

# Data Analysis in Geophysics

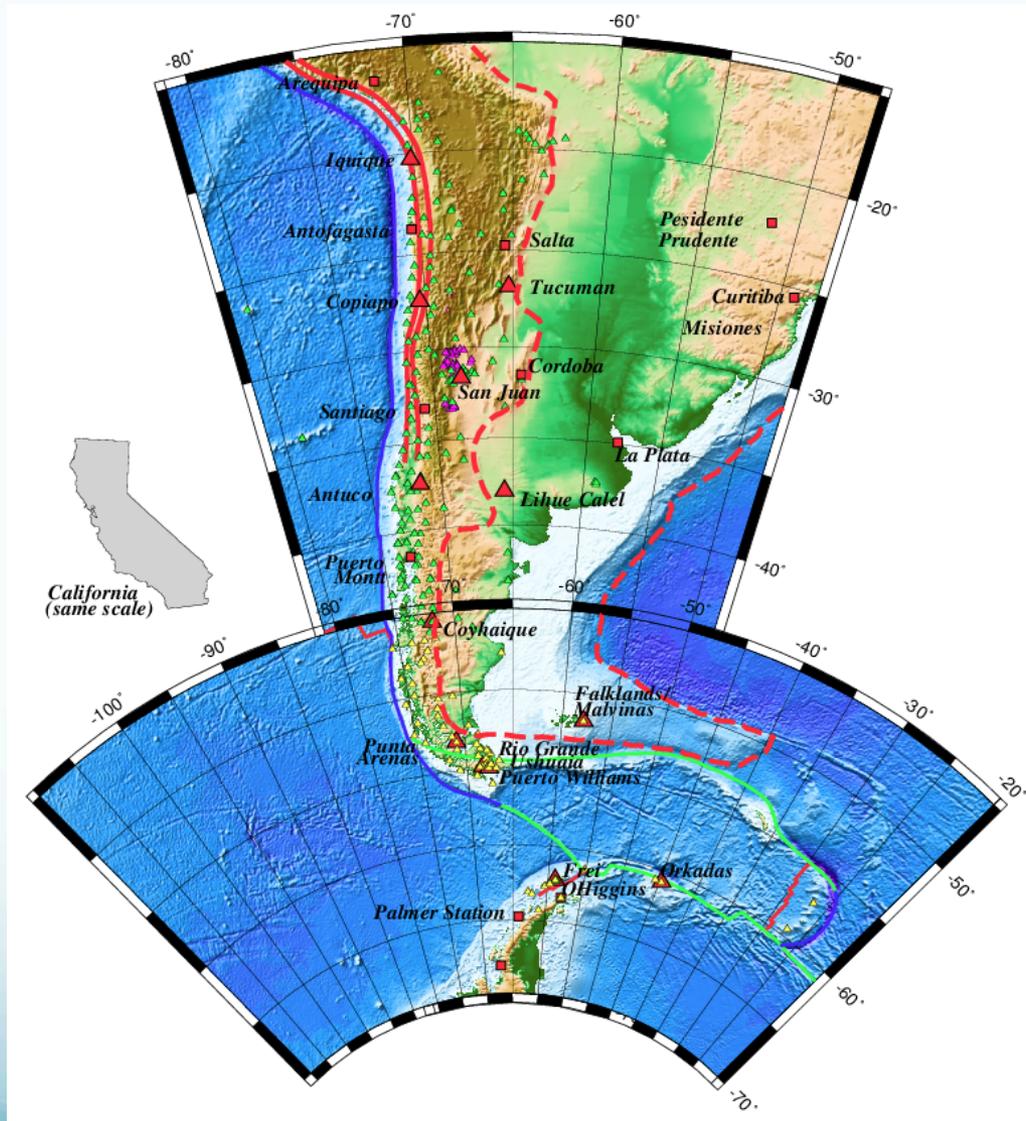
## ESCI 7205

Class 13

Bob Smalley

GMT loose ends, representation of numbers.

# Azimuthal projections



# Part I of shell script

## Set stuff up

```
#!/bin/sh
#call with "stn_az_map lon lat name"
ROOT=/gaia/home/smalley
WORLDCOAST=0/360/0/180
RED=250/50/50
BLUE=50/50/255
GREEN=50/255/50
MOREPS=-K
ADDPS=-O
CONTINUEPS="-K -O"
FILL=200
SCALE=1.75
XOFFSET=0.75
YOFFSET=1.5
GRIDCNTR=180/90/7/90
OUTPUTFILE=$0_$3.ps
rm $OUTPUTFILE
```

Notice abundant (lack of) “comments” (use variable names that are self documenting)

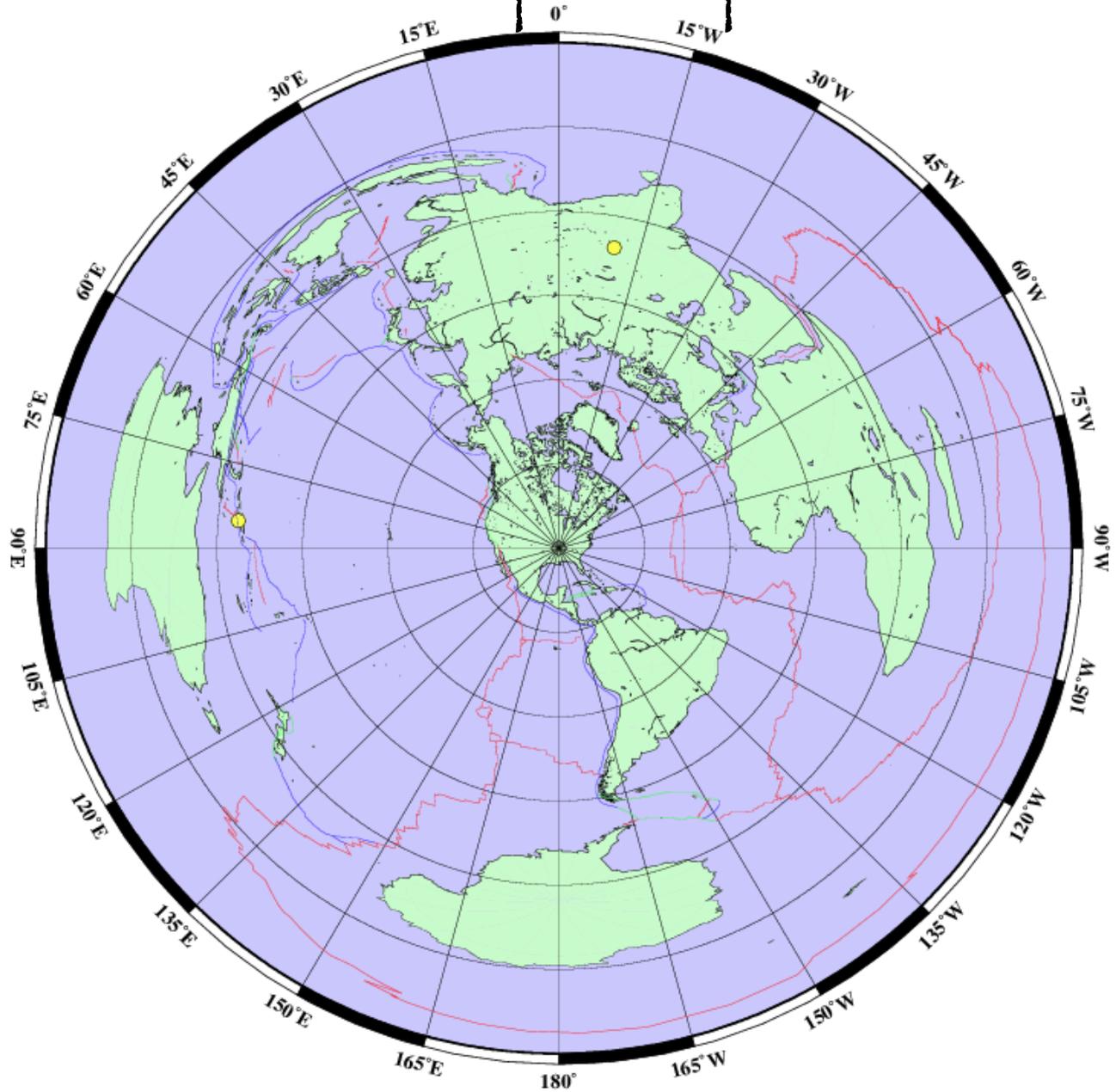
```
#set up map to be centered on lat lon given in command line
#draw crude coastlines, ocean blue, land green
#do not draw lat long grid (no frame specs on -B, could put w/next)
pscoast -R$WORLD Coast -Je$1/$2/$SCALE/180 -B:."Station $3 Map": -
S200/200/255 -G200/250/200 -W1 -Dc -P $MOREPS -X$XOFFSET -Y$YOFFSET >
$OUTPUTFILE
```

```
#set up new map centered on north pole and draw only the lat long grid
psbasemap -R$WORLD Coast -Je$GRIDCNTR -B15g15 -O -K >> $OUTPUTFILE
```

```
#RESET map to be centered on lat lon given in command line
#to put on some earthquake data read from this file
#data specified in lat long order, psxy assumes long lat (x,y) so
#use the "-:" switch to let psxy know (another common gotcha)
psxy -R$WORLD Coast -Je$1/$2/$SCALE/180 -Sc0.1 -G250/250/50 -W1/0/0/0
$CONTINUEPS -: <<END >> $OUTPUTFILE
-9.09 158.44
35.35 78.13
END
```

```
#add plate boundaries, notice don't have to respecify details of region
and projection but do need -R -Je
psxy -R -Je -M$ -W1/$RED $CONTINUEPS $ROOT/ptect/ridges >> $OUTPUTFILE
psxy -R -Je -M$ -W1/$GREEN $CONTINUEPS $ROOT/ptect/xforms >> $OUTPUTFILE
psxy -R -Je -M$ -W1/$BLUE $ADDPS $ROOT/ptect/trenches >> $OUTPUTFILE
```

