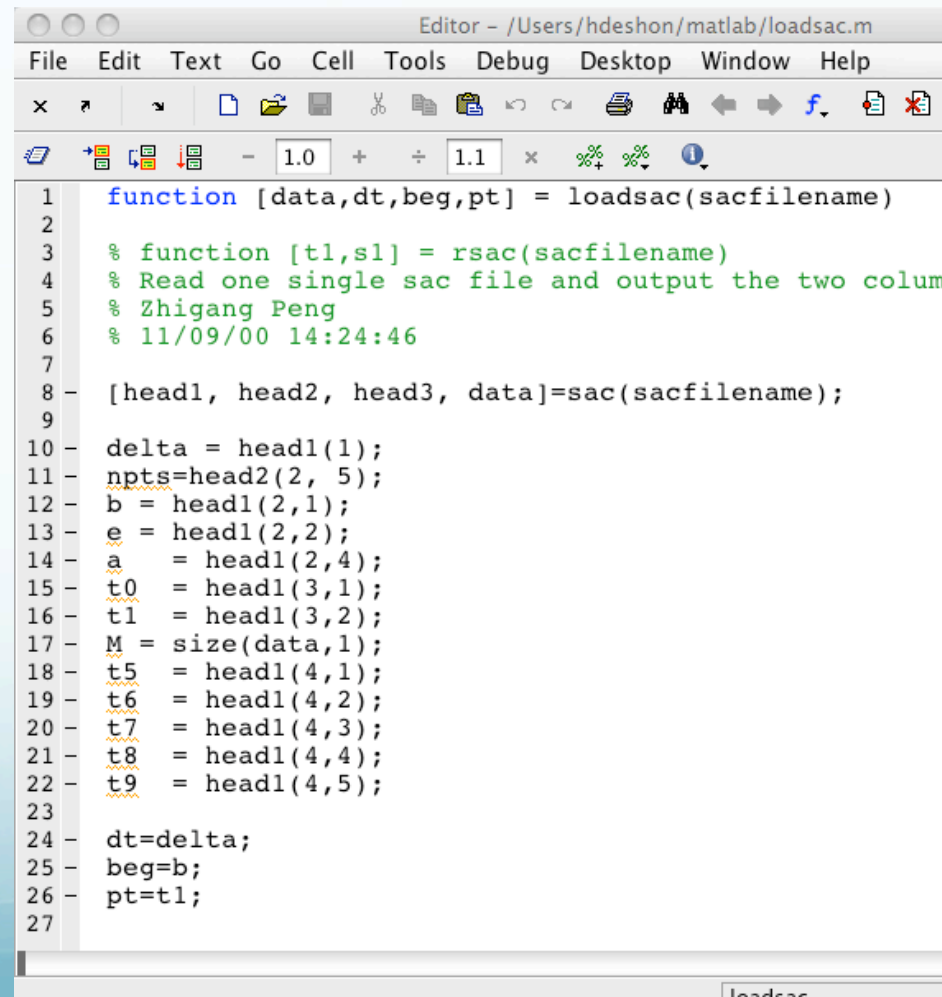
```
[c(:,3),delta,start,numberpts]=...  
loadsac('MEM.BHZ.SAC')
```

Notice that the names for things in the calling program and in the function do not have to match.



```
Editor - /Users/hdeshon/matlab/loadsac.m  
File Edit Text Go Cell Tools Debug Desktop Window Help  
x [Icons] 1.0 1.1 x [Icons]  
1 function [data,dt,beg,pt] = loadsac(sacfilename)  
2  
3 % function [t1,s1] = rsac(sacfilename)  
4 % Read one single sac file and output the two column  
5 % Zhigang Peng  
6 % 11/09/00 14:24:46  
7  
8 [head1, head2, head3, data]=sac(sacfilename);  
9  
10 delta = head1(1);  
11 npts=head2(2, 5);  
12 b = head1(2,1);  
13 e = head1(2,2);  
14 a = head1(2,4);  
15 t0 = head1(3,1);  
16 t1 = head1(3,2);  
17 M = size(data,1);  
18 t5 = head1(4,1);  
19 t6 = head1(4,2);  
20 t7 = head1(4,3);  
21 t8 = head1(4,4);  
22 t9 = head1(4,5);  
23  
24 dt=delta;  
25 beg=b;  
26 pt=t1;  
27
```

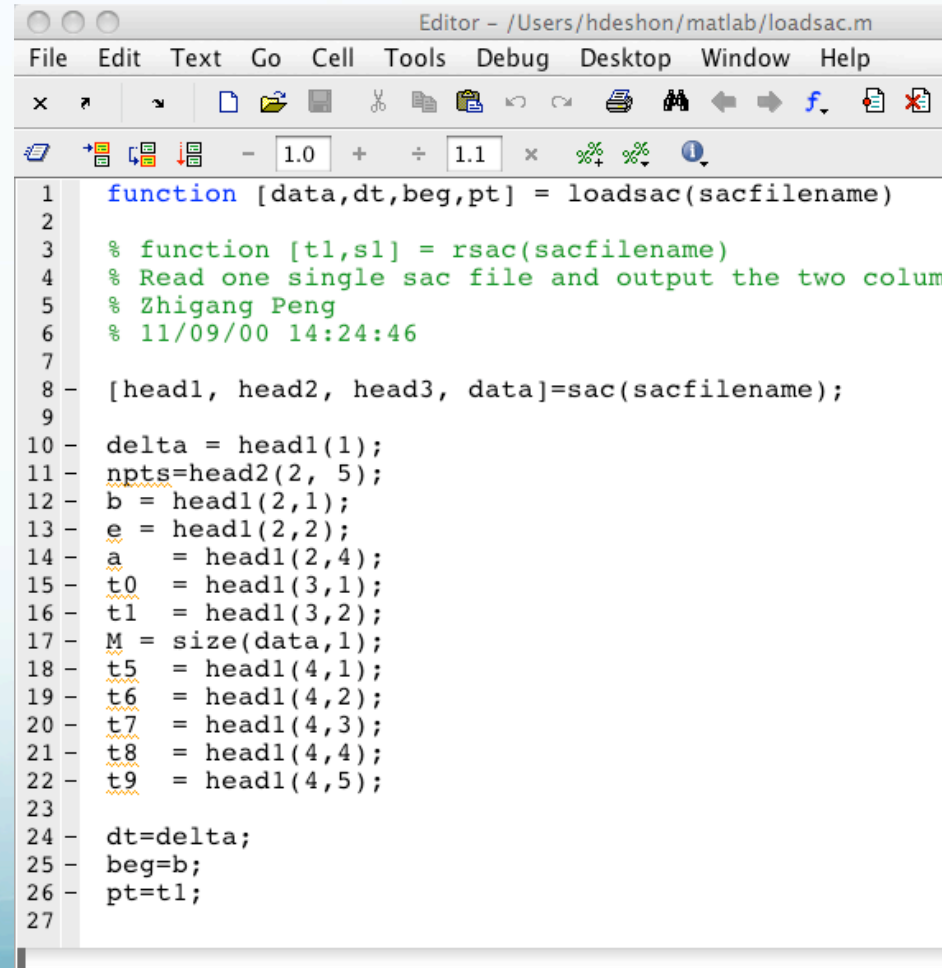
Unless passed into the function or returned, variables in a function are known only to the function (here there are a bunch of them and then don't do anything – they are set, but never used and they are not returned to the calling program)



The screenshot shows a MATLAB Editor window titled 'Editor - /Users/hdeshon/matlab/loadsac.m'. The window has a menu bar (File, Edit, Text, Go, Cell, Tools, Debug, Desktop, Window, Help) and a toolbar with various icons. Below the toolbar is a numeric display showing '1.0' and '1.1'. The main area contains MATLAB code for a function 'loadsac'. The code includes comments and several assignments. The function signature is 'function [data,dt,beg,pt] = loadsac(sacfilename)'. The code uses 'rsac' to read a file, 'sac' to get header information, and then extracts specific fields from the header into variables like 'delta', 'npts', 'b', 'e', 'a', 't0', 't1', 'M', 't5', 't6', 't7', 't8', and 't9'. Finally, it assigns 'dt=delta;', 'beg=b;', and 'pt=t1;'.

```
1 function [data,dt,beg,pt] = loadsac(sacfilename)
2
3 % function [t1,s1] = rsac(sacfilename)
4 % Read one single sac file and output the two column
5 % Zhigang Peng
6 % 11/09/00 14:24:46
7
8 [head1, head2, head3, data]=sac(sacfilename);
9
10 delta = head1(1);
11 npts=head2(2, 5);
12 b = head1(2,1);
13 e = head1(2,2);
14 a = head1(2,4);
15 t0 = head1(3,1);
16 t1 = head1(3,2);
17 M = size(data,1);
18 t5 = head1(4,1);
19 t6 = head1(4,2);
20 t7 = head1(4,3);
21 t8 = head1(4,4);
22 t9 = head1(4,5);
23
24 dt=delta;
25 beg=b;
26 pt=t1;
27
```

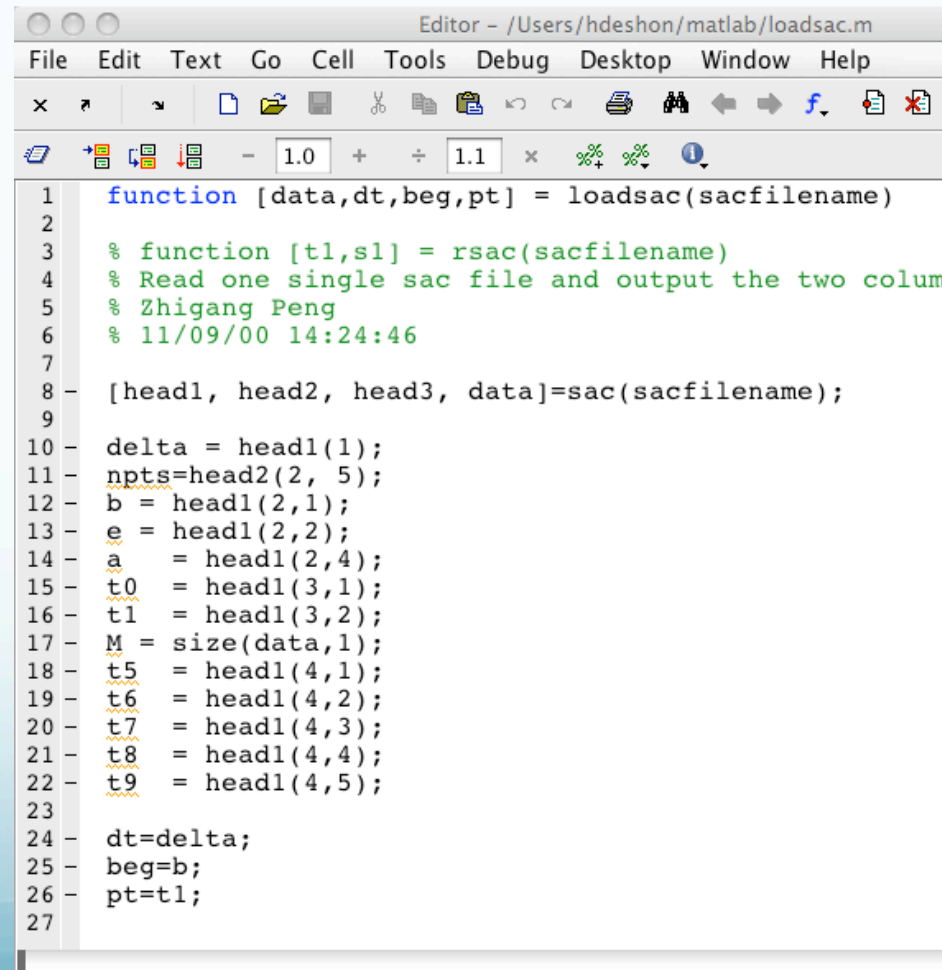

Scalars are passed by value (a copy of the variable) so any changes to such a variable inside the function is not known outside the functions by the calling routine (what happens in functions, stays in functions).



```
Editor - /Users/hdeshon/matlab/loadsac.m
File Edit Text Go Cell Tools Debug Desktop Window Help
x [Icons] - 1.0 + ÷ 1.1 x [Icons]
1 function [data,dt,beg,pt] = loadsac(sacfilename)
2
3 % function [t1,s1] = rsac(sacfilename)
4 % Read one single sac file and output the two column
5 % Zhigang Peng
6 % 11/09/00 14:24:46
7
8 [head1, head2, head3, data]=sac(sacfilename);
9
10 delta = head1(1);
11 npts=head2(2, 5);
12 b = head1(2,1);
13 e = head1(2,2);
14 a = head1(2,4);
15 t0 = head1(3,1);
16 t1 = head1(3,2);
17 M = size(data,1);
18 t5 = head1(4,1);
19 t6 = head1(4,2);
20 t7 = head1(4,3);
21 t8 = head1(4,4);
22 t9 = head1(4,5);
23
24 dt=delta;
25 beg=b;
26 pt=t1;
27
```

Arrays are passed by reference or address. The function gets the address with the starting location of the array (and some information about it's size)

Any thing done to the array from within the function is seen by the calling routine and all other functions that use that array.

