BASH programming
http://www.gnu.org/software/bash/

Basic tutorial to BASH programming

Sébastien LE ROUX sebastien.leroux@ipcms.unistra.fr

INSTITUT DE PHYSIQUE ET DE CHIMIE DES MATÉRIAUX DE STRASBOURG,
DÉPARTEMENT DES MATÉRIAUX ORGANIQUES,
23 RUE DU LOESS, BP43,
F-67034 STRASBOURG CEDEX 2, FRANCE

NOVEMBER 14, 2016

Contents

Linux system

For detailed references see [1, 2].

1.1 User management

The Linux system is designed as a multi-user system, it means that many users can work with the system at one time. That is, on a Linux system it is possible to distinguish two type of users:

- The "standard" user who plays, works, in short who uses the computer but with a restricted access.
- The "super" user or administrator (often called 'root') who can do everything in particular administrate and thus change the configuration of the system.

Depending on the Linux distribution the super user can:

- Have an account on the computer and therefore access his/her personal home directory (`/root`). In that case to administrate the computer the super user has to log-in using the "`su`" command.
  → Red-Hat based Linux.
- Not have an account, then only users that have been granted permission can administrate the computer using the "`sudo`" command.
  → Debian based Linux.

Notice that on a Linux system it is usual to find group of users, ie. users that share the same privileges and that have been gathered into a group.

1.2 File permissions

The purpose of this section is to introduce the basic ideas on the file permission system on a Linux/Unix system. On a Linux system file permissions does not necessarily means administrating the system by installing the driver for the newly installed piece of hardware or updating

the program library of the computer. Understanding the file permission system of the Linux system is the most basic prerequisite to becoming a 'Power' user, ie. a user having the power to understand what he/she is doing with the computer.

The golden rule of a Linux/Unix system is that everything is a file. The computer is seen by the operating system as a file tree and hence each component (screen, keyboard, mouse, graphic card ...) is seen as a file. When a hardware component is added to the computer, a file is added in the tree of the operating system `/`. Therefore permissions to manipulate the hardware from the software point of view are handled like the one of a file on the hard drive.
In a file tree one can distinguish two objects the first are the 'simple files' and the second the 'directories'. For both permissions are handled on the same way:

- The different permissions that can be granted for a simple file are:
  - read: to visualize its content
  - write: to modify its content (ex: editing)
  - execute: to execute its content (ex: program)
- The different permissions that can be granted for a directory are:
  - read: to visualize its content
  - write: to modify its content (ex: adding new files)
  - execute: to go inside this directory (ex: changing directory)

To obtain the information regarding the permissions of a file in the Linux tree one case use the "`ls -l`" command (see section [Sec. 3.1]):

```
user@localhost ]$ ls -lh MyFile
-rwxrw-r--. 1 user ipcms 1.0K  15 févr.  2011 MyFile
```

The syntax for this line is the following:

- "`-`" means "file", alternatively "`d`" means directory and "`s`" means symbolic link.
- "`rwxrw-r-`" are the permissions on the file.
- "`1`" is the number of physical links of the file with the hard drive.
- "`user`" is the name of the owner of the file.
- "`ipcms`" is the name of the group the owner of the file belongs to.
- "`1.0K`" is the size of the file on the hard drive.
- "`15 févr.  2011`" is the last modification date of the file.
- "`MyFile`" is the name of the file

The permissions on the file can be decomposed in 3 series of 3 letters: r (for read), w (for write) and x (for execute), also the symbol "-" in place of a letter means that the permission is denied. The first 3 letters refer to the owner of the file, the second to the group the owner belong to, and the third to all other users of the computer. In the case of the file "MyFile" the owner of the file ("user") has all permissions (read, write execute). The members of the group the owner of the file belongs to (so the members of the group "ipcms") can read and modify the file. Finally other users can only read the file "MyFile".
It is possible to define the access, utilization or modification permissions of a file with the "chmod" command (see section [Sec. 3.1]) and using 3 numbers:

- 1 = execute
- 2 = write
- 4 = read

As well as their combinations:

- 3 = 1 + 2 = execute + read
- 5 = 1 + 4 = execute + write
- 6 = 2 + 4 = write + read
- 7 = 1 + 2 + 4 = execute + write + read

Finally we distinguish the 3 classes of users for whom it is possible to grant permissions on a file, the owner, the group the owner belongs to and all other users recognized by the system. The permissions on the file being granted for each category of user by a single combination of the number 1, 2 and 4.
For example it is possible to grant permissions 644 to a file:

```
user@localhost ]$ chmod 644 MyFile
```

The command will grant permission to read and modify the file to the owner, thus removing the permission to execute it. It will grant permission to read the file to the members of the group the owner belongs too, thus removing the permission to modify the file. And it will not modify the permissions granted to all the other users of the computer.

