

# 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 - 6, 09/12/13

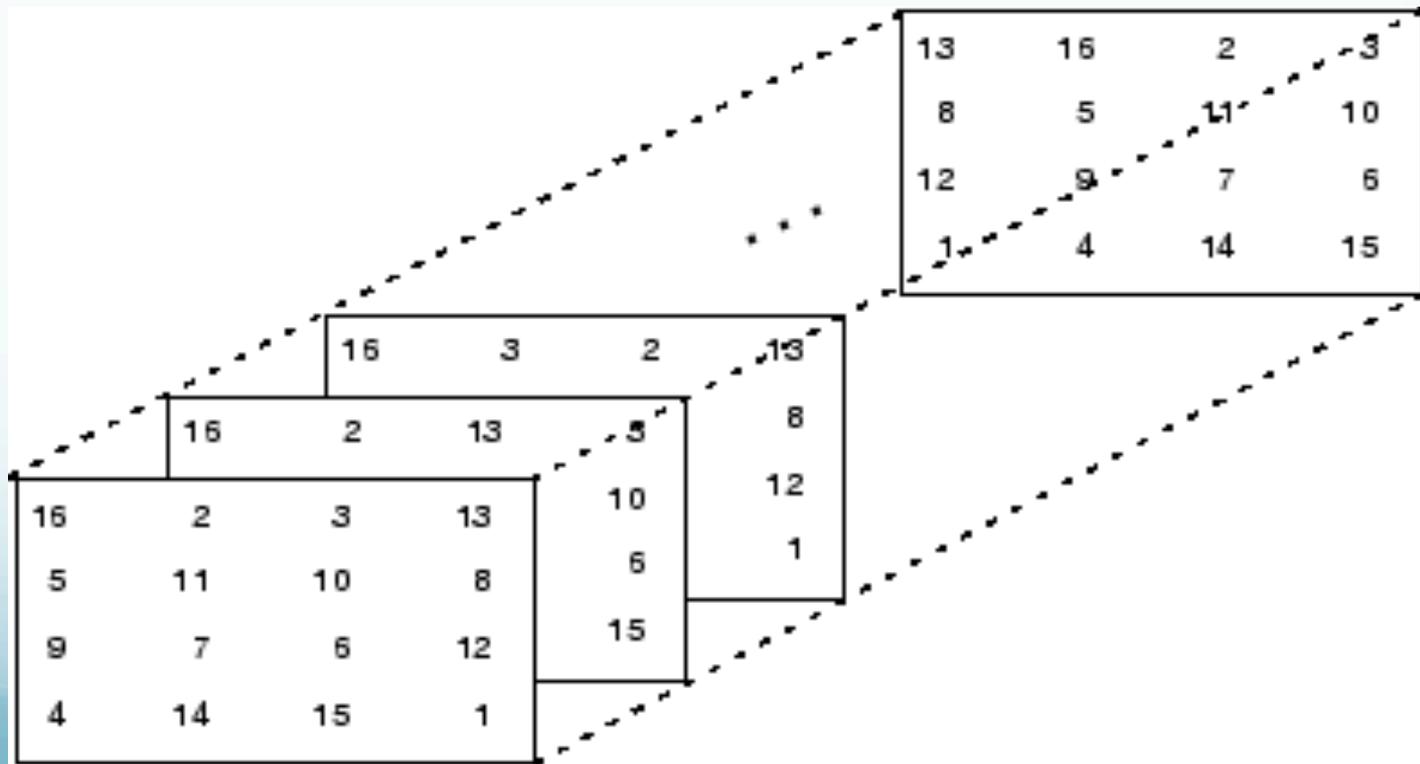
# Matlab

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
>>
```

# Other ways (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!) (`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
>>
```

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.

# How Tony's trick works

What does this do? How does it work?

```
>> x=val([1 1 1; 1 1 1])
```

We know what `x=val(1)` does.

We know what `x=val(1,1)` does.

We know (or can find out) that `x=val(1,2)` does not work.

What about `x=val([1])`?

From there easy to generalize a matrix as index.