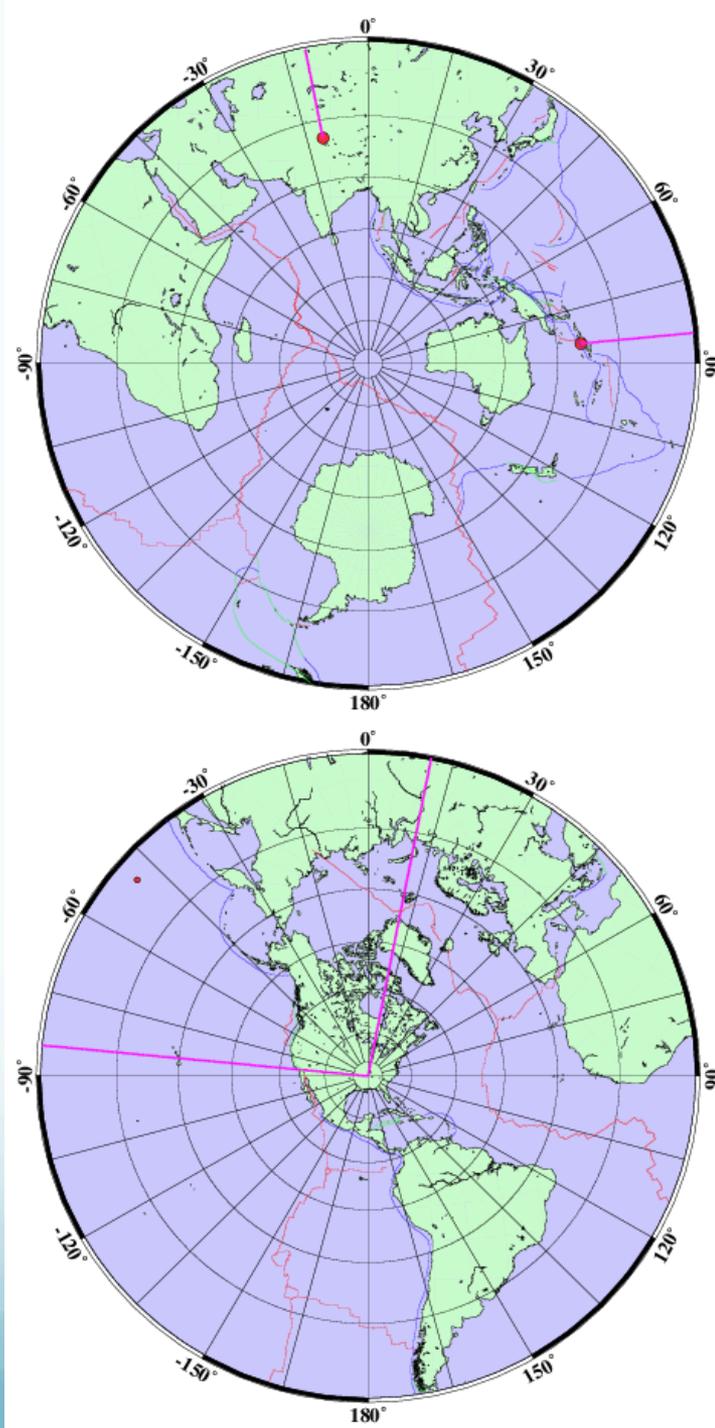
# Example map



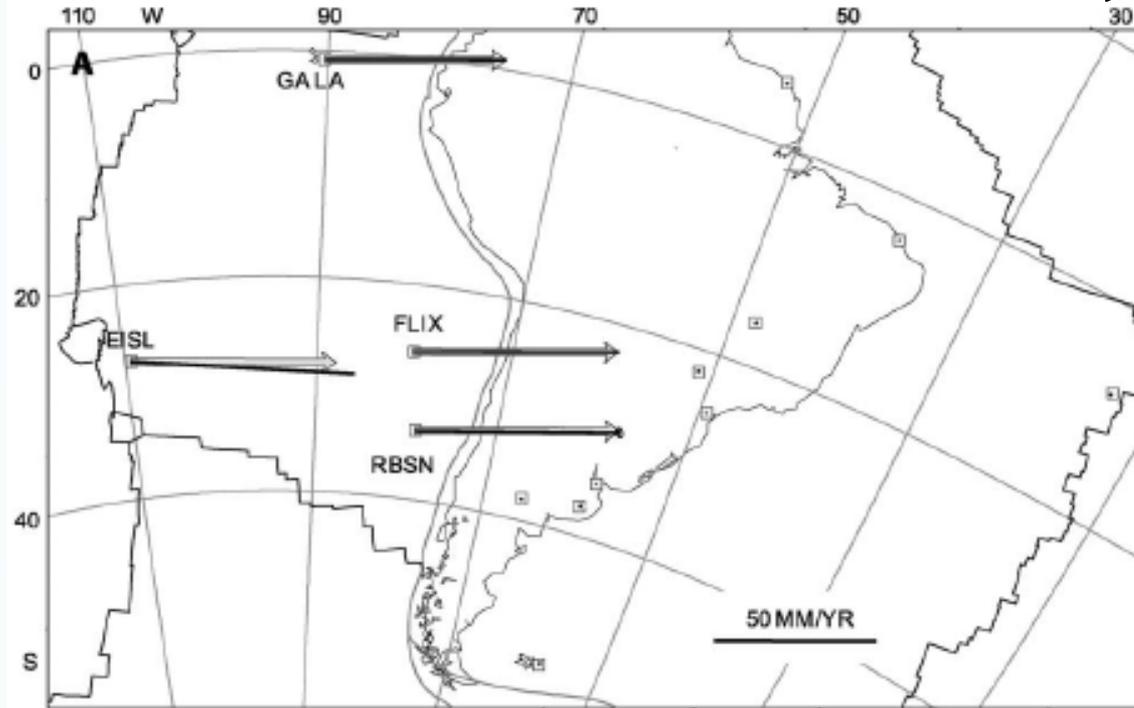
Another version of an azimuthal, equiangular map centered on Memphis and it's anti-pode.

Now it's a lot easier to identify landmasses on the other side of the globe by their shapes.

Also shows that great circles (the radial lines) converge at the anti-pode.



