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 – 13, 10/8/13

OLNY SRMAT POELPE CAN RAED TIHS

I cdnuolt blveiee taht I cluod aulaclty uesdnatnrd waht I was rdanieg. The phaonmneal pweor of the hmuan mnid. Aoccdrnig to rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer be in the rghit pclae. The rset can be a taotl mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe. Amzanig huh? yaeh and I awlyas tghuhot slpeling was ipmorantt!

Tihs deos not wrok for the cetupmor!

Shells

Basics of the Unix/Linux Environment

What is a shell?

As far as Unix is concerned, the shell is just another program.

As far as the user in concerned, it is the traditional command line user interface with the Unix operating system…it interprets your typing.

What is a shell?

Just as there are many flavors of Unix and Unix-like systems, there are many types of shells.

If you don't like any of the shells in existence, this is Unix – write your own!

Common shells

Bourne Shell	sh
Bourne Again Shell	bash

(current default on MAC OS X)

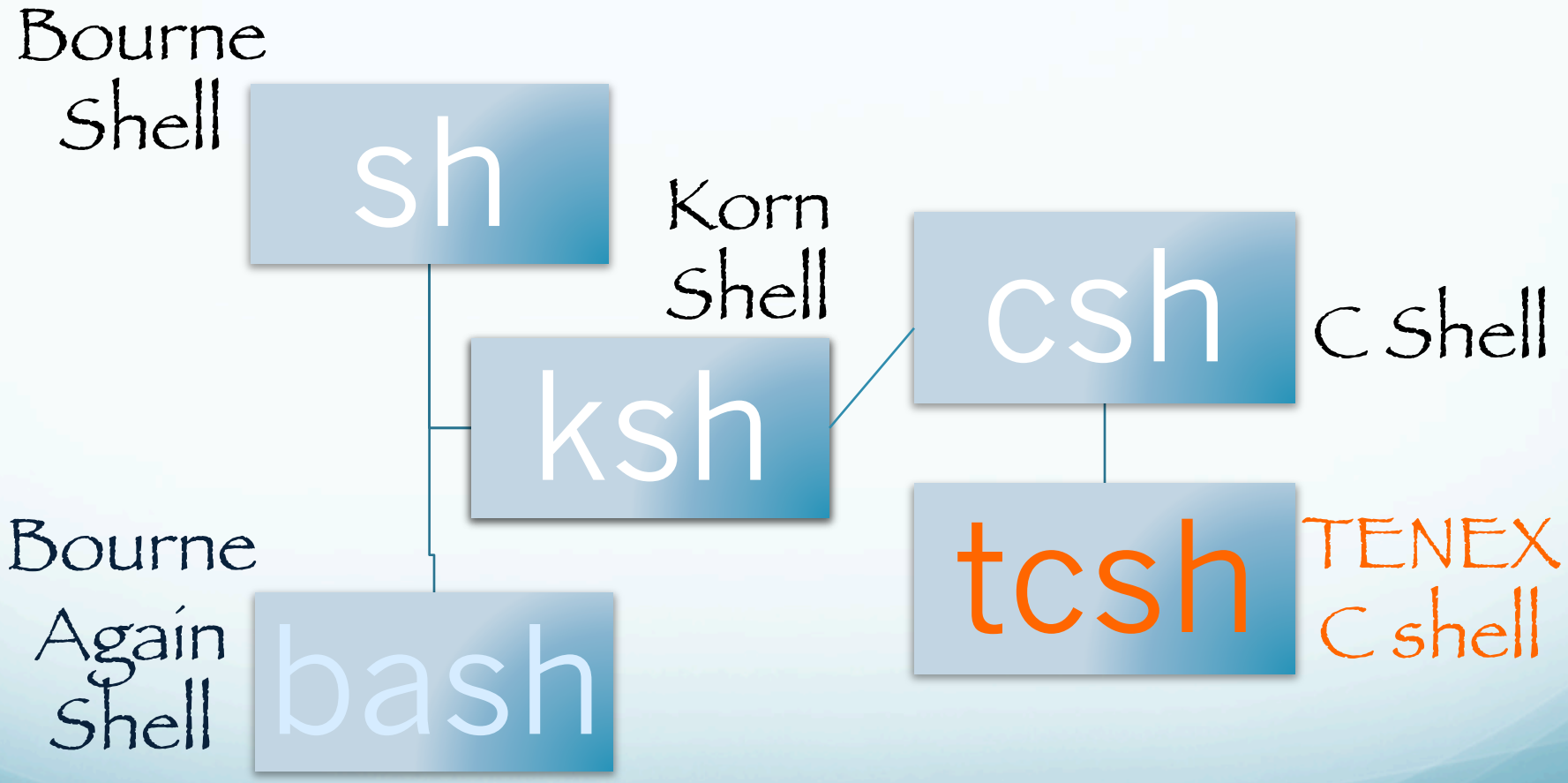
C Shell	csh
TENEX C Shell	tcsh

(This is the default shell at CERI)

Korn Shell	ksh

(mix between two shell families above)

Common shells



sh

Bourne shell

The original Unix shell.

Pro: Flexible and powerful scripting shell.

Con: Not interactive or particularly user friendly.

csh

C shell

designed for the BSD Unix system.

syntax closely follows C programming.

Pro: easy for C programmers to learn and comes with many interactive features such as file completion, aliases, history.

Con: not as flexible or powerful a scripting language as sh or bash.

ksh

Korn shell

derived from the Bourne shell so has a shared syntax.

job control taken from the C shell.

bash

Bourne-Again shell

Combines the "best" of sh, ksh, and csh.

Default shell (out of the box) on Linux and Mac OSX operating systems.

Pro: Flexible and powerful scripting language with all the interactive features of csh plus command completion.
This shell is great for complicated GMT scripts.

tcsh

TENEX C shell

Default shell of the CERI unix environment.

Pro: User friendly on the command line.

Con: It is not as suitable for long and involved scripts.

It is perfectly OK for most daily geophysics work on the command line & most faculty here use it on a daily basis so there are many experts around.

Features bash and tcsh Shells

Basics of the Unix/Linux Environment

Useful features of tcsh & bash

-file completion-

key the tab key, or the escape key twice, to "complete" the name of a long file.

Say I have a file named

`largest-deadliest-eqs-last-100-years.ai`

I can type just enough so the system can continue (i.e. there are no options for the next letter – assume I also have a file `lapilona.dat`)

`$ls lar<tab>` will produce this

`$ls largest-deadliest-eqs-last-100-years.ai`

Useful features of tcsh & bash

-file completion-

Say I have 2 files file named

```
ls largest-deadliest-eqs-last-50-years.ai
ls largest-deadliest-eqs-last-100-years.ai
```

Actually I can type just enough so it can continue on its own for a while

`$ls lar<tab>` will produce this

`$ls largest-deadliest-eqs-last-`

At which point it gets stuck. I help it along

`$ls largest-deadliest-eqs-last-1<tab>`

`$ls largest-deadliest-eqs-last-100-years.ai`

Useful features of tcsh & bash

history command

list the previous commands entered during the active session.

```
148:> history
. . .
   145   21:30    pwd
   146   21:30    DEM
   147   21:30    cd srtm
   148   21:30    history
```

Useful features of tcsh & bash

~history "feature"~

Shell keeps "history" of commands

up and down arrow keys: allow you to move up and down through previous commands.

right and left arrow keys: allow you to edit command lines (backspace to remove, type at cursor to insert) without starting from scratch.

Useful features of tcsh & bash

bang (“ ! ”) command/shortcut

Bang is used to search backward through your *Bash/tcsh* history until it finds a command that matches the string that follows the bang and returns/executes it.

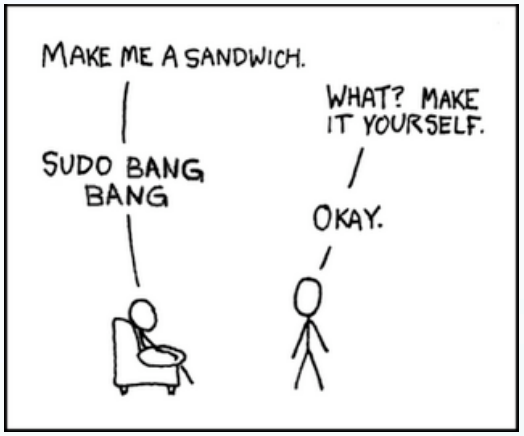
bang ("!") command/shortcut

!!: reruns the last command in the history list.

```
% vi foo.c bar.c
% !!
```

Becomes:

```
% vi foo.c bar.c
```



!vi: reruns the last command in the history file beginning with "vi".

```
% vi foo.c bar.c
% ls
% !vi
```

Becomes:

```
% vi foo.c bar.c
```

bang ("!") command/shortcut

!XXX<CR> returns the command numbered XXX in the history list. It runs it after you enter the <CR>.)

```
148:> history
. . .
   145  21:30   pwd
   146  21:30   DEM
   147  21:30   cd srtm
   148  21:30   history
149:> !146
DEM
/gaia/home/rsmalley/dem
150:>
```

bang ("!") command

!-X: returns the command X back in the history list and runs it at the <CR>.

```
151:> history
. . .
   147   21:30    cd srtm
   148   21:30    cd ~
   149   21:30    history
   150   21:46    DEM
   151   21:55    history
152:> !-4
cd ~
/gaia/home/rsmalley
153:>
```

bang ("!") command/shortcut is actually more general – use it to return commands from history and do something with them.

For the purposes of these tips, every tip will assume these are the last three commands you ran:

```
% which firefox
% make
% ./foo -f foo.conf
% vi foo.c bar.c
```

Getting stuff from the last command:

Get the last argument ("$") from command :

```
% svn ci !$
```

Becomes:

```
% svn ci bar.c
```

Various shells have options that can affect this.

Be careful with shells that let you share history among "instances" (if you have 5 terminals open you have a shell running in each one. Each running copy is an "instance"). You can also have shells running in the "background" (almost never needed with modern gui's, was essential with single terminal).

Some shells also allow bang commands to be expanded with tabs or expanded and reloaded on the command line for further editing when you press return.

bang ("!") command/shortcut.

For the purposes of these tips, every tip will assume these are the last three commands you ran:

```
% which firefox
% make
% ./foo -f foo.conf
% vi foo.c bar.c
```

Getting stuff from the last command:

All arguments ("*", special definition):

```
% svn ci !*
```

Becomes:

```
% svn ci foo.c bar.c
```

bang ("!") command/shortcut.

For the purposes of these tips, every tip will assume these are the last three commands you ran:

```
% which firefox
% make
% ./foo -f foo.conf
% vi foo.c bar.c
```

Getting arguments from the last command:

First argument ("`:N`"):

```
% svn ci !!:1
```

Becomes:

```
% svn ci foo.c
```

bang ("!") command/shortcut

For the purposes of these tips, every tip will assume these are the last three commands you ran:

```
% which firefox
% make
% ./foo -f foo.conf
% vi foo.c bar.c
```

Accessing command lines by pattern: (saw this already, but now with ./, need to go to first letter)

Full line:

```
% !./f
```

Becomes:

```
% ./foo -f foo.conf
```

bang ("!") command/shortcut

```
% ls -d a*.f
atantest.f
% make
% ./foo -f foo.conf
% vi foo.c bar.c
```

Accessing command lines by pattern and command substitution:

This:

```
% vi `!ls`
```

Becomes:

```
% vi `ls -d a*.f`
```

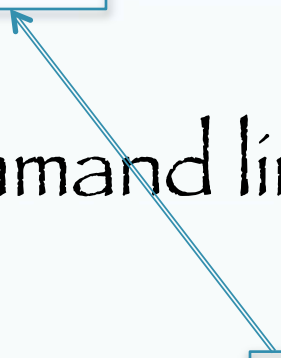
Which becomes:

```
% vi atantest.f
```

bang ("!") command/shortcut

For the purposes of these tips, every tip will assume these are the last three commands you ran:

```
% which firefox
% make
% ./foo -f foo.conf
% vi foo.c bar.c
```



Accessing command lines by pattern:

All args : `% ./bar !./f:*`

Becomes: `% ./bar -f foo.conf`

We are looking for the command that begins with "./f", and then we want (the colon, ":") all of its arguments (the splat, "*")

bang (“!”) command/shortcut

Notice how this makes perfect sense under the Unix philosophy.

Make a tool and (mis/ab)use it.

(the basic commands are really very simple, but in tricky combination they become very powerful - and confusing.)

Most normal people are not going to use all these shortcuts, they are just too complicated.

I showed them, however, to present additional application and appreciation of the Unix philosophy.

When you Google for help with Unix the answers/ examples are usually maximally Unixified, so you will have to figure it out.

bang ("!") command/shortcut

you can also view the command that bang finds without immediately executing it.

```
!cat:p<CR>
```

Now, instead of executing the command it finds, bang prints the command to Standard OUT for you to look at.

bang ("!") command/shortcut

```
!cat:p<CR>
```

That's not all though, it also copies the command to the end of your history (even though it was not executed).

This is useful because if you do want to execute that command, you can now use the *bang bang* shortcut to run it (*bang bang* runs the last thing in history).

How typically Unix.

bang ("!") command/shortcut

```
$ !cat:p<CR>
cat tst.sh
$ !! | grep "hello"<CR>
```

Here, the most recent command containing *cat* is printed, and copied to the end of your history.

Then, that command is executed with its results being piped into the **grep** command, which has been specified to print those lines containing the string "hello".
(We are following Unix philosophy)

bang ("!") command/shortcut

To find a lot of this "neat" stuff, I GOOGLEd

"unix bang command"

---

you will not find it in the man pages

```
147:> man !
No manual entry for !.
148:>
```

Modify last command in history list using caret or circumflex accent, "^", to fix typos or make small changes.

Replaces text inside first two carets with that between second and third.

(can sometimes skip closing caret as shown below in second example.)

```
$ ls trk1.kml
trk1.kml
$ ^1^2^
ls trk2.kml
trk2.kml
$ !!:p
ls trk2.kml
$ ^2^1
ls trk1.kml
trk1.kml
$
```

First it shows it to you and executes the edited command.

Environment (esoteric and essential)

Basics of the UNIX/Linux Environment

The UNIX Environment (general and CERI specific)

Mitch/Bob/Deshone have set up the basic CERI environment on both the Macs and Suns so that everyone can access the standard UNIX tools and geophysics packages available on the UNIX systems at CERI.

The UNIX Environment

But what does this mean?

Many UNIX utilities, including the shell, need information about *you* and what *you're* doing in order to do a reasonable job.

What kinds of information?

Well, to start with, a lot of programs (particularly editors) need to know what kind of terminal *you're* using.

Your environment is composed of a number of

environment variables

which provide this important information to the operating system.

Rather than forcing you to type this (hard to remember, where does one find it?) information with every command

such as (`% mail -editor vi -term aardvark48`)

UNIX uses *environment variables* to store information that you'd rather not worry about.

For example, the *TERM* environment variable tells programs what kind of terminal you're using. Any programs that care about your terminal type know (or ought to know) that they can read this variable, find your terminal type, and act accordingly.