What about `x=val([1 1])`?

# How Tony's trick works

What does this do? How does it work?

```
>> x=val([1 1 1; 1 1 1])
```

```
x =
```

```
3.141592653589793    3.141592653589793    3.141592653589793  
3.141592653589793    3.141592653589793    3.141592653589793
```

Tony's trick just replaces the matrix definition above

```
>> x=val(ones(2,3))
```

```
x =
```

```
3.141592653589793    3.141592653589793    3.141592653589793  
3.141592653589793    3.141592653589793    3.141592653589793
```

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
```

```
>>
```

# Object on right does not have to be scalar

```
>> a=[1:3;3:5].^2
```

```
a =
```

```
     1     4     9
     9    16    25
```

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

```
b =
```

```
     1     9     1
```

```
>> b=a([1:2 5 6])
```

```
b =
```

```
     1     9     9    25
```

```
>> b=a([1:2;5:6])
```

```
b =
```

```
     1     9
     9    25
```

```
>>
```

Can write very obscure, compact code that nobody can figure out (including you 6 mos later).

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)
```

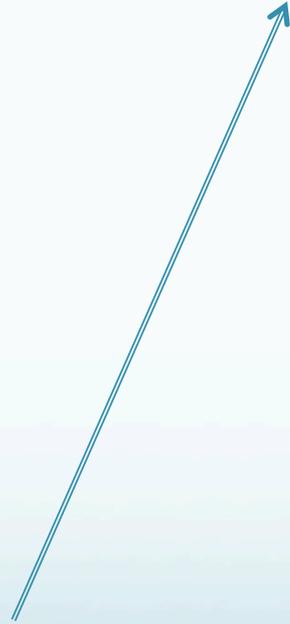
```
ans =  
     3     6  
     2     5  
     1     4
```

```
>> 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
```

```
>> x.x
```

```
ans =
```

```
    1
```

```
    1
```

```
. . . 7 more times . . .
```

```
>> x.n
```

```
ans =
```

```
ceri
```

```
. . . 7 more times . . .
```

```
>> x(2,2,2)
```

```
ans =
```

```
    x: 1
```

```
    n: 'ceri'
```

Tony's trick

# 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)) (same as prod(A(:)) – probably better)

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
(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
```

Look at formatted input and output and characters using examples from mathworks web pages.

## 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
```

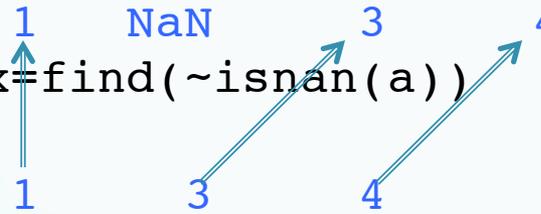
```
>>
```

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).

# How to find values in matrix - you will need this for the hw:

```
>> a=magic(3)
```

```
a =
```

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

```
>> b=find(a>5)
```

```
b =
```

```
1
6
7
8
```

```
>> [c d]=find(a>5)
```

```
c =
```

```
1
3
1
2
```

```
d =
```

```
1
2
3
3
```

```
>>
```

	index	value
linearly	- 2-d	
1	1,1	8
6	3,2	9
7	1,3	6
8	2,3	7

But no way I can see to use the 2-d result to index directly into matrix as a 2-d matrix - stuck with linear indexing.

# For 2-d indexing you have to do this

```
>> A = [1 3 5 2 7;6 0 1 16 12; 8 11 2 3 6];  
[value, index] = min(abs(A(:)-10))  
[row, col] = ind2sub(size(A), index)  
A(index)  
A(row, col)  
value =  
    1  
index =  
    6  
row =  
    3  
col =  
    2  
ans =  
    11  
ans =  
    11  
>>
```

# Can go other way also – 2-d index to linear index

```
>> rng(0,'twister'); % Initialize random number generator.  
>> A = rand(3, 4, 2)
```

```
A(:,:,1) =
```

```
0.8147 0.9134 0.2785 0.9649  
0.9058 0.6324 0.5469 0.1576  
0.1270 0.0975 0.9575 0.9706
```

```
A(:,:,2) =
```

```
0.9572 0.1419 0.7922 0.0357  
0.4854 0.4218 0.9595 0.8491  
0.8003 0.9157 0.6557 0.9340
```

Find the linear index

```
%corresponding to (2, 1, 2):
```

```
>> linearInd = sub2ind(size(A), 2, 1, 2)
```

```
linearInd =
```

```
14
```

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.

