Basic Bash programming

Hans Petter Langtangen[1,2] Joakim Sundnes[1,2]
Ola Skavhaug[3] Simon Funke[1,2]

Center for Biomedical Computing, Simula Research Laboratory[1]

Dept. of Informatics, UiO[2]

Expert Analytics[3]

Aug 31, 2016

Some more course information

Reminder: Sign up to Piazza

- Great knowledge and discussion platform.
- Allows students to ask and answer questions.
- Lecturers/teaching assistants will be active on the platform as well.
- We will announce course updates (for example the video recordings).
- Sign up: http://piazza.com/uio.no/fall2016/inf3331inf4331

Assignment 1

- Deadline: Extended to this Friday
- Make sure that your solutions are pushed to your github repository. All files should be in `https://github.com/UiO-INF3331/INF3331-<UiO-Username>`

Assignments 2

- Assignment 2 is online now.
- Topic: Bash scripting (lecture today).

.

Overview of Unix shells

- The original scripting languages were (extensions of) command interpreters in operating systems
- Primary example: Unix shells
- Bourne shell (`sh`) was the first major shell
- C and TC shell (`csh` and `tcsh`) had improved command interpreters, but were less popular than Bourne shell for programming
- Bourne Again shell (Bash/'bash'): GNU/FSF improvement of Bourne shell
- Other Bash-like shells: Dash (`dash`), Korn shell (`ksh`), Z shell (`zsh`)
- Bash is the dominating Unix shell today

Why learn Bash?

- Learning Bash means learning Unix
- Learning Bash means learning the roots of scripting (Bourne shell is a subset of Bash)
- Shell scripts, especially in Bourne shell and Bash, are frequently encountered on Unix systems
- Bash is widely available (open source) and the dominating command interpreter and scripting language on today's Unix systems

Why learn Bash? (2)

- Shell scripts evolve naturally from a workflow:
  1. A sequence of commands you use often are placed in a file
  2. Command-line options are introduced to enable different options to be passed to the commands
  3. Introducing variables, if tests, loops enables more complex program flow
  4. At some point pre- and postprocessing becomes too advanced for bash, at which point (parts of) the script should be ported to Python or other tools
- Shell scripts are often used to glue more advanced scripts in Perl and Python

Remark

- The plain Bourne shell can be used (`/bin/sh`) when special features of Bash (`/bin/bash`) are not needed
- Most of our examples can in fact be run under Bourne shell (and of course also Bash).
- In Mac OSX, the Bourne shell (`/bin/sh`) is just a link to Bash (`/bin/bash`). In Ubuntu, the Bourne shell (`/bin/sh`) is a link to Dash, a minimal, but much faster shell than bash.

More information

- Offline documentation: `man bash`
- Bash reference manual: http://www.gnu.org/software/bash/manual/bashref.html
- Advanced Bash-Scripting Guide: http://www.tldp.org/LDP/abs/html
- Simple and good tutorial: http://www.linuxintro.org/wiki/Shell_scripting_tutorial

What Bash is good for

- File and directory management
- Systems management (build scripts)
- Combining other scripts and commands
- Rapid prototyping of more advanced scripts
- Very simple output processing, plotting etc.

What Bash is not good for

- Cross-platform portability
- Graphics, GUIs
- Interface with libraries or legacy code
- More advanced post processing and plotting
- Calculations, math etc.