UNIX commands receive information from three potential sources.

-Arguments on the command line

-Data coming down their standard input channel.

-The *environment*. When a command is started, it is sent a list of *environment variables* by the shell.

Since you generally want the computer to behave the same way everyday, these

environment variables

are setup and stored in

configuration files

that are accessed automatically at login.

What are your environment variables?

The commands **`env`**, or
`setenv` with no parameters,

print the current environment variables to the Standard Out.

```
141:> env
USER=rsmalley
LOGNAME=rsmalley
HOME=/gaia/home/rsmalley
PATH=.:/gaia/home/rsmalley:/gaia/home/rsmalley/bin:/gaia/home/
rsmalley/shells:/gaia/home/rsmalley/dem:/gaia/home/rsmalley/
defm:/gaia/home/rsmalley/defm/src:/gaia/home/rsmalley/
visco1d_pollitz/viscoprogs_rs:/gaia/home/rsmalley/gg:/gaia/home/
rsmalley/gg/com:/gaia/home/rsmalley/gg/gamit/bin:/gaia/home/
rsmalley/gg/kf/bin:/gaia/dunedain/d2/gps/bin:/gaia/smeagol/local/
passcal.2006/bin:/gaia/smeagol/local/gmt/GMT4.2.1/bin:/usr/sbin:/
usr/local/teTeX/bin/sparc-sun-solaris2.8:/gaia/home/rsmalley/
bin:/opt/local/sbin:/opt/sfw/bin:/usr/bin:/usr/ccs/bin:/usr/
local/bin:/opt/SUNWspro/SC5.0/bin:/opt/local/bin:/usr/bin:/usr/
dt/bin:/usr/openwin/bin:/bin:/usr/ucb:/gaia/smeagol/local/bin:/
net/gps4/d1/Noah/rbh/usr/PROGRAMS.330/bin:/gaia/home/rsmalley/X/
bin:/gaia/home/rsmalley/X/com:/gaia/home/rsmalley/record_reading/
bin:/gaia/home/rsmalley/record_reading/scripts
MAIL=/var/mail//rsmalley
SHELL=/usr/bin/tcsh
TZ=US/Central
LC_CTYPE=en_US.ISO8859-1
LC_COLLATE=en_US.ISO8859-1
```

```
LC_TIME=en_US.ISO8859-1
LC_NUMERIC=en_US.ISO8859-1
LC_MONETARY=en_US.ISO8859-1
LC_MESSAGES=C
SSH_CLIENT=75.66.47.230 50561 22
SSH_CONNECTION=75.66.47.230 50561 141.225.157.63 22
SSH_TTY=/dev/pts/12
TERM=xterm
HOSTTYPE=sun4
VENDOR=sun
OSTYPE=solaris
MACHTYPE=sparc
SHLVL=1
PWD=/gaia/home/rsmalley
GROUP=user
HOST=alpaca.ceri.memphis.edu
REMOTEHOST=c-75-66-47-230.hsd1.tn.comcast.net
MANPATH=/gaia/smeagol/local/passcal.2006/man:/gaia/smeagol/local/
gmt/GMT4.2.1/man:/ceri/local/man:/usr/dt/man:/usr/man:/usr/
openwin/share/man:/usr/local/man:/opt/SUNWspro/man:/opt/sfw/man:/
usr/local/teTeX/man:/gaia/smeagol/local/man
LD_LIBRARY_PATH=/gaia/smeagol/local/gmt/lib:/gaia/opt/SUNWspro/
lib:/gaia/opt/SUNWspro/SC5.0/lib:/usr/lib:/usr/openwin/lib
```

```
LM_LICENSE_FILE=/gaia/opt/licenses/licenses_combined
EDITOR=vi
AB2_DEFAULTSERVER=http://stilgar.ceri.memphis.edu:8888
PRINTER=3892
```

You get all the stuff shown so far automatically.

If you can figure it out, you can change it to suit yourself.

(But when you break-it, don't ask [or humbly ask] the system managers for help. If you were smart enough to break it, you're smart enough to fix it.)

```
GMTHOME=/gaia/smeagol/local/gmt/GMT4.2.1
NETCDFHOME=/gaia/smeagol/local/gmt
GMT_GRIDDIR=/gaia/smeagol/local/gmt/GMT4.2.1/share/dbase
GMT_IMGDIR=/gaia/smeagol/local/gmt/GMT4.2.1/DATA/img
GMT_DATADIR=/gaia/smeagol/local/gmt/GMT4.2.1/DATA/misc
CWD=/gaia/home/rsmalley
HELP_DIR=/gaia/home/rsmalley/gg/help/
INSTITUTE=uom
RECORD_READING=/gaia/home/rsmalley/record_reading
RECORD_READING_BIN=/gaia/home/rsmalley/record_reading/bin
RECORD_READING_SCR=/gaia/home/rsmalley/record_reading/scripts
RECORD_READING_SRC=/gaia/home/rsmalley/record_reading/src
latestrtvel=rtvel4_9305_5bv19
LATESTRTVEL=rtvel4_9305_5bv19
ANONFTP=/gaia/midtown/mid4/smalley/public_ftp
ANONFTP_IN=/gaia/midtown/mid4/smalley/public_ftpinbox
SACDIR=/gaia/tesuji/d1/local/sac
SACXWINDOWS=x11
SACAUX=/gaia/tesuji/d1/local/sac/aux
SACSUNWINDOWS=0
GPSHOME=/gaia/dunedain/d2/gps
```

Plus you can add our own stuff (above).

Unless you are running Linux (in which case you are the system manager), you can forget about setting up most of this as the system managers do it for you.

There are a few environment variables, however, that you need to know about and/or set up yourself.

HOME*

This environment variable controls what UNIX commands consider your (base) home directory.

This is how "**cd**" and "~" know which directory to refer to

```
% echo $HOME
/gaia/home/rsmalley
```

To refer to the value of an environment variable put a **$** in front of the name.

*these environment variables should not be changed by the user

The $ therefore has a special meaning to the shell.

(As do the characters " `~, !, /, *,?,^,\` " all of which we have already seen.

By the time we are done we will have used up most of the non alpha-numeric characters with special meanings.)

SHELL*

This variable stores your default shell

```
% echo

/usr/bin/tcsh
```

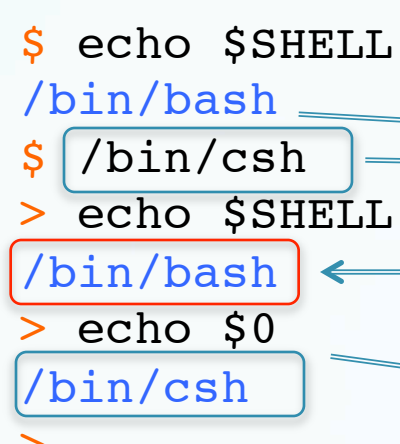
Seems pretty simple!

How to find your SHELL

```
$ echo $SHELL
/bin/bash
$ /bin/csh
> echo $SHELL
/bin/bash
```

Start csh

OOPS!



SHELL

How to really find your shell

Start bash (inside csh, inside bash)

0 is the shell variable containing the name of the program that is running - the shell. The shell is just another program to UNIX. $0 is value of shell variable 0).

SHELL
How to really find your shell

```
$ echo $SHELL
/bin/bash
$ /bin/csh
> echo $SHELL
/bin/bash
> echo $0
/bin/csh
> ps -p $$
PID TTY           TIME CMD
91456 ttys003    0:00.02 -bin/csh
> /bin/tcsh
>> echo $SHELL
/bin/bash
> echo $0
/bin/tcsh
> ps -p $$
PID TTY           TIME CMD
91467 ttys003    0:00.03 -bin/tcsh
> exit
exit
> exit
exit
$ echo $0
-bash
```

$ is the shell variable containing the process id (pid), $$ is value of of shell variable $ (very UNIX).

What happens if we enter $SHELL all by itself?

```
$ $SHELL
```

The shell sees

```
$ /bin/bash
```

Since a shell variable is just a character string, it replaces the $SHELL with the character string.

So if the shell variable is a command or otherwise interpretable by the shell it will try to do it.

Can also id the shell by the prompts ($, >, etc., once you know which is which).

These examples also show that the shell is just another program – the only thing special about it is that it is the program that is started automatically for you when you login.

Finally, What is my shell?

This seems to be the best way to find out.

```
% echo $0
```

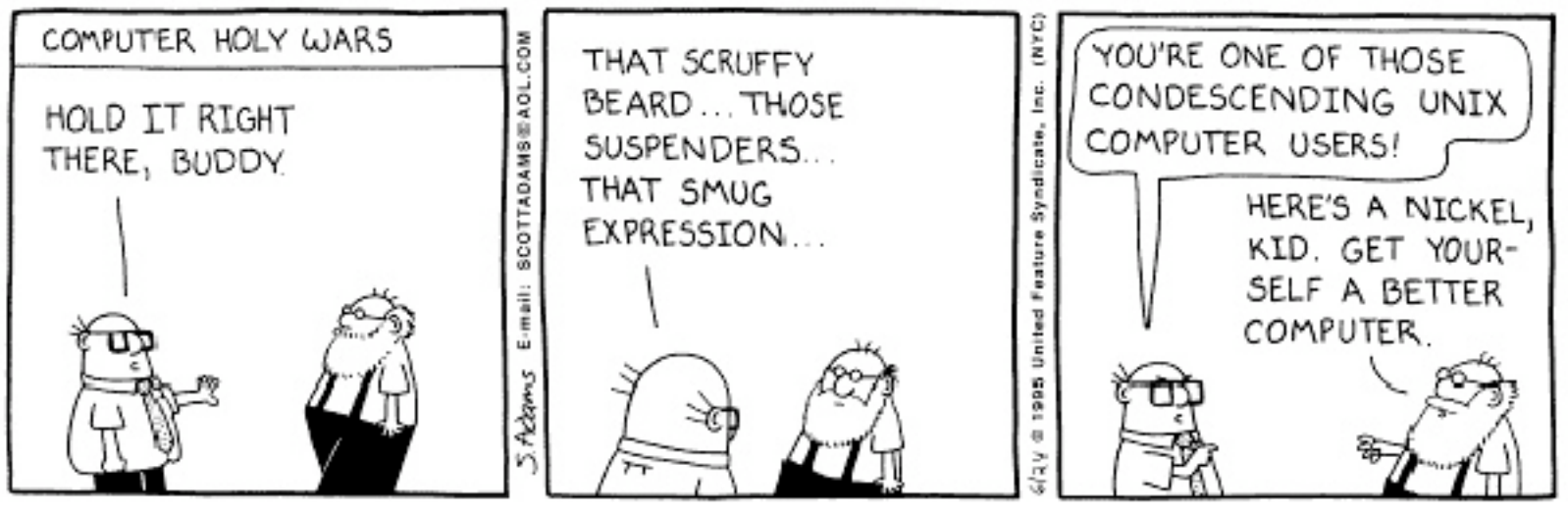
Works for csh, tcsh, sh, and bash.

($0 does not refer to the shell in general, this is one of the UNIX "standards" that $0 is the program you are running!!

[which in this case is the shell – perfect UNIX logic]).

"people who have trouble with typing commands should not be using a computer."

Response of the UNIX community to criticism that UNIX ignored the needs of the unsophisticated user.



Environment variables are managed by your shell.

The difference between
environment variables
and regular
shell variables
is that

a shell variable is local to a particular instance of the shell (such as your current shell or a shell script), while environment variables are "inherited" by any program you start, including another shell.

That is, the new process gets its own copy of these variables, which it can read, modify, and pass on in turn to its own children.

In fact, every UNIX process (not just the shell) passes its environment variables to its child processes.

Example (very important) environment variable, what it is used for, and how to maintain it (you will probably need to do this at some point).

PATH

To see the value of the environment variable PATH, echo it to the screen.

PATH

This environment variable tells the shell where to find executable files

```
%echo $PATH
.:/gaia/home/rsmalley:/gaia/home/rsmalley/bin:/gaia/home/
rsmalley/shells:/gaia/home/rsmalley/dem:/gaia/home/rsmalley/
defm:/gaia/home/rsmalley/defm/src:/gaia/home/rsmalley/
visco1d_pollitz/viscoprogs_rs:/gaia/home/rsmalley/gg:/gaia/home/
rsmalley/gg/com:/gaia/home/rsmalley/gg/gamit/bin:/gaia/home/
rsmalley/gg/kf/bin:/gaia/dunedain/d2/gps/bin:/gaia/smeagol/local/
passcal.2006/bin:/gaia/smeagol/local/gmt/GMT4.2.1/bin:/usr/sbin:/
usr/local/teTeX/bin/sparc-sun-solaris2.8:/gaia/home/rsmalley/
bin:/opt/local/sbin:/opt/sfw/bin:/usr/bin:/usr/ccs/bin:/usr/
local/bin:/opt/SUNWspro/SC5.0/bin:/opt/local/bin:/usr/bin:/usr/
dt/bin:/usr/openwin/bin:/bin:/usr/ucb:/gaia/smeagol/local/bin:/
net/gps4/d1/Noah/rbh/usr/PROGRAMS.330/bin:/gaia/home/rsmalley/X/
bin:/gaia/home/rsmalley/X/com:/gaia/home/rsmalley/record_reading/
bin:/gaia/home/rsmalley/record_reading/scripts
```

The “:” is used to separate each full path name in sh, bash (space for csh, tcsh).

When you run a command (from the terminal or a shell script), your shell looks through each directory in your *PATH* variable , in order, until it finds the first instance of an executable file with the name of the command.

It then runs the command.

```
%echo $PATH
.:/gaia/home/rsmalley:/gaia/home/rsmalley/bin:/gaia/home/
rsmalley/shells:/gaia/home/rsmalley/dem:/gaia/home/rsmalley/
defm:/gaia/home/rsmalley/defm/src:/gaia/home/rsmalley/
visco1d_pollitz/viscoprogs_rs: etc.
```

My path starts with dot ("**.**").

This is convenient (or else you have to type the relative path **./myprog** to execute programs in the working directory) but this is considered a security weakness.

Next I have a number of my directories where I've written Fortran or C programs and shell scripts.

In the "standard" UNIX organization that you will see in most books, one is supposed to put all your executable programs in your

"`~/bin`" directory

and all your shell scripts in your

"`~/scripts`" directory.

You will probably not find many people (or systems) that do this anymore.

So how does this work?

If you are working a program to do least squares analysis and decide to call it "`ls`" what will happen when you enter the command "`ls`"?

It depends.

What happens depends on your path.

Remember that to UNIX, everything outside the kernel (including the shell) is just a file.

Some of these files are executable (programs).

When the shell goes looking through your path for an executable file (has to have executable set in file permissions) named "`ls`", it will run the first one it finds.

If the directory containing your least squares program (executable file), "`ls`", is in your path

Before

the directory containing the UNIX list command, "`ls`", it will run your program and you will not be able (at least simply) to get a listing of your directory!