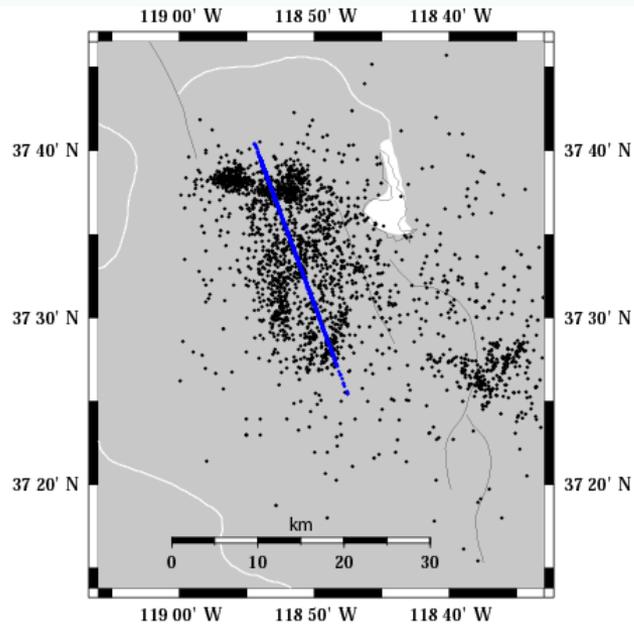
# Nazca-South America Euler pole



Data plotted in South America reference frame  
using oblique Mercator projection  
referenced to Euler pole

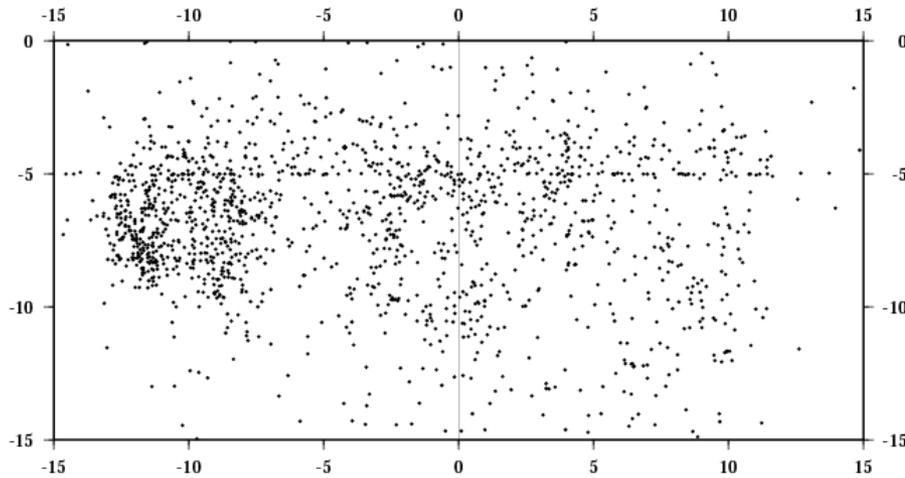
(points on South America plate have zero – or near zero – velocities.)

Plate motion follows lines of latitude  
(horizontal lines).



Make a cross section

(2 parts, draw map,  
draw cross section)



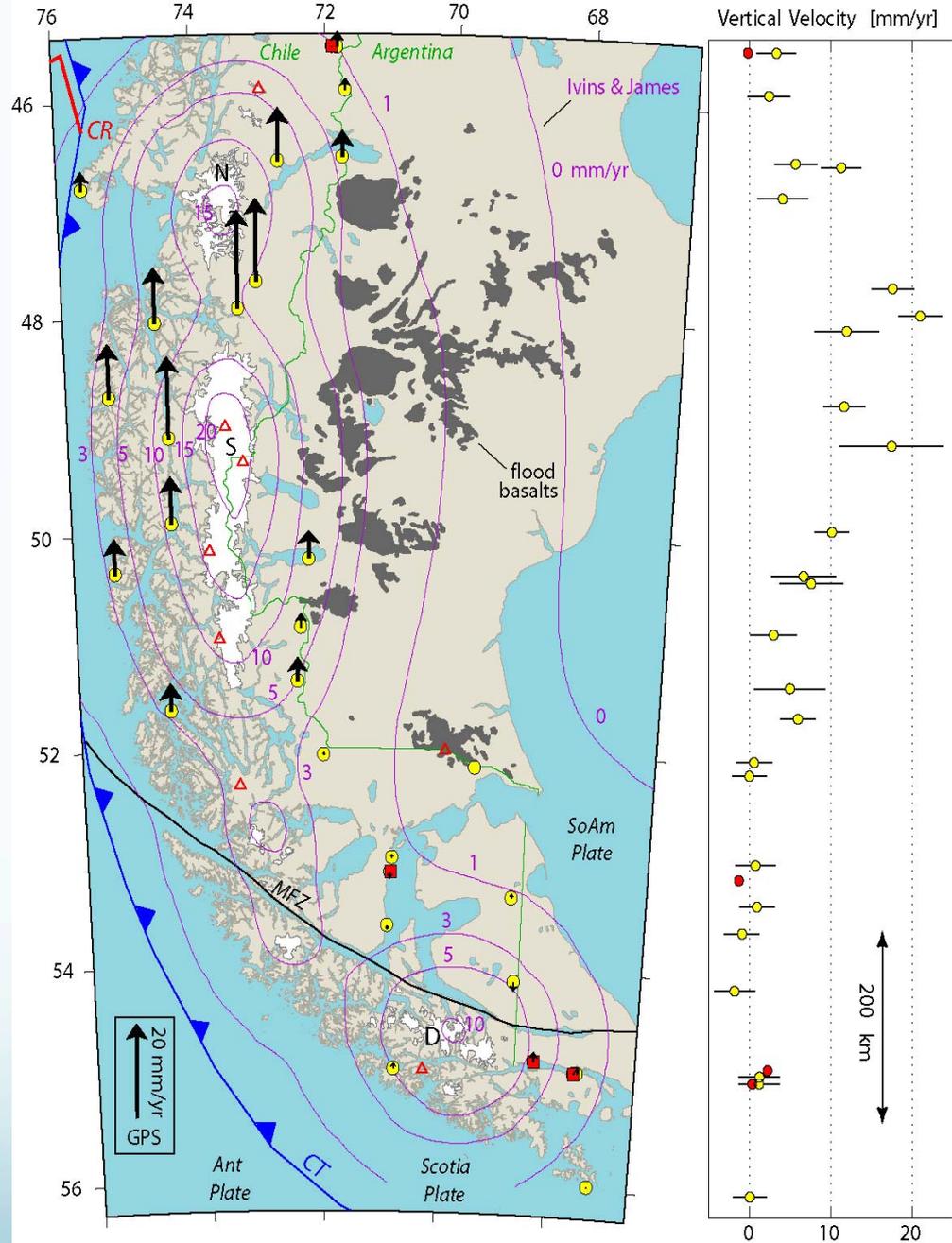
Data and non working version of shell script from  
[http://www-geology.ucdavis.edu/~gps/GMT/LONG\\_VALLEY/hypocenter.html](http://www-geology.ucdavis.edu/~gps/GMT/LONG_VALLEY/hypocenter.html)

```
# Set PARAMETERS FOR CROSS-SECTION PLOT
center="-118.85/37.55"
azimuth="160.0"
#3. DEFINE A BOX
width="-5/5"
length="-15/15"
\rm LV_seismicity.tmp
nawk '{print $1,$2, $3}' LV_seismicity.dat | project -C${center}\ -A$
{azimuth} -Q -W${width} -L${length} -V > LV_seismicity.tmp

# PLOT CROSS-SECTION HYPOCENTERS ON MAP
nawk '{print $6,$7}' LV_seismicity.tmp | psxy -J${projection} \
-R${range} -P -M -Sc0.03 -G0/0/255 -O -V -K >> ${psfile}
# PLOT CROSS-SECTION BOX
# SET PARAMETERS TO PLOT
brange="-15/15/-15/0"
bprojection="x0.2/0.2"
btick="a5f5g0/a5f5g0"
psxy box_dim -R${brange} -J${bprojection} -B${btick} -W1 -P -O \
-K -X-1.25 -Y-4 -V >> ${psfile}
# PLOT HYPOCENTERS ON CROSS-SECTION
nawk '{print $4, $3*(-1.0)}' LV_seismicity.tmp | psxy -P -M \
-J${bprojection} -R${brange} -Sc0.03 -G0/0/0 -O -V >> ${psfile}
```

psvelo does not plot vertical vectors – fake it. Plot vertical as N, E=0.