Some common tasks in Bash

- file writing
- for-loops
- running an application
- pipes
- writing functions
- file globbing, testing file types
- copying and renaming files, creating and moving to directories, creating directory paths, removing files and directories
- directory tree traversal
- packing directory trees

Bash example 1: Hello world

```bash
#!/bin/bash
echo "Hello world"
```

Two options to run this script:

- Type the commands directly in the bash shell (only feasible for small scripts).
- Save the code as `helloworld.sh` and run with:

```bash
chmod a+x helloworld.sh  # Make script executable
./helloworld.sh
```

Bash example 2: Hello world, v2

```bash
#!/bin/bash
x="world"
echo "Hello ${x}!"
```

Bash example 3: Line counter

```
#!/bin/bash
filename="text.txt"
declare -i count; count=0

echo "Start counting..."
# loop over all lines of  file
while read p; do
    # increase line counter
    ((count++))
done < $filename
echo "done"

echo "Number of lines in $file: ${count}"
```

Bash variables and commands

- Assign a variable by `x=3.4`, retrieve the value of the variable by `$x` or `${x}` (also called *variable substitution*).
- Variables passed as command line arguments when running a script are called `positional parameters`.
- Bash has a number of built in commands. Type `help` or `help | less` to get a list.
- The real power comes from all the available Unix commands, in addition to your own applications and scripts. Try:

```
curl -s http://www.yr.no/place/Norway/Oslo/Oslo/Oslo/ |
    grep -m 1 "temperature" | cut -d"\"" -f4
```

Bash variables (1)

Variables in Bash are untyped!

Generally treated as character arrays, but permit simple arithmetic and other operations

Variables can be explicitly declared to integer or array;

```
declare -i i       # i is an integer
declare -a A       # A is an array
declare -r r=10    # r is read only
```

Bash variables (2)

The `echo` command is used for writing:

```
s=42
echo "The answer is $s"
```

and variables can be inserted in the text string (variable interpolation)

Frequently seen variables:

- Command line arguments:

```
$1  $2  $3  $4 and so on
```

- All command line arguments as array: `$@`
- Number of command line arguments: `$#`
- The exit status of the last executed command: `$?` (is 0 if command was succesfull)

Bash variables (3)

Comparison of two integers use a syntax different from comparison of two strings:

```
if [ $i -eq 10 ]; then          # integer comparison
if [ "$name" == "10" ]; then  # string  comparison
```

Unless you have declared a variable to be an integer, assume that all variables are strings and use double quotes (strings) when comparing variables in an if test

```
if [ "$?" != "0" ]; then  # this is safe
if [  $?  !=  0  ]; then  # might be unsafe
```

Convenient debugging tool: -x

Each source code line is printed prior to its execution if you add -x as option to /bin/bash

Either in the header

```
#!/bin/bash -x
```

or on the command line:

```
unix> /bin/bash -x hw.sh
unix> sh -x hw.sh  # only works if sh links to bash
unix> bash -x hw.sh
```

Very convenient during debugging

Combining bash commands (1)

- The power of Unix lies in combining simple commands into powerful operations
- Standard bash commands and Unix applications normally do one small task
- Text is used for input and output – easy to send output from one command as input to another

Combining bash commands (2)

Two standard ways to combine commands:

- The pipe, sends the output of one command as input to the next:

```
ls -l | grep 3331
```

Will list all files having 3331 as part of the name

- Executing a command, storing the result as a variable:

```
time=$(date)
# or time=`date`
echo $time
```

Combining bash commands (3)

More useful applications of pipes:

```
# send files with size to sort -rn
# (reverse numerical sort) to get a list
# of files sorted after their sizes:

/bin/ls -s | sort -rn


cat $case.i | oscillator
# is the same as
oscillator < $case.i
```

Make a new application: sort all files in a directory tree `root`, with the largest files appearing first, and equip the output with paging functionality:

```
du -a root | sort -rn | less
```

Bash redirects

Redirects are used to pass output to either a file or stream.

```
echo "Hei verden" > myfile.txt  # Save output to file
wc -w < myfile.txt    # Use file content as command input
```

Note: Pipes can be (in a clumpsy way) reimplemented with redirects:

```
prog1 > myfile && prog2 < myfile
```

is the same as

```
prog1 | prog2
```

Redirects and stdin, stdout, stderr

```
rm -v *.txt 1> out.txt        # Redirect stout to a file
rm -v *.txt 2> err.txt        # Redirect stderr to a file
rm -v *.txt &> outerr.txt     # Redirect stdout and stderr to file
rm -v *.txt 1>&2              # Redirect stdout to stderr
rm -v *.txt 2>&1              # Redirect stderr to stdout
```