(How to solve this? Have to give full path to the system `ls`, `/bin/ls` for example. You need to know where the system `ls` lives.)

If the directory containing your least squares program, "`ls`", is in your path

after

the directory containing the UNIX list command, "`ls`", it will run the UNIX `ls` command and you will not be able (at least simply) to run your program!

(How to solve this? Give full path to your program `ls`, e.g. `~/myprogs/ls` or relative path `./ls`, `../myprogs/ls` etc.

Can't solve this with the "\" we saw before since this undoes an alias. It does not change the path.

which

Command that shows what the shell finds for the command name.

```
$ which ls
/bin/ls
```

Or if you have redefined `ls` and it is found in your path first

```
$ which ls
/user/smalleyb/in/ls
```

To run a specific executable file – give its full path.

```
$ /bin/ls
!              Public
Adobe SVG 3.0 Installer Log   Sites
Desktop              bin . . .
```

More examples.
Can use all the tricks in specifying paths.

Run from one directory up.

```
$ pwd
/Users/smalley/bin
$ ../hello.sh
Hello
$
```

If the file is in the working directory, and that directory is not in your path, use the dot.

```
$ ./hello.sh
Hello
$
```

This behavior is not a "bug", it is considered to be desirable and an example of the POWER of UNIX.

(This is also where the security problem comes in when dot is in your path. If someone compromises your system and puts a malicious file in your directory with the name "`ls`" and you have dot in your path and don't notice the file "`ls`" and enter "`ls`" to get a list, you execute the bad file instead.)

How you make your path is up to you.

Will see how to do it next.

Modifying your environment

Modifying your environment

If you mess up modifying the environment in your current window – you may "break" your current window (shell).

This is generally not a problem on the sun, mac, etc.

The environment is local to that window/shell.

Just close it and open another window.

How to change/set shell and environment variables.

In csh/tcsh use commands

set for regular (local) shell variables and

setenv for environment (global) variables.

```
set term = xterm
setenv TERM = xterm
```

We already mentioned the difference between regular shell and environment variables.

(you have to know that xterm is something that the shell will understand.)

If you need to deal with this level of UNIX, go find a wizard
(Bob Debula, Mitch Withers).

This syntax is also specific to csh/tcsh.

```
set term = xterm
setenv TERM = xterm
```

(note that since UNIX is case sensetive this is two environment variables. A local one “term” and a global one “TERM”).

To do the same thing in bash.

```
term=xterm
TERM=xterm
```

Note that there are NO SPACES on either side of the equals sign here.

How do we tell the difference between a regular shell variable and an environment variable in bash.

(there is really no difference within an instance of a shell).

(**set** with no parameters lists shell variables, **env** also lists environment variables. In csh - **setenv** with no parameters lists environment variables,)

In sh/bash, when we define a variable, it is a regular shell variable.

To make it an environment variable (one that is inherited) you export it.

```
term=xterm
TERM=xterm
EXPORT term
EXPORT TERM
```

setenv:

The csh/tcsh command to change environment settings.

Can be run on the command line,
from within a local configuration file
(.cshrc or .login),
or in a shell script.

When run it without specifying an environment variable, it will print all environment variables to the screen

How to change/set your path in csh/tcsh.

```
% setenv   PATH    {$PATH}:/gaia/home/rsmalley/scripts
```

This adds the (text string that is the) name of the directory

`'/gaia/home/rsmalley/scripts'`

to the end of the current environment variable PATH associated with the active shell.

When UNIX starts, you automatically get a path environment variable (it may be, but probably is not, empty) and this is the best candidate for the one you will have to change.

The environment variable is just a text string.

The shell interprets it.

setenv:

```
% setenv   PATH    /gaia/home/rsmalley/scripts:{$PATH}
```

Operationally it adds the the directory

`/gaia/home/rsmalley/scripts`

to the path, this time at the beginning.

What are the braces "{" "}" for?

They delimit the shell variable used with the `$`.
They are needed when characters that could be in a variable name follow it without a space.

```
SANJUAN=/volumes/seismicdata/panda/sanjuan
```

… `${SANJUAN}_disk1` or …`{$SANJUAN}_disk1`

expands to `/volumes/seismicdata/panda/sanjuan_disk1`

while

… `$SANJUAN_disk1`

tries to expand a variable named `SANJUAN_disk1`

Which probably does not exist.

setenv:

```
% setenv   PATH    /gaia/home/rsmalley/scripts:{$PATH}
```

Note PATH is used twice. On the right with the $ it refers to the current value of the environment variable.

On the left it refers to the name of the environment variable that is being set to the string on the right (including the old value).

setenv:

```
% setenv   PATH    /gaia/home/rsmalley/scripts:{$PATH}
```

In this case it will append the current value of PATH to the new information and put everything in a new version of PATH.

(sort of like `vari=vari+1` in Fortran, Matlab, c, etc. This is not a mathematical algebraic equation.)

If you don't write any of your own programs (or always use the path to the program/file) you will not have to change your path from the default.

(The default path at CERI will give you the path to the tcsh (and other) shell(s), and the paths to the tools such as MATLAB, SAC GMT, and some others.)

Modifying your default environment.

We already saw that you can always change things in your current environment [and that of any new child process] using the setenv command.

But it will get old changing everything to the way you want it each time you log in/open a new window/start a new shell.

And this being UNIX, there is a (easy) way to set up your own personal environment.

Modifying your default environment.

The setup of your personal environment (personal changes/preferences for how you want the shell to work for you) in csh and tcsh is stored in the file named

.cshrc

(there is also a file .login, but it is not likely you will have to change it (it get's used when you log in, not each time you start a shell) – so I'll mention it for completeness, but let's ignore it.)

When to make your own environment variables.

Anytime you want a global definition of something.

```
417:> grep rtvel .cshrc
setenv latestrtvel rtvel4_9305_5bv19
setenv LATESTRTVEL $latestrtvel
```

Modifying your default environment variable PATH using the .cshrc (.bashrc) file.

We are now doing brain surgery on ourselves.

In a mirror.

This is dangerous.

So--

Make a back up of the current, working .cshrc (.bashrc) file before you change it.

Have a second terminal window open in case you mess your file up so completely and break your active window.

This way you have another window open to delete the offending file and restore things from the backup file. (Unless you run the command to change it in a window, the environment is static once a window is open.)

You want this window open BEFORE you make the change, as any window opened after the file is saved will use the modified, bad, .cshrc (.bashrc) file.

For your path, you will see something like this in your .cshrc file.

```
set path = (. ~ ~/bin ~/shells ~/dem ~/defm ~/defm/src $path )
```

Which uses the set command (local) rather than the setenv command (global).

The man page for set says
var = value set assigns value to var, where value is one of:
word - A single word (or quoted string).
(wordlist) - A space-separated list of words enclosed in parentheses.

Ex. using the command set with the environment variable path also sets the environment variable PATH (tcsh).

```
265:> set path = ( $path ~/ESCI7205 )
```

Now look at the environment variable PATH (using a script I wrote to put out each entry on a separate line)

```
266:> ExaminePath.sh
.
/gaia/home/rsmalley
/gaia/home/rsmalley/bin
. . .

/gaia/home/rsmalley/record_reading/scripts
/gaia/home/rsmalley/ESCI7205
267:>
```

The ESCI7205 entry was not there before.

When you set path, it also changes PATH.
When you setenv PATH, is also changes path.
They seem to track.

I've not been able to find documentation on how this works. (I think one is for sh/bash and one for csh/tcsh)

But this is what you will see in both the universal .cshrc (/etc/.cshrc), and if you make changes, in your own .cshrc file.

It has been copied down through the ages.

After changing your path in the current shell.

First see what your path is.

```
266:> ExaminePath.sh
.
/gaia/home/rsmalley
/gaia/home/rsmalley/bin
. . .

/gaia/home/rsmalley/record_reading/scripts
267:>
```

.cshrc (csh resource script) configuration file (aka dot file)

```
setenv PATH .:/gaia/home/rsmalley/bin:$PATH
setenv PATH ${PATH}:/gaia/home/rsmalley/record_reading/bin
setenv PATH {$PATH}:/gaia/home/rsmalley/record_reading/scripts
setenv PRINTER 3892
alias cd              'cd \!*;echo $cwd'
alias home             "cd ~"
alias del            'rm -i'
set history=500
set ignoreeof
set savehist=500
set filec
```

.bashrc (bash resource script) configuration file (aka dot file)

Slightly different

```
PATH=.:/Users/robertsmalley:/Users/robertsmalley/bin:$PATH
```

Or (usually – so children inherit it)

```
export PATH=.:/Users/robertsmalley:/Users/robertsmalley/bin:$PATH
```

```
export PATH=$PATH:/Users/robertsmalley/gamit_globk_10.4/com:/Users/robertsmalley/gamit_globk_10.4/gamit/bin:/Users/robertsmalley/gamit_globk_10.4/kf/bin
```

Once you have made changes to your .cshrc (.bashrc) (and saved them), which is just a file, how do you have them activated in your current window/shell?

(at this point they will be activated in any new shell/window/login)

You could log out and then log back in (not very efficient as you loose your history, but it works), or open a new window (ditto) and work there.

Use the source command with the .cshrc (.bashrc) file as input. (don't need the input redirect "<")

```
151:> source .cshrc
152:>
```

source: executes configuration files

If you change your configuration file, you will need to execute source in all open terminal windows for the changes to take effect. The changes automatically will take effect when new terminal windows/shells are opened.

Say you have edited the .cshrc file.

```
%nedit  ~/.cshrc
%source ~/.cshrc
```

The default .cshrc file that everyone at CERI gets when they login, open a window, or start a shell is stored (on the SUN) in the file

/etc/.cshrc

And on the Mac in the file

/etc/csh.cshrc

After that the shell looks in your home directory for a .cshrc, which is used to expand upon and/or override the CERI values.

HOST*: environment variable with the name of the machine you are currently logged into.

REMOTEHOST*: environment variable with the name of the machine you are sitting in front of, if different (e.g. you are in the class on a PC and have used the program ssh to log into a sun at CERI.).

```
161:> echo $HOST $REMOTEHOST
alpaca.ceri.memphis.edu
162:>
```

SSH_CLIENT: the IP (internet protocol) address and port of the HOST machine.

SSH_CONNECTION: the IP addresses and ports of the HOST machine and the REMOTEHOST machine.

```
162:> echo $SSH_CLIENT $SSH_CONNECTION
75.66.47.230 51704 22 75.66.47.230 51704 141.225.157.63 22
163:>
```

If you want to get as much info as you can about the IP addresses. (Can also put in the name and get the address.)

```
169:> nslookup 141.225.157.63
Server:  dns1.memphis.edu
Address:  141.225.253.21

Name:    alpaca.ceri.memphis.edu
Address:  141.225.157.63

170:> nslookup 75.66.47.230
Server:  dns1.memphis.edu
Address:  141.225.253.21

Name:    c-75-66-47-230.hsd1.tn.comcast.net
Address:  75.66.47.230

171:>
```

Aside ---

How to destroy your input data file and how to prevent doing that (i.e. accidently doing it).

First – look at file.

```
262:> more flong.dat
1
2
3
4
5
6
7
8
9
10
6
7
8
9
10
263:>
```

Sort it, using the sort command.

```
263:> sort flong.dat
1
10
10
2
3
4
5
6
6
7
7
8
8
9
9
264:>
```

So far OK.

Say we want to save the sorted output to a file. Use redirection.

```
264:> sort flong.dat > flong.dat
265:> more flong.dat
266:>
```

We just erased our file!

Say we want to save the sorted output to a file. Use redirection.

```
264:> sort flong.dat > flong.dat
265:> more flong.dat
266:>
```

We just erased our file!

UNIX says we will need an output file, and (unless your sys admin has done the non-UNIX philosophy action of setting "no-clobber") it has permission to clobber a pre-existing output file – so it does. It then goes looking for the input file , which it cannot find because it just erased it!!

(Notice that this is not consistent with the PATH=$PATH:/mybin or x=x+1 model!)
(consistency is the hob-goblin of little minds)

It sorts nothing (the now empty input file) and puts it into the output file.

It sees no reason to complain, warn you, etc.

You are an adult, this behavior is "obvious" from the operating principles of UNIX.

(and if you had not figured that out by logically thinking everything through like a lawyer, you have now been told!)

Bet you erase a few files before you learn this.

Say we want to save the sorted output to a file. Use redirection.

```
264:> sort flong.dat > flong.dat
flong.dat: File exists.
```

UNIX says we will need an output file but your sys admin has done the non-UNIX philosophy action of defining "no-clobber" to protect you from yourself so UNIX cannot make the output file (since it cannot erase the pesky file with the same name).

Having no-clobber set prevents you from inadvertently erasing existing files

It protects you from yourself.

Very non-UNIX philosophy.

(and no-clobber typically only works from the terminal, in a shell script it will gladly clobber your file.)

Say we want to save the sorted output to a file. Use redirection.

```
264:> sort flong.dat > flong.dat
flong.dat: File exists.
265:> sort flong.dat >! flong.dat
266:> more flong.dat
267:>
```

But if you insist, you can still erase your input file!

The `>!` says to redirect the output to the file `flong.dat` and clobber a file with that name if you need to (i.e. a file with that name exists).

Tricky distinctions in sh and bash

set is a shell command to set the value of a shell attribute variable; these are internal variables used by the shell program.

env is a program that runs another program with modified environment variables.

Tricky distinctions

The major difference is that the env command will never modify the shell's own environment (only that of the child process), while set will.

set can also change settings like brace expansion within the shell.

You might also want to look at export, which changes the environment variables for all future commands.

Tricky distinctions

Enter the commands set and env and look at the differences (it will help if you sort the output of env)

(e.g. - noclobber is a shell attribute variable, not an environment variable – it is "set" and "unset" using set)

How to set/clear noclobber

```
-bash 532 ~ # set -o | grep noclobber
noclobber          off
-bash 533 ~ # set -o noclobber
-bash 534 ~ # set -o | grep noclobber
noclobber          on
-bash 535 ~ # set +o noclobber
-bash 536 ~ # set -o | grep noclobber
noclobber          off
-bash 537 ~ #
```

Aliases

Basics of the UNIX/Linux Environment

Alias

The `alias` and `unalias` commands allow you to rename, or define/undefine "shortcuts" (including mental), for commands.

Their use parallels their name – you are using another name, that is easier to type/remember, for something.

You can set an alias in your shell interactively (you will only have it locally and in child processes)

or set in your configuration files (.cshrc/.bashrc) so it is available every time you login, start a shell or open a new terminal window (which starts a shell for that terminal window).

Typical UNIX think.

When to make/use aliases.

Anytime you find yourself typing the same command over and over, you could make an alias.

Anytime you prefer to type a command "your way".

Typical UNIX think.

When to make/use aliases.

Anytime you find yourself mis-typing the same thing over and over, you could make an alias

("mroe" is usually aliased to the more command for example {why learn to type?}.
The original, interactive spelling corrector!).