A screenshot of a MATLAB editor window titled 'Editor - /Users/hdeshon/matlab/loadsac.m'. The window has a standard menu bar (File, Edit, Text, Go, Cell, Tools, Debug, Desktop, Window, Help) and a toolbar with various icons. Below the toolbar is a command window area showing zoom levels (1.0, 1.1) and other controls. The main area displays the source code of a function named 'loadsac'. The code is as follows:

```
1 function [data,dt,beg,pt] = loadsac(sacfilename)
2
3 % function [t1,s1] = rsac(sacfilename)
4 % Read one single sac file and output the two column
5 % Zhigang Peng
6 % 11/09/00 14:24:46
7
8 [head1, head2, head3, data]=sac(sacfilename);
9
10 delta = head1(1);
11 npts=head2(2, 5);
12 b = head1(2,1);
13 e = head1(2,2);
14 a = head1(2,4);
15 t0 = head1(3,1);
16 t1 = head1(3,2);
17 M = size(data,1);
18 t5 = head1(4,1);
19 t6 = head1(4,2);
20 t7 = head1(4,3);
21 t8 = head1(4,4);
22 t9 = head1(4,5);
23
24 dt=delta;
25 beg=b;
26 pt=t1;
27
```


Global Variables

(another way to pass variables between functions and the main routine)

If you want more than one function to share a single copy of a variable, simply declare the variable as global in all the functions.

Do the same thing at the command line if you want the base workspace to access the variable.

The global declaration must occur before the variable is actually used in a function.

Although it is not required, using capital letters for the names of global variables helps distinguish them from other variables.

Global variables

When you define a variable at the matlab prompt, it is defined inside of matlab's "workspace."

Running a script does not affect this, since a script is just a collection of commands, and they're actually run from the same workspace.

If you define a variable in a script, it will stay defined in the workspace.

Global variables

Functions, on the other hand, do not share the same workspace.

A function won't know what a variable is unless

- the function gets the variable as an argument, or
- the variable is defined as a variable that is shared by the function and the matlab workspace, i.e. a global variable.

Global variables

To use a global variable, every place (function, script, or at the matlab prompt) that needs to share that variable must have a line near the top identifying it as a global variable, ie:

```
global phi;
```

Then when the variable is assigned a value in one of those places, it will have a value in all the other places that have the global statement.

In an .m file called falling.m

```
function h = falling(t)
global GRAVITY
h = 1/2*GRAVITY*t.^2;
```

In the workspace, enter the statements

```
>> global GRAVITY
>> GRAVITY = 32;
>> y = falling((0:.1:5)');
```

The two global statements make the value assigned to **GRAVITY** at the command prompt available inside the function. You can then modify **GRAVITY** interactively and obtain new solutions without editing any files.

In an .m file called falling.m

```
function h = falling(t)
global GRAVITY
h = 1/2*GRAVITY*t.^2;
```

In the workspace, enter the statements

```
>> global GRAVITY
>> GRAVITY = 32;
>> y = falling((0:.1:5)');
```

This also shows the basic function syntax.

```
function [output variables] =... FunctionName(input variables)
body of function
```

There is no statement to end the function (no return or
end needed, uses EOF or new definition)

return

return: returns to invoking function

allows for termination of function 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
```

```
function tsting
global c
c=4
b=2
[a d]=tstfn(b)
whos
return
function [out1 out2]=tstfn(in)
global c
out1=in.^2
out2=c*out1
whos
return
```

Can put multiple functions in one m file.

The variables “b” and “c” are declared global in both, but only assigned in one of them.

```
>> tstingfuns
```

```
c =
    4
b =
    2
out1 =
    4
out2 =
   16
```

Name	Size	Bytes	Class	Attributes
c	1x1	8	double	global
in	1x1	8	double	
out1	1x1	8	double	
out2	1x1	8	double	

```
a =
    4
d =
   16
```

Name	Size	Bytes	Class	Attributes
a	1x1	8	double	
b	1x1	8	double	
c	1x1	8	double	global
d	1x1	8	double	

Rethinking code for taking advantage of matlab vectorization.

More than just defining vectors and matrices using matlab definitions.

```
%cheating a little – creation of x is vectorized  
x = rand(1,100);
```

```
%In place of :  
for k=1:100  
y(k) = sin(x(k));  
end
```

```
% We can use :  
y = sin(x);
```

Given $a_n = n$ and $b_n = (1000 - a_n)$, $n = 1, 2, \dots, 1000$,
calculate

$$ssum = \sum_{n=1}^{1000} a_n b_n$$

Solution: It might be tempting to implement the
above calculation as

```
a = 1:1000;  
b = 1000 - a;  
ssum=0;  
for n=1:1000 %poor style...  
    ssum = ssum +a(n)*b(n);  
end
```

Given $a_n = n$ and $b_n = (1000 - a_n)$, $n = 1, 2, \dots, 1000$, calculate

$$ssum = \sum_{n=1}^{1000} a_n b_n$$

Recognizing that the desired sum/calculation is the inner or dot product of the vectors a and b , or multiplication of the matrices ab^T , we can do better (we even have a more than one way to do it!):

```
ssum = a*b'           %Vectorized, better!  
ssum = dot(a,b)       %Vectorized, better!
```

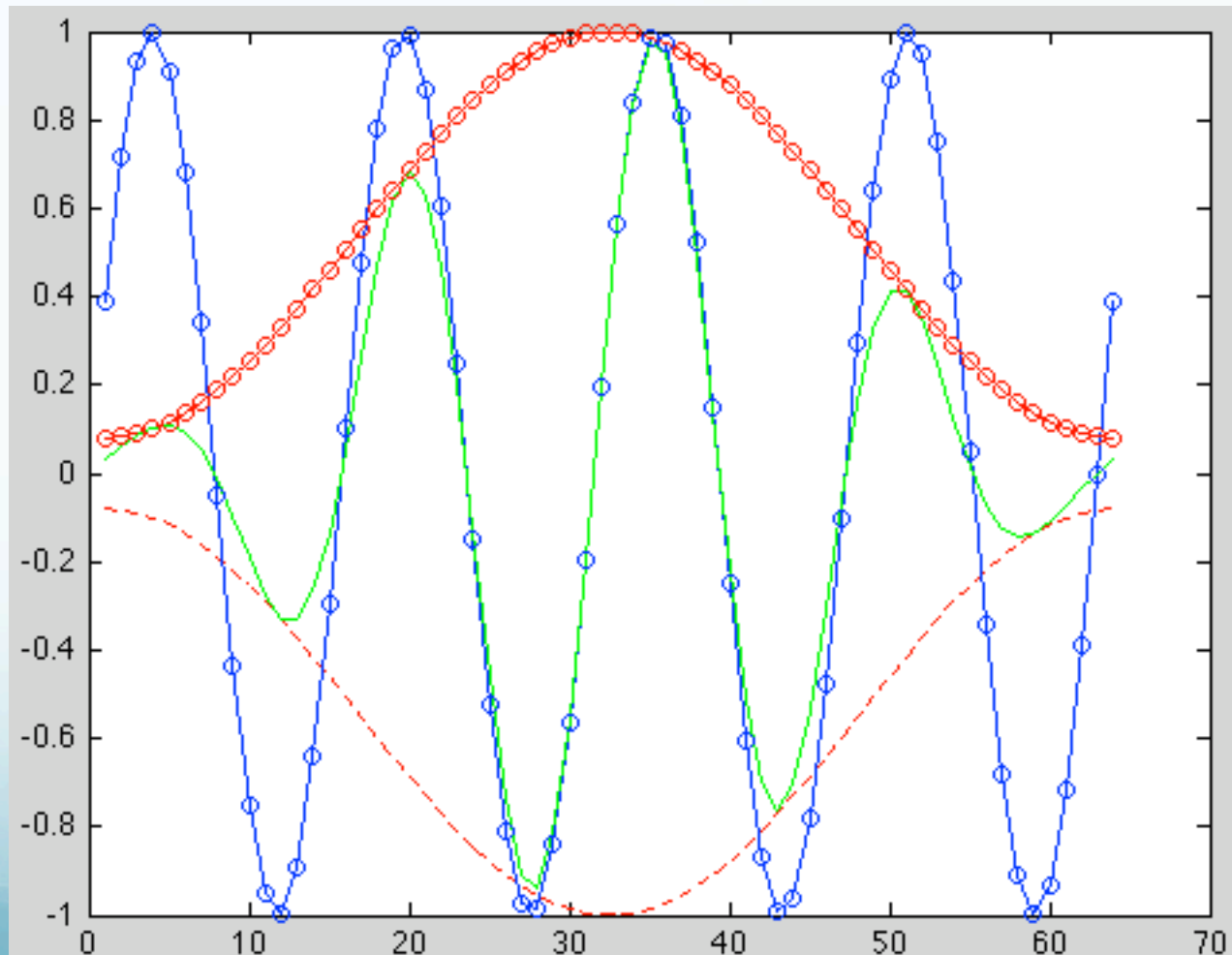
Say we have a number of seismograms and we would like to “window” and scale each one.

First – what is the “window” process?

(Blue trace is original signal, red trace is “window”, dashed red trace is negative of window to show “envelope” – will use it to scale the original signal.

Green trace is the final result after applying the “window” to the blue trace.

In this case the windowing is done using a point by point multiply of the blue and red vectors, $b \cdot r$, [64 pts]).



What if we want to do this to a number of seismograms?

We could use a loop, doing the vectorized multiply on each seismogram.

But can we do better than that?

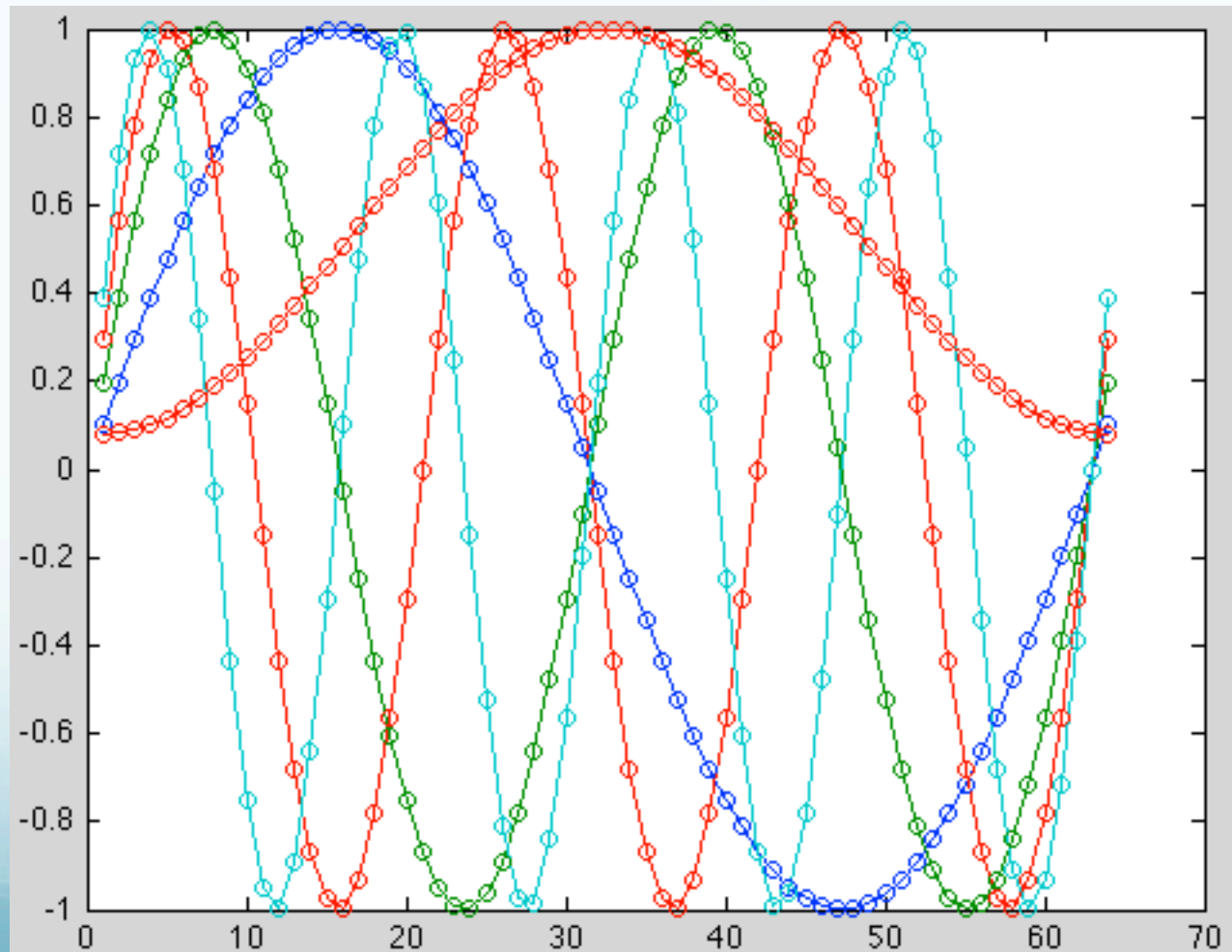
We would not be asking leading questions if not!

So how do we do it?

(Now we want to do point
by point multiplies of each
trace by the window ~
 $T_1 \cdot w$ and $T_2 \cdot w$, etc.

How can we do this in one
shot?

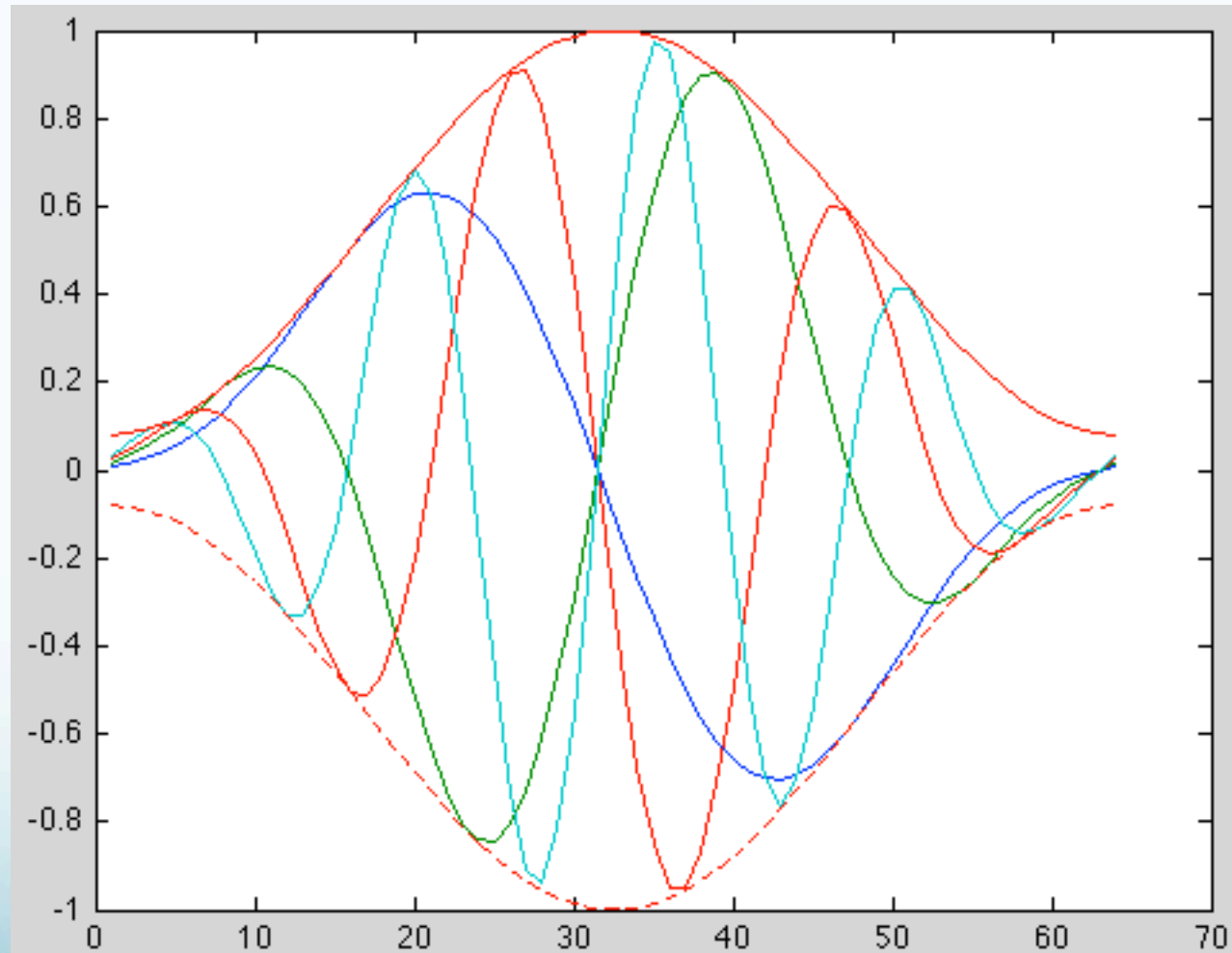
What if we make a diagonal
matrix of the window vector
[elements of the vector on
the diagonal and all else
zero]?)



Looking at what happens when we do matrix multiplication, we see that this does what we need.