1.3 Environment variables

Adapted from the Wikipedia web page

1.3.1 Introduction

Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They can be said in some sense to create the operating environment in which a process runs.
In a Linux system each process has its own private set of environment variables. By default, when a process is created it inherits a duplicate environment of its parent process, except for explicit changes made by the parent when it creates the child. From shells such as bash, you can change environment variables for a particular command invocation by indirectly invoking it via `env` or using the `ENVIRONMENT_VARIABLE=VALUE` command notation (see section [Sec. 3.3].
Examples of environment variables include:

- **`PATH`**: lists directories the shell searches, for the commands the user may type without having to provide the full path.
- **`HOME`**: indicates where a user's home directory is located in the file system.
- **`PWD`**: indicates the working directory.
- **`TERM`**: specifies the type of computer terminal or terminal emulator being used.
- **`SHELL`**: specifies the type of command interpreter being used (BASH, TCSH ...).
- **`PS1`**: specifies how the prompt is displayed in the Bourne shell and variants.

Shell scripts and batch files use environment variables to communicate data and preferences to child processes. They can also be used to store temporary values for reference later in the script.
In Unix/Linux, an environment variable that is changed in a script or compiled program will only affect that process and possibly child processes. The parent process and any unrelated processes will not be affected.
In Unix, the environment variables are normally initialized during system start-up by the system init scripts, and hence inherited by all other processes in the system. Users can modify them in the profile script for the shell they are using.
The variables can be used both in scripts and on the command line. They are usually referenced by putting special symbols in front of or around the variable name. For instance, to display the program search path, in most scripting environments, the user has to type:

```
user@localhost ]$ echo $PATH
```

The command `env` (see section [Sec. 3.3]), displays all environment variables and their values.

1.3.2 Working principles of environment variables

A few simple principles govern how environment variables achieve their effect.

- Local to process
  Environment variables are local to the process in which they were set. That means if we open two terminal windows (Two different processes running shell) and change value of environment variable in one window, that change will not be seen by other window.

- Inheritance
  When Parent process creates a child process, the child process inherits all the environment variable and their values which parent process had.

- Case sensitive
  The names of environment variables are case sensitive.

- Persistence
  Environment variables persistence can be session-wide or system-wide.

The BASH "Bourne-Again" SHell

2.1 What is a shell ?

From the official GNU Bash web page:

At its base, a shell is simply a macro processor that executes commands. The term macro processor means functionality where text and symbols are expanded to create larger expressions.

A Unix shell is both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of gnu utilities. The programming language features allow these utilities to be combined. Files containing commands can be created, and become commands themselves. These new commands have the same status as system commands in directories such as `/bin`, allowing users or groups to establish custom environments to automate their common tasks.

Shells may be used interactively or non-interactively. In interactive mode, they accept input typed from the keyboard. When executing non-interactively, shells execute commands read from a file.

A shell allows execution of commands, both synchronously and asynchronously. The shell waits for synchronous commands to complete before accepting more input; asynchronous commands continue to execute in parallel with the shell while it reads and executes additional commands. The redirection constructs permit fine-grained control of the input and output of those commands. Moreover, the shell allows control over the contents of commands environments.

Shells also provide a small set of built-in commands (built-ins) implementing functionality impossible or inconvenient to obtain via separate utilities. For example, `cd`, `break`, `continue`, and `exec` cannot be implemented outside of the shell because they directly manipulate the shell itself. The `history`, `getopts`, `kill`, or `pwd` built-ins, among others, could be implemented in

separate utilities, but they are more convenient to use as built-in commands.