Example aliases taken from .cshrc on CERI SUN system (so you get these automatically).

```
alias   settitlebar     'echo -n "^[]2;$CWD^G"'
alias   cd              'chdir \!* && cwdcmd && settitlebar'
alias   howmuch         'du -sk .'
alias   a               alias
alias   h               'history'
alias   u               unalias
alias   m               more
alias   mroe            more
alias   l               'ls -F'
alias   c               clear
alias   src             source
```

Example aliases taken from my SUN .cshrc file.

```
alias mjday '/gaia/dunedain/d2/gps/oldbin/mjday'
alias home  "cd ~"
alias x     'chmod +x'
alias dir   'ls -lt | more'
alias hp    "lpr -Php_3890 "
alias tek   "lpr -P3904_tek"
alias nb    "lpr -P3892_grad "
alias nbcolor   "lpr -P3892_hpcolor "
alias DEM    "cd $home/dem"
alias ssh_yang 'ssh -l gps yang.soest.hawaii.edu'
alias ftp_jpl 'ftp bodhi.jpl.nasa.gov'
alias matlab_term 'matlab -nodesktop –nosplash'
```

You can find all the aliases that are defined by using the command alias without any arguments.

Dealing with file names with special characters

Basics of the UNIX/Linux Environment

Say I have a file named "!". (this is probably because I used >! at some time while in bash, but this syntax is for tcsh not bash, so I redirected my output to a file called !)

```
$ rm !
remove !? Y
```

That was easy.

What about a file named "~"

Make a file named "-" with touch command

(use man to see what the touch command does)

```
> touch -
> ls
-               f2.dat          HW              hw1a.txt        SCRIPTS
f1.dat          f_1_2_3.dat     hw1.txt         NOTES           SRC
```

Try to remove it.

```
> rm -
usage: rm [-fiRr] file ...
```

What is the problem? (you tell me.)

We have to let the shell know that the "-" is NOT a switch.

Use the "-" switch all by itself.

```
> rm - -
rm: remove - (yes/no)? Y
>
```

Remember that filenames can have any character but the "/" (used to define the path), so sooner or later you are going to get a file name that will be hard or dangerous to reference.

You will have to be especially careful/creative if you get a file named "*" as

```
%rm *
```

can be disastrous

(and the more privileges you have and the higher up you are in the directory structure, the more disastrous it is.)

File Permissions

Basics of the UNIX/Linux Environment

Every user on a UNIX system has a unique username, and is a member of at least one group (the primary group for that user).

A user can also be a member of one or more other groups.

Only the administrator can create new groups or add/delete group members (one of the shortcomings of the system).

Every file (directories are files) on the system has an owner, and also an associated group.

Every file also has a set of permission flags which specify separate read, write and execute permissions for the

'user' (owner),
'group',
and 'other'
(everyone else with an account on the computer)

Permissions

Read
ability to read the file (r).

Write
ability to write or overwrite the file (w).

Execute
ability to execute or run the file and allow others to view directories (x).
(if a directory is not executable, non-owner's cannot cd into it or see what is in it at all.)

How to view the ownership & permissions of files/direcories (review)

ls -l: lists long format

```
> ls -l
total 2201712
-rw-rw-rw-  1 rsmalley user 54847 Mar  7 2009 *CHARGE-2002-107*
-rw-rw-rw-  1 rsmalley user   413 Oct 30 2006 022285A.cmt
-rwxrwxrwx  1 rsmalley user 13092 Aug 13 2007 a.out
Drwxrwxrwx  3 rsmalley user   512 Oct 10 2008 adelitst
Drwxrwxrwx  5 rsmalley user   512 Aug 29 2007 ANT_GMT
```

Permissions

How to view the ownership & permissions of files/direcories (review)

ls -l: lists long format

```
> ls -l
total 2201712
-rw-rw-rw- 1 rsmalley user 54847 Mar  7 2009 *CHARGE-2002-107*
-rw-rw-rw- 1 rsmalley user   413 Oct 30 2006 022285A.cmt
-rwxrwxrwx 1 rsmalley user 13092 Aug 13 2007 a.out
Drwxrwxrwx 3 rsmalley user   512 Oct 10 2008 adelitst
Drwxrwxrwx 5 rsmalley user   512 Aug 29 2007 ANT_GMT
```

Owner

How to view the ownership & permissions of files/direcories (review)

ls -l: lists long format

```
> ls -l
total 2201712
-rw-rw-rw- 1 rsmalley user 54847 Mar  7 2009 *CHARGE-2002-107*
-rw-rw-rw- 1 rsmalley user   413 Oct 30 2006 022285A.cmt
-rwxrwxrwx 1 rsmalley user 13092 Aug 13 2007 a.out
Drwxrwxrwx 3 rsmalley user   512 Oct 10 2008 adelitst
Drwxrwxrwx 5 rsmalley user   512 Aug 29 2007 ANT_GMT
```

Group

Changing owners and groups.

If you create a file, you are the owner/user.

Mitch and Bob have the SUN and Mac systems set up to automatically set the group to 'user', or all users of the CERI UNIX system.

Default permissions on the SUN are

rw-r--r--

(numerically 644)

And on the Mac are (seem to be)

rwx------

(numerically 700)

chmod

Command to change file or directory permissions (change mode in the normal UNIX philosophy of naming commands).

```
%chmod ugo+x hello.sh
%ls -lF hello.sh
-rwxr-xr-x   1 rsmalley user     21 Sep 16 08:36 hello.sh*

go-x Removes execute privilages from group and other
o+r Adds read privilages to other
```

Flags (or numerical values) allows you to set the permissions. Using wildcards you can set permissions globally within a directory, and with the –r flag all subdirectories.

Changing Permissions

you can also use octal values (numbers) to change ownership

644 represents u=rw; go=r
755 represents u=rwx; go=rx

(using this puts you is a special eunuch class)

Manipulating & Printing Files

Basics of the UNIX/Linux Environment

Printing Commands

`lpr`: submit files for printing

```
% lpr -P3892_grad file.txt
```

Printing Commands

lpq: show printer queue status useful to find out if other jobs are before yours.

```
%lpq -P3892_grad
3892_grad is ready and printing
Rank    Owner   Job     File(s)         Total Size
active  hdeshon 146     junk.pdf    108544 bytes
```

Identifies the job.

lprm: cancel print job (by number)

```
%lprm -P3892_grad 146
```

lpstat: printer status information
useful for finding out printer names on Macs, which are not necessarily the same as on the SUN system

```
%lpstat –a
_3876langston accepting requests since Wed Aug 27 13:11:36 2008
hp_color_LaserJet_4600 accepting requests since Mon Aug 4
                                                11:50:47 2008
```

CERI Printers

Long Building (3892 Central)

`3892_grad` -- B & W printer in Mac Lab
`3892_hpcolor` -- Color printer in Mac Lab
`3892_hpxlfp` -- Poster printer in Mac Lab
`3892_Mitch` -- B & W printer in Mitch's office
`3892_colorps` -- B & W printer in

CERI Printers

House 3 (3876 Central)

3876_langston --
3876_hpcolor -- Color printer
3876_grad -
3876_bodin -
3876_powell -

CERI Printers (Continued)

House 2 (3890 Central)

`3890_hpcolor` – Color printer in copier room

`3890_copy` – B & W printer in copier room

`3890_sheila` – B & W printer in Michelle's office

House 0 (3918 Central)

`3918_usgs` –

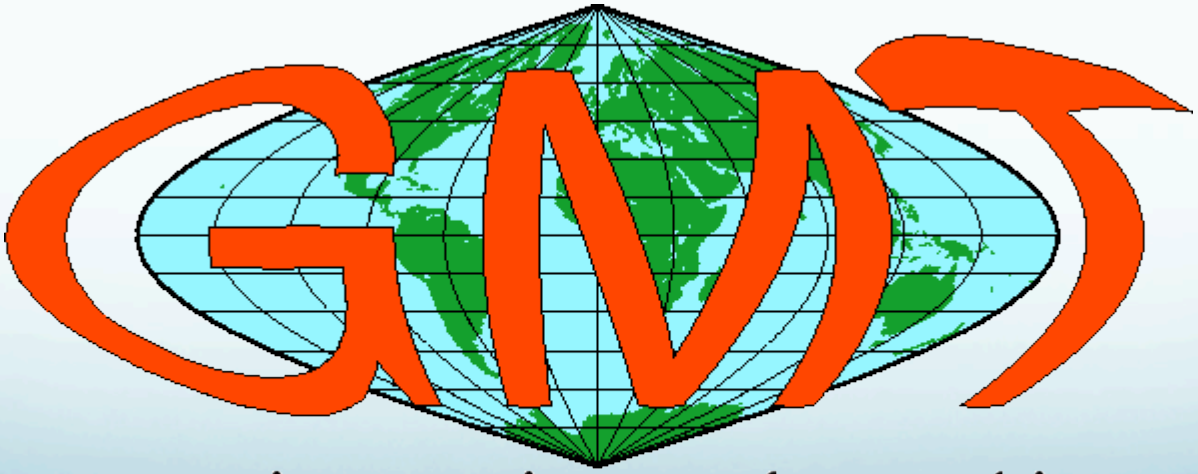
CERI Printers (Continued)

House 1 (3904 Central)

`3904_tek` -- Color printer
`3904_tekdup` -- Color printer
`3904_hallway` -- B & W printer
`3904_brother` --

Data Analysis in Geophysics
ESCI 7205

Bob Smalley



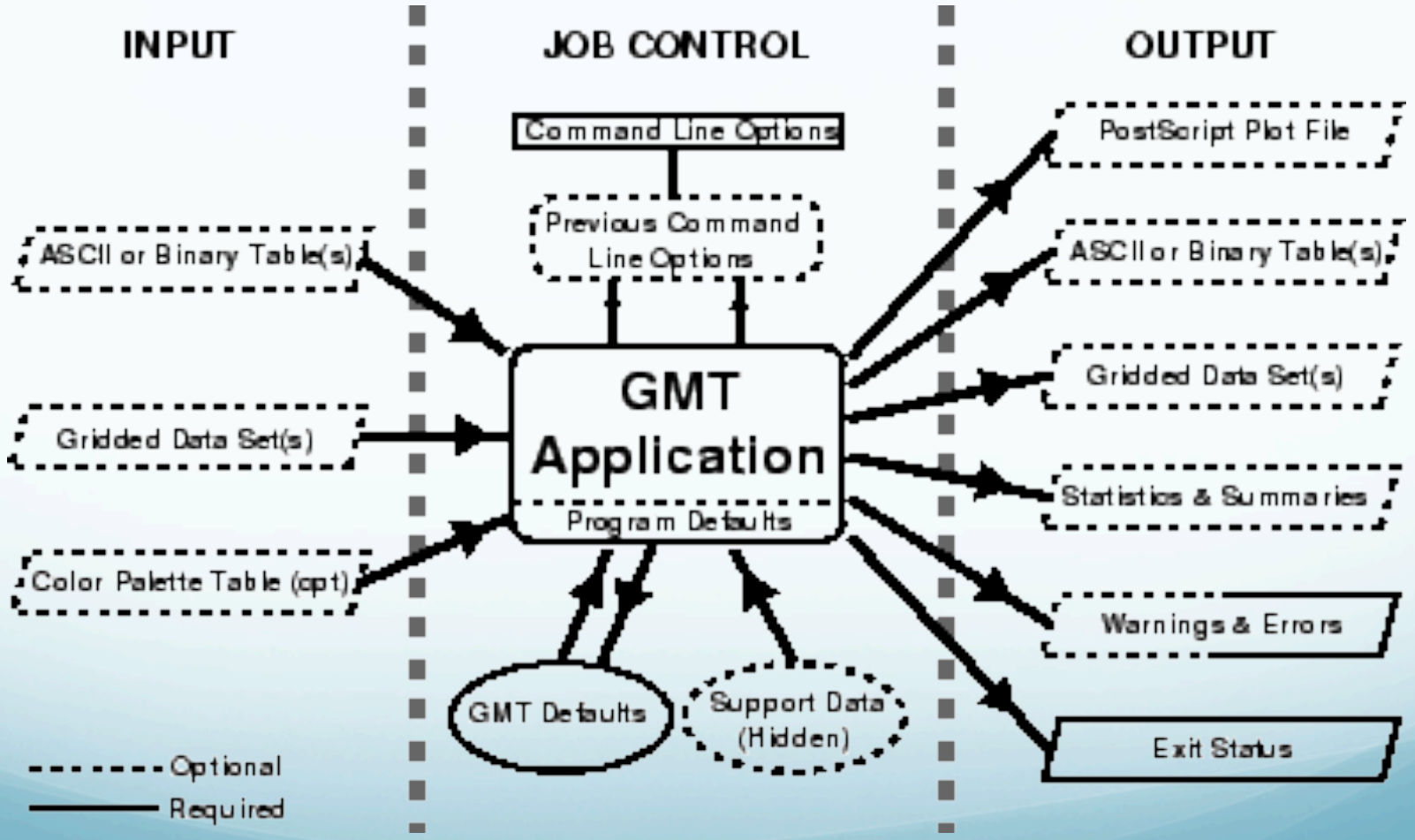
Easiest way to get started

1) Find system with GMT already set up

2) Get working program (shell script) from someone else and modify (hack) it.

Lots examples in
- Tutorial
- available on www
- available from your "friends"

What goes on in GMT



Sources of operational parameters/job control

i) command line options/switches or program defaults

ii) carried over from execution of previous commands

iii) from your .gmtdefaults file

(looks first in working directory, then in your home directory, finally the system, program defaults)

Sources of operational parameters/job control

Why a defaults file?

- too many parameters to require setting all explicitly (powerful)

- customize – can have different defaults in different directories

Basic GMT use

Most GMT programs

read input from terminal (*stdin*) or files, and

write output to terminal (*stdout*) (a few write to files)

– follow UNIX philosophy.

To write output to files one can use *UNIX* redirection (else goes to screen - uselessly):

```
GMTprogram switches >> Outputfile
```

Most GMT programs will accept input-file names and pipes in lieu of stdin

```
GMTprogram input-file switches > outputfile

GMTprogram switches < input-file > outputfile

Someprogram | GMTprogram1 | GMTprogram2 > outputfile
```

Many GMT programs will also accept input redirection (in-line input) – reads whatever follows -- up to character string XXX -- as input.

```
GMTprogram switches << END > output-file
.1 .1
.2 .2
END
```

Can also do with "command substitution":

```
GMTprogram switches << FIN > output-file
`someprogram swithches < input-file…`
FIN

echo `someprogram swithches < input-file…` | GMTprogram switches
> output-file
```

Some GMT programs require input-file names (usually when need more than one input file, or input usually so big that one would be forced to pipe or redirect input all the time, or binary file, etc.)

GMT and scripts

GMT commands act much like regular UNIX commands.

Generally, commands are enacted within a shell script so that they may be combined with other UNIX commands such as `awk` (`nawk`, `gawk`).

`bash` and `csh` are the most commonly encountered shells in academia and passing down GMT scripts is how much of seismology gets illustrated

Use Comments!