Temporarily done with Matlab.

Move on to SAC.

(Seismic Analysis Code)

Basic Data Manipulation

Seismic analysis code (sac)

## SAC (Seismic Analysis Code)

General purpose interactive program designed for the study of sequential signals, especially seismic timeseries data (seismograms).

Emphasis has been placed on analysis tools used by research seismologists in the detailed study of seismic events.

# SAC (Seismic Analysis Code)

## Analysis capabilities include:

- General arithmetic operations
  - Fourier transforms
  - integration/differentiation
- spectral estimation/processing techniques
  - IIR and FIR filtering
  - Signal stacking
- Decimation and Interpolation,
  - Correlation,
- seismic phase (time and amplitude) picking
  - Instrument correction
  - Particle motion rotation
  - Trace envelopes
  - Linear regressions
- Frequency-wavenumber analysis
  - various types of plotting.

# SAC (Seismic Analysis Code)

SAC also contains an extensive (for the early 1980's) graphics capability.

# SAC

Seismic Analysis Code was developed at Lawrence Livermore National Laboratory (LLNL) and University of California in the early 1980's.

LLNL is one of the 3 US Nuclear weapons laboratories.

Seismology is one of the principle tools in Nuclear Test Ban treaty verification.

SAC

SAC was developed using PRIME  
Microcomputers under the PRIMEOS.

It was written in FORTRAN.

# SAC

It is a command-driven, as opposed to a menu or GUI driven, program.

It took advantage of several features of the PRIMEOS (the OS from the computer company PRIME), such as its command line processor that passes commands that are not part of the program to the OS.

(This means, that if one is running SAC and you need to get a directory listing, you just enter the "ls" command. Since this is not a SAC command, the command line processor will pass the command to the OS and you will get a directory listing. You don't have to leave SAC, do the "ls", write down/remember the file name, and restart SAC. This was very important in the pre -GUI days.)

## SAC

Although it can run in batch mode, it was principally designed to be interactive and have interactive graphics.

It was designed to use the “state-of-the-art” Tektronix 401X “storage tube” line of graphics terminals.



### Tektronix 4010 Specifications

Data Transfer Rate: 150 to 9600 baud

Screen Size: 8 1/4 by 6 3/8 inches

Character Set: UPPER CASE ONLY 63 total 5x7 matrix

Format: 74 characters per line 35 lines per screen (2590 characters)

Character Draw Time: 1200 per second

Vector Resolution: 1024 by 780

Vector Draw Time: 2.6 milliseconds maximum

Usable Storage Time: Up to one hour without permanent damage

Operating Environment: 10 to 40 C (50-104F)

Power: 192 watts maximum

Weight: 78 pounds

Notice the blistering speed -

Data Transfer Rate: 150 to 9600 baud.

This will come back to haunt us.

# SAC

After the demise of PRIME in the early 90's, SAC was beaten into submission to run under the UNIX operating system, specifically, SOLARIS, the SUN (which has followed PRIME into oblivion) OS.

(as with UNIX, SAC dragged along many of the idiosyncrasies of its birth associated with the PRIMEOS and the hardware limitations of the time – such as the TEK401X. The UNIX implementation was the simplest “make it run” under UNIX effort – amounted to writing a graphics translator from tektronix graphics commands to the current ones – no rethinking, etc. due to new hardware and capabilities – just translate line by line.)

It now runs on most UNIX/LINUX systems and has become one of the standard data manipulation tools in seismology.

# SAC

SAC's data format, especially for binary data, is one of the principle data formats used today in storing, transferring, and manipulating (earthquake) seismological time series data.