You can print to stderr with:

```
echo "Wrong arguments" >&2
```

Redirects and pipes can be combined:

```
./compile 2>&1 | less  # View both stdout and stderr in less
```

Example: the classical Unix script

A combination of commands, or a single long command, that you use often;

```
./pulse_app -cmt WinslowRice -casename ellipsoid
    < ellipsoid.i | tee main_output
```

(should be a single line) In this case, flexibility is often not a high priority. However, there is room for improvement;

- Not possible to change command line options, input and output files
- Output file `main_output` is overwritten for each run
- Can we edit the input file for each run?

Problem 1; changing application input

In many cases only one parameter is changed frequently;

```
CASE='testbox'
CMT='WinslowRice'
if [ $# -gt 0 ]; then
   CMT=$1
fi
INFILE='ellipsoid_test.i'
OUTFILE='main_output'

./pulse_app -cmt $CMT -cname $CASE
  < $INFILE | tee $OUTFILE
```

Still not very flexible, but in many cases sufficient. More flexibility requires more advanced parsing of command line options, which will be introduced later.

Problem 2; overwriting output file

A simple solution is to add the output file as a command line option, but what if we forget to change this from one run to the next?

Simple solution to ensure data is never over-written:

```
jobdir=$PWD/$(date)
mkdir $jobdir
cd $jobdir

./pulse_app -cmt $CMT -cname $CASE  < $INFILE | tee $OUTFILE
cd ..
if [ -L 'latest' ]; then
    rm latest
fi
ln -s $jobdir latest
```

Problem 2; overwriting output file (2)

Alternative solutions;

- Use process ID of the script ($$, not really unique)
- `mktemp` can create a temporary file with a unique name, for use by the script
- Check if subdirectory exists, exit script if it does;

```
dir=$case
# check if $dir is a directory:
if [ -d $dir ]
  #exit script to avoid overwriting data
  then
    echo "Output directory exists, provide a different name"
    exit
fi
mkdir $dir    # create new directory $dir
cd $dir       # move to $dir
```

Alternative if-tests

As with everything else in Bash, there are multiple ways to do if-tests:

```
# the 'then' statement can also appear on the next line:
if [ -d $dir ]; then
  exit
fi

# another form of if-tests:
if test -d $dir; then
  exit
fi

# and a shortcut:
[ -d $dir ] && exit
test -d $dir && exit
```

Problem 3; can we edit the input file at run time?

- Some applications do not take command line options, all input must read from standard input or an input file
- A Bash script can be used to equip such programs with basic handling of command line options
- We want to grab input from the command line, create the correct input file, and run the application

File reading and writing

File writing is efficiently done by 'here documents':

```
cat > myfile <<EOF
multi-line text
can now be inserted here,
and variable substitution such as
$myvariable is
supported.
EOF
```

The final EOF must start in column 1 of the script file.

EOF is an arbitrary keyword here.

Parsing command-line options

```
# read variables from the command line, one by one:
while [ $# -gt 0 ]  # $# = no of command-line args.
do
    option=$1; # load command-line arg into option
    shift;       # eat currently first command-line arg
    case "$option" in
        -m)
            m=$1; shift; ;;  # ;; indicates end of case
        -b)
            b=$1; shift; ;;
                ...
        *)
            echo "$0: invalid option \"$option\""; exit ;;
    esac
done
```