```
% length
%Number traces
N = 64;
M=1:4;
TH=1:N;
X = sin(TH'*M*...
pi*2/(N-1));
plot(X)

% Make a window
w = hamming(N);
W = diag(w);
% Windowed signals
%without loops
XW = W * X;
```



To do the point by point multiply we need to match the length of the seismograms (64 points in this case).

```
>> whos
```

Name	Size	Bytes	Class	Attributes
M	1x4	32	double	
N	1x1	8	double	
TH	1x64	512	double	
W	64x64	32768	double	
X	64x4	2048	double	
XW	64x4	2048	double	
g	1x4	32	double	
w	64x1	512	double	

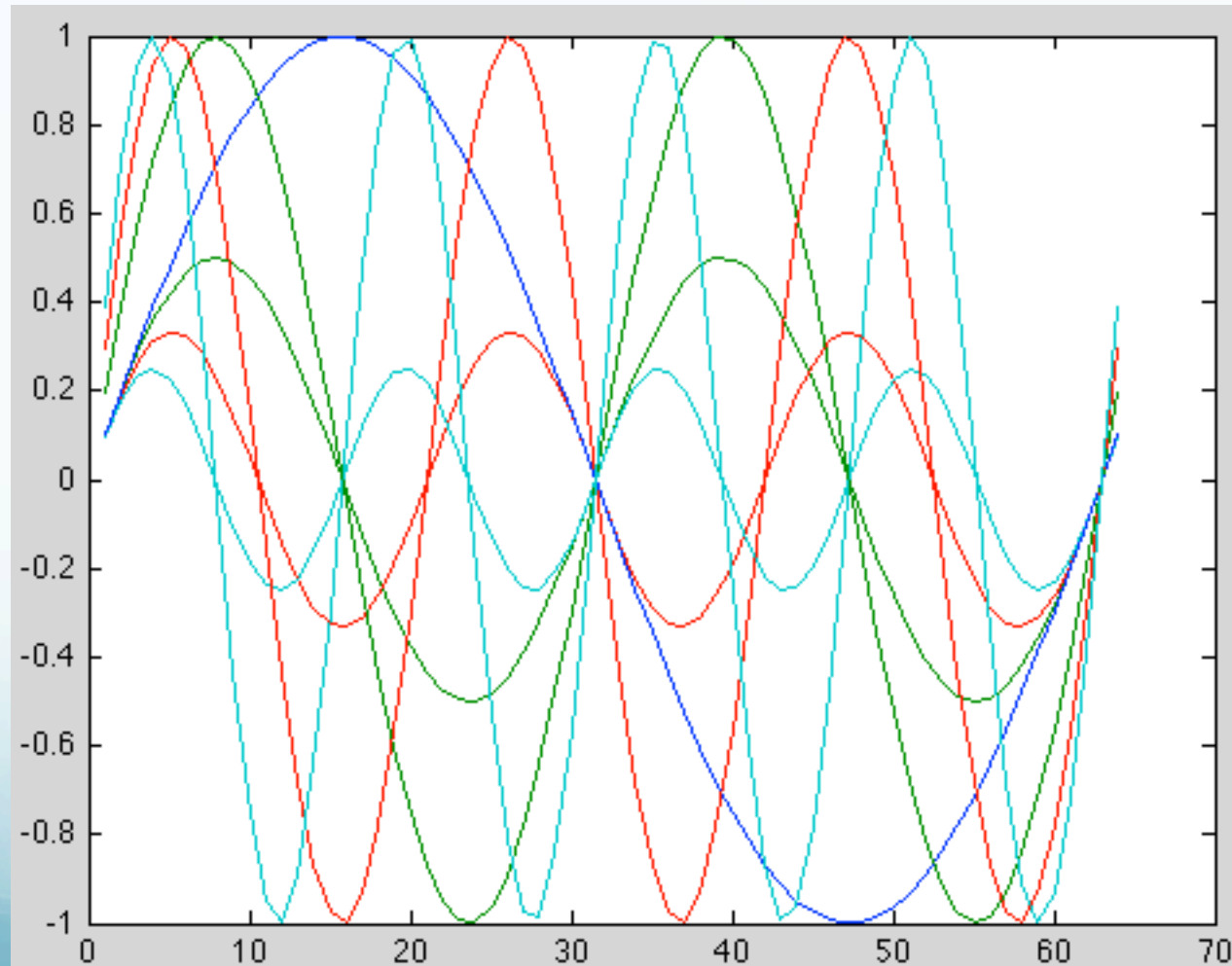
So we have a 64x64 matrix * 64x4 matrix producing a 64x4 result matrix .

Say we want to scale each seismogram (there are 4 of them). We have to multiply each point in the seismogram by the same number).

(Now we want to multiply each trace by its scale – $T_1 * w_1$ and $T_2 * w_2$, etc. How can we do this in one shot?

What if we make a diagonal matrix of the weights [elements of the vector on the diagonal and all else zero]?)

(really same as last time, but elements on diagonal are now scaling weights for each seismogram – boxcar window of height other than 1)



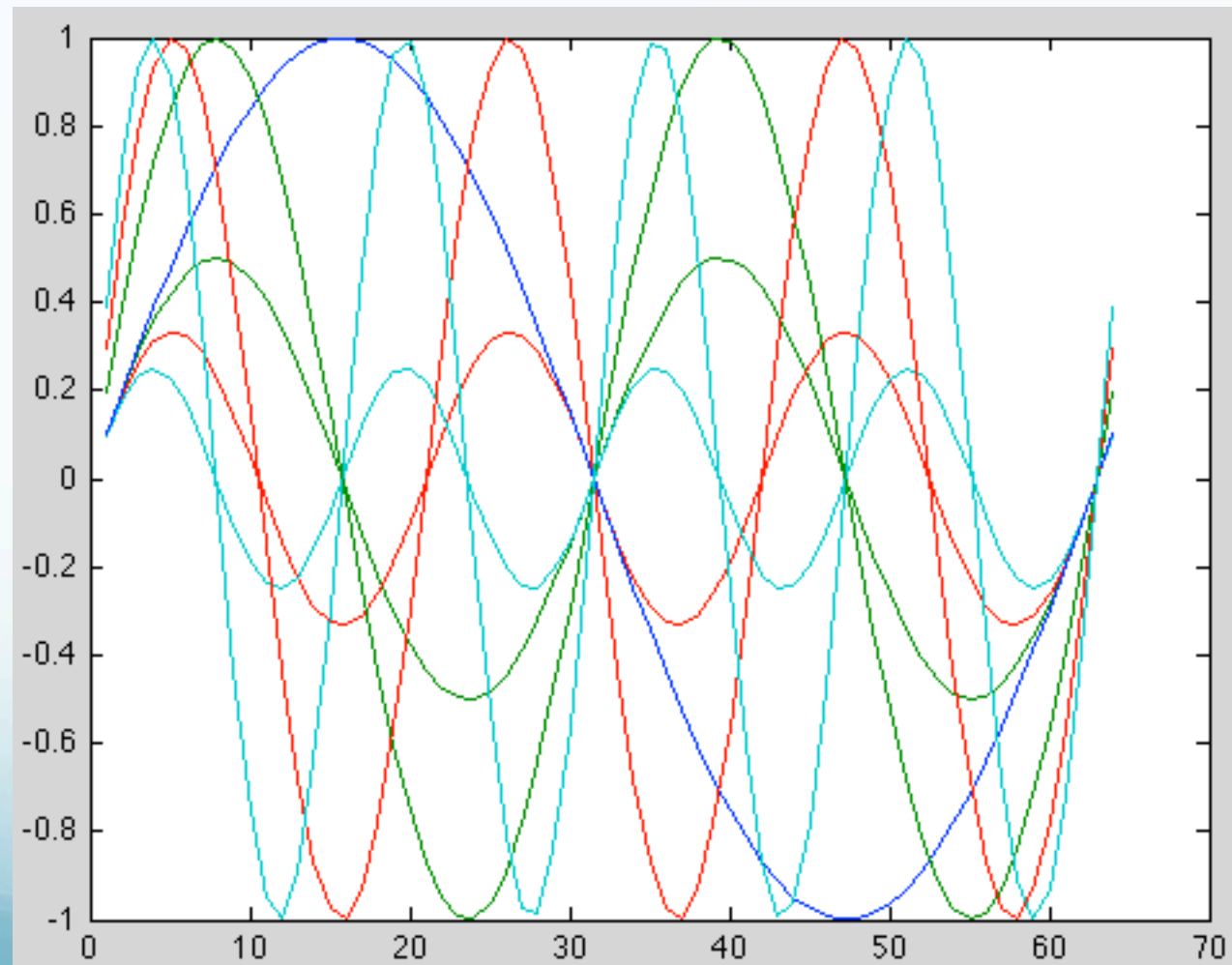
```
% Make a vector of gain factors
```