Comments are very popular to forget but if you don't comment your script, 2 years later you may not remember what you were doing (especially if you write tight UNIX code [as compact as possible] that took 20 iterations to get "correct").

Spaces and blank lines make your script readable. While it may take more paper if you print it, it only takes two bytes to make new line or a space.

Keeping track of your scripts

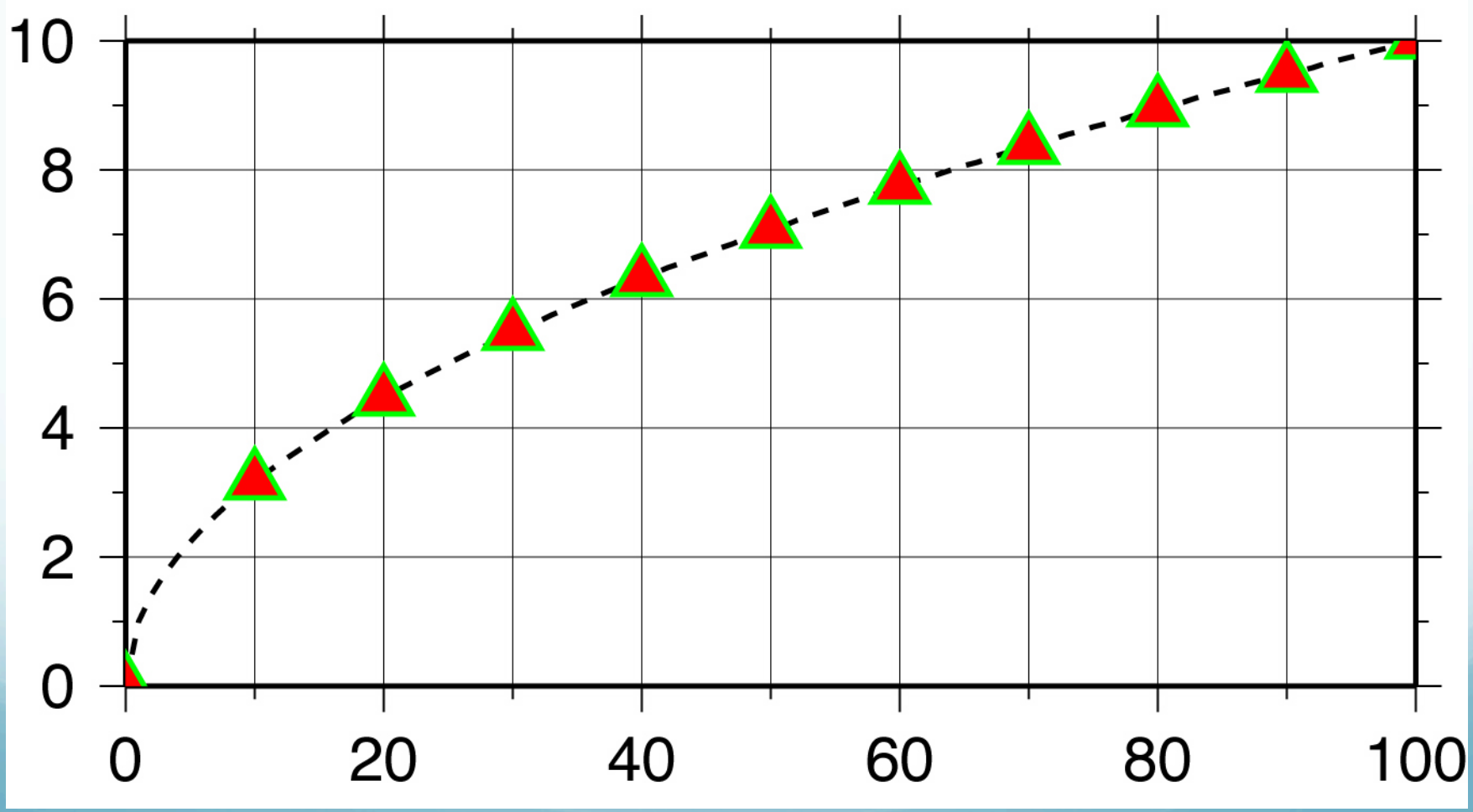
You will be glad (someday) if you set up directories and subdirectories to keep your maps, data, and scripts organized.

Mitch suggests have something like this in `~/GMT`

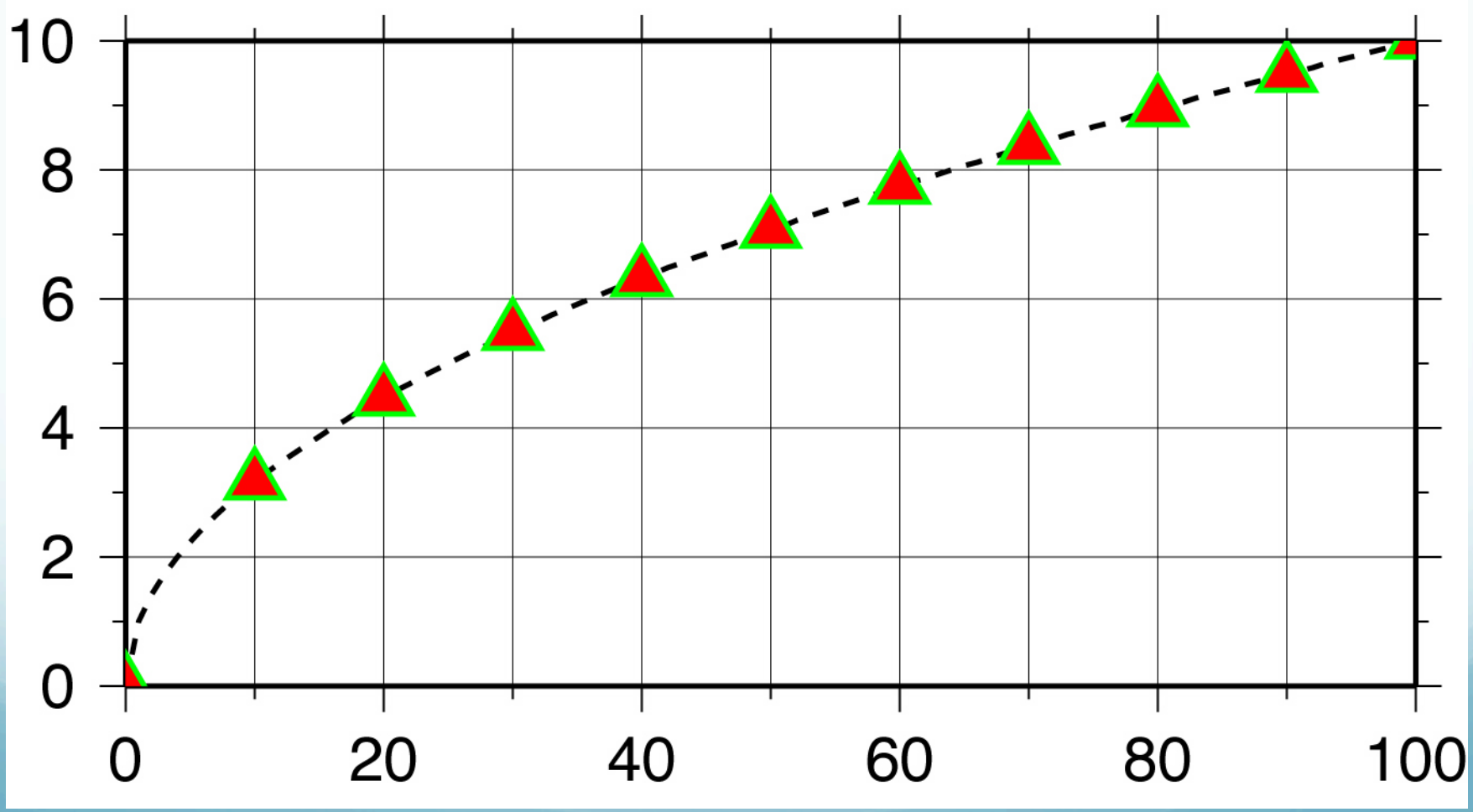
```
csh/      data/     ps/       scratch/  sh/
```

There is a directory for `csh` scripts, for `sh` scripts, for data files, for postscript files, and for scratch files.

OK lets look at some "simple" examples:



Plot $x^{1/2}$ from 0 to 100 as a dashed line, using red triangles with green borders at x=n*10.



1) We start by making the `basemap` frame for a linear *x*-*y* plot.

2) We want it to go from `0` to `100` in `x`, with ticks, grid and annotation every `10`, and from `0` to `10` in `y`, with ticks, grid and annotation every `2`.

3) The final plot should be 4 by 3 inches in size.

Note GMT does not make any helpful assumptions such as

a) You want to plot the whole x and y range of the data and

b) You want it to fit nicely on the page.

You have to specify EVERYTHNG (comes under the excuse of being "powerful")

Here's how we do it:

```
psbasemap -R0/100/0/10 -JX4i/3i -B10/1:."My first plot": -P \
> plot.ps
```

We will first look at how we specify to GMT how to make the map/figure.

This is done using the command line options/ switches.

psbasemap

draws a map frame and sets up the map parameters

(so they don't have to be re-specified in later GMT program calls, although it is a good idea – variables make it easy).

```
psbasemap -R0/100/0/10 -JX4i/3i -B10/1:."My first plot": -P \
> plot.ps
```

Requirements 1 (projection) and 3 (axis sizes) are specified to GMT together

1) We start by making the `basemap` frame for a linear (the projection, or lack of one) `x-y` plot.

3) The final plot should be 4 by 3 inches in size.

psbasemap

The –J option selects the type of projection and the scale.

In this case we want a linear x-y plot, or no projection, which is specified by

x or X.

```
psbasemap -R0/100/0/10 -JX4i/3i -B10/1:."My first plot": -P \
> plot.ps
```

There are 25 projections available in GMT, each specified by one letter (case sensitive to set options).

There are no provisions for providing your own projection.

(short of using the open source to roll your own.)

Requirements 1 and 3 are specified to GMT together

The –J option also sets the axis scales (distance per unit, x) or (axis length, x)

Where the "unit" is specified in `.gmtdefaults` or explicitly – inches, `i`, or cm, `c`.

```
psbasemap -R0/100/0/10 -JX4i/3i -B10/1:."My first plot": -P \
>! plot.ps
```

2) We want it to go from 0 to 100 in x, with grid and annotation every 10, and from 0 to 10 in y, annotating every 1.

This is really two conditions

i) We want it to go from 0 to 100 in x, and from 0 to 10 in y.

Specified by the REGION (-R) option, which (in the usual form) is

```
-Rxmin/xmax/ymin/ymax
```

2) We want it to go from 0 to 100

`-Rxmin/xmax/ymin/ymax`

Notice that unlike MATLAB, GMT does not make any assumptions about what you want

(such as the reasonable one that you just might want the region to show all the input data).

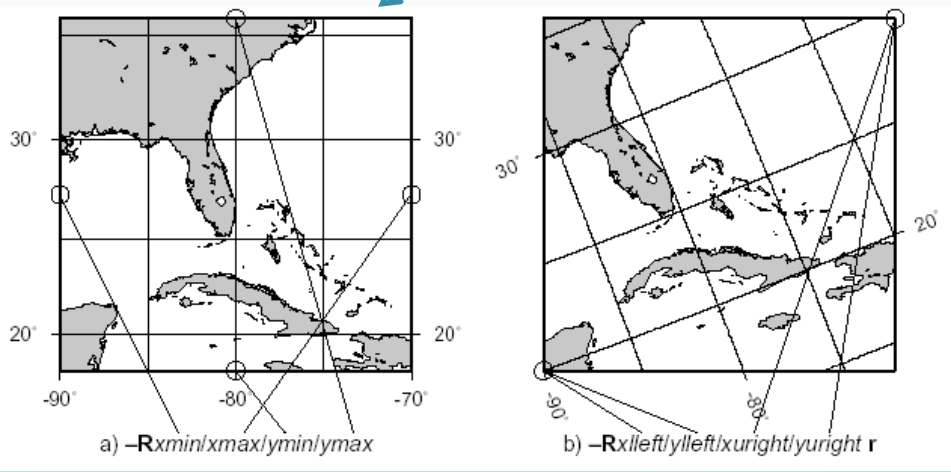
You have to specify every detail. (i.e. powerful)

(why should the writers of gmt work hard when they can convince the user that it is "better" if the users do!)

```
psbasemap -R10/100/10 -JX4i/3i -B10/1:."My first plot": -P \
>! plot.ps
```

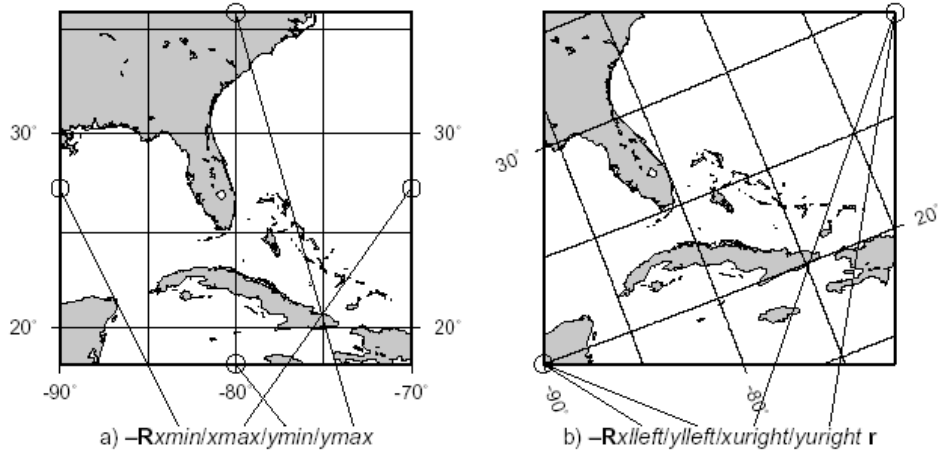
There are two forms for the –R option

1) For projections where the boundaries follow lines of latitude and longitude ("rectangle" on sphere) – specify sides.



There are two forms for the –R option

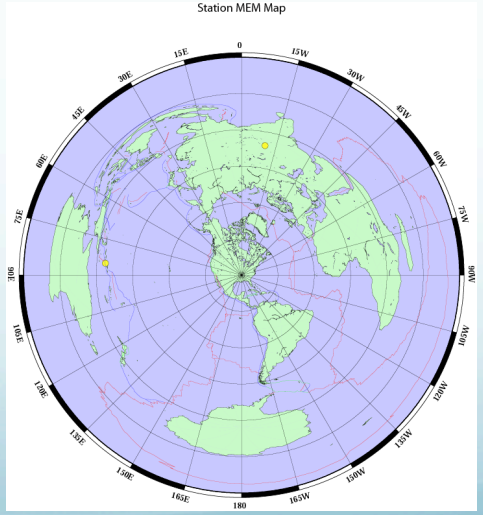
2) For regions where the sides do not follow lines of latitude and longitude (will make more sense when we do map projections) - specify corners by appending an "r" to end



The idea of a "region" to plot specified this way breaks down for azimuthal projections

(outside border of plot is a circle, you really want to specify center and radius)

will see how to do this later.