Alternative to case: if

case is standard when parsing command-line arguments in Bash, but if-tests can also be used. Consider

```
case "$option" in
    -m)
        m=$1; shift; ;;   # load next command-line arg
    -b)
        b=$1; shift; ;;
    *)
        echo "$0: invalid option \"$option\""; exit ;;
 esac
```

versus

```
 if [ "$option" == "-m" ]; then
     m=$1; shift;  # load next command-line arg
 elif [ "$option" == "-b" ]; then
     b=$1; shift;
 else
     echo "$0: invalid option \"$option\""; exit
 fi

echo "Command line arguments:"
[ -n "$m" ] && echo "m=$m"
[ -n "$b" ] && echo "b=$b"
```

After assigning variables, we can write the input file

```
# write to $infile the lines that appear between
# the EOF symbols:

cat > $infile <<EOF
        gridfile='test2.grid'
        param_a=4.5
EOF
```

Then execute the program as usual

Redirecting input to read from the new input file

```
../pulse_app < $infile
```

We can add a check for successful execution. The shell variable $? is 0 if last command was successful, otherwise `$? != 0`.

```
if [ "$?" != "0" ]; then
  echo "running pulse_app failed"; exit 1
fi

# exit n sets $? to n
```

Other uses of cat

```
cat myfile               # write myfile to the screen
cat myfile >  yourfile # write myfile to yourfile
cat myfile >> yourfile # append myfile to yourfile
cat myfile | wc          # send myfile as input to wc
```

For-loops

What if we want to run the application for multiple input files?

```
./run.sh test1.i test2.i test3.i test4.i
```

or

```
./run.sh *.i
```

A for-loop over command line arguments

```
for arg in $@; do
  ../../build/app/pulse_app < $arg
done
```

Can be combined with more advanced command line options, output directories, etc...

For-loops (2)

For loops for file management:

```
files=`ls *.tmp`

for file in $files
do
  echo removing $file
  rm -f $file
done
```

Counters

Declare an integer counter:

```
declare -i counter
counter=0
# arithmetic expressions must appear inside (( ))
((counter++))
echo $counter  # yields 1
```

For-loop with counter:

```
declare -i n; n=1
for arg in $@; do
  echo "command-line argument no. $n is <$arg>"
  ((n++))
done
```

C-style for-loops

```
declare -i i
for ((i=0; i<$n; i++)); do
  echo $c
done
```

Example: bundle files

Idea:

- Scripts packs a series of files into one file
- Executing this single file as a Bash script packs out all the individual files again

Usage:

```
bundle file1 file2 file3 > onefile  # pack
bash onefile # unpack
```

Writing `bundle` is easy:

```
#/bin/sh
for i in $@; do
    echo "echo unpacking file $i"
    echo "cat > $i <<EOF"
    cat $i
    echo "EOF"
done
```

The bundle output file

Consider 2 fake files; file1

```
Hello, World!
No sine computations today
```

and file2

```
1.0 2.0 4.0
0.1 0.2 0.4
```

Running `bundle file1 file2` yields the output

```
echo unpacking file file1
cat > file1 <<EOF
Hello, World!
No sine computations today
EOF
echo unpacking file file2
cat > file2 <<EOF
1.0 2.0 4.0
0.1 0.2 0.4
EOF
```

Running an application

Running in the foreground:

```
cmd="myprog -c file.1 -p -f -q";
$cmd < my_input_file

# output is directed to the file res
$cmd < my_input_file > res

# process res file by Sed, Awk, Perl or Python
```

Running in the background:

```
myprog -c file.1 -p -f -q < my_input_file &
```

or stop a foreground job with Ctrl-Z and then type bg

Functions

```
function system {
# Run operating system command and if failure, report and abort

  "$@"
  if [ $? -ne 0 ]; then
    echo "make.sh: unsuccessful command $@"
    echo "abort!"
    exit 1
  fi
}
# function arguments: $1 $2 $3 and so on
# return value: last statement
# call:
name=mydoc
system pdflatex $name
system bibtex $name
```

How to return a value from a function? Define a new variable within the function - all functions are global!