While executing commands is essential, most of the power (and complexity) of shells is due to their embedded programming languages. Like any high-level language, the shell provides variables, flow control constructs, quoting, and functions.

2.2 The BASH command interpreter

From the official GNU Bash web page:

Bash is the shell, or command language interpreter, that will appear in the GNU operating system. Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.

The improvements offered by BASH include:

- Command line editing
- Unlimited size command history
- Job Control
- Shell Functions and Aliases
- Indexed arrays of unlimited size
- Integer arithmetic in any base from two to sixty-four

The manual is available online at http://www.gnu.org/software/bash/manual/.

Start-up scripts

When Bash starts it executes commands in a variety of different scripts.

- When Bash is invoked as an interactive login shell it:
  1. First reads and executes commands from the file `/etc/profile`, if that file exists.
  2. Then looks for `~/.bash_profile` (in Unix/Linux language `~ = $HOME`), `~/.bash_login` and `~/.profile`, in that order, and reads and executes commands from the first one that exists and is readable.
- When an interactive shell that is not a login shell is started, Bash reads and executes commands from `~/.bashrc`, if that file exists.
- When a login shell exits, Bash reads and executes commands from the file `~/.bash_logout`, if it exists.

2.3 Bash scripting

The command line

The command line or prompt is the interactive mode offered to users to enter commands typed from the keyboard. In most Linux system the prompt looks like:

```
user@localhost ]$
```

From this command line the user can enter directly any command available in the `$PATH` or alternatively the direct path to the target command. On the prompt command, option(s) and argument(s) separated by empty spaces are numbered with the number that corresponds to their order of appearance on the line starting from 0 for the command:

```
user@localhost ]$ ls -l
```

the command `ls` has the number 0 and its option `-l` has number the 1.

```
user@localhost ]$ cat MyFile | grep MyKeyword
```

the command `cat` has the number 0, the argument `MyFile` has the number 0, the redirecting pipeline `|` (for details see section [Sec. 3.2]) has the number 2, the command `grep` has the number 3 and finally the argument `MyKeyword` has number 4.

Bash script

A script is always composed of at least two lines:

```
#!/bin/bash

# This little example illustrates how to say 'Hello' in Bash programming
echo "Hello"
```

the first tells the system with program to use to run the script, here the Bash command interpreter; the second is the command to be run.
Comments can be inserted at any place using the `#` followed by a space.

In Bash like in any other programming language you will find many, somewhat different and somewhat alike, possibilities to achieve the same goal. In the next pages I provide a very short introduction, as well as few very simple examples, to the most useful (in my very humble opinion) Bash scripting instructions. Since these examples are based on my small experience, I am pretty sure that it exists other tools worth being presented thereafter but still missing so far. Time will (hopefully) help me to correct this, otherwise, for, I quote, "An in-depth exploration of the art of shell scripting", simply read the amazing Advanced Bash-Scripting Guide written by Mendel Cooper [2].

2.4 Scripting instructions

Option(s) and argument(s)

It is possible to pass option(s) and/or argument(s) to a script by referring to the number they will appear on the command line (see section [Sec. 2.3]), ex:

```
user@localhost ]$ MyScript MyOption1 MyOption2 MyArg
```

Corresponding possible structures of the file `MyScript` that contains command(s):

```
#!/bin/bash

ls -$1$2 $3
```

or

```
#!/bin/bash

OPT1=$1
OPT2=$2
ARG=$3

ls -$OPT1$OPT2 $ARG
```

If `MyOPtion1`, `MyOption2` and `MyArg` are respectively `l`, `t` and `MyFile` then the script `MyScript` will simply be equivalent to:

```
user@localhost ]$ ls -lt MyFile
```

`let` - numerical variables

In Bash scripting it possible to assign a numerical value to a variable, in that case the declaration of the variable is particular and uses the `let` command:

Examples

Ex 1:

```
#!/bin/bash

let VAL=0
```

Ex 2:

```
#!/bin/bash

NUM=`cat MyFile | wc -l`
let VAL=$NUM
```

The following syntax can be used to increment a numerical variable in Bash scripting using the `let` command:

```
#!/bin/bash

# Option 1
let NUM=1
let NUM=$NUM+1

# Option 2
let NUM=1
let "NUM+=1"
```

and similarly with the - operator, notice that the increment (here 1) could be different.