What about the contours? If exist as digital line data plot with psxy. If from raster data with values on a grid have to convert first then draw with psxy.



**ICE AND FIRE**

Postglacial rebound in Patagonia

```
echo make pgr contours
```

```
PGRFILE=pgr5e18
```

```
SPACING=4m
```

```
xyz2grd $SAMDATA/$PGRFILE -G$SAMDATA/$PGRFILE.grd -ISPACING /
```

```
 -: -R$REGION
```

```
grdinfo $SAMDATA/$PGRFILE.grd
```

```
grdcontour $SAMDATA/$PGRFILE.grd -C1 -Jx1.0 -D$PGRFILE.con -M /
```

```
-R$REGION > /dev/null
```

```
#have to hand edit the contour file to do 2 things -- as made the  
first point in each contour
```

```
#is stuck on the end of the new contour separator line - have to  
add <cr>, also does VERY bizzare
```

```
#stuff with > for segment separator, change to $ and works fine.
```

```
#exit
```

```
psxy -R$REGION -$PROJ/$SCALE -M -W$LINETHICK/$ICECOLOR /
```

```
$CONTINUE $PGRFILE.con $VBSE >> $OUTPUTFILE
```

First grid data for gmt.

Then contour.

Contour SRTM DEM data w/ GMT and convert into KML (using a program I wrote) but there is a new GMT routine to do this, `gmt2kml`, for points and lines.



`gmt2kml` reads one or more GMT table file and converts them to a single output file using Google Earth's KML format. Data may represent points, lines, or polygons, and you may specify additional attributes such as title, altitude mode, colors, pen widths, transparency, regions, and data descriptions. You may also extend the feature down to ground level (assuming it is above it) and use custom icons for point symbols.

Representing numbers on the computer.

Computer memory/processors consist of items that exist in one of two possible states (binary states).

These states are usually labeled  
0 and 1.

Each item in memory stores one “bit” of information. (whether it is a 0 or a 1).

How can we combine these “bits” into something useful?

We can let the two states represent the digits 0 and 1 of a positional, base 2 system.

(similar to our base 10 system, but with only 2 digits, 0 and 1, rather than 10 digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

This leads to a simple way to represent integers -  
- just use base 2.

Base 10		Base 2
0		0
1		1
2		10
3		11
4	When you run out of symbols, combine using idea of 0 and positional value	100
5		101
6		110
7		111
8		1000
9		1001
10		1010
11		1101

$a_n b^n a_{n-1} b^{n-1} \dots a_2 b^2 a_1 b^1 a_0 b^0$ , "b" = base, "a" element of set of symbols for base  $\{0, 1, \dots, b-1\}$

glitches

All the integers we can write down are finite (use some finite number digits), but size otherwise arbitrary  
(8, 147, 987346036. etc).

Computer takes this one step further by stating up front the number of digits (bits) in a number in memory. All numbers will have this number of bits (or multiples of it).

Introduce the “byte” – a group of 8 bits.

In general the computer does not work with individual bits – it works with bytes (8 bits at a time) or words (some number of bytes).

Bytes can be combined into “words”. Words were originally defined as 2 bytes (16 bits), but as computers got more powerful, words grew to 4 bytes (32 bits), and now 8 bytes (64 bits) with 16 bytes (128 bits on the horizon). (how things are combined will come back to bite us later)

Half a byte, 4 bits, is a “nibble”.

On the SUN, a word is 4 bytes  
("32 bit machine").

On the newest MacPro's (and PC's, since they  
both use the same INTEL chips) a word is 8 bytes  
("64 bit machine").

So let's say we are on a machine with 3 bit  
“words”.

(so we can write out the numbers)

We have seen that we can represent positive  
integers in base 2.

With 3 bits we can count to 7.  
(with  $n$  bits we can count to  $2^n - 1$ )

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

What about negative integers?

Now we have a problem.

We don't have a minus sign.

All we have are 0 and 1.

One "simple" solution is to use the most significant (MSB) or highest order bit (the leftmost one) as a "sign" bit.

000	0
001	1
010	2
011	3
100	-0
101	-1
110	-2
111	-3

Why the simple solution is not so simple.

This solution has some problems.

We have two values for zero (positive and negative, and we only have 6 distinct numbers, not 7).

This representation will also make binary arithmetic (add, subtract) on computers awkward (i.e. if numbers are expressed in this fashion).

Actual solution:

Use MSB to indicate the sign of the number – but in a slightly funky manner.

Use the idea of the

- Additive inverse – each positive number has an additive inverse.

combined with the fact that the computer has

- Finite precision – uses a fixed number of bits to represent a number.

How does this work? (using 4 digit numbers)

$$A=0101_2$$

Want additive inverse of  $A$ , a number that when added to  $A$  produces 0.

If we add  $1_2$  we get

$$A' = 0101_2 + 0001_2 = 0110_2$$

So we have made the least significant bit (LSB) a zero.

If we add  $11_2$  we get

$$A' = 0101_2 + 0011_2 = 1000_2$$

Now the 3 LSBs are zero.

If we add  $1011_2$  we get

$$A' = 0101_2 + 1011_2 = \mathbf{1}0000_2$$

Not a problem for us – we just write down the answer's 5 bits ---- but – our numbers in the computer are only 4 bits in size, the **1** is a carry into a 5<sup>th</sup> bit – but we don't have a 5<sup>th</sup> bit, so it goes into the bit bucket (gets thrown out).

$$A' = 0101_2 + 1011_2 = 10000_2$$

So with a limit of 4 bits we cannot add the two 4 bit positive numbers  $0101_2$  and  $1011_2$  because the answer needs 5 bits.

Keeping only the 4 bits we have.

$$A' = 0101_2 + 1011_2 = 0000_2$$

and we see that  $1011_2$  is the (4 bit) additive inverse of  $0101_2$ .

So we want  $0101_2$  to represent  $5_{10}$ ,  
and  $1011_2$  to represent  $-5_{10}$ ,  
While maintaining positional notation.

$$0101_2 = 0 * \text{something} + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5_{10}$$

and

$$1011_2 = 1 * \text{something} + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = -5_{10}$$

$$1011_2 = 1 * \text{something} + 3_{10} = -5_{10}$$

$$\text{So something} = -8_{10} = -2^3$$

So now we have,

$$abcd_2 = a * (-2^3) + b * 2^2 + c * 2^1 + d * 2^0$$

$$abcd_2 = a * (-8_{10}) + b * 4_{10} + c * 2_{10} + d * 1_{10}$$

which also maintains positional notation.

We can now count from  $-2^{n-1}$  to  $2^{n-1}-1$

(ex. for  $n=3$ ,  $2^3=8$ , so we can represent 8 values  
{-4, -3, -2, -1, 0, 1, 2, 3})

This representation of numbers is called two's complement.

Numbers written this way are two's complement numbers.

Two's complement numbers can be made using the "theory" presented, or by noticing that you can also form them by inverting all the bits and then adding 1!

$(5_{10}) = 0101_2 \rightarrow$  invert bits to obtain  $1010_2$   
then add 1

$$1010_2 + 0001_2 = 1011_2 = (-5_{10}). \text{ (compare to before)}$$

This method is trivial to do on a computer (which can really only do logical operations [and, or, exclusive-or, negate] on pairs of bits or bits [negate].)

So – to add on a computer just add the two binary numbers.

To subtract on a computer, just add the two's compliment of the number you are subtracting.