# SAC's competitors (data format) include

**ah** (ad-hoc, used by IRIS/PASSCAL program, born about the same time as SUN)

**SEED** (Standard for the Exchange of Earthquake Data, native format IRIS-DMC)

**CSS** (Center for Seismic Studies, associated with treaty verification)

**SUDS** (Seismic Unified Data System, from Willie Lee PC based system/USGS)

**SEG-Y** (the standard for seismic reflection data)

**Others** (Panda,...)

(new ones crop up every 5-10 years to address the chaotic state of affairs.)

SAC's competitors (analysis) include

- IRIS/PASSCAL ah (ad-hoc) system
- Various versions (with various names) of DATASCOPE  
(now Antelope)
- XPICK
- Seismic UNIX
- MATSEIS
- others

SAC is used for a range of seismic analysis tasks from quick preliminary analyses to routine processing and testing new techniques creating publication quality graphics, etc.

Luckily for us we are protected from the power of UNIX and all the UNIX setup details for running SAC (and GMT and MATLAB, etc.) have been set up for us in the global `.cshrc` file.

To run `sac`, simply type “`sac`” at the prompt.

```
ceri% sac  
SEISMIC ANALYSIS CODE [8/8/2001 (Version 00.59.44)]  
Copyright 1995 Regents of the University of California  
SAC>
```

SAC is now ready to start accepting commands.

# Commands

SAC commands fall into 3 main categories

Parameter-setting: change values of internal SAC parameters.

Action-producing: perform some operation on the signals currently in selected memory based upon the values of these parameters.

Data-set: determine which files are in active (selected) memory and therefore will be acted upon.

# Commands

help calls up a list of all commands.

help command shows the manual page for the command.

## Defaults

Based on typical use at CERL, default values for all operational parameters are set when you start SAC.

Almost all of these parameters are under direct user control.

SAC can be reinitialized to the default state at any time by executing the `INICM` command.

# Data File Command Module

This module is used to read, write, and access SAC data files.

`read` (can be shortened to “`r`”): reads data files from disk into memory

```
sac> r *.SAC
```

Uses standard UNIX wildcards: reads all files whose filenames end in “.SAC”

# Data File Command Module

`write` (“w”): writes the data currently in memory to disk

You can write the data into a range of file formats and file names or simply overwrite the current set of files.

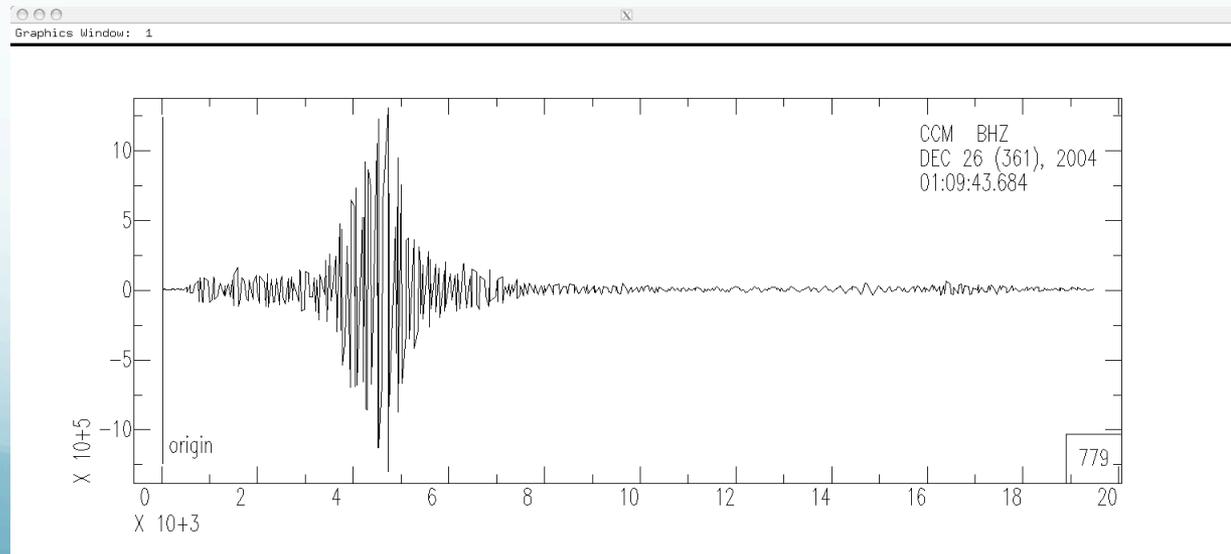
(so be careful, you’ve been warned)