`if` - the conditional tests

Comparison operators

- Basic comparisons:

  `-f` = exists and is a file
  `-d` = exists and is a directory
  `-h` = exists and is symbolic link
  `-r` = has read permission
  `-w` = has write permission
  `-x` = has execute permission
  `-O` = you are the owner of the file
  `-nt` = is newer than
  `-ot` = is older than
  `!` = is false

- Integer comparisons:

  `-eq` = is equal to
  `-ne` = is no equal to
  `-gt` = is greater than
  `-ge` = is greater than or equal to
  `-lt` = is less than
  `-ge` = is less than or equal to

- String comparisons:

  `=` = **`==`** = is equal to
  `!=` = is not equal to
  `-z` = string is null
  `-n` = string is not null

General syntax of the "`if`" command

```
#!/bin/bash

if [ TEST ]; then
  We do something
fi
```

General syntax of the "`if-else`" command

```
#!/bin/bash

if [ TEST ]; then
  We do something
else
  We do something else
fi
```

General syntax of the "`if-elif-else`" command

```
#!/bin/bash

if [ TEST ]; then
  We do something
elif [ TEST ]; then
  We do something else
else
  We do something else
fi
```

Examples

Ex 1:

```
#!/bin/bash

if [ -f MyFile ]; then
  cp MyFile MyFile.bkp
fi
```

Ex 2:

```
#!/bin/bash

NUM=`cat MyFile | wc -l`
let VAL=$NUM
if [ $VAL -eq 100 ]; then
  echo "MyFile is 100 lines long"
fi
```

Ex 3:

```
#!/bin/bash

if [ -d MyDirectory ]; then
  mkdir MyDirectory/data
elif [ -f MyDirectory ]; then
  mv MyDirectory MyFile
  mkdir -p MyDirectory/data
elif [ -h MyDirectory ]; then
  mv MyDirectory MyLink
  mkdir -p MyDirectory/data
else
  mkdir -p MyDirectory/data
fi
```

Ex 4:

```
#!/bin/bash

A="Car"
B="Truck"

if [ $A != $B ]; then
  C=`echo $A | sed 's;C;B;1'`
  echo $C
else
  echo $A" "$B | awk '{printf $NF}'
fi
```

`for` - the loops

General syntax of the "`for`" command

```
#!/bin/bash

# Option 1
LIST="1 2 3 4 ... 1000"
for VAR in $LIST
do
  We do something
done

# Option 2
let NUM1=1
let NUM2=1000
for VAR in $(seq $NUM1 $NUM2)
do
  We do something
done

# Option 3 - integer loops
let NUM=1000
# Double parenthesis and limit variable, NUM, written with no "$" symbol
for (( VAR=1 ; VAR <= NUM ; VAR++ ))
do
  We do something
done
```

Examples

Ex 1:

```
#!/bin/bash

LIST= `ls -l | grep '^-' | awk '{printf $NF" "}'`
for LI in $LIST
do
  NLIGN=`cat $LI | wc -l`
  echo "File= "$LI", Number of ligne(s)= "$NLIGN
done
```

Ex 2:

```
#!/bin/bash

for LI in $(seq 1 100)
do
  for MI in $(seq 1 100)
  do
    mkdir -p $li/$MI
    echo "The path: "$LI"/"$MI" has been created"
  done
done
```

`while` - conditional loops

General syntax of the "`while`" command

```
#!/bin/bash

while [ TEST ]
do
  we do something
done
```

Examples

Ex 1:

```
#!/bin/bash

let A=50
while [ $A -ge 1 ]
do
  echo "A= "$A
  let "A-=1"
done
```

Ex 2:

```
#!/bin/bash

STATUS=`ps auwx | grep 'MyCmd' `
while [ -n $STATUS ]
do
  sleep 60
  STATUS=`ps auwx | grep 'MyCmd' `
done
```

`until` - conditional loops

General syntax of the "`until`" command

```
#!/bin/bash

until [ TEST ]
do
  we do something
done
```

Examples

Ex 1:

```
#!/bin/bash

let A=50
until [ $A -lt 1 ]
do
  echo "A= "$A
  let "A-=1"
done
```