## Sizes of numbers (integers)

8 bit (byte) : -128 to +127  
(unsigned: 0 to +255)

16 bit (half word, word, short, int) : -32,768 to  
+32,767 (32K)  
(unsigned: 0 to +65,535 or 64K)

32 bit (longword, word, int) : -2,147,483,648  
to +2,147,483,647  
(unsigned: 0 to +4,294,967,295)

64 bit (double word, long word, quadword,  
int68) : -9,223,372,036,854,775,808 to  
+9,223,372,036,854,775,807  
(unsigned: 0 to  
+18,446,744,073,709,551,615)

128 bit (octaword) :

-170,141,183,460,469,231,731,687,303,715,884,105,728 to  
+170,141,183,460,469,231,731,687,303,715,884,105,727  
(unsigned: 0 to  
+340,282,366,920,938,463,463,374,607,431,768,211,455)

{ $10^{38}$  – a pretty big number – but not big enough to count the atoms in the universe –  
estimated to be  $10^{80}$ .}

So now we can add and subtract integers (also known as fixed point numbers)

(and multiplying by repetitive addition, division by repetitive subtraction).

What about -

Non-integer numbers?

Numbers outside the range of integers?

Enter – floating point numbers.

Non-integer numbers on the computer are limited to a subset of the rational numbers ( $a/b$  where  $a$  and  $b$  are integers and  $a/b$  can be represented exactly in the number of bits used by the computer).

Akin to scientific notation

$a.cde \dots * b^n$

where  $b$  is the base.

Floating point numbers can represent a wider range of numbers (bigger range of exponents) than fixed point numbers.

As with scientific notation – floating point numbers will have a number of digits in a decimal number (with a decimal point, not base 10) plus an exponent, which is used to multiply the decimal number by the base raised to that power.

$$2.235 \times 10^6$$

But now our number and base will be binary.

To maximize the precision, we will normalize floating point numbers such that there are no leading zeros to the left of the decimal point.

This determines the exponent.

After normalization all floating point numbers will have a 1 in their most significant bit (as a  $u$  always follows a  $q$ ).

Modern floating point format is IEEE 754 standard (there were at least as many as computer manufacturers for a long time).

Normalized (move decimal point so is after first non-zero digit) non-integer numbers are represented as

$$\left( 1. + \sum_{n=1}^{p-1} bit_n 2^{-n} \right) 2^m$$

$$\pi = 1.5707964 * 2^1 =$$

1.10010010000111110110110011111101010110010

$$(1 + 1 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} + \dots + 1 * 2^{-7} + 1 * 2^{-11}) * 2^1$$

In IEEE format the decimal point will always be after (a choice, it could be before as in DEC's version) the most significant non-zero digit (which can only be a 1 in base 2).

Implicit or hidden bit.

To milk another digit out of our floating point representation use the fact that for all numbers but zero, the first binary digit will be a 1, so we can throw it out.

(i.e. - not store it in the number. We have to remember to stick the implicit or hidden bit back in for calculations. This is done automatically by the hardware in the CPU.)

$$\pi = 1.5707964 * 2^1 =$$

1.1001001000011111101101100111111010101100

$$(1 + 1 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} + \dots + 1 * 2^{-7} + 1 * 2^{-11}) * 2^1$$

The decimal number is called the mantissa.

The exponent is called the exponent.

Typical size for floating point number is 32 bits – called single precision (double precision is 64 bits, quad precision 128 bits).

The IEEE floating point number consists of a

- 24 bit mantissa (including hidden bit),

- an 8 bit exponent (with a bias or excess to handle positive and negative values),

- and a sign bit (total 33 bits including hidden bit).

Rounding is done by adding 1 to the 24<sup>th</sup> bit if the 25<sup>th</sup> bit is a 1.

(we have 23 bits in the floating point number for the mantissa after taking the sign bit into account,

but one bit [the MSB] is implicit, so the last bit is really the 24<sup>th</sup> in the number)

To handle negative exponents we will just add 127  
(in the IEEE standard, some other floating point formats use 128) to the value of  
the exponent (in base 2). The exponent is an  
unsigned integer.

The base for the exponent is 2

(it was 16 on the IBM, which gave a much wider range of values for the exponent, but also  
much bigger round off errors because every change in the exponent shifted 4 rather  
than 1 bits).

So our floating point number is

1.   
SEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

example

$\pi$ 's first 32 bits =  $11.00100100001111110110101010001_2$

Round to 24 bits, normalize

$1.10010010000111111011011_2 \times 2^1$  

sign bit = 0

exponent = 1       $(11.0... = 1.10... \times 2^1)_2$

excess or bias  $127_{10}$  exponent =  $127_{10} + 1_{10} = 128_{10} = 10000000_2$

So our floating point number is  
example

Assemble

$\pi \approx 0$  10000000 1. 10010010000111111011011

Run together

01000000010010010000111111011011

(in hex 40490fdb)

Range of floating point numbers.

Single precision  
(32 bit number, 24 in mantissa)

About 7 decimal digits.

How to approximate number of base 10 digits  
from number of base 2 digits

$$2^{10} = 1024 \sim 10^3$$

$$\text{base 2 exponent} / \text{base 10 exponent} = \\ 10 / 3 = 3.3$$

$$(24 / 3.3 = 7.2)$$

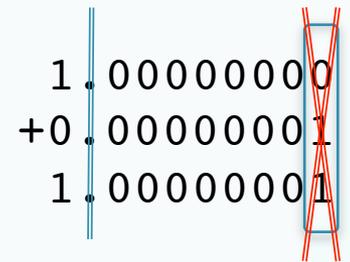
# Ranges for various sizes of floating point numbers

Type	Sign	Exponent	Mantissa	Total bits	Exponent bias	Bits precision
Single	1	8 ( $\pm 38$ )	23	32	127	$24_2$ ( $7_{10}$ )
Double	1	11 ( $\pm 308$ )	52	64	1023	$53_2$ ( $16_{10}$ )
Quad	1	15 ( $\pm 4,965$ )	112	128	16383	$113_2$ ( $34_{10}$ )

To add floating point numbers – have to line up the mantissas (shift based on exponent).

Potential problem when adding two numbers of very different magnitude.

eg.  $1.0 + 0.0000001 = 1.0$

$$\begin{array}{r} 1.00000000 \\ +0.00000001 \\ \hline 1.00000001 \end{array}$$


in single precision  
(does not give expected 1.00000001)

because we do not have enough bits to represent correct value (only 7 decimal digits).

(solution here is to go to double precision.)

$$\begin{array}{r} 1.0000000000000000 \\ +0.00000001000000 \\ \hline 1.00000001000000 \end{array}$$


## Another potential problem

Loss of significance when subtracting two numbers that are almost the same.

$$1.234567 - 1.234566 = 0.000001$$

Start out with 7 digit numbers, end up with single significant digit in new - seemingly 7 significant digit - number ( $1.000000 \times 10^{-6}$ ).

This problem is not solved by  
Increasing precision on computer.

```
1.23456700000000
-1.23456600000000
0.00000100000000
```

To multiply floating point numbers – add exponents, multiply mantissas.

On computer – result has same number significant digits (7 for single precision) as the two factors.

## Special values:

- Zero (no 1 bit anywhere to normalize on – all zeros)

+/- infinity

- NaN (result of operations such as divide by zero, sqrt -1 [except in matlab])

-Others

# Machine precision

Characterizes precision of machine representation.

epsilon or  $\epsilon_{\text{mach}}$

Value depends on number bits in mantissa and how rounding is done.

With rounding to zero,

$$E_{\text{mach}} = B^{\wedge}(1-P)$$

With rounding to nearest,

$$E_{\text{mach}} = (1/2) * B^{\wedge}(1-P)$$

Where  $B^{\wedge}(M) = B^M$ .

$$E_{\text{mach}}$$

Quantifies bounds on the *relative error* in representing any non-zero real number  $x$  within the normalized range of a floating point system:

$$\left| (f(x) - x) / x \right| \leq E_{\text{mach}}$$

Math vs what the computer does.

Due to finite precision and rounding the computer will (generally) not give what you might expect mathematically.

Mathematically  $\sin^2\theta + \cos^2\theta = 1$ .

But on the computer (finite precision, rational values only, ...) the test

$\sin^2\theta + \cos^2\theta == 1$

will return FALSE!

One solution to this problem is to test against a small number – the machine precision, rather than zero.

So test

$$\text{abs}(\sin^2\theta + \cos^2\theta - 1) < \text{epsilon}$$

if this is true consider, then we can consider  
 $\sin^2\theta + \cos^2\theta = 1.$

(same with test  $a == b$ ,  
use  $\text{abs}(a - b) < \text{epsilon}$ ).