2) We want ticks, grid and annotation in x every 10, and in y every 1.

This is specified by the –B option (Border?).

```
psbasemap -R0/100/0/10 -JX4i/3i -B10/1:."My first plot": -P \
>! plot.ps
```

This is the most complicated GMT option.

Ticks and annotation – every 10 for x (first one) and every 1 for y (second one).

If you wanted the same ticks and annotation for x and y you would only have to specify it once.

```
psbasemap -R0/100/0/10 -JX4i/3i -B10/1:."My first plot": -P \
>! plot.ps
```

Not given in our specifications for the plot, but controlled by the –B option, the plot title.

This is a little more complicated.

Labels are between colons, with

"." for plot title,
nothing for x axis label,
"," for y axis label.

If label/title is more than one word, has to be in double quotes.

```
psbasemap -R0/100/0/10 -JX4i/3i -B10/1:."My first plot": -P \
>! plot.ps
```

If this sounds confusing you can look at the man page for `psbasemap` for the full explanation and more examples.

The man page for the `-B` option, however, is practically incomprehensible.

The `BUGS` section of the man page states

"The `-B` option is somewhat complicated to explain and comprehend. However, it is fairly simple for most applications (see examples)."

Remaining options/switches

`-P`

Sets the output to Portrait (long side vertical) mode.

"Default" is Landscape (long side horizontal) mode.

```
psbasemap -R10/70/-3/8 -JX4i/3i -B10/1:."My first plot": -P \
>! plot.ps
```

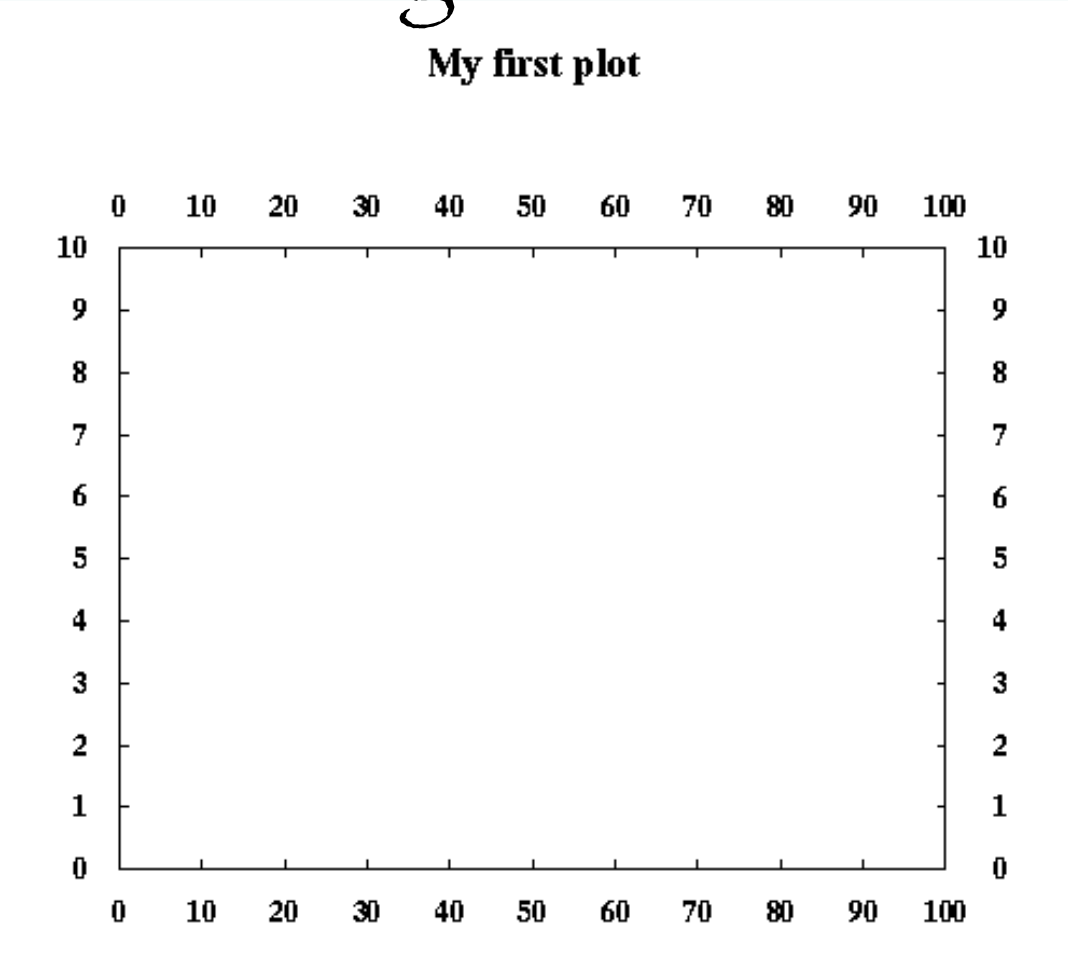
This option actually switches "states".

Remaining options/switches

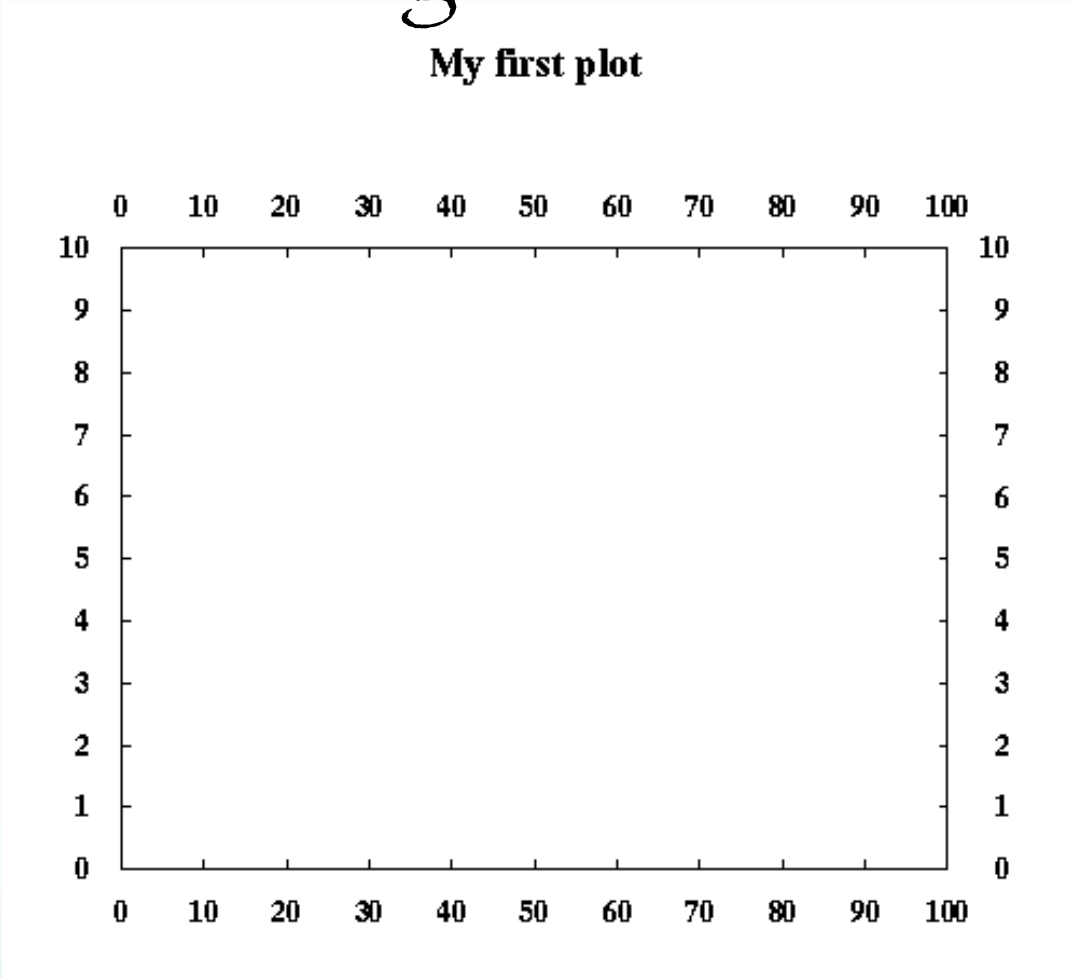
If `.gmtdefaults` defines portrait mode as the default, then `-P` will send it to landscape.

(make a figure and see how it comes out, if you don't like the orientation stick in a `-P`).

So, what did we get for all our effort?



So, what did we get for all our effort?



Good start – but usually we make plots to show some sort of data

– so how do we do that?

Now let's look at a little more complicated example:
Lets call it "`full_court_press.sh`"

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

This is more than "a little more" complicated.

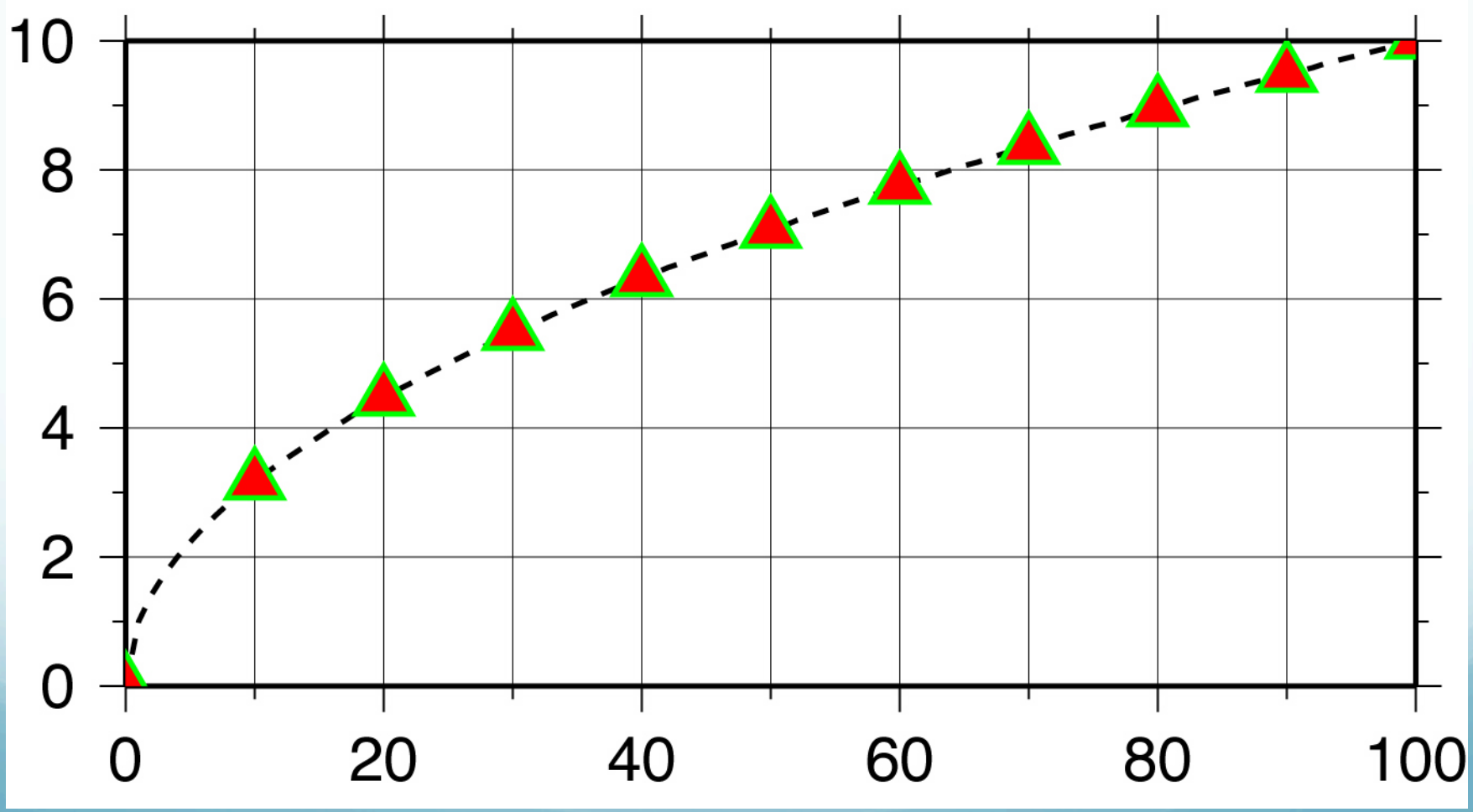
```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

But it follows the UNIX philosophy – a bunch of simple things stuck together to do something more complex.

Gives you the idea that most useful GMT produced figures are going to be a LOT of GMT calls

Here's what the output looks like

(actually the output is a ascii file containing a PostScript program, this is what it looks like after displaying with GhostScript or GhostView to the screen or printing to a PostScript printer).



Let's look at it piece, by simple piece.

Set shell

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1,
...
```

Set shell to Bourne Shell.

Could also have set it to bash or csh

(change first line to #!/usr/bin/csh -f, this works because this script does not contain anything that is specific to one shell script – such as variable name definition. Use -f, fast, option which stops it from running your .cshrc).

Next piece.

Get (actually make) input data – part 1

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

`sample1d`, resamples (here interpolates) the input, which in this case is redirected (<<) to being in-line from the shell script (from the end of this command line, which is somewhat far away, to `END`) and pipes it to the next guy.

```
#!/bin/sh
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

We have to specify the resampling step (-I1, which is steps of 1).

We will leave everything else at the default values.

(see man page if you want more info)

sample1d provides a list of numbers from 0 to 100 in steps of 1 to stnd out.

Next piece. Get (make) input data – part 2
We want `x` and `sqrt(x)`

```
#!/bin/sh
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

Pipe the re-sampled data into the program `nawk`.

`nawk` is a great tool for preprocessing data for GMT.

Next piece.
Generate input data – part 2

```
#!/bin/sh
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

The `nawk` command says to print the first column (`$1`) and the square root of the first column (`sqrt($1)`) of every line.

We will (break the UNIX philosophy and) make an intermediate file as we will need the data more than once.

Next piece.

Plot it

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

The graphics part of GMT

psxy is the GMT program that plots points and lines.

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

psxy accepts the "standard"/"global" options of the GMT filters that produce PostScript output.

(one can put all the plot/map specifications into the first GMT call rather than use **psbasemap**, everything we said before holds, so we don't have to re-say it.)

We already know what **–R**, **-J**, **–B** and **–P** do, although the **–B** option here is a bit more complicated looking.

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

Output file `$0.ps` (is first instance so use `>`, append in second instance so use `>>` – this takes care of UNIX part)

Use `\` to continue command on next line

Next piece.

```
...
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
...
```

So, what's all that extra stuff on the –B?

Each of the letters controls a different feature/ aspect of the plotting of the axis

```
...
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
...
```

a is for annotation spacing

g is for grid spacing

WSne says to plot the annotation and ticks on the West and South sides and ticks only on the north and east sides.

(how would you put annotation without ticks?)

Next piece

Name the output file.