Ex 2:

```
#!/bin/bash

STATUS=`ps auwx | grep 'MyCmd'`
until [ -z $STATUS ]
do
  sleep 60
  STATUS=`ps auwx | grep 'MyCmd'`
done
```

`case` - testing and branching

General syntax of the "`case`" command

```
#!/bin/bash

case "$variable" in

  "$condition1" )
    we do something
  ;;

  "$condition2" )
    we do something
  ;;

esac
```

Examples

Ex 1:

```
#!/bin/bash

case "$1" in

  "" )
     echo "Usage: SCRIPT <filename>"
     exit
  ;;

  * )
     FILENAME=$1
  ;;

esac
```

Ex 2:

```
#!/bin/bash

ARG=$1
case "$ARG" in

  "Hello" )
    echo "You too, thank you"
  ;;

  "Bye Bye" )
    echo "See you next time"
    exit
  ;;

  * )
    echo "What do you want ?"
  ;;

esac
```

`function` - advanced scripting

General syntax of the "`function`" command

Like any other programming languages, Bash has functions, though in a somewhat limited implementation. A function is a subroutine, a code block that implements a set of operations, a "black box" that performs a specified task. Furthermore as it is possible to pass option(s) and/or argument(s) on the command line calling the main script, it is possible to pass option(s) and/or argument(s) to the function.
Wherever there is repetitive code, such as a repeating task with only slight variations in procedure, it is worth using a function.

```
#!/bin/bash

function my_function {
  we do something
}

my_function
```

or

```
#!/bin/bash

function my_function {
  FuncVar=$1

  we do something
}

MyVar=`ls -l | wc -l`
my_function $MyVar
```

Examples

Ex 1:

```
#!/bin/bash

function calc_length {
  TestFile=$1
  VAR=$2

  echo "File "$TestFile" is "$VAR" line(s) long"
}

LISTEF=`ls -l | grep '^-' | awk '{printf $NF" "}'`

for FILE in $LISTEF
do
  FLENGTH=`cat $FILE | wc -l`
  calc_length $FILE $FLENGTH
done
```

Ex 2:

```
#!/bin/bash

function calc_length {
  TestFile=$1

  VAR=`cat $TestFile | wc -l`
  echo "File "$TestFile" is "$VAR" line(s) long"
}

LISTEF=`ls -l | grep '^-' | awk '{printf $NF" "}'`

for FILE in $LISTEF
do
  calc_length $FILE
done
```

2.5 Examples

Example 1 - psclean

- **psclean** *"clean (kill -9) all processes with [ARG] in name"*
  - usage: **psclean [ARG]**
  - ex: `user@localhost ]$ psclean cpmd`

```
#!/bin/bash

job=$1

listps=`ps auwx | grep '$job' | awk '{printf $2" "}'`
for idps in $listps
do
  kill -9 $idps
done

# One more time the other way around to be sure

listps=`ps auwx | tac | grep '$job' | awk '{printf $2" "}'`
for idps in $listps
do
  kill -9 $idps
done
```

Example 2 - archdata

- **archdata** *"archive (tar.bz2 format) and then delete archived files"*
  - usage: **archdata [ARG]**
  - ex: `user@localhost ]$ archdata my_results`

```
#!/bin/bash

toarch=$1

mv $toarch $toarch-saved
tar -jcf --remove-files $toarch-saved.tar.bz2 $toarch-saved
```