Let's try it (and also jump ahead to graphics action module to `plot` ("p") it) ~

```
alpaca.ceri.memphis.edu504:> sac  
SEISMIC ANALYSIS CODE [8/8/2001 (Version 00.59.44)]  
Copyright 1995 Regents of the University of California
```

```
SAC> read ccm_sumatra_.bhz  
SAC> plot
```

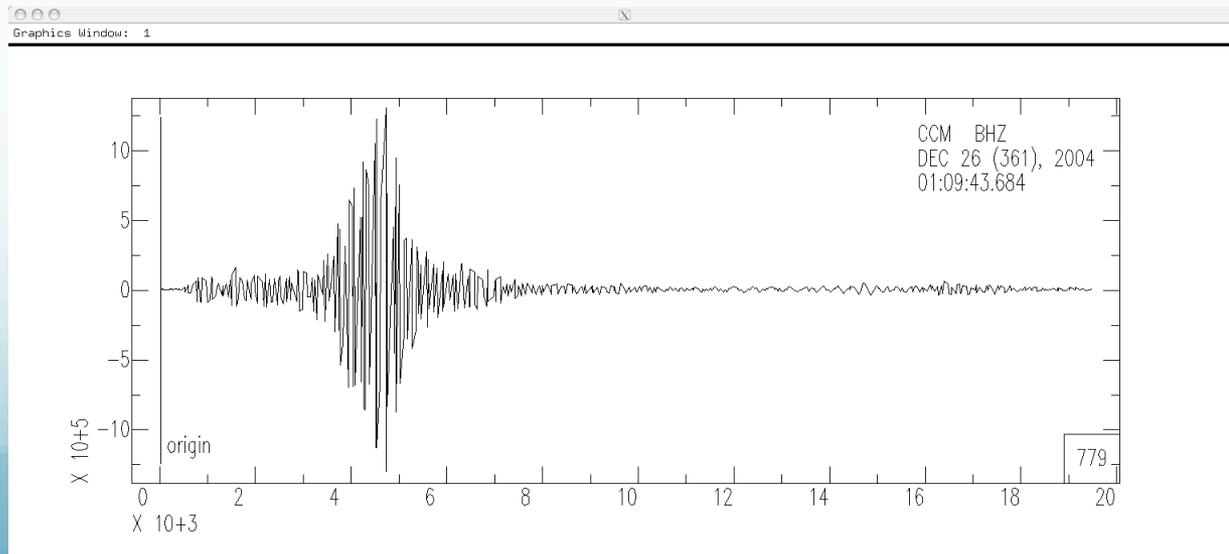
Which produces the following plot.



This plot shows the heritage of the SAC program.