```
g = 1./M;
```

```
G = diag(g);
```

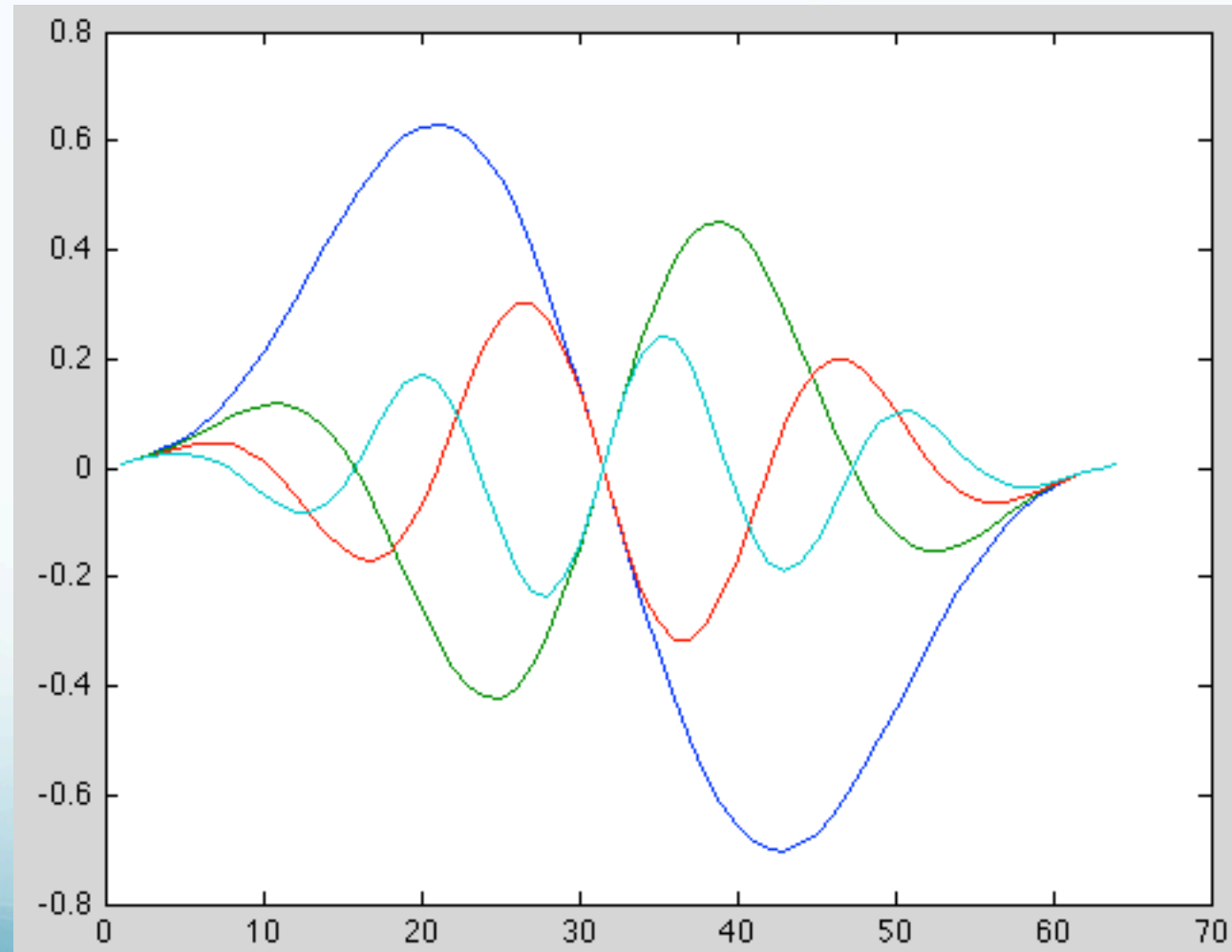
```
% scale each seismogram by corresponding gain factor
```

```
XG = X * G;
```



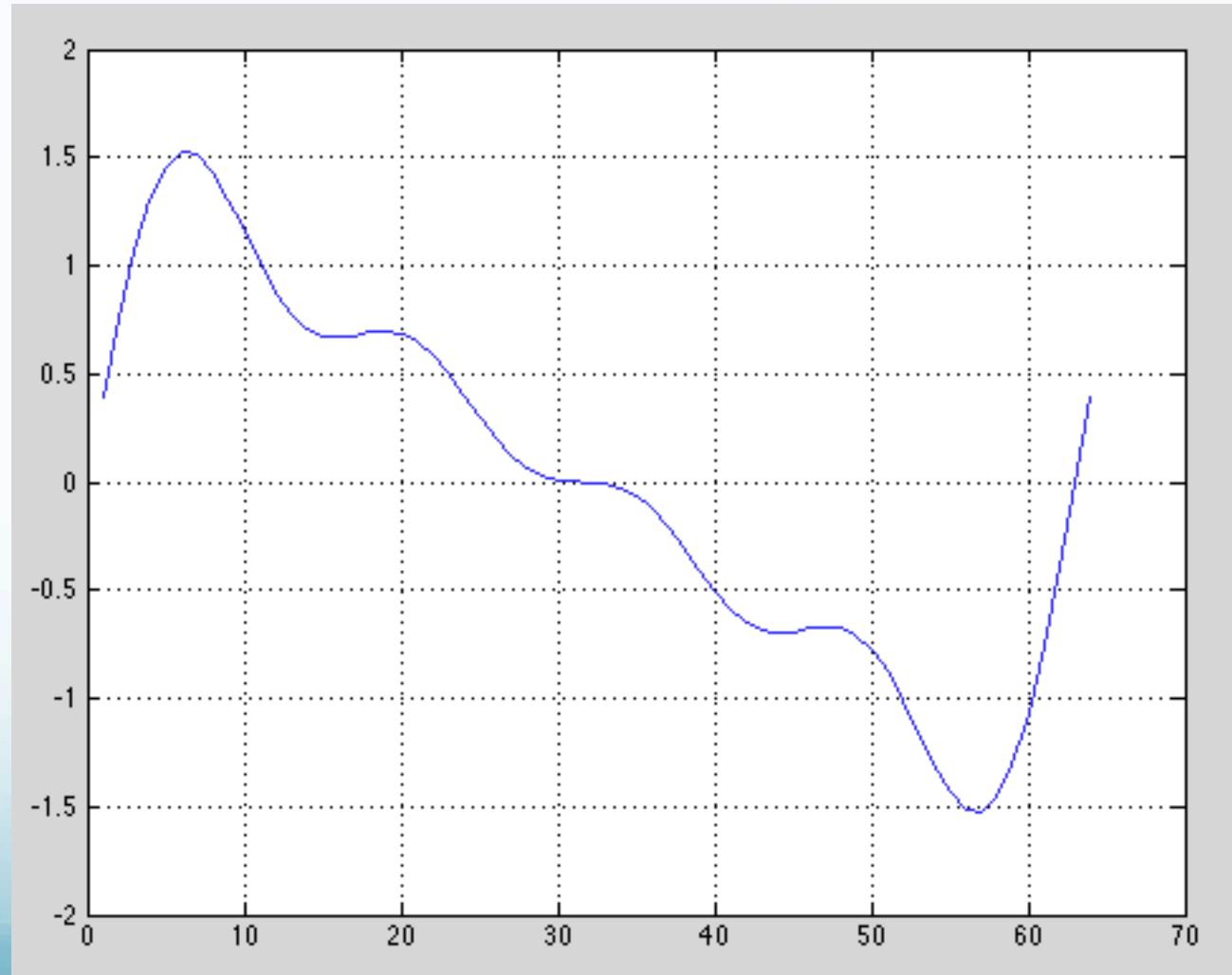
Do both windowing and gain scaling together.

```
% Windowing and gain scaling is just left and right  
% multiplication with appropriate diagonal matrices.  
XWG = W * X * G;
```



Sum the basis functions to get the Fourier series.

```
>> s=sum(XG' );  
>> plot(s)  
>> grid
```



Notice that vectorization requires a different thought pattern in the approach to solving problems.

It will initially take longer to develop the program.

But with practice (effort!, and seeing/looking for examples), it will become more natural and faster to code.

(If doing the previous example “for real” one would use the sparse matrix feature for the diagonal matrices

```
W = diag(sparse(w));
```

This will save on both memory use and execution time.)

Why Use Vectorized Code?

Advantages

Increased Speed:

Vectorized code runs significantly faster.

How much faster?

This depends on the commands used and the application. And although Matlab has made great strides in accelerating low-level code, vectorized code still runs faster. But in general, vectorized code is faster than its low-level counterpart.

Why Use Vectorized Code?

Advantages

Compact: Vectorized code is more compact and can be easier to read and understand.

("Looks-like" the underlying Math)

Why Use Vectorized Code?

Disadvantages

Difficulty:

Most people with programming experience are used to doing things in a low-level manner (i.e. FOR loops).

Vectorizing code can be a challenge because of the different thinking that is required. In addition, there is no set formula on how to vectorize code.

A good working knowledge of the available functions within Matlab is certainly helpful when it comes to vectorization.

Why Use Vectorized Code?

Disadvantages

Compact:

Being compact is both a blessing and a curse.

Vectorized code can be difficult to understand because it is so compact.

If the code is undocumented and does not have any comments, it can be a real pain to figure what the code does.

(but probably not worse than any other undocumented code.

As least it is not spaghetti code – can't write spaghetti code in Matlab since don't have a goto).

Although vectorization can make your code simpler at times, it can also make your code archaic and difficult to understand.

In addition, it can be difficult to vectorize your code at times, so it may not be worth the time and effort to do so.

Thus, you may be wondering if it is really worth your time to vectorize your code.

If you find it too difficult to vectorize your code,
you may be better off just using a low level
method.

The most important thing is to make sure that
your code works!

After you get your code working, you can
consider optimizing it through vectorization.

In conclusion, vectorization is not required, but it can certainly be beneficial.

Unless the arrays you are dealing with are quite large (and depending on the operations performed), it can be difficult to see the benefits of vectorization.

But in general, I believe that it's good practice to get into the habit of using vectorized code as it is more efficient.

Another look at arithmetic between a matrix and a vector.

Saw how to do it using

- repmat
- Tony's trick
- Multiply by matrix of ones
(plus a few others).

`bsxfun` - Apply element-by-element binary operation to two arrays with singleton expansion enabled

Syntax

`C = bsxfun(fun,A,B)`

Description

`C = bsxfun(fun,A,B)` applies an element-by-element binary operation to arrays `A` and `B`, with singleton expansion enabled.

The inputs must be of the following types:
numeric, logical, char, struct, cell.

`fun` is a function handle, and can either be a MATLAB function or one of the following built-in functions:

The size of the output array `C` is equal to:

`max(size(A),size(B)).*(size(A)>0 & size(B)>0).`

@plus	Plus
@minus	Minus
@times	Array multiply
@rdivide	Right array divide
@ldivide	Left array divide
@power	Array power
@max	Binary maximum
@min	Binary minimum
@rem	Remainder after division
@mod	Modulus after division
@atan2	Four quadrant inverse tangent
@hypot	Square root of sum of squares
@eq	Equal
@ne	Not equal
@lt	Less than
@le	Less than or equal to
@gt	Greater than
@ge	Greater than or equal to
@and	Element-wise logical AND
@or	Element-wise logical OR
@xor	Logical exclusive OR

@plus	Plus
@minus	Minus
@times	Array multiply
@rdivide	Right array divide
@ldivide	Left array divide
@power	Array power
@max	Binary maximum
@min	Binary minimum
@rem	Remainder after division
@mod	Modulus after division
@atan2	Four quadrant inverse tangent
@hypot	Square root of sum of squares
@eq	Equal
@ne	Not equal
@lt	Less than
@le	Less than or equal to
@gt	Greater than
@ge	Greater than or equal to
@and	Element-wise logical AND
@or	Element-wise logical OR
@xor	Logical exclusive OR

Example, use `bsxfun` to subtract the column means from the corresponding columns of matrix `A`.