# One last detail

Combining bytes into words.

Many ways to do it, and all were used (of course).

Two of the most popular are both still around.

Can cause value of numbers to be interpreted incorrectly.

Can cause major headaches

(some operating systems/programs can figure it out and fix it for you, others can't and you have to do it).

# Endianness

In byte (unsigned integer)

$2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0$

MSB on left, LSB on right.

What happens when I combine 2 bytes into a 16 bit number?



If I was starting with 16 bit numbers I'd do

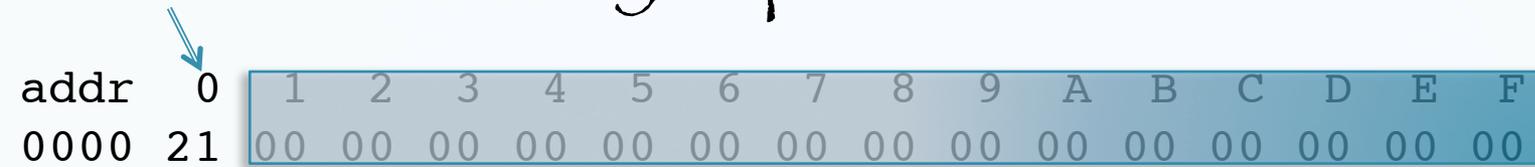
$2^{15}, 2^{14}, 2^{13}, 2^{12}, 2^{11}, 2^{10}, 2^9, 2^8, 2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0$

Two possible ways to combine.  
(and also several possible ways to visualize memory).

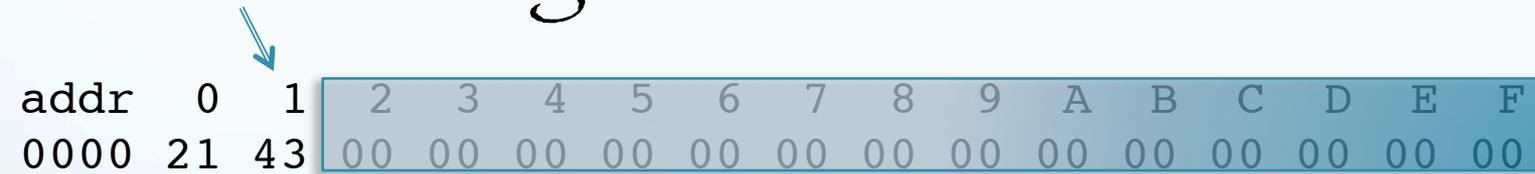
	MSByte	LSByte
Address	0	1

	LSByte	MSByte
Address	0	1

A number made up of just one byte would have that byte placed at address 0.



How do we expand this number to two bytes? We have 2 options. We could allow it to grow towards the right – the little endian form).



This puts the numbers "backwards", but allows us to extend the size of number to the limits of memory without having to move the least significant parts.

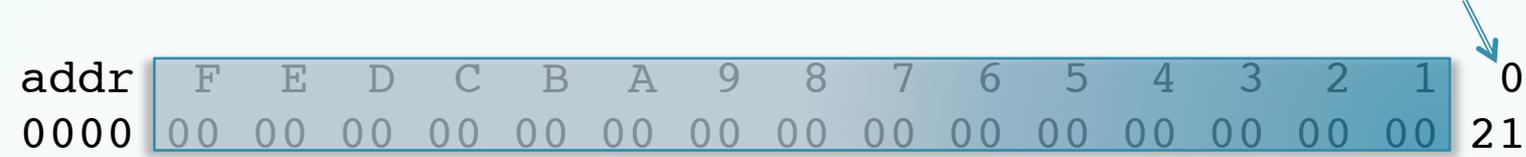
Alternately, we could slide the first byte to the right, changing its address, and then extend the number toward the left, the big endian form.

addr	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	43	21	00	00	00	00	00	00	00	00	00	00	00	00	00	00

This keeps the digits in the “correct” order, but forces a definite size into the number (one has to move the bytes with lower significance as add more bytes).

(the arrow indicates the base address when referring to the number).

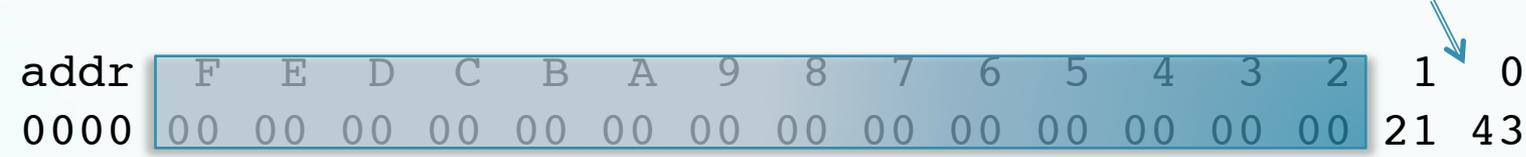




grows to become either (Little Endian):



or (Big Endian)

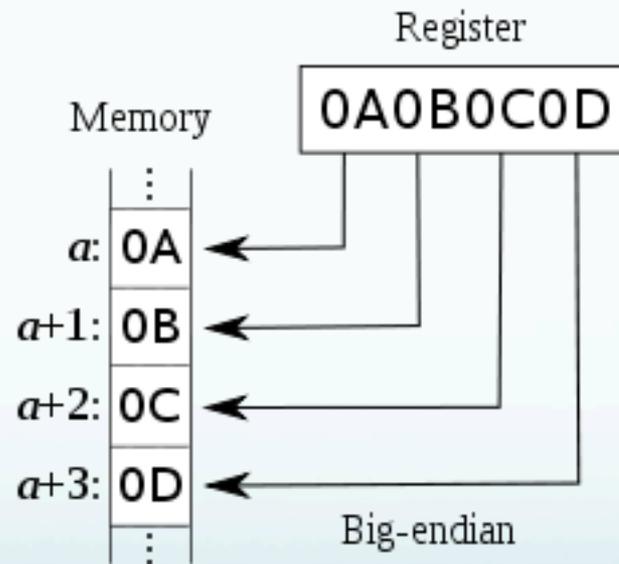


Suddenly, little endian not only looks correct, but also behaves correctly, grows left without affecting existing bytes.

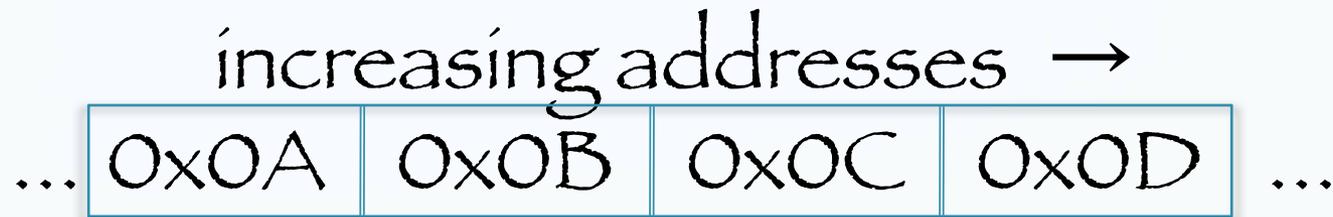
And, just as suddenly, big endian turns onto a bizarre rogue whose byte ordering doesn't follow the "rules".

# Big-endian

(Looking at memory as column going down.)