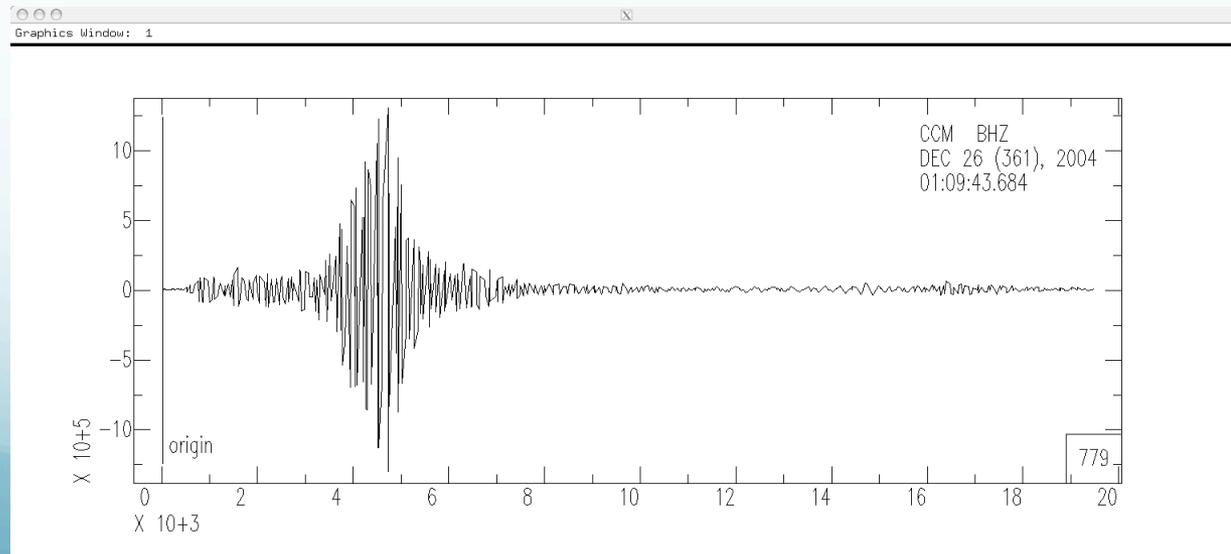
The plot is a straight “port” of the TEK 401X graphics over to an X-Window display.

(It looks exactly the same as it did on the TEK 401X.)



This seismogram is 20,000 seconds long, with samples 20 times per second.

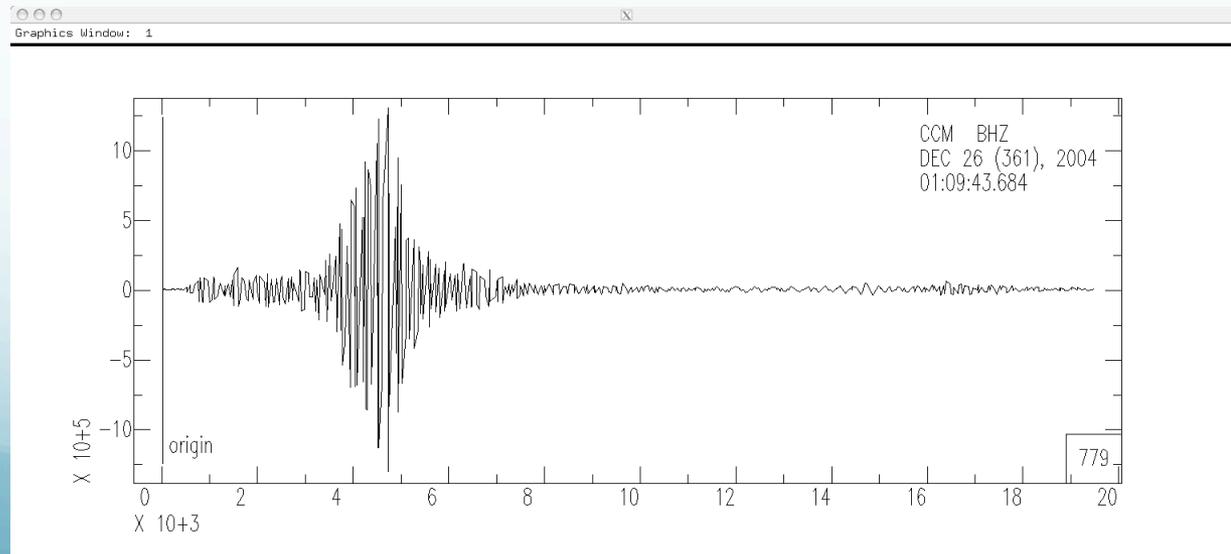
It has over 3,890,000 points and would take almost an hour to draw at 9600 baud.