```
A = magic(5); A = bsxfun(@minus, A, mean(A));
```

`bsxfun` is even faster than building the matrices and does not use as much memory (speed increase comes from not having to build the matrix, also saves space).

Problem is that it is not very readable.

Matlab

Graphics

Basics

Types of Graphics

Predefined graph types, or
Create your own graphics

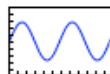
Creating a Graph

Use plotting tools to create graphs interactively.

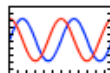
Use the command interface to enter commands in the Command Window or create plotting programs.

Line Graphs

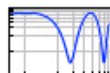
[plot](#)



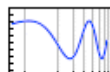
[plotyy](#)



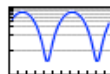
[loglog](#)



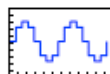
[semilogx](#)



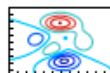
[semilogy](#)



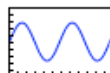
[stairs](#)



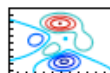
[contour](#)



[ezplot](#)

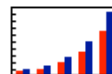


[ezcontour](#)

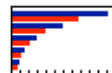


Bar Graphs

[bar](#) (grouped)



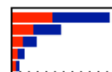
[barh](#) (grouped)



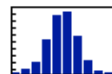
[bar](#) (stacked)



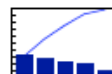
[barh](#) (stacked)



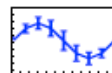
[hist](#)



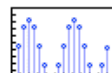
[pareto](#)



[errorbar](#)

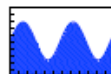


[stem](#)



Area Graphs

[area](#)



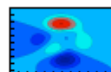
[pie](#)



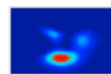
[fill](#)



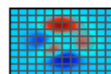
[contourf](#)



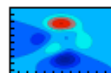
[image](#)



[pcolor](#)

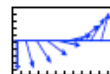


[ezcontourf](#)

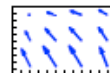


Direction Graphs

[feather](#)



[quiver](#)



[comet](#)



Radial Graphs

[polar](#)



[rose](#)



[compass](#)

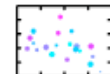


[ezpolar](#)



Scatter Graphs

[scatter](#)



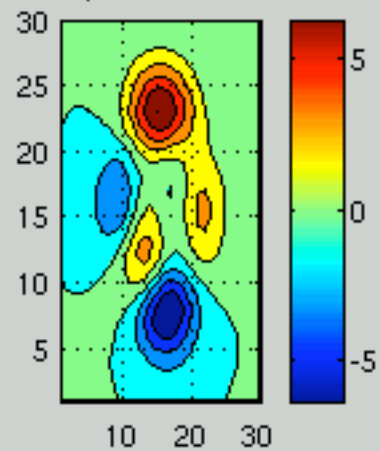
[spy](#)



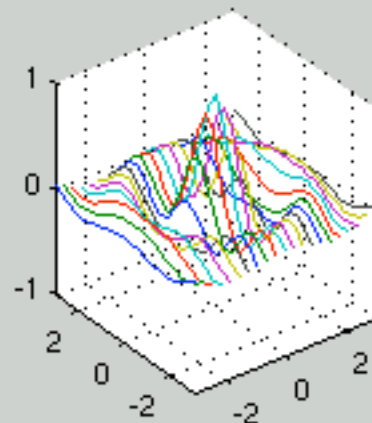
[plotmatrix](#)



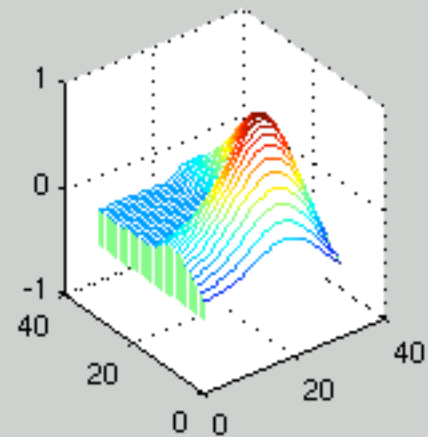
Peaks Function - (CONTOURF & COLORBAR)



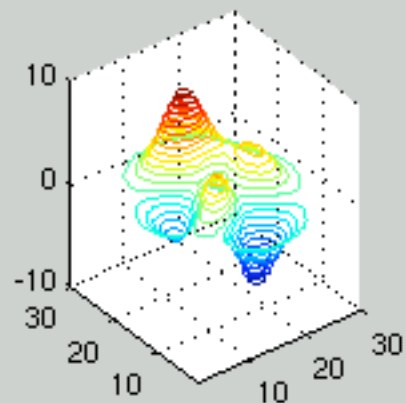
Sinc Function - (PLOT3)



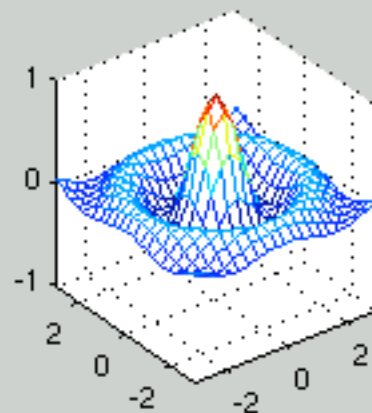
L-shaped Membrane - (WATERFALL)



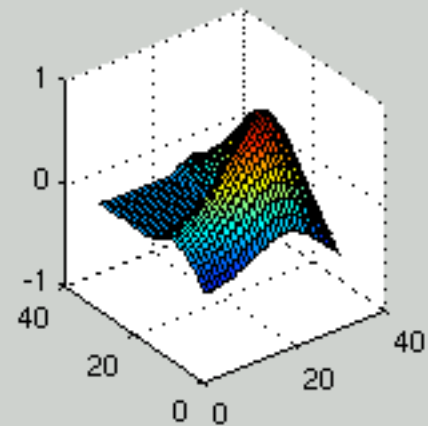
Peaks Function - (CONTOUR3)



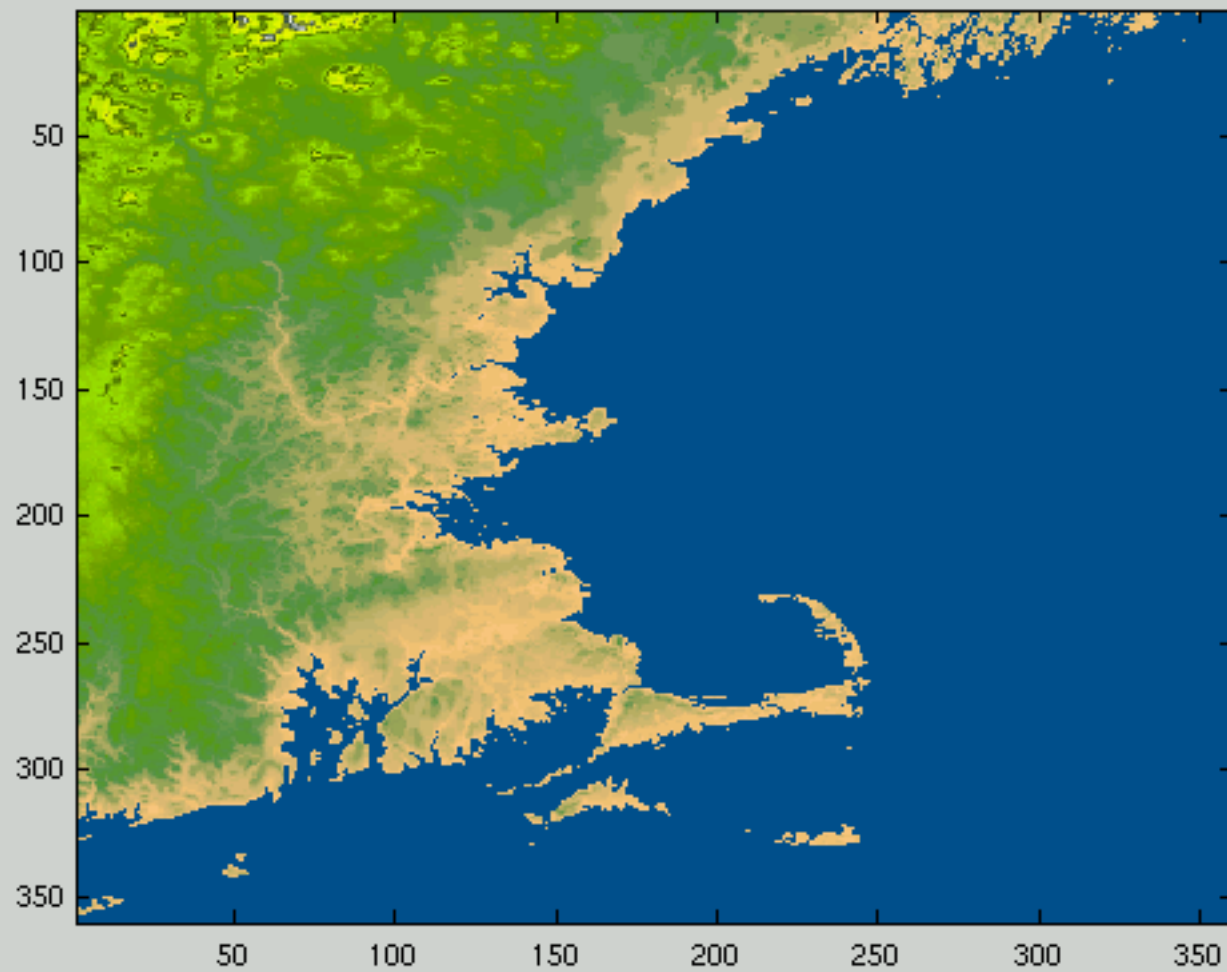
Sinc Function - (MESH)

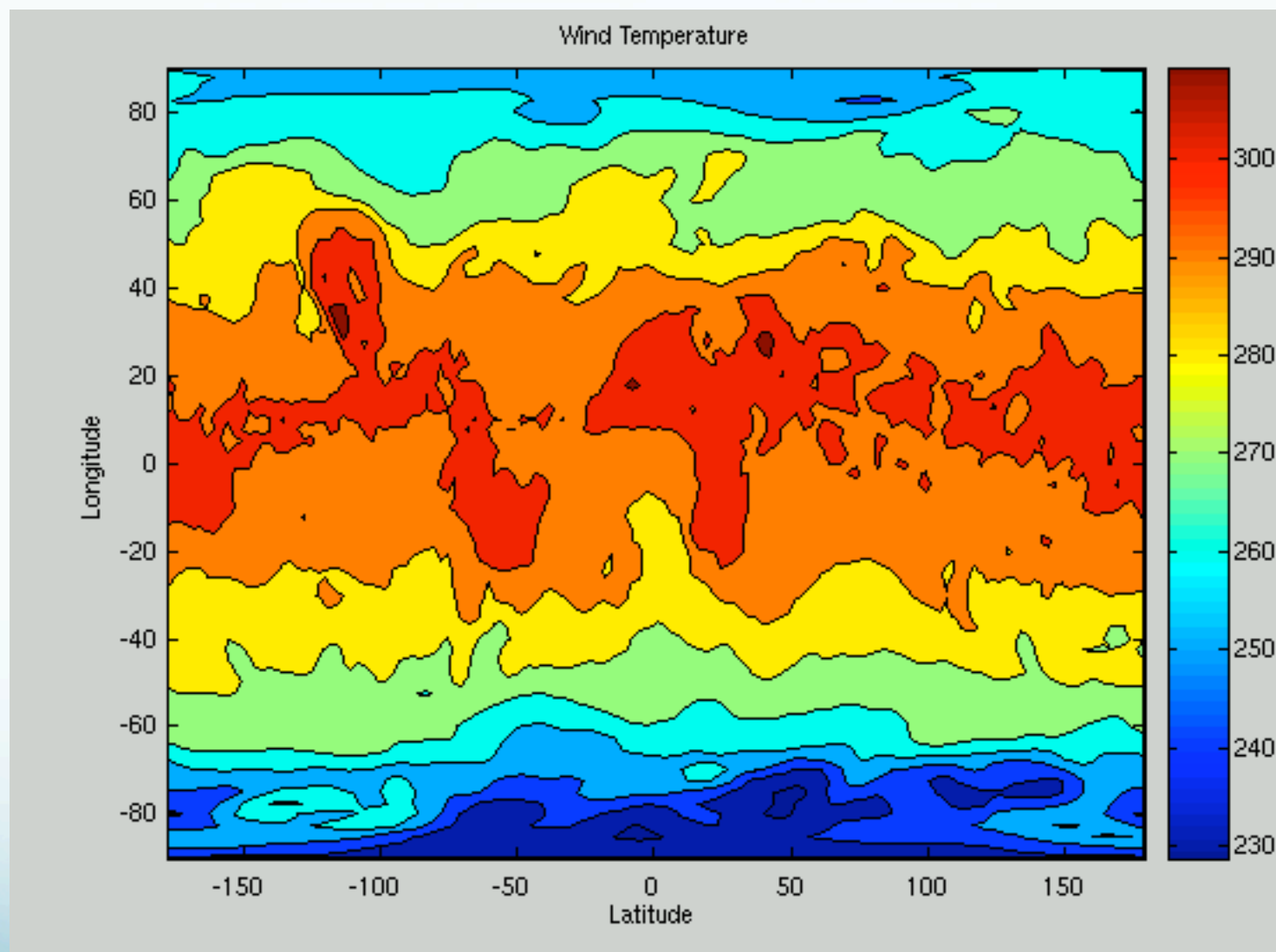


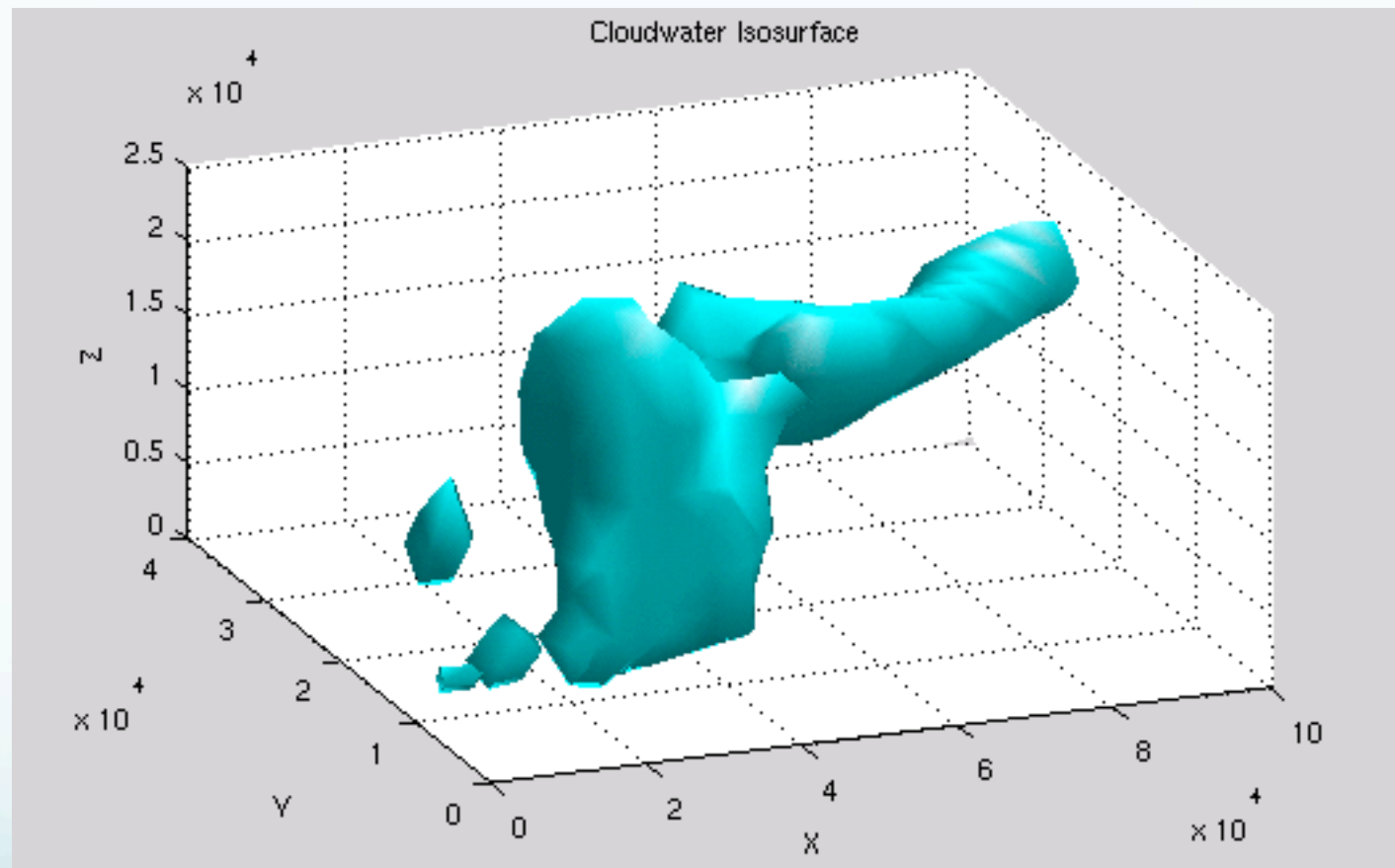
L-shaped Membrane - (SURF)



Elevation map of area around Cape Cod, Massachusetts







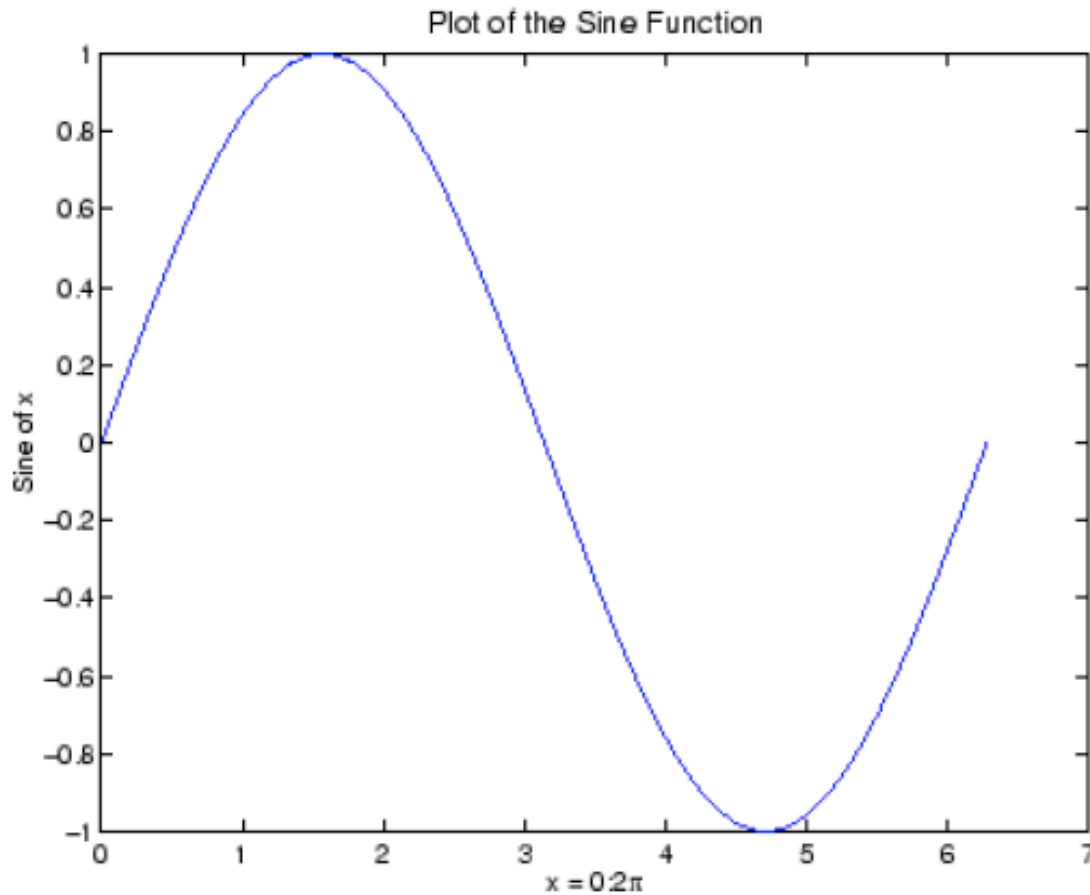
Creating a plot

The `plot` function has many different forms, depending on the input arguments.

If `y` is a vector, `plot(y)` produces a piecewise linear graph of the elements of `y` versus the index of the elements of `y`.

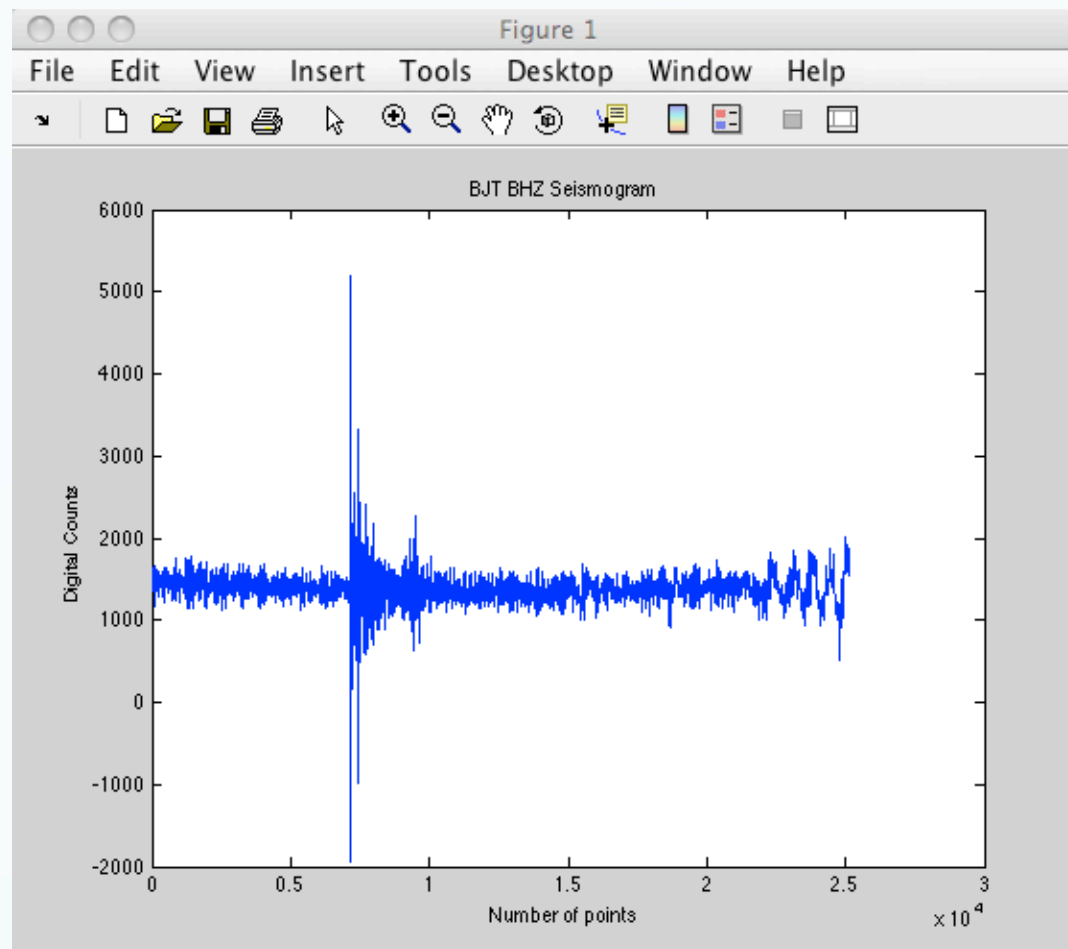
If you specify two vectors as arguments to plot, `plot(x,y)` produces a graph of `y` versus `x`.