Example 3 - checkjobs

- **checkjobs** *"check multiple running calculations on remote server"*
  - usage: **checkjobs [ARG1] [ARG2] [ARG3]**
  - ex: `user@localhost ]$ checkjobs MyServer MyJOB MyXYZfile`

```
#!/bin/bash

function checkjob {
# We check if the calculation directory already exists
  if [ ! -d $HOME/jobs/check/$1/$2 ]; then
    mkdir $HOME/jobs/check/$1/$2
  fi
  cd $HOME/jobs/check/$1/$2
# We check if some calculation(s) have already been completed
# if yes we save the previous XYZ coordinate files
  if [ -f $3.xyz ]; then
    val=`ls -l *.xyz | |wc -l`
    let num=$val
    mv $3.xyz saved-$num.xyz
  fi
# We prepare the R.I.N.G.S. code calculation
  if [ ! -d rings ]; then
    mkdir -p rings/data/
    cp $HOME/jobs/check/input rings/
    cp $HOME/jobs/check/options rings/
  fi
# We copy the data from the remote server, this required a SSH
# connexion managed using RSA (or DSA) encryption keys (no passwords)
  scp $1:files/glasses-files/my_running_jobs/$2/$1.xyz .
  cp $1.xyz rings/data/
  cd rings
# We run the analysis, see http://rings-code.sourceforge.net
  rings input
# We append the results to the survey files
  cat gr/gr-xrays.dat >> $HOME/jobs/survey/$2-gr.dat
  echo " " >> $HOME/jobs/survey/$2-gr.dat
  cat sq/sq-xrays.dat >> $HOME/jobs/survey/$2-sq.dat
  echo " " >> $HOME/jobs/survey/$2-sq.dat
}
```

```
SERV=$1
JOB=$2
XYZ=$3
# First we append the experimental data to the survey files,
# The -gr.dat and -sq.dat files will contain respectively the calculated
# quantities at different time step during the calculation
cat $HOME/jobs/check/grexp-$XYZ.dat >> $HOME/jobs/survey/$XYZ-gr.dat
echo " " >> $HOME/jobs/survey/$2-gr.dat
cat $HOME/jobs/check/sqexp-$XYZ.dat >> $HOME/jobs/survey/$XYZ-sq.dat
echo " " >> $HOME/jobs/survey/$2-sq.dat
# 1000 time steps each separated by 1800 s
for tps $(seq 1 1000)
do
# We call the function to check on the jobs
  checkjob $SERV $JOB $XYZ
  sleep 1800
done
```

Example 4 - runcalc

- **runcalc** *"run calculation using configurations stored in CPMD trajectory file"*
  - usage: **runcalc**
  - ex: `user@localhost ]$ runcalc`

```
#!/bin/bash

# We want to run a CPMD dos calculation on 100 configurations
# These configurations are selected from a CPMD trajectory file
# that contains the history of the atomic positions saved during the MD
for li in $(seq 1 100)
do
  mkdir $li
# We have a standard PBS script to run CPMD calculation, we need to correct the PATH
# where the PBS program will look for the data, to do that in the appropriate location
# we inserted the 'MYDIRECTORY' keyword, now we substitute this keyword by
# the correct location and we put the new PBS script in that location
  sed 's/MYDIRECTORY/\/scratch\/$USERNAME\/'$li'/1' cpmd.pbs > $li/cpmd.pbs
# Now we create the CPMD input file, for that we already prepared
# the generic part of the input which are the same for all calculation
  cat top.in > $li/ge2se3.in
# Only atomic positions change, we will split the complete the CPMD trajectory file
# to keep 100 configurations each separated by the same MD time, we do have 120 atoms
  let num=$li
  let num=$num*120
  tac full-conf.trj | tail --lines=$num | tac | tail --lines=120 > tmp.trj
# Now we insert the atomic coordinates of the 48 Ge atoms
  tac tmp.trj|tail --lines=48|tac|awk '{printf $2"  "$3"  "$4"\n"}' >> $li/ge2se3.in
  cat mid.in >> $li/ge2se3.in
# Afterwards we insert the atomic coordinates of the 72 Se atoms
  tail --lines=72 tmp.trj | awk '{printf $2"  "$3"  "$4"\n"}' >> $li/ge2se3.in
  echo "&END" >> $li/ge2se3.in
  cd $li
# Finally we run the calculation
  qsub cpmd.pbs
  cd ..
done
```

Example 5 - submit.pbs

- **submit.pbs** *"Script for CPMD calculations on HPC"*
  - usage: **qsub submit.pbs**
  - ex: `user@localhost ]$ qsub submit.pbs`

```
#!/bin/bash

# First we specify the PBS options, using the '#PBS' command

#PBS -l nodes=4:ppn=4
#PBS -l walltime=01:00:00
#PBS -N CPMD
#PBS -M leroux@ipcms.u-strasbg.fr
#PBS -o errout.o
#PBS -e errout.e

# We want to use the intel tools
module load compilers/intel10
module load mpi/openmpi.i10
module load libs/mkl

# Path for the directory storing pseudo-potential files:
export PPLIB=$HOME/pseudo

WORKDIR=$PWD

# Threads Open-MP
export OMP_NUM_THREADS=1
export MKL_NUM_THREADS=1

mpirun $HOME/appz/bin/cpmd.x  $WORKDIR/ge2se3.in $PPLIB > $WORKDIR/ge2se3.out
```