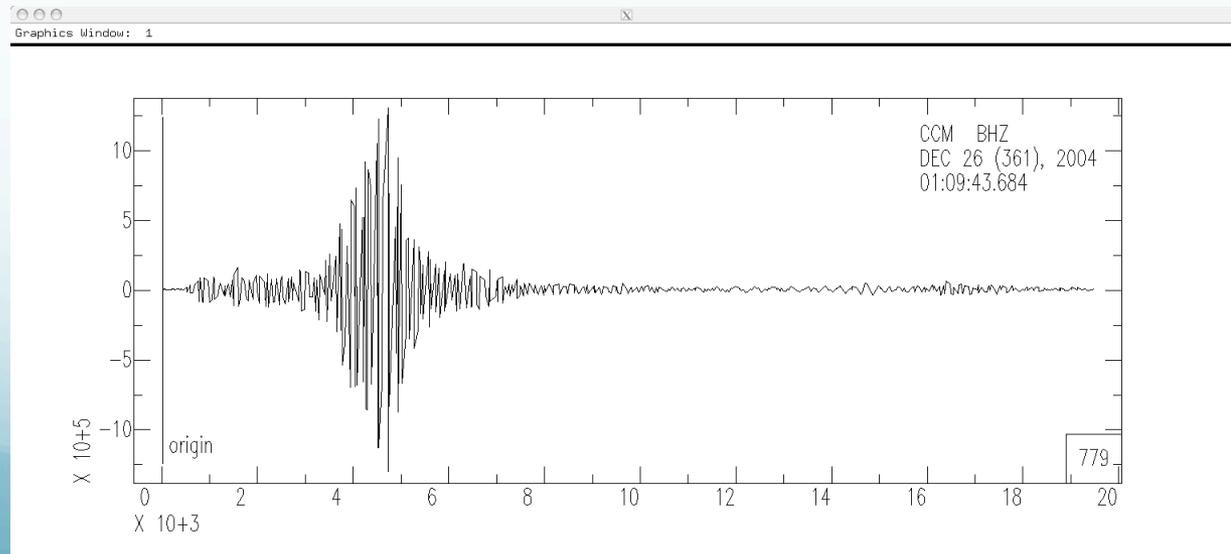
Enter QDP (Quick and Dirty Plot mode) to the rescue.

Look at the lower right corner. There is a box there with the number 779.

This tells us that SAC is displaying every 779<sup>th</sup> point (that's one point every 39 seconds!).



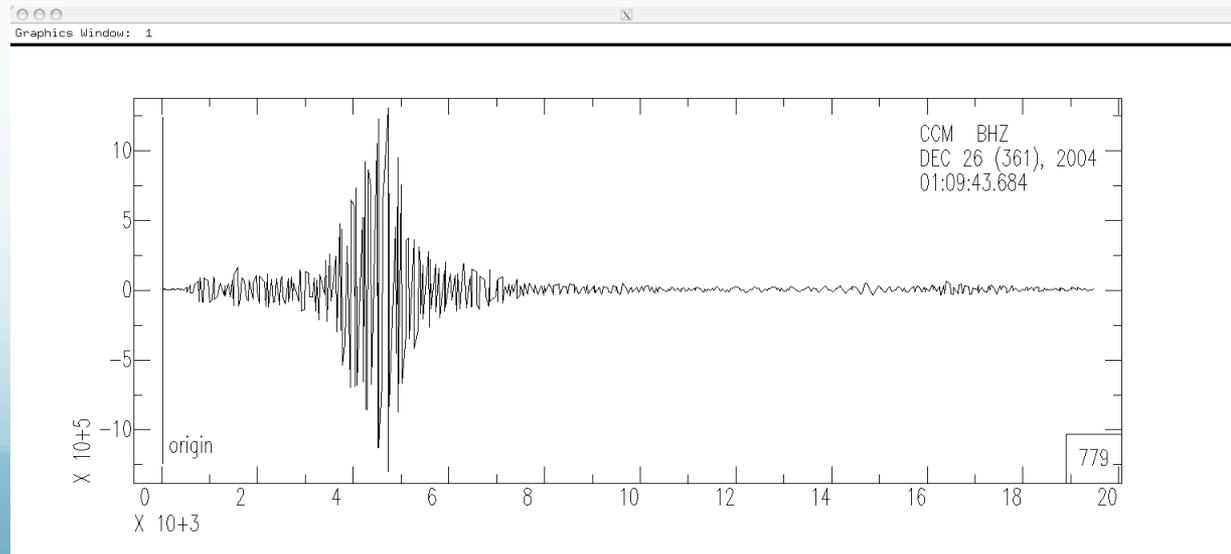
SAC automatically cuts down the amount of data it shows when the number of input points is **>1000** (the resolution of the TEK401X series of devices is 1024) so that it only takes a few seconds to draw it at 9600 baud.