```
>> x = 0:pi/100:2*pi;  
>> y = sin(x);  
>> plot(x,y)
```



```
>> xlabel('x = 0:2\pi')  
>> ylabel('Sine of x')  
>> title('Plot of the Sine Function','FontSize',12)
```

(Notice that the fontsize specification is sort of verbose. This aspect of setting plot parameters is worse than GMT! It will improve you typing however.)

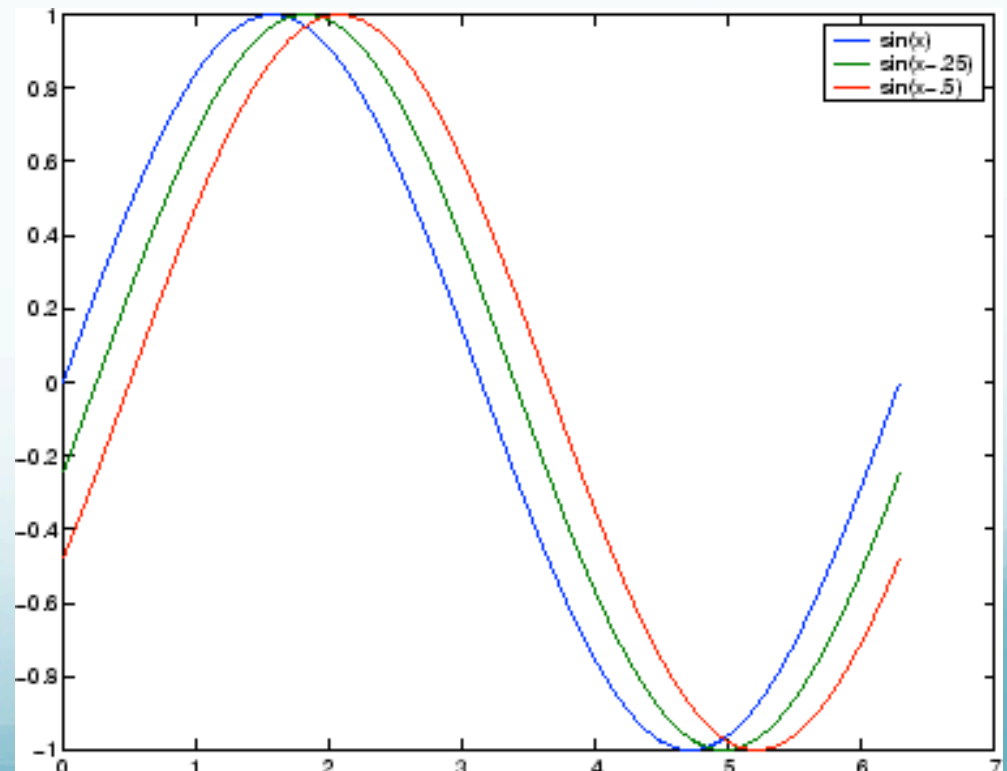


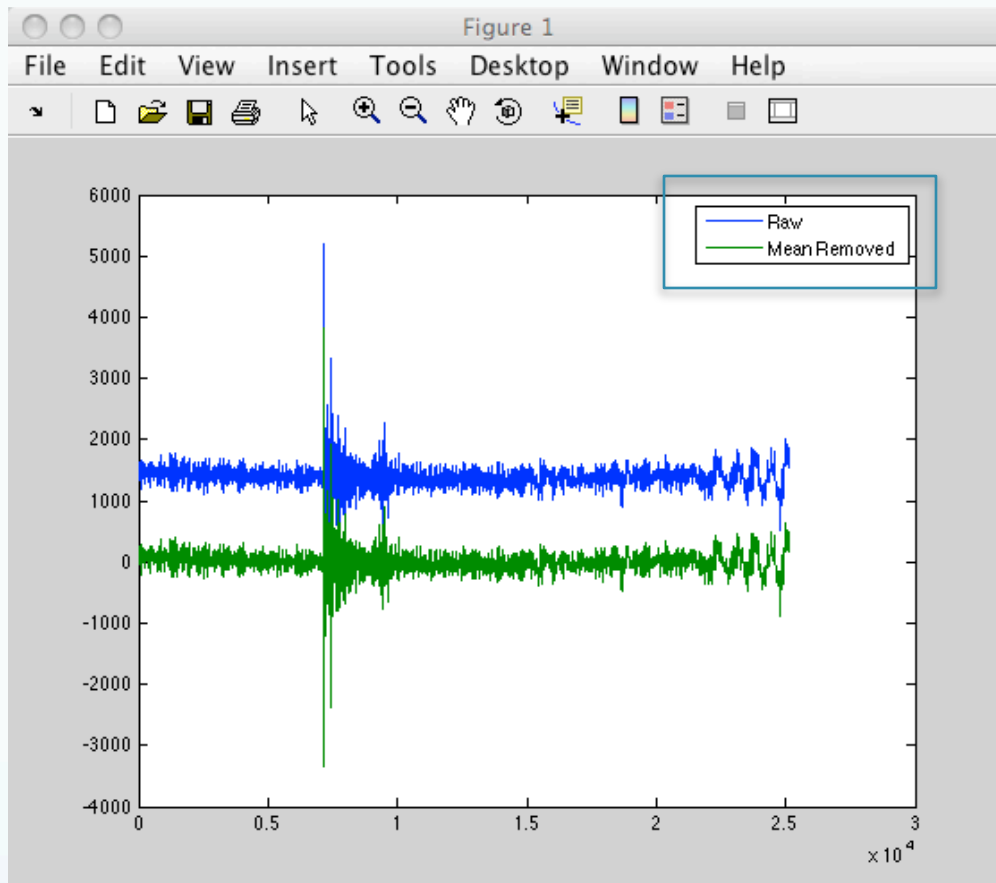
```
>> seis=loadsac('BJT.BHZ_00.Q.2005:01:23:41');  
>> plot(seis)  
>> ylabel('Digital Counts')  
>> xlabel('Number of points')  
>> title('BJT BHZ Seismogram')
```


Plotting multiple data sets

Multiple x-y pair arguments create multiple graphs with a single call to plot, which automatically cycles through a predefined (but customizable) list of colors

