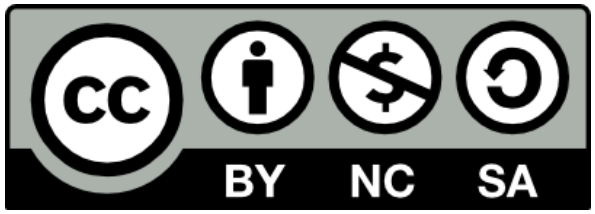
bash is Awesome!

Andy Turner, EPCC
a.turner@epcc.ed.ac.uk

Reusing this material



archer

epcc

THE UNIVERSITY OF EDINBURGH

www.epcc.ed.ac.uk
www.archer.ac.uk

Outline

- Basic Stuff
- Arithmetic
- String search and replace
- Default values for variables
- Flow control: for and if
- Arrays
- printf
- xargs

Why bash scripting?

- Everyone on ARCHER writes shell scripts for job submission
  - Usually bash
- Extremely useful for file manipulation and automation
  - More people having to deal with this as data volumes increase
- Scripting in csh/tcsh is “considered harmful”:
  - http://www.perl.com/doc/FMTEYEWTK/versus/csh.whynot
- Familiarity with bash features opens up possibilities for automation and improving workflows

Basic stuff: variables

- Variable assignment:

```
my_var="Hello world"
my_number=7
```

- Referring to variables

```
echo $my_var                  ➔ Hello World
echo "my_var is $my_var"      ➔ my_var is Hello World
echo 'my_var is $my_var'      ➔ my_var is $my_var
```

- Capturing command output:

```
result=$(echo "my_var is $my_var")
echo $result ➔  my_var is Hello World
```

Basic stuff: redirection and piping

- Redirect stdout to file:

```
echo “Hello World!” > hello.txt
```

- Append stdout to file

```
echo “Goodbye World!” >> hello.txt
```

- *Pipe* results of one command into another:

```
qstat | grep “ H “
```

- View and save stdout to file using *tee:*

```
qstat | tee current_queue.txt
```

Arithmetic

- Integer arithmetic is simple:

```
meaning_of_life=$(( 6 * 7 ))
echo $meaning_of_life
        ➔ 42
```

- Floating point arithmetic requires an external program:

```
big=189.0
small=4.5
echo “$big $small” | awk ‘{print $1/$2}’   ➔ 42
echo “print $big/$small” | python          ➔ 42.0
```

String search and replace

- Search and replace within string (first match):

```
infile="myjob.in"
outfile=${infile/.in/.out}
echo $outfile ➔ myjob.out
```

- All matches:

```
${string//substring/replacement}
```

- Match at start:

```
${string/#substring/replacement}
```

- Match at end:

```
${string/%substring/replacement}
```

Default value for variable

- Set variable value and provide default if referenced variable is not set:

```
initialfile=“”
infile=“${initialfile:-job1}.in”
echo $infile
        ➔ job1.in

initialfile=“bigjob1”
infile=“${initialfile:-job1}.in”
echo $infile
        ➔ bigjob1.in
```

Flow control: for

- for loop, basic form:

```
list=$(ls)
for item in $list; do
   echo $item
done
```

- for loop, C syntax:

```
for ((i=0; i<10; i++)); do
   echo $i
done
```

Example: run benchmarking

```
…PBS options…


module load vasp5/5.4.1
size_list=“24 48 96 192 384 768”
resfile=“runtimes.dat”
rootdir=$PBS_O_WORKDIR
for size in $size_list; do
        rm WAVECAR
        aprun –n $size vasp_gam > $size.stdout
        runtime=$(grep Elapsed OUTCAR)
        echo $size $runtime >> $resfile
        mv OUTCAR OUTCAR.$size
done
```

Flow control: if, string comparisons

```
if [ $var == “One” ]; then
   echo “The answer is one”
elif [ $var == “Two” ]; then
   echo “The answer is two”
else
   echo “I do not know the answer”
fi
```

Flow control: if, arithmetic comparison

- Note “((“ instead of “[“:

```
if (( $var == 1 )); then
   echo “The answer is one”
elif (( $var > 2 )); then
   echo “The answer is greater than one”
else
   echo “I do not know the answer”
fi
```

Flow control: if, other tests

- File tests, e.g.:

```
if [ -e file.dat ]      Test that file exists
if [ ! -d test ]        File is not directory
```

- String tests, e.g.:

```
if [ -n “$var” ]        Variable has a value
if [ -z “$var” ]        String has zero length
```

Arrays

- Basic array usage:

```
array=(red green blue yellow orange)
echo ${#array[@]} ➔ 5
echo ${array[2]}  ➔ blue
array[5]=pink
echo ${#array[@]} ➔ 6
```

- Looping over arrays:

```
len=${#array[@]}
for ((i=0; i<$len; i++)); do
   echo ${array[$i]}
done
```

Generating arrays

- From lines in a file:

```
IFS=$'\n' lines_array=($(<data.txt))
```

- From a string with elements separated by spaces:

```
line="4 3 5 10 6 12"
read -ra my_array <<< "$line"
```

- From a string with elements separated by commas:

```
line="4,3,5,10,6,12"
IFS=',' read -ra my_array <<< "$line"
```

printf

- Formatted printing in the style of C:

```
pi=3.14159265359
printf “pi is %.2f\n” $pi
   ➔ pi is 3.14
```

xargs

- Allows you to run commands on multiple results from another command
  - For example, identify files with a particular name and move then to specific directory:

```
find . –name “*.res” –type f –print0 \
    | xargs -0 –I {} mv {} $RDF/result_files/
```

  - `-print0` print file name followed by ASCII NULL
  - `-0` deal correctly with spaced in file names
  - `-I {}` argument indicator
- Really useful for file manipulation and data management

Example: parameter sweep script

- Example file with list of job directories and number of cores:

```
calc1 384
calc2 384
calc3 768
```

- Calculation input could be set up ahead of submission or on the fly.

Example: parameter sweep script

```
…PBS options…

module load vasp5/5.4.1
job_list="job_list.txt"
resfile="energies.dat"
rootdir=$PBS_O_WORKDIR
IFS=$'\n' jobarray=($(<$job_list))
for ((i=0; i<${#jobarray[@]}; i++)); do
          read -ra tokens <<< "${jobarray[$i]}"
          cd $rootdir/${tokens[0]}
          aprun -n ${tokens[1]} vasp_gam > ${jobarray[$i]}.stdout &
done
wait

for ((i=0; i<${#jobarray[@]}; i++)); do
          read -ra tokens <<< "${jobarray[$i]}"
          cd $rootdir/${tokens[0]}
          enline=$(grep 'free  e' OUTCAR)
          read -ra toten <<< "${enline}"
          printf "%s: %.7d\n" ${tokens[0]} ${toten[4]} >> $rootdir/$resfile
done
```

Summary

- Bash scripting is powerful and useful
- Large number of useful features built in that you may not be aware of
- Particular uses on ARCHER:
  - Ensemble jobs
  - Benchmarking runs
  - Collating results from multiple jobs
  - File and data management
- Further information, Advanced Bash-Scripting Guide:
  - http://tldp.org/LDP/abs/html/index.html