Introduction to Unix Shell Scripting

Tuesday, Aug 16, 2011

Short course on USArray data processing for the next generation of seismologists III

Suzan van der Lee

Go to a work directory/folder and "poke around", for example:

```
> pwd
> ls ..
> ps
> ls
> uptime
> whoami
> echo -
> echo `man gmtinfo | grep Software | awk '{print substr($11,1,3)}'``uptime | awk '{print substr($8,1,2), " th" substr($7,3,2) substr($9,7,1)}'`
```

Let's assume you want to issue this set of commands (for no good reason) each time you cd to a different directory, but are too lazy to type them all in. We can group these "unix commands" into a file that we will then execute and call a "shell script". Open an editor and place each of the above commands on a separate line in this new file. Save and close the file, lets call it "script1".

To execute any program, including a shell script, the script needs to have "execute permissions".

```
> ls –l script1
```

produces:

```
-rw-r--r--  1 usarray  staff  229 Aug 14 13:53 script1
```

If the fourth character is an "x" then the file has "execute permissions". Our script does not and thus you'll need to give it that permission:

```
> chmod +x script1
> ls –l script1
```

now produces:

```
-rwxr--r--  1 usarray  staff  229 Aug 14 13:53 script1
```

Your script is ready to run:

```
> script1
```

Great. But there's more to it; shells come in different flavors. Traditionally, lots of scripting was developed using the csh, but on the iMacs our defaults shell is the bash. The computer need to know within which shell to run your script, particularly if different from the login shell (bash for us). Add that information to your script by making the script's first line "#!/bin/csh" for the C shell or "#!/bin/bash" for the Bourne again shell. Normally a "#" as the first chracter of a line in a shell script means that the line is a comment rather than a command; the exception is the very first line of the script, where the "#" is *required* for declaring the shell. Now add a conditional statement to your script:

```
if ($user == usarray) then
```

```
        echo Hooray for USArray
else
        echo Hello there
endif
```

for the csh script and for bash you would need to use the bash syntax:

```
if [ $USER = usarray ]; then
        echo Hooray for USArray
else
        echo Hello there
fi
```

Now you have experimentedwith different shells and their shared and unique syntax. Let's examine some of the shell scripts we used yesterday in the response exercise and that live in the "codes" subdirectory.

Query1

```
#!/bin/csh                                          (use the C shell)
# list the file names of those stations that I have more than 1 BHZ data file for
set sp = "9999"                                     (initialization of station variable)
foreach s ( `ls *BHZ*SAC | awk -F. '{print $8}'` )  (extract station name from file name
  set a = `ls *$s*BHZ*SAC | wc`                     and for each station count
how many Z files there are
  if ( $a[1] !=  1 && $s != $sp ) then              for this station. If one, do nothing.
    ls *$s*BHZ*SAC                                  Otherwise list the offending files)
  endif                                             (end of conditional statement)
  set sp = $s                                       (remember this station when
end                                                 checking the next. End of loop)
```

Query2

```
#!/bin/csh                                          (use the C shell)
# list the data files that are not part of the TA network        (comment)
foreach f ( `ls *BHZ.R.SAC` )                       (loop over each raw file)
set a = `echo $f | awk -F. '{print $7}'`            (extract network code from filename)
if ( $a !=  "TA" ) then                             (if the network code is not TA,
  echo $f                                           echo the file name to screen)
endif
end                                                 (End of loop)
```

Seedresp

Seedresp is a shall script that needs input arguments, in this case a seismogram in counts, a seismogram name for the ground motion output record, and the "RESP" file that rdseed produced, which contains the full instrument response. To run the script:

```
> seedresp I23A.BHZ I23A.dz RESP.TA.I23A..BHZ
```

The script:

```
#!/bin/csh                                          (should look familiar by now)
```

```
if (${#argv} != 3) then                          (if there are not 3 input arguments,
  echo Usage: `basename $0` input_sacfile output_sacfile response_file
  exit                                           then tell the
endif                                            user how to call the script and exit)

../codes/responsee<<!                (call compiled fortran program, which needs its
0                                    own input. 0 = not verbose)
$1                                   (raw, input seismogram in counts)
$2                                   (name of output seismogram (ground motion units))
1                                    (method of supplying response information)
$3                                   (name of response file)
2                                    (deconvolve rather than convolve)
2                                    (ground motions units choice (2=nm))
3
0.0036 0.006 2.0 2.8                 (frequencies of band pass filter)
!                                    (code to signal end of input)
```

So this shell script calls a fortran program and parses its input arguments to become input to the same program ("responsee"). Behind the scenes "responsee" calls the IRIS "evalresp" function (if *method*=1).

Getorder

Here's an example of a shell script that calls SAC for header information and then creates a SAC macro that the user can run in SAC to read the seismograms files in in a particular order.

```
#!/bin/csh                           (C shell again)

set list = ($*)                      (place all input arguments (sac files) in the list)

if (-f /tmp/tlist.$user) then        (remove a previously used temporary, scratch file,
  /bin/rm /tmp/tlist.$user           if it exists)
  echo "removed old /tmp/tlist.$user "
endif

foreach file ($list)                 (loop through all sac files provided as arguments)
set dist = `sac ../codes/getsacdist.m $file | awk '{if ($1=="dist") print $3}'`
echo $file $dist                     (run the sac macro to list the distance in header
echo $file $dist >> /tmp/tlist.$user value DIST and grab just its value, then list it with
end                                  file name in the scratch file, then do next file)
sort -n -k2 /tmp/tlist.$user | awk '{if (NR==1) printf "r %s ", $1; else if (NR%10==1) printf "r more
%s ", $1; else if (NR%10==0) printf "%s \n", $1; else printf "%s ", $1}' > olist.m
                                     (sort the list by distance and change into a simple
echo "created ordered list olist.m"  SAC macro, olist.m, that can read the files in order)
```

Getsacdist.m is a SAC macro that lists the distance between event and station of the given file. Of course you can grab any SAC header value (kztime, azimuth, npts, etc.) instead of distance for use in ordering the sac files.

```
r $1
lh dist                              (or any other header field)
q
```