File globbing, for loop on the command line

List all .ps and .gif files using wildcard notation:

```
files=`ls *.ps *.gif`

# or safer, if you have aliased ls:
files=`/bin/ls *.ps *.gif`

# compress and move the files:
gzip $files
for file in $files; do
  mv ${file}.gz $HOME/images
```

Testing file types

```
if [ -f $myfile ]; then
    echo "$myfile is a plain file"
fi

# or equivalently:
if test -f $myfile; then
    echo "$myfile is a plain file"
fi

if [ ! -d $myfile ]; then
    echo "$myfile is NOT a directory"
fi

if [ -x $myfile ]; then
    echo "$myfile is executable"
fi

[ -z $myfile ] && echo "empty file $myfile"
```

Rename, copy and remove files

```
# rename $myfile to tmp.1:
mv $myfile tmp.1

# force renaming:
mv -f $myfile tmp.1

# move a directory tree my tree to $root:
mv mytree $root

# copy myfile to $tmpfile:
cp myfile $tmpfile

# copy a directory tree mytree recursively to $root:
cp -r mytree $root

# remove myfile and all files with suffix .ps:
rm myfile *.ps

# remove a non-empty directory tmp/mydir:
rm -r tmp/mydir
```

Directory management

```
# make directory:
$dir = "mynewdir";
mkdir $mynewdir
mkdir -m 0755 $dir  # readable for all
mkdir -m 0700 $dir  # readable for owner only
mkdir -m 0777 $dir  # all rights for all

cd $dir # move to $dir
cd      # move to $HOME

# create intermediate directories (the whole path):
mkdir -p  $HOME/bash/prosjects/test1
```

The find command

Very useful command!

`find` visits all files in a directory tree and can execute one or more commands for every file

Basic example: find the `oscillator` codes

```
find $scripting/src -name 'oscillator*' -print
```

Or find all PostScript files

```
find $HOME \( -name '*.ps' -o -name '*.eps' \) -print
```

We can also run a command for each file:

```
find rootdir -name filenamespec -exec command {} \; -print
# {} is the current filename
```

Applications of find (1)

Find all files larger than 2000 blocks a 512 bytes (=1Mb):

```
find $HOME -name '*' -type f -size +2000 -exec ls -s {} \;
```

Remove all these files:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \; -exec rm -f {} \;
```

or ask the user for permission to remove:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \; -ok rm -f {} \;
```

Applications of find (2)

Find all files not being accessed for the last 90 days:

```
find $HOME -name '*' -atime +90 -print
```

and move these to /tmp/trash:

```
find $HOME -name '*' -atime +90 -print \
    -exec mv -f {} /tmp/trash \;
```

Tar and gzip

The `tar` command can pack single files or all files in a directory tree into one file, which can be unpacked later

```
tar -cvf myfiles.tar mytree file1 file2

# options:
# c: pack, v: list name of files, f: pack into file

# unpack the mytree tree and the files file1 and file2:
tar -xvf myfiles.tar

# options:
# x: extract (unpack)
```

The tarfile can be compressed:

```
gzip mytar.tar

# result: mytar.tar.gz
```

Two find/tar/gzip examples

Pack all PostScript figures:

```
tar -cvf ps.tar `find $HOME -name '*.ps' -print`
gzip ps.tar
```

Pack a directory but remove CVS directories and redundant files

```
# take a copy of the original directory:
cp -r myhacks /tmp/oblig1-hpl
# remove CVS directories
find /tmp/oblig1-hpl -name CVS -print -exec rm -rf {} \;
# remove redundant files:
find /tmp/oblig1-hpl \( -name '*~' -o -name '*.bak' \
 -o -name '*.log' \) -print -exec rm -f {} \;
# pack files:
tar -cf oblig1-hpl.tar /tmp/tar/oblig1-hpl.tar
gzip oblig1-hpl.tar
# send oblig1-hpl.tar.gz as mail attachment
```