```
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

Being lazy and disorganized - I don't want to have to type the output file name in lots of times nor keep track of which shell script made which postscript file in my directory.

Next piece

Name the output file.

```
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

So I want to find a short and easy way to name the file, and I might want to associate the output file name with the name of the shell script that made it.

Enter UNIX argument passing to the rescue.

When you call a shell script, the system passes predefined, pre-named "arguments" to the shell script from the command line.

So if I enter

```
"myscript arg1 arg2"
```

UNIX automatically gives me (in this case 3 arguments)

```
$0      the name of the shell script
$1      the value of arg1 (character string)
$2      the value of arg2
```

All I have to do to use these arguments in my shell script (within some constraints) is stick them in.

The Shell will expand them to their proper values.

So my output file will be named

"`full_court_press.sh.ps`",

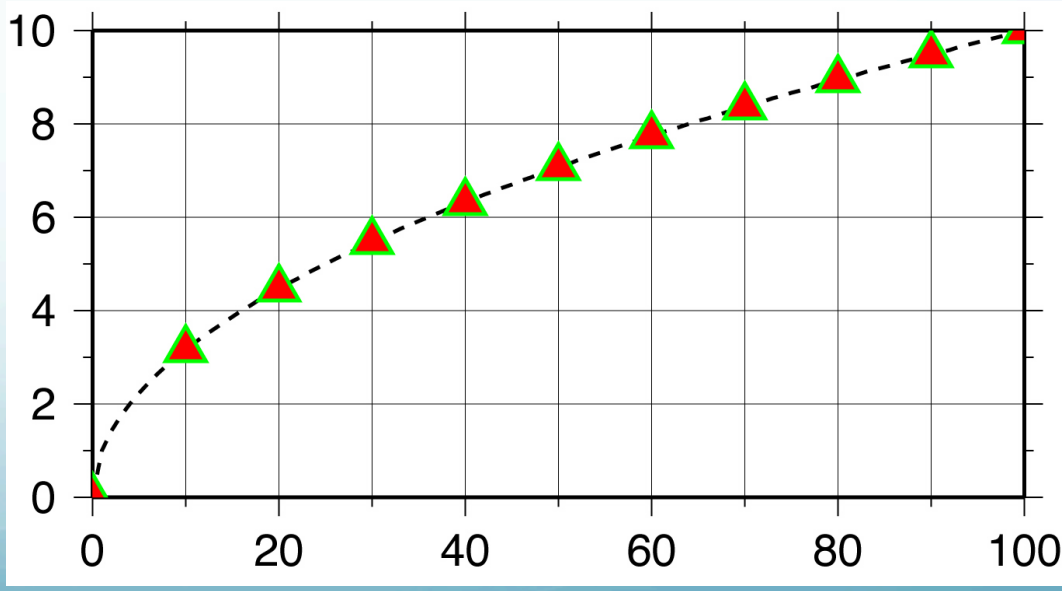
since `$0` will get expanded to
"`full_court_press.sh`"

(the name of the shell script)

Next piece.

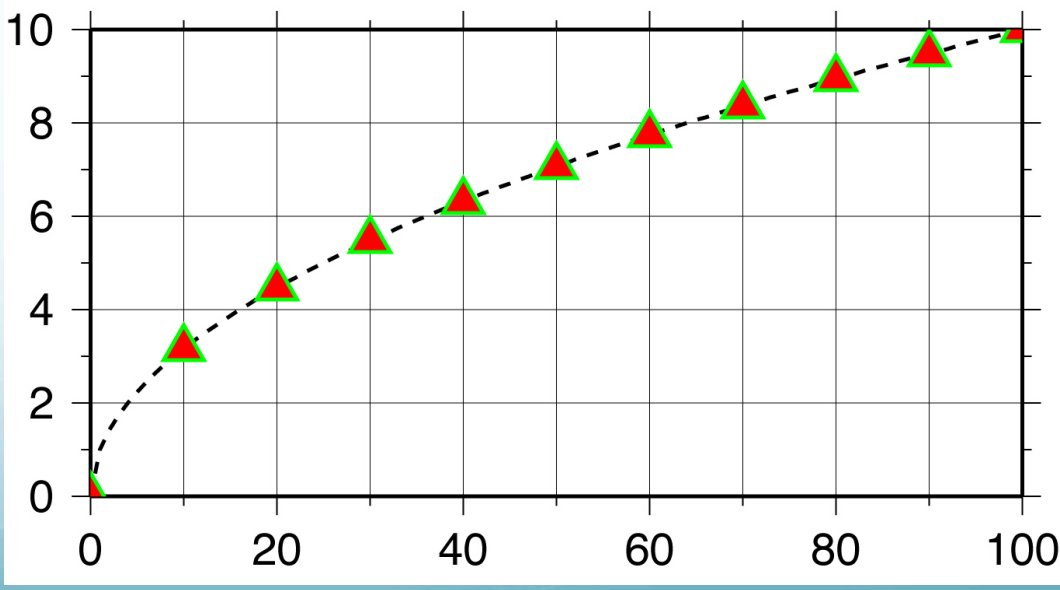
Draw a line -W5t15_15:0

```
...
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
. . .
```



Make line 5 units thick (where units depends on the device and default settings)

```
-W5t15_15:0
```

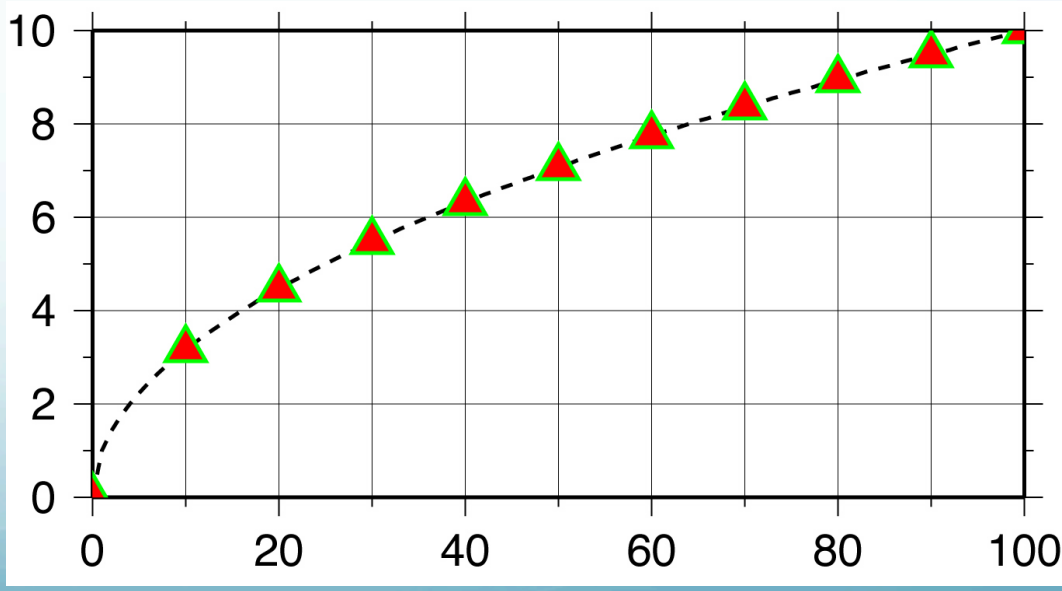


Can also specify color

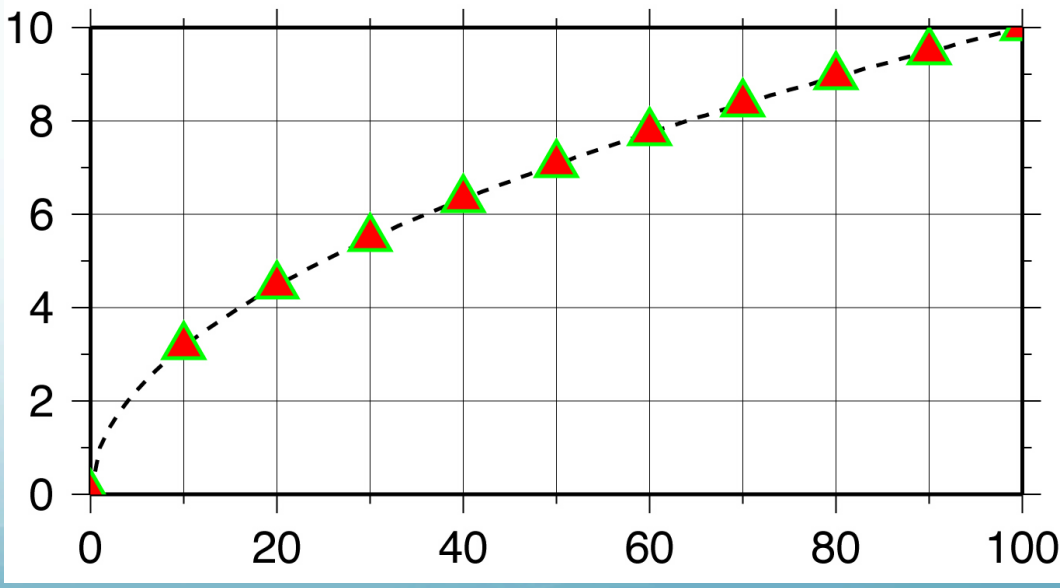
-W5/0t15_15:0 for black line or

-W5/255/0/0t15_15:0 for red line

```
...
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
. . .
```



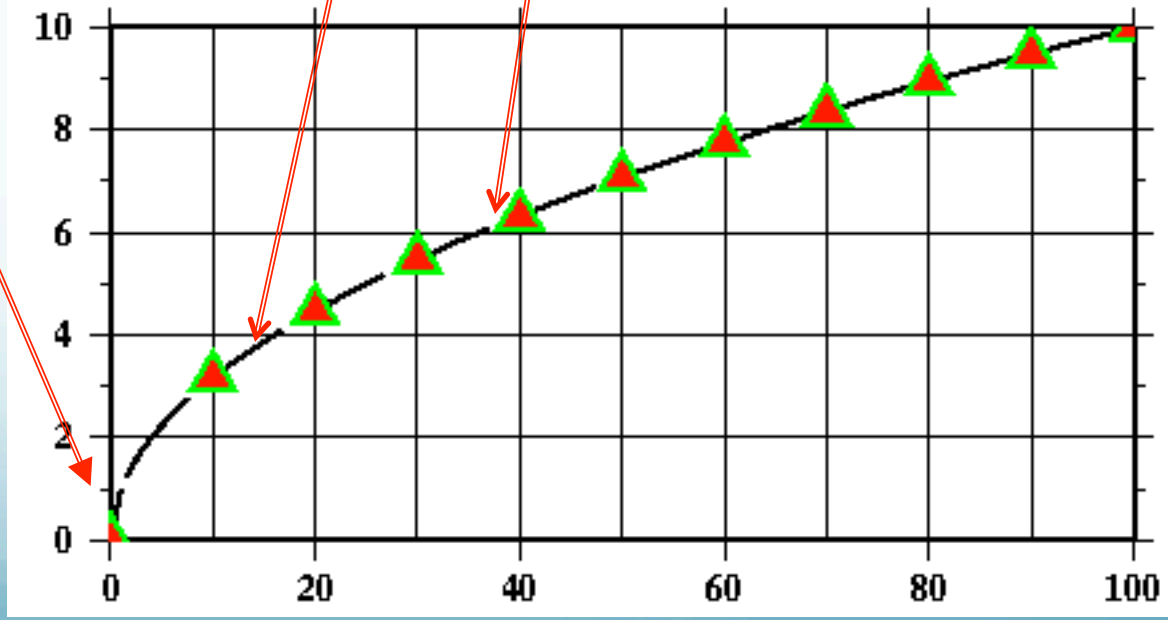
Make it dashed with dashes 15 units long followed by 15 unit long open spaces -W5t15_15:0, and a "phase offset" for the dashes of zero -W5t15_15:0



What is "phase offset"

compare to `-W5t120_15:60`

(line segments specified to be 120 long and then 15 blank, first line segment only 60 long [phase])



This format also works

`W[width],[color],[texture]`

`-W5,0,15_15:-`

`-W5,255/0/0,15_15:-`

`-W5,red,15_15:-`

Gives the same plot.

Generate input data w/out temporary file

Using nawk, one does not have to write programs to make intermediate files in GMT input format,

but can go right to the source data file,

read it,

modify each line into GMT input format

and pipe this directly into the GMT program.

```
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' | \
psxy -R0/100/0/10 -JX4/2 -Ba20f10g10/a2g2WSne -W5t15_15:0 \
-Y2 -P -K > $0.ps
0 0
100 0
END
```

Generate input data