```
>> x = 0:pi/100:2*pi;  
>> y = sin(x);  
>> y2 = sin(x-.25);  
>> y3 = sin(x-.5);  
>> plot(x,y,x,y2,x,y3)  
>> legend('sin(x)',...  
'sin(x-.25)', 'sin(x-.5)')
```



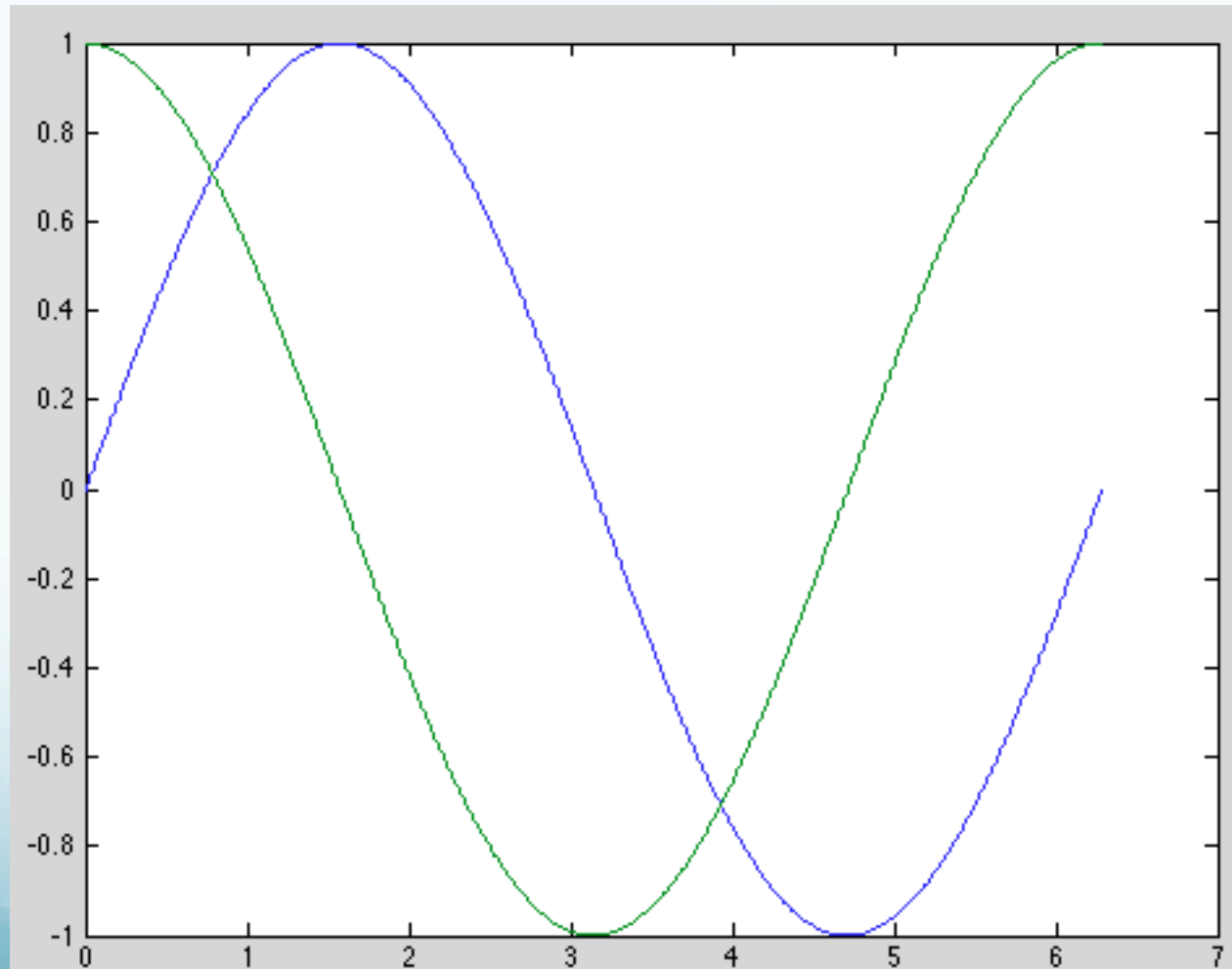


```
>> size(seis)
    25138    1
>> x=1:25138;
>> plot(x,seis,x,seis2)
>> legend('Raw','Mean Removed')
```

Plotting multiple data sets

If you have a matrix of y vectors for a single x vector, plots all the y s against x .

```
>> x=[0:.01:2*pi];  
>> y=[sin(x); cos(x)];  
>> plot(x,y)  
>>
```



Specifying line colors/styles

It is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the plot command:

```
>> plot(x,y, 'specify_color_linestyle_markertype')
```

Change color

```
>> plot(x,seis,'r',x,seis2,'b')  
>> plot(x,seis,'r',x,seis2,'b:')
```

Color

'c'

cyan

'm'

magenta

'y'

yellow

'r'

red

'g'

green

'b'

blue

'w'

white

'k'

black

Line style

no character

' _ '

solid

' _ _ '

dashed

' : '

dotted

' . - '

dash-dot

Specifying lines and markers

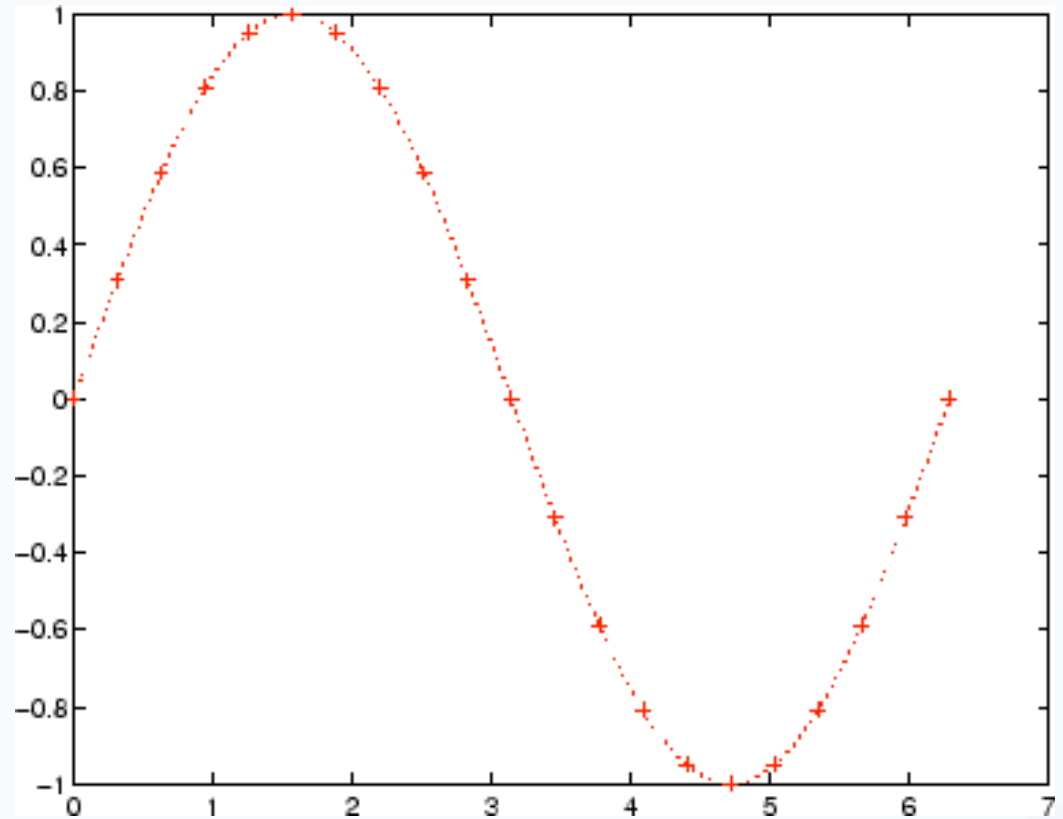
If you specify a marker type but not a line style, only the marker is drawn.

```
>> plot(x,y,'ks')
```

plots black (**k**) squares (**s**) at each data point,
but does not connect the markers with a line

```
>> plot(x,y,'r:+')
```

plots a red-dotted line and places plus sign
markers at each data point



```
>> x1 = 0:pi/100:2*pi;  
>> x2 = 0:pi/10:2*pi;  
>> plot(x1,sin(x1),'r:',x2,sin(x2),'r+')
```

Second part only plots the + every 10 points. So does.

```
>> plot(x1,sin(x1),'r:',x1(1:10:end),sin(x1(1:10:end)),'r+')
```

Marker Type

'+'

plus mark

'o'

unfilled circle

'*'

asterisk

'x'

letter x

's'

filled square

'd'

filled diamond

'^'

filled upward triangle

'v'

filled downward triangle

'>'

filled right-pointing triangle

'<'

filled left-pointing triangle

'p'

filled pentagram

'h'

filled hexagram

no character

no marker

Graphing imaginary and complex data

Reminder: complex numbers can be represented by the expression $a+bi$ where a and b are real numbers and i is a symbol with the property $i^2 = -1$

Complex numbers can be plotted using Real and Imaginary axes.

When the arguments to plot are complex, the imaginary part is ignored *except* when you pass plot a single complex argument. For this special case, the command is a shortcut for a graph of the real part versus the imaginary part.

```
>> t = 0:pi/10:2*pi;  
>> plot(exp(i*t), '-o')  
>> axis equal  
>> xlabel('Real')  
>> ylabel('Imaginary')  
>> hold on  
>> plot(t,t, 'r+-')
```

Plus plotting second
data set with
“hold” (else erases existing figure
with new call to plot)

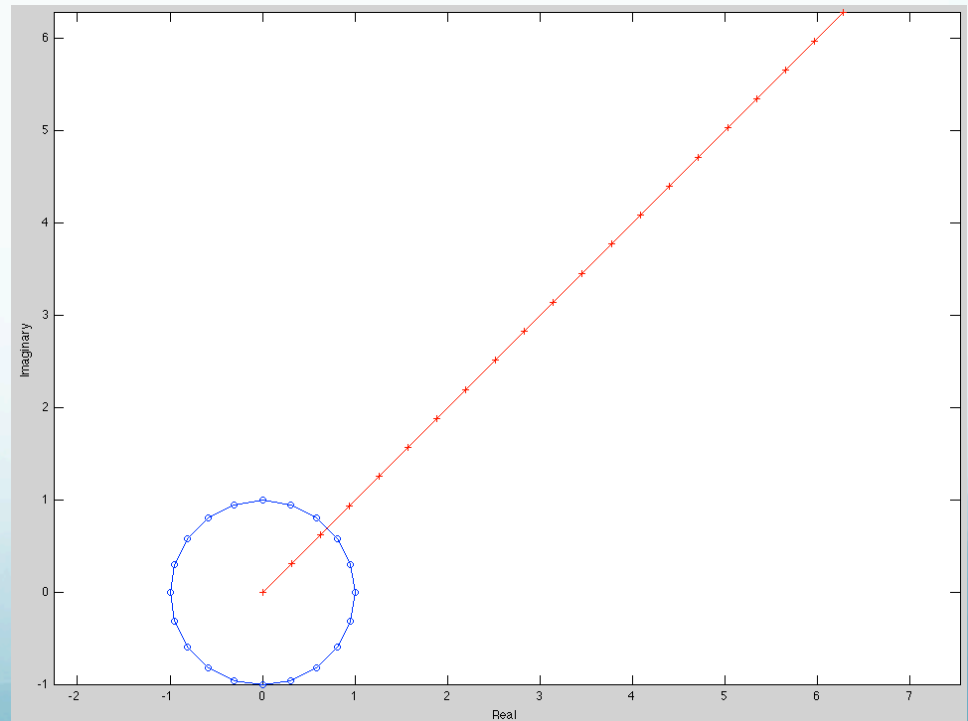


Figure Handling

Graphing functions automatically open a new figure window if there are no figure windows already on the screen.

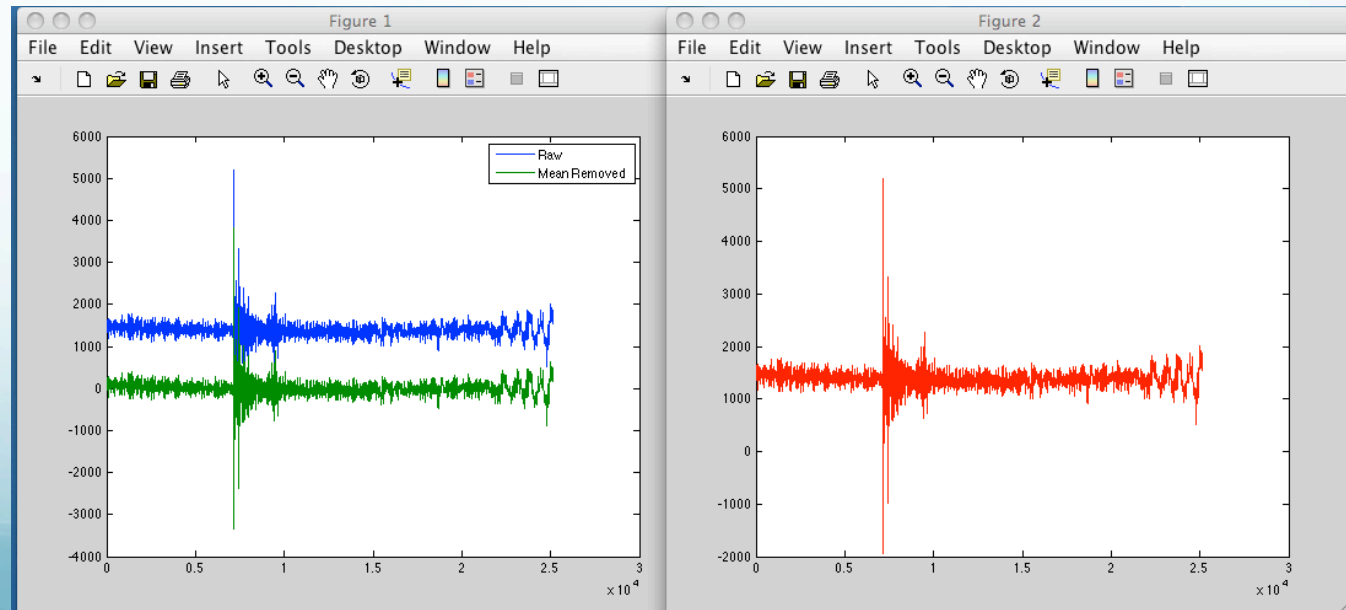
If a figure window exists, it is used for graphics output (and clobbers what's there if hold is off).

The default is to graph to the *current figure* (usually the last active figure)

To create a new figure without overwriting the old, use the figure command

When multiple figures already exist, you can set one of them to the current figure with the command figure(n) where n is the number at the top of the figure window.