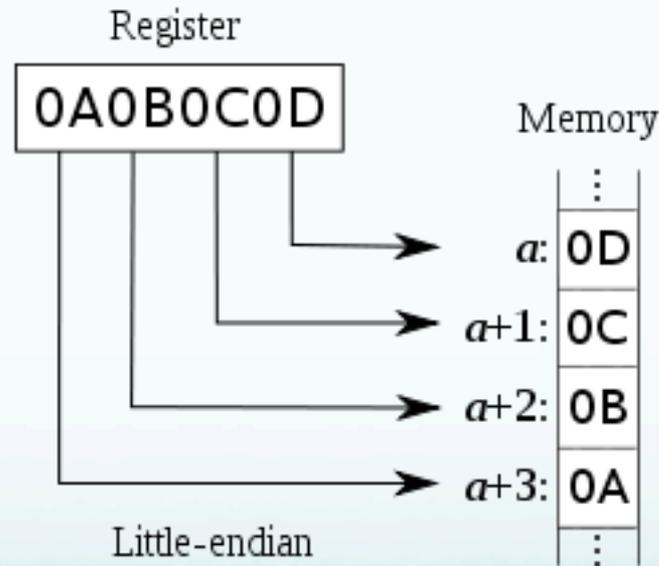
Big-endian - with 8-bit atomic element size and 1-byte (octet) address increment:



The most significant byte (MSB) value, which is 0x0A in our example, is stored at the memory location with the lowest address, the next byte value in significance, 0x0B, is stored at the following memory location and so on. This is akin to Left-to-Right reading in hexadecimal order.

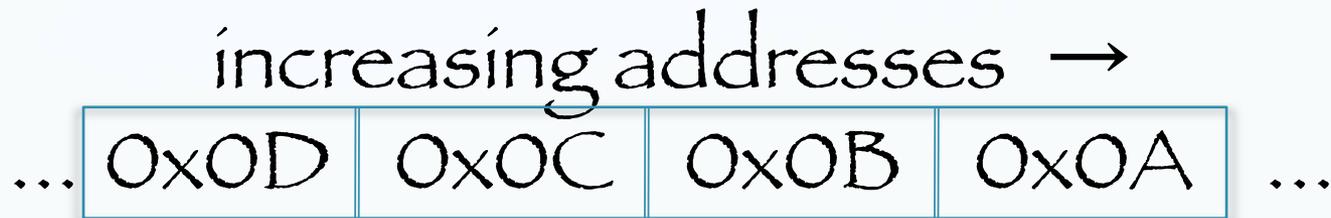
# Little-endian

(Looking at memory as column going down.)



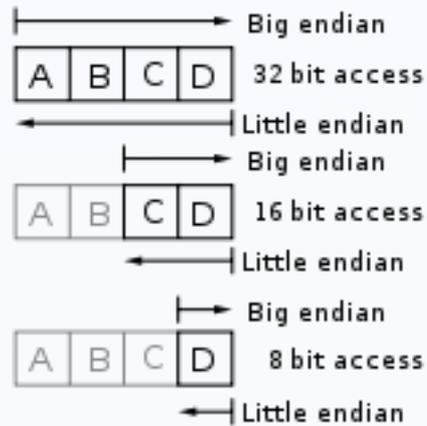
(As with left2right or right2left ordering of row form, reasonableness of behaviors would “switch” if looked at memory as column going up.)

Little-endian - with 8-bit atomic element size and 1-byte (octet) address increment:

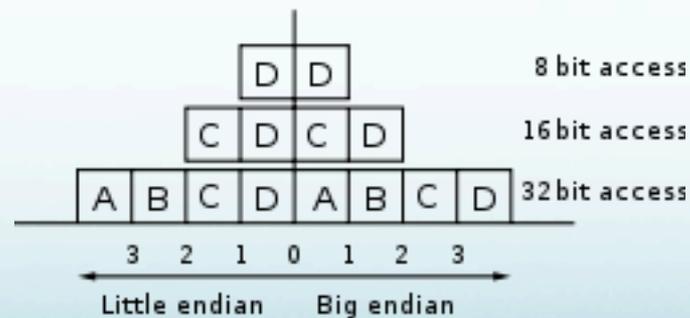


The least significant byte (LSB) value, 0x0D, is at the lowest address. The other bytes follow in increasing order of significance.

## Register



## Memory



Which way makes “more sense” depends on how you picture memory and which way they are numbered.

As rows, or columns.

Whether the rows go from left2right or right2left, or the columns go up or down.

Machines that use little-endian format include x86 (Intel), 6502, Z80, VAX, and, largely, PDP-11

Machines that use big-endian format include Motorola (pre Intel Macs), IBM, SUN (SPARC)

(machines/companies that started out with 8 bits typically used little-endian when they combined bytes. Machines/companies that started out with 16 bits typically used big-endian to break words into bytes since bit and byte counting go same way.)

What you need to know.

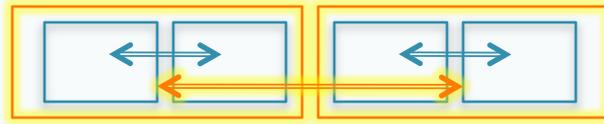
For binary data (not ASCII [basically letters] which is stored in a single byte) you have to know how it is stored. If it is stored the wrong way for your machine, you have to do a “byte swap” to fix it.

There are programs to do this.

(plus some programs, like the latest version of SAC, can figure it out – so you don’t have to worry about it.

Sometimes a problem with GMT and other geophysical binary data sets – especially since universities were in the SUN world).

When you byte swap, you also have to swap each grouping of  $2^n$  (e.g. for 32 bit numbers you have to swap words also).



Etc. for 64 bit, 128 bit, values.

When converting floating point (assuming base 2 exponent) have to worry about

- the exponent's excess value (IEEE uses 127, some other formats use 128 – a factor of 2)
- and position of assumed decimal point (before or after most significant bit with value of 1 (another factor of 2)).

Only have to worry about this stuff when moving (usually old) binary stuff between machines/architectures.

# Matlab

Introduction

MATLAB = MATrix LABoratory

Interactive system.

Basic data element is an array that does not require dimensioning.

“Efficient” computation of matrix and vector formulations (in terms of writing code – it is interpreted so loses efficiency there) relative to scalar non-interactive language such as C or Fortran.