```
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' | psxy \
-R0/100/0/10 -JX4/2 -Ba20f10g10/a2g2WSne -W5t15_15:0 -Y2 -P -K \
> $0.ps
0 0
100 0
END
```

The `nawk` command says to print the first column (`$1`) and the square root of the first column (`sqrt($1)`) of every line.

Then pipe the data straight to psxy for plotting.

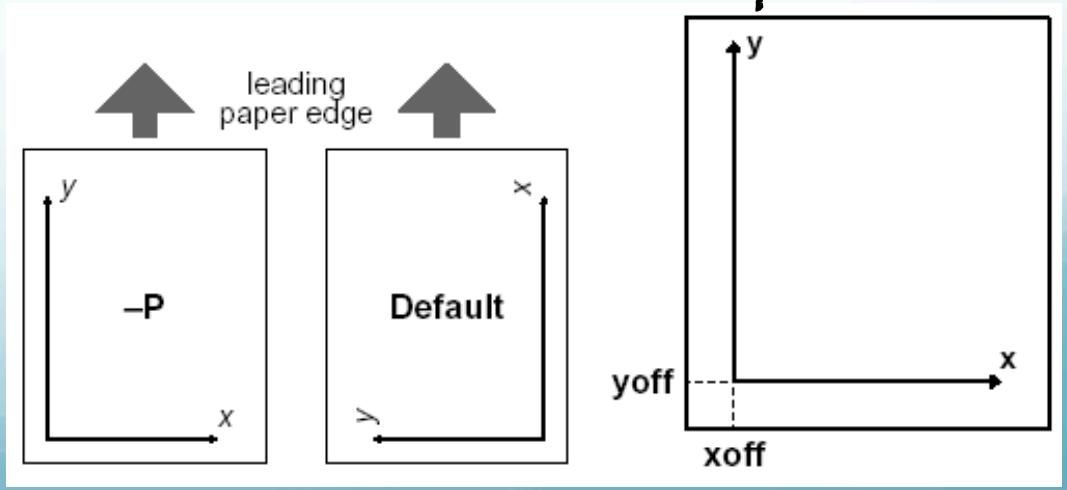
```
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' | psxy \
-R0/100/0/10 -JX4/2 -Ba20f10g10/a2g2WSne -W5t15_15:0 -Y2 -P –K \
> $0.ps
0 0
100 0
END
```

Note: sort of confusing since the stuff in blue is associated with the first part (`sample1d...`).

Next piece. Misc. 1

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

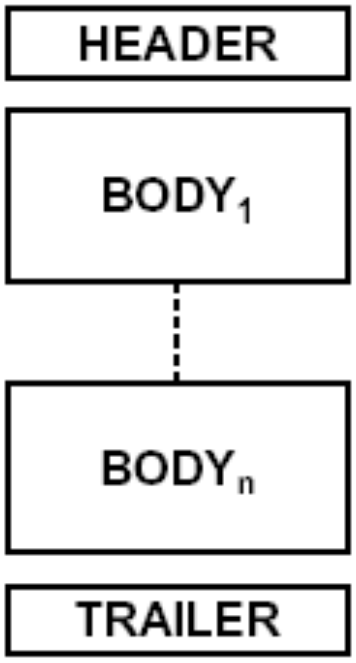
-Y2 offset plot 2 units in the Y direction (else x axis labels get cut off across bottom of plot)



Misc. 2

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

-K do not close PostScript file
(we will be adding to the plot so don't close it.
Unless otherwise specified each gmt call will put out both the header and trailer.)



1st needs -K omits the trailer

2nd through next to last needs both -K and –O to omit both the header and trailer

Last needs -O omits the header

-K do not close PostScript file (don't output "showpage") so more PostScript can be appended to the file

-O do not initialize PostScript (does not output PostScript header stuff) so this can be appended to existing file (that hopefully does not have a showpage at the end).

Misc. 3

Several common "gotchas"

- no `showpage` (can see on screen, but does not print – actually prints a blank page) (error: have a `–K` in last GMT call)

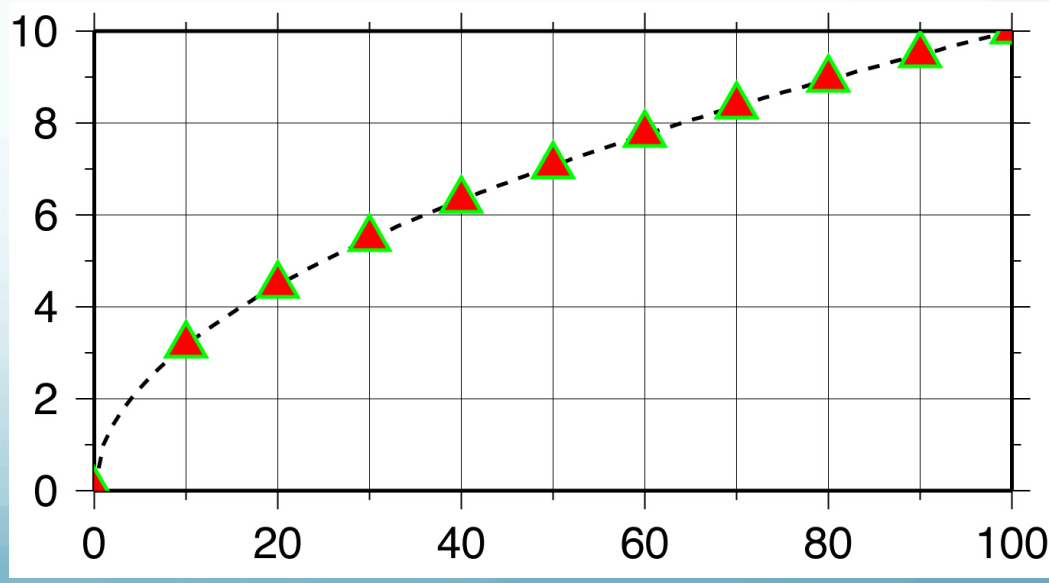
- `showpage` in middle of file (error: forgot the `–K` somewhere) – only get part of file on screen or in final print or get ghostscript error message.

- Have header in middle of file (error: forgot `–O` somewhere), get ghostscript error message.

Next piece.
Draw symbols every 10th point

```
#!/bin/sh
#plot square root x
sample1d -I1 << END | nawk '{print $1, sqrt($1)}' > {$0}_1.dat
0
100
END
psxy -R0/100/0/10 -JX4/2 -Ba20g10/a2g2WSne -W5t15_15:0 \
-Y2 -P {$0}_1.dat -K > $0.ps
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

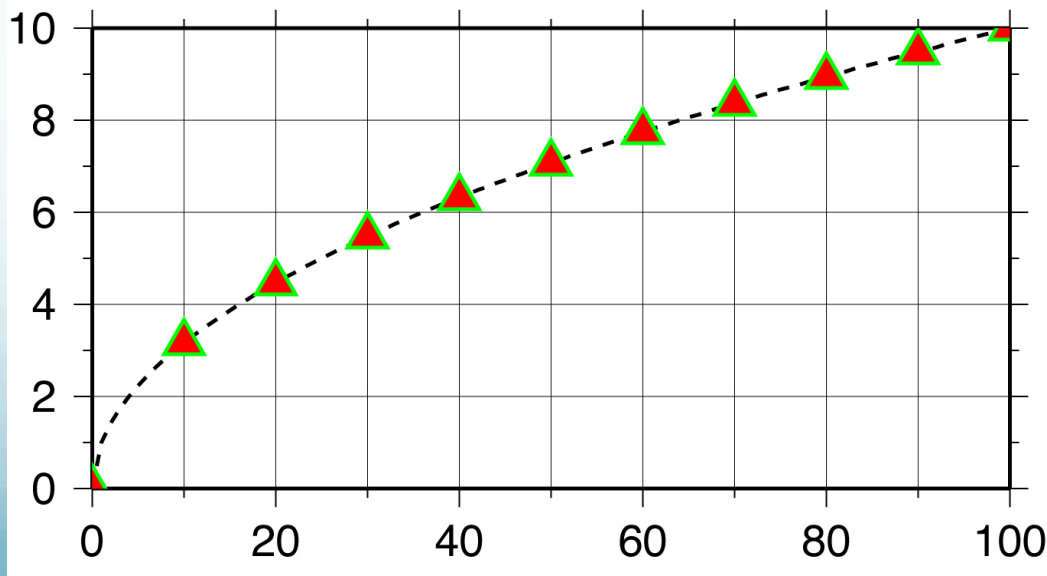
Resample our temporary file – taking every 10th point (-I10). Pipe output to psxy



Next piece.
Draw symbols -St0.2

```
...
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

Draw symbols `-S`, make triangles, `t`, with a size of `0.2` units (see man page for what dimension size relates to), `-St0.2`

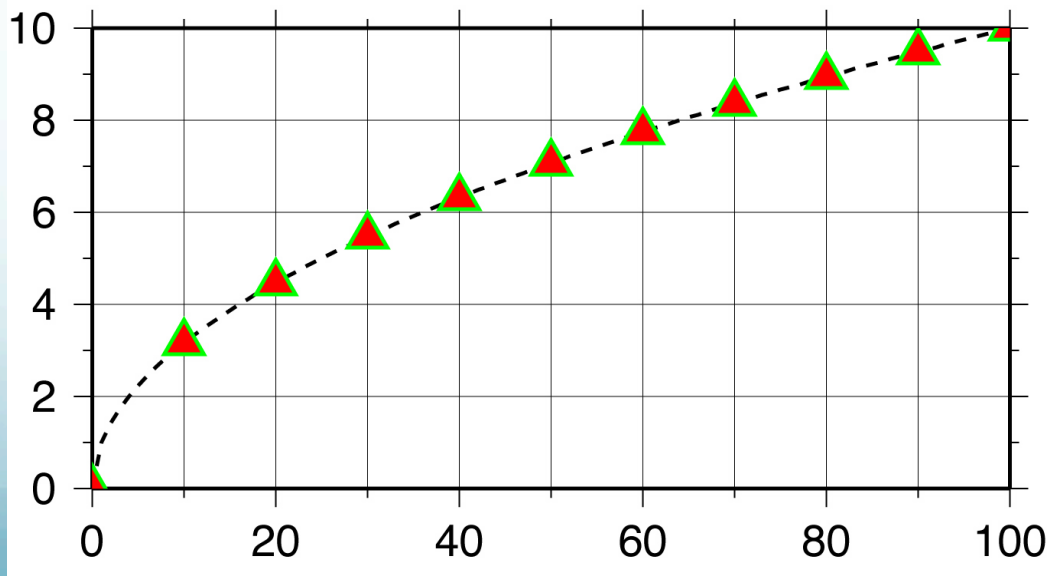


Next piece.
Draw symbols

```
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

Fill the symbol, `-G` flag, with color red, `-G255/0/0`, or `-Gred`

Can also use color names (red, green, blue. There are over 700 X-11 color names).



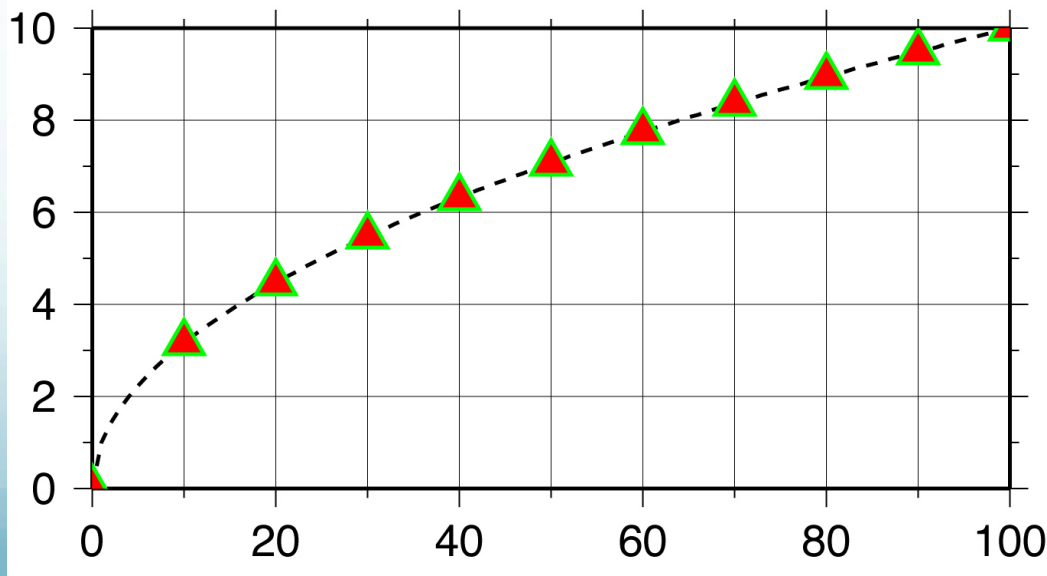
Next piece.
Draw symbols

```
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```

Make the line outlining/drawing the symbols, `-W` flag, that is 5 units wide, and draw outline in green (R/G/B) `-W5/0/255/0` or `W5/green`

Colors specified in R/G/B format

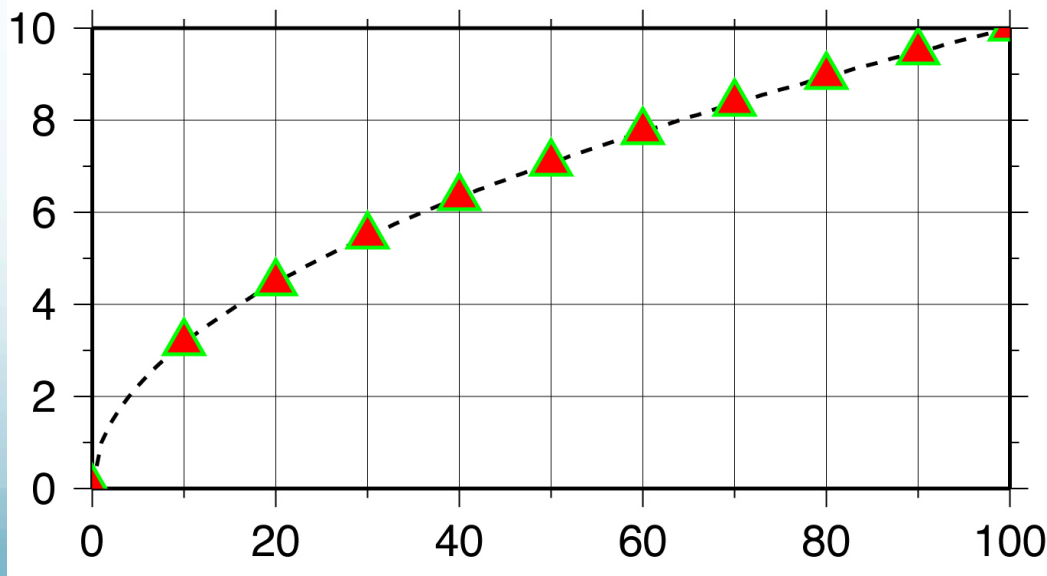
(intensity of Red, Green and Blue color guns – primary colors for additive system).



Next piece.
Will append to previous plot so use –O to prevent initialization of PostScript

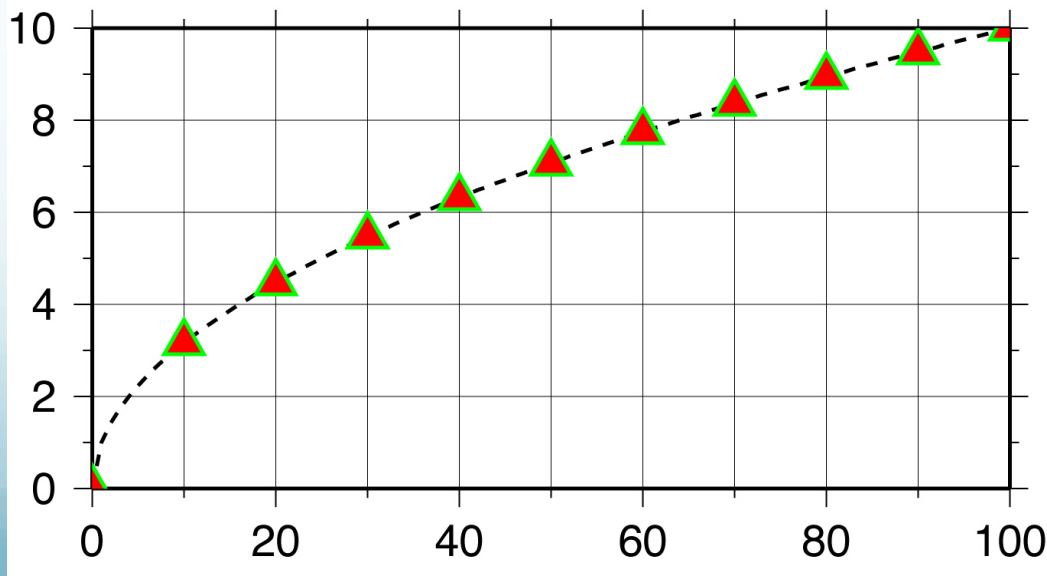
(use when adding, so it does not start new plot)

```
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```



Next piece.
Append to previously started file using >>.

```
sample1d {$0}_1.dat -I10 | psxy -R -JX4/2 -St0.2 \
-G255/0/0 -W5/0/255/0 -O >> $0.ps
```



We're done!

That wasn't so bad now, was it?