Command glossary

3.1 Standard commands

- **man** *"call for help - manual pages"*
  - usage: **man [ARG]**
  - ex: `user@localhost ]$ man ls`

- **ls** *"list content"*
  - usage: **ls [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ ls`
  - options: `-l` (detailed list), `-t` (sort by time), `-a` (show hidden files), `-h` (human readable)

- **cd** *"change directory"*
  - usage: **cd [ARG]**
  - ex: `user@localhost ]$ cd MyDirectory`

- **mv** *"move"*
  - usage: **mv [ARG1] [ARG2]**
  - ex: `user@localhost ]$ mv MyFile MyNewFile`

- **cp** *"copy"*
  - usage: **cp [ARG1] [ARG2]**
  - ex: `user@localhost ]$ cp MyFile MyNewFile`

- **rm** *"remove"*
  - usage: **rm [OPTION] ... [ARG] ...**
  - ex: `user@localhost ]$ rm MyFile`

- **mkdir** *"make directory"*
  - usage: **mkdir [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ mkdir MyNewDirectory`
  - options: `-p` (with parents)

- **df** *"display information on file system"*
  - usage: **df [OPTION] ...**
  - ex: `user@localhost ]$ df -h`
  - options: `-h` (human readable)

- **du** *"disk usage"*
  - usage: **du [OPTION] ... [ARG] ...**

- ex: `user@localhost ]$ du -sh MyFile`
- options: `-s` (total size), `-h` (human readable)

- **cat** *"display in standard output"*
  - usage: **cat [ARG]**
  - ex: `user@localhost ]$ cat MyFile`

- **tac** *"opposite of cat"*
  - usage: **tac [ARG]**
  - ex: `user@localhost ]$ tac MyFile`

- **more** *"display in standard output screen by screen"*
  - usage: **more [ARG]**
  - ex: `user@localhost ]$ more MyFile`

- **less** *"opposite of more"*
  - usage: **less [ARG]**
  - ex: `user@localhost ]$ less MyFile`

- **clear** *"clean the terminal"*
  - usage: **clear**
  - ex: `user@localhost ]$ clear`

- **echo** *"display something"*
  - usage: **echo [ARG]**
  - ex: `user@localhost ]$ echo`

- **cut** *"cut, extract columns"*
  - usage: **cut [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ cut -c5-10 MyFile`
  - options: `-c` (character numbers: `-cA-B,C-D,E-F...`

- **tail** *"show me the tail"*
  - usage: **tail [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ tail -f MyFile`
  - options: `-f` (last ten lines with update), `-lines=n` (last n lines)

- **wc** *"words and lines count"*
  - usage: **wc [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ wc -l MyFile`
  - options: `-l` (number of lines)

- **ln** *"create a link"*
  - usage: **ln [OPTION] ... [ARG1] [ARG2]**
  - ex: `user@localhost ]$ ln -s MyFile MyNewFile`
  - options: `-s` (create symbolic link)

- **chmod** *"change permissions"*
  - usage: **chmod [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ chmod 755 MyFile`
  - options: `xyz` with `x`, `y` and `z` between 0 and 7, `-R` (recursive)

- **chown** *"change owner"*
  - usage: **chown [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ chown newuser.newgroup MyFile`
  - options: `-R` (recursive)