Unfortunately, about the only thing this plot tells us is that something was recorded.

The wiggles you see are absolutely completely useless for analysis (the data in memory is OK however)

(you will learn the technical reasons for this – known as aliasing – in signal analysis).

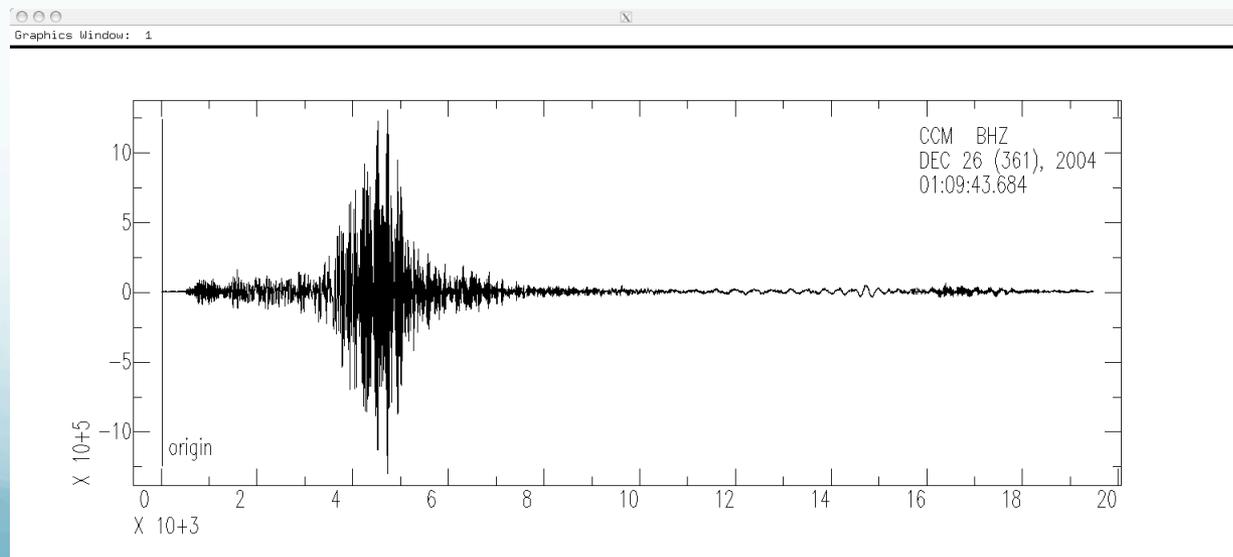


Since we are on a modern computer we can afford to plot all the data (although it is still sub-optimal to do so. Our plot will now legally represent the seismic signal).

So we turn the QDP “feature” off  
(you can guess how to turn it back on.)

```
SAC> qdp off  
SAC> plot
```

This plot is now “good” (compare to previous slide).



qdp is the correct idea (don't waste time displaying stuff that you can't see due to screen resolution), but it is very badly implemented.

qdp should have taken the max and min of each of the sections of N points (instead of each Nth point) and plotted a vertical line between them at each time (it would have taken twice as long to display, but the qdp display was relatively quick).

The display would be identical to the full display on the last slide (and look like a paper record).

(this would have cost some computer time to calculate what to display, but that is minimal compared to the data transfer time to the TEK401X. As it is now it takes the decimation factor longer to calculate what gets drawn on the screen – you don't notice it, but the drawing is instantaneous).