# The 5 parts

-

1 - Desktop Tools and Development

2 - Mathematical Functions

3 - The Language

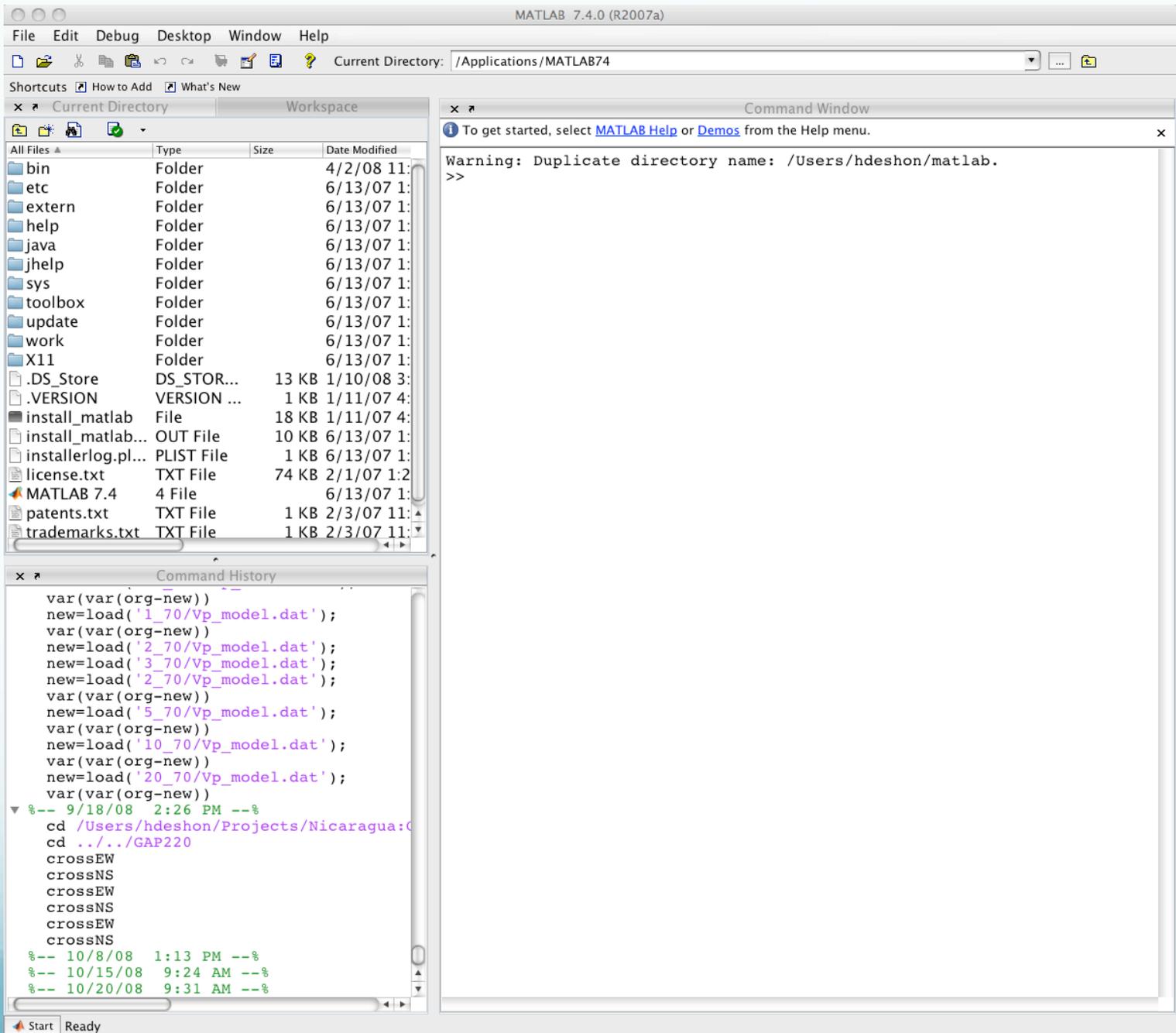
4 - Graphics

5 - External Interfaces

# Desktop Tools & Development

## Graphical user interfaces:

- MATLAB desktop and Command Window
  - Command history window
  - Editor and debugger
  - A code analyzer and other reports
- Browsers for viewing help, the workspace, files, and the search path.





# Mathematical Functions

Large collection of computational algorithms including but not limited to:

Elementary functions, like sum, sine, cosine

Complex arithmetic

Matrix math – inverse, eigenvalues/vectors, etc.

Fast Fourier transforms

Bessel functions

etc.

# Interactive help and documentation.

The screenshot shows the MATLAB Help Navigator window. The left pane displays a tree view of the help content, with 'Mathematics' selected. The right pane shows the 'Mathematics' page, which includes a list of topics and their descriptions.

**Help Navigator**

Search for:  Go

Example: "plot tools" OR plot\* tools

Contents Index Search Results Demos

- Begin Here
- Release Notes
- Installation
- MATLAB
  - Getting Started
  - Examples
  - Desktop Tools and Development Environment
  - Mathematics**
  - Data Analysis
  - Programming
  - Graphics
  - 3-D Visualization
  - Creating Graphical User Interfaces
  - Functions – By Category
  - Functions – Alphabetical List
  - Handle Graphics Property Browser
  - External Interfaces
  - C and Fortran Functions – By Category
  - C and Fortran Functions – Alphabetical List
  - Release Notes
  - Printable Documentation (PDF)
- MATLAB Compiler
- Control System Toolbox
- Image Processing Toolbox
- Mapping Toolbox
- Partial Differential Equation Toolbox
- Signal Processing Toolbox
- System Identification Toolbox

**Mathematics**

MATLAB<sup>®</sup> provides many functions for performing mathematical operations and analyzing data. The following list summarizes the contents of this collection:

- [Matrices and Linear Algebra](#) Describes matrix creation and matrix operations that are directly supported by MATLAB. Topics covered include matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations.
- [Polynomials and Interpolation](#) Describes functions for standard polynomial operations such as polynomial roots, evaluation, and differentiation. Additional topics covered include curve fitting and partial fraction expansion.
- [Fast Fourier Transform \(FFT\)](#) Describes what you can do with the fast Fourier transform (FFT) in MATLAB.
- [Function Functions](#) Describes MATLAB functions that work with mathematical functions instead of numeric arrays. These function functions include plotting, optimization, zero finding, and numerical integration (quadrature).
- [Differential Equations](#) Describes the solution, in MATLAB, of initial value problems for ordinary differential equations (ODEs) and differential-algebraic equations (DAEs), initial value problems for delay differential equations (DDEs), and boundary value problems (BVPs) for ODEs. It also describes the solution of initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs). Topics covered include representing problems in MATLAB, solver syntax, and using integration parameters.
- [Sparse Matrices](#) Describes how to create sparse matrices in MATLAB, and how to use them in both specialized and general mathematical operations.

Matrices and Linear Algebra

Biggest resource

GOOGLE/WEB

There are trillions of matlab tutorials, program exchanges, discussions, “toolboxes”, etc., on the web.

# The Language

## High-level matrix/array language

Includes control flow statements, functions, data structures, input/output, and object-oriented programming features

It allows both “programming in the small” to rapidly create quick and dirty throw-away programs, and “programming in the large” to create large and complex application programs.

Graphics:

Two-dimensional and three-dimensional data  
visualization.

Image processing.

Animation.

Presentation graphics.

## Graphics:

It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build complete GUIs for your own applications.

## External Interfaces

Library that allows you to write C and Fortran programs that interact with MATLAB.

It includes facilities for calling routines from MATLAB (dynamic linking), for calling MATLAB as a computational engine, and for reading and writing MAT-files.

# Toolboxes

Add-on application-specific solutions

Comprehensive collections of MATLAB functions (M-files) to solve particular classes of problems.

Examples include:

- Signal processing
- Image processing
- Partial differential equations
  - Mapping
  - Statistics

Search for:  Go

Example: "plot tools" OR plot\* tools

Contents Index Search Results Demos

- Begin Here
- ▶ Release Notes
- ▶ Installation
- ▼ MATLAB
  - ▶ Getting Started
  - ▶ Examples
  - ▶ Desktop Tools and Development Environment
  - ▶ **Mathematics**
  - ▶ Data Analysis
  - ▶ Programming
  - ▶ Graphics
  - ▶ 3-D Visualization
  - ▶ Creating Graphical User Interfaces
  - ▶ Functions - By Category
    - ▶ Functions - Alphabetical List
  - ▶ Handle Graphics Property Browser
  - ▶ External Interfaces
  - ▶ C and Fortran Functions - By Category
    - ▶ C and Fortran Functions - Alphabetical List
  - ▶ Release Notes
  - ▶ Printable Documentation (PDF)
- ▶ MATLAB Compiler
- ▶ Control System Toolbox
- ▶ Image Processing Toolbox
- ▶ Mapping Toolbox
- ▶ Partial Differential Equation Toolbox
- ▶ Signal Processing Toolbox
- ▶ System Identification Toolbox



Title: Mathematics (Mathematics)

## Mathematics

# Mathematics

MATLAB® provides many functions for performing mathematical operations and analyzing data. The following list summarizes the contents of this collection:

### [Matrices and Linear Algebra](#)

Describes matrix creation and matrix operations that are directly supported by MATLAB. Topics covered include matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations.

### [Polynomials and Interpolation](#)

Describes functions for standard polynomial operations such as polynomial roots, evaluation, and differentiation. Additional topics covered include curve fitting and partial fraction expansion.

### [Fast Fourier Transform \(FFT\)](#)

Describes what you can do with the fast Fourier transform (FFT) in MATLAB.

### [Function Functions](#)

Describes MATLAB functions that work with mathematical functions instead of numeric arrays. These function functions include plotting, optimization, zero finding, and numerical integration (quadrature).

### [Differential Equations](#)

Describes the solution, in MATLAB, of initial value problems for ordinary differential equations (ODEs) and differential-algebraic equations (DAEs), initial value problems for delay differential equations (DDEs), and boundary value problems (BVPs) for ODEs. It also describes the solution of initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs). Topics covered include representing problems in MATLAB, solver syntax, and using integration parameters.

### [Sparse Matrices](#)

Describes how to create sparse matrices in MATLAB, and how to use them in both specialized and general mathematical operations.

# Starting MATLAB

Runs on SUNS, MACS, PC's – same interface.

From CERI unix machines, just type

```
%matlab
```

On a PC/Mac, double-click the Matlab icon.

# Starting MATLAB

In an X11 window (assuming it is in your path), type

```
%matlab
```

Useful trick from remote machines

```
%matlab -nojvm
```

*or*

```
%matlab -nodesktop -nosplash
```

turns off the graphical interface – which is SLOW and buggy over net (actually does not work).