```
>> plot(x,seis,x,seis2)
>> legend('Raw','Mean Removed')
>> figure %creates 2
>> plot(seis,'r')
>> figure(1) %makes 1 current
```



Creating subplots

The subplot command enables you to display multiple plots in the same window or print them on the same piece of paper.

```
t = 0:pi/10:2*pi;  
[X,Y,Z] = cylinder(4*cos(t));  
subplot(2,2,1); mesh(X)  
subplot(2,2,2); mesh(Y)  
subplot(2,2,3); mesh(Z)  
subplot(2,2,4); mesh(X,Y,Z)
```

%creates a 2 x 2 matrix of
%subplots

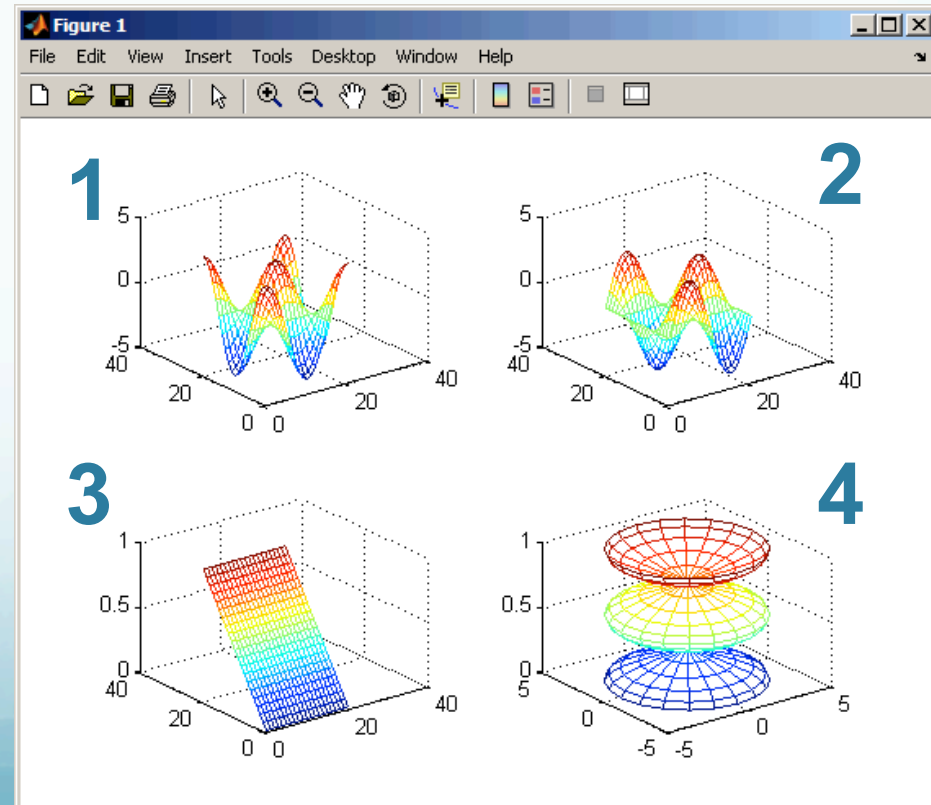
```
>> help cylinder
```

CYLINDER Generate cylinder.

[X,Y,Z] = CYLINDER(R,N) forms the unit cylinder based on the generator curve in the vector R. Vector R contains the radius at equally spaced points along the unit height of the cylinder. The cylinder has N points around the circumference. SURF(X,Y,Z) displays the cylinder.

[X,Y,Z] = CYLINDER(R), and [X,Y,Z] = CYLINDER default to N = 20 and R = [1 1].

Omitting output arguments causes the cylinder to be displayed with a SURF command and no outputs to be returned.



Controlling axes

The axis command provides a number of options for setting the scaling, orientation, and aspect ratio of graphs.

Set the axis limits

```
axis auto  
axis([xmin xmax ymin ymax zmin zmax])
```

Set the axis aspect ratio

```
axis auto normal  
axis square; axis equal
```

The axis command provides a number of options for setting the scaling, orientation, and aspect ratio of graphs.

Set axis visibility

`axis on; axis off`

Set grid lines

`grid on; grid off`

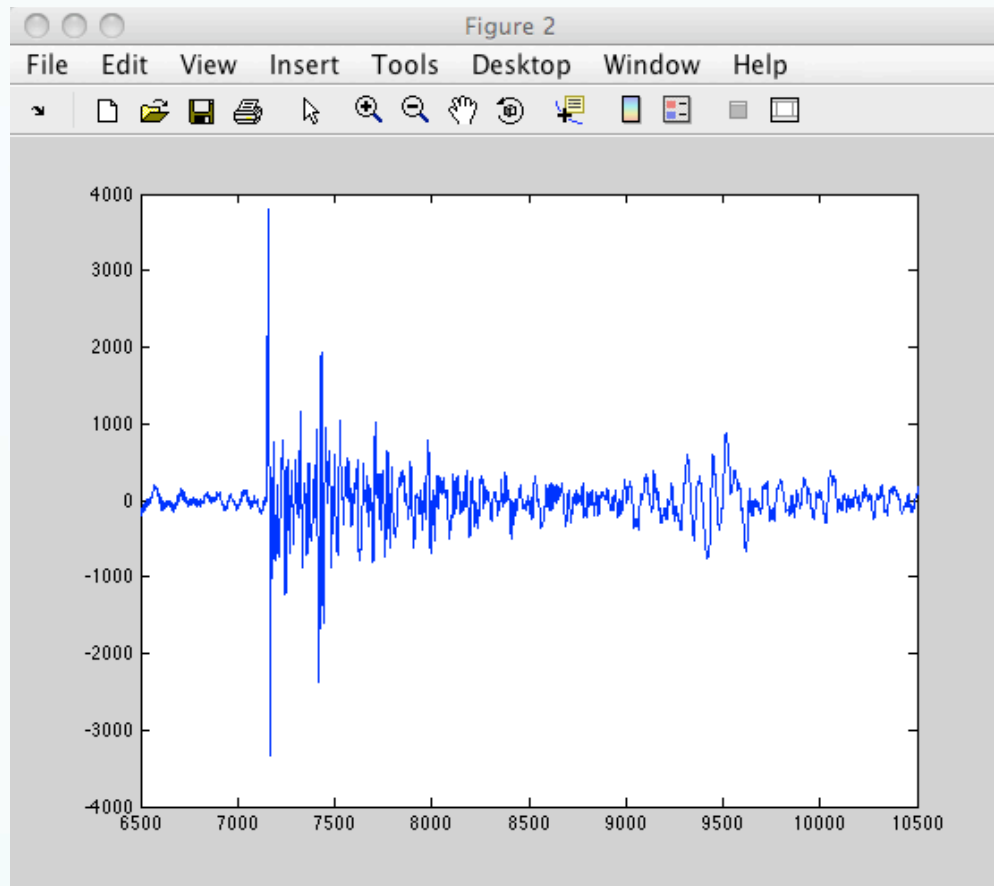
axis vs axes

```
>> help axis
```

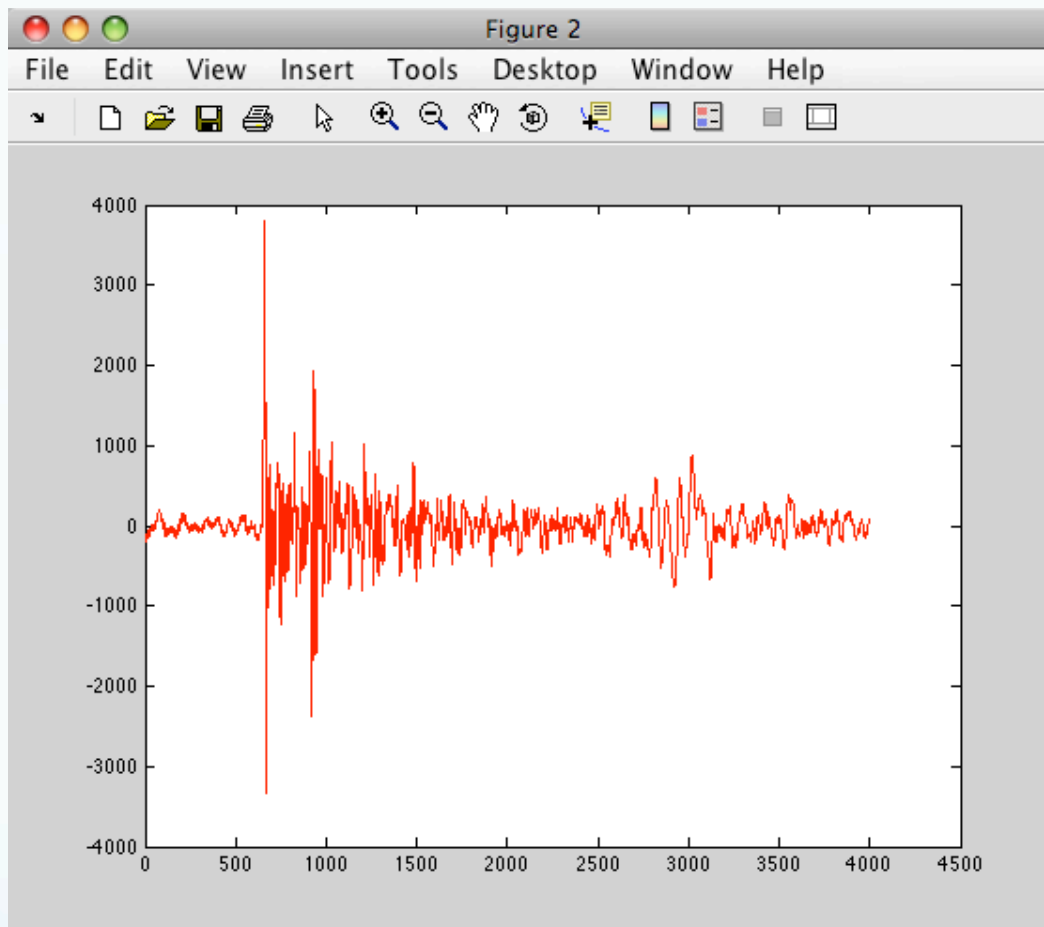
```
AXIS  Control axis scaling and appearance.
```

```
>> help axes
```

```
AXES  Create axes in arbitrary positions.
```



```
>> figure(2)
>> plot(seis2)
>> axis([6500 10500 -4000 4000])
```



```
>> axis auto  
>> plot(seis2(6500:10500,:), 'r')
```

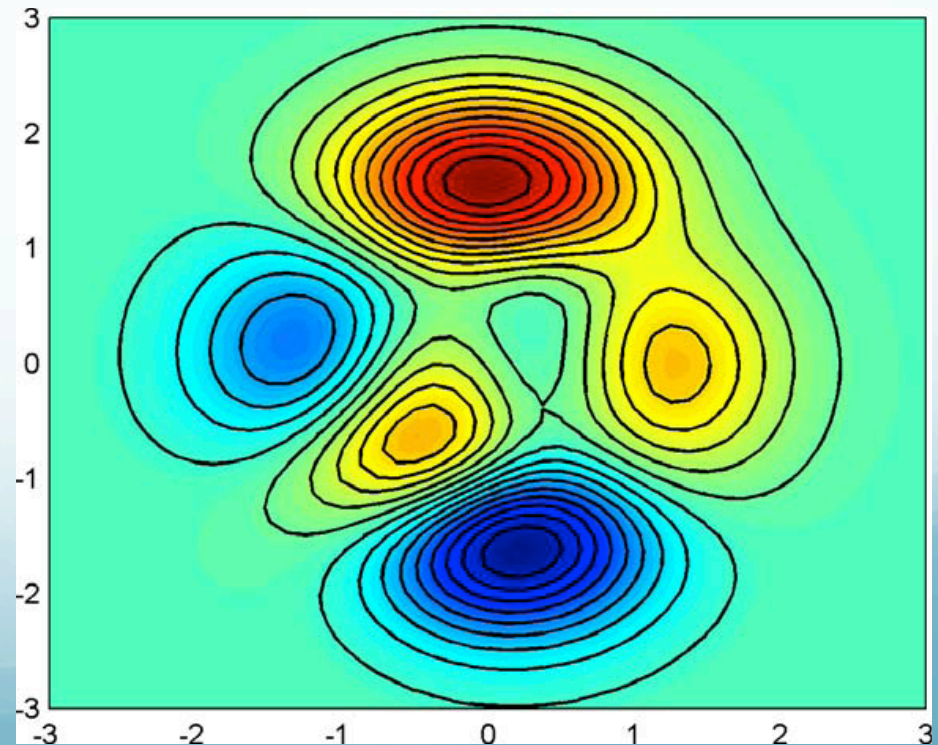
Overlaying new graphs
Use the command

`hold on`

or just
`hold`

to overlay different types of plots on one another

```
>> [x,y,z] = peaks;  
>> pcolor(x,y,z)  
>> shading interp  
>> hold on  
>> contour(x,y,z,20,'k')  
>> hold off
```



```
>> surf(x,y,z)
>> clf
>> [x y z]=peaks;
>> surf(x,y,z)
>> shading interp
```