- **ps** *"process information"*
  - usage: **ps [OPTION] ...**
  - ex: `user@localhost ]$ ps -auwx`
  - options: `-ax` (all processes), `-u` (user oriented format), `-w` (wide output)
- **top** *"system usage"*
  - usage: **top [OPTION] ...**
  - ex: `user@localhost ]$ top`
- **kill** *"terminate process"*
  - usage: **kill [OPTION] ... [ARG] ...**
  - ex: `user@localhost ]$ kill -9 PID`
  - options: `-9` (send kill signal)
- **su** *"switch user"*
  - usage: **su [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ su - user`
  - options: `-` (use shell login)
- **sudo** *"run command as root"*
  - usage: **sudo [OPTION] ... [CMD] [OTION] ... [ARG]**
  - ex: `user@localhost ]$ sudo ls -l MyDirectory`
- **env** *"know you environment"*
  - usage: **env**
  - ex: `user@localhost ]$ env`
- **diff** *"difference between two files"*
  - usage: **diff [OPTION] ... [ARG1] [ARG2]**
  - ex: `user@localhost ]$ diff MyOldFile MyNewFile`
- **sleep** *"wait some time please"*
  - usage: **sleep [TIME]**
  - ex: `user@localhost ]$ sleep 3600`
- **which** *"check `$PATH` to know which binary is used"*
  - usage: **which [CMD]**
  - ex: `user@localhost ]$ which cpmd`
- **whereis** *"find out the location of binary file(s), library file(s) and manual page(s)"*
  - usage: **whereis [ARG]**
  - ex: `user@localhost ]$ whereis gimp`
- **tar** *"create archive"*
  - usage: **tar [OPTION] ... [ARG] ...**
  - ex: `user@localhost ]$ tar -jcf MyFile.tar.bz2 MyFile`
  - options: `-c` (create archive), `-x` (extract archive), `-f` (filename), `-j` (bzip2), `-z` (gzip), `-t` (list file in archive)
- **find** *"look for a file"*
  - usage: **find [PLACE] [OPTION] ... [ARG]**
  - ex: `user@localhost ]$ find /home -name "MyFile`
  - options: `-name` (name is)
- **passwd** *"change the password"*
  - usage: **passwd [OPTION] ... [USERNAME]**
  - ex: `user@localhost ]$ passwd`

3.2 Redirection commands

- **&** *"send job to the background"*
  - usage: **[CMD] [OPTION] ... [ARG] &**
  - ex: `user@localhost ]$ gedit &`

- **Ctrl-z + bg** *"send foreground job to the background"*
  - usage: **Ctrl-z** in terminal, followed by: **user@localhost]$ bg**
  - ex: `user@localhost ]$ bg`

- **>** *"redirect standard output in file (erase existing)"*
  - usage: **[CMD] [OPTION] ... [ARG] > MyFile**
  - ex: `user@localhost ]$ ls -l > MyFile`

- **>>** *"redirect standard output and append at the end of file"*
  - usage: **[CMD] [OPTION] ... [ARG] >> MyFile**
  - ex: `user@localhost ]$ ls -l >> MyFile`

- **>& (or &>)** *"redirect standard output and standard error in file (erase existing)"*
  - usage: **[CMD] [OPTION] ... [ARG] >& MyFile**
  - ex: `user@localhost ]$ ls -l >& MyFile`

- **&>>** *"redirect standard output and standard error and append at the end of file"*
  - usage: **[CMD] [OPTION] ... [ARG] &>> MyFile**
  - ex: `user@localhost ]$ ls -l &>> MyFile`

- **| (called pipe or pipeline)** *"redirect in an other command"*
  - usage: **[CMD] [OPTION]...[ARG] | [CMD] [OPTION]...[ARG]**
  - ex: `user@localhost ]$ ls -l | more`
  - ex: `user@localhost ]$ ls -l | grep MyFile`
  - ex: `user@localhost ]$ | awk 'printf $NF"\n"'`

- **tee** *"read from standard output and redirect to standard output and file (erase existing)"*
  Special invocation from a pipe |
  - usage: **[CMD] [OPTION]...[ARG] | tee [OPTION]...[ARG]**
  - ex: `user@localhost ]$ ls -l | tee MyFile`
  - ex: `user@localhost ]$ cpmd file.inp | tee file.out`

3.3 Bash commands

- **alias** *"create command alias"*
  - usage: **alias CMD='[CMD] [OPTION] ... [ARG]'**
  - ex: `user@localhost ]$ alias ll='ls -lth'`

- **ENVIRONMENT_VARIABLE=** *"create environment variable"*
  - usage: **ENVIRONMENT_VARIABLE= [VALUE]**
  - ex: `user@localhost ]$ MYVAR= 100`

- **export ENVIRONMENT_VARIABLE=** *"create inheritable environment variable"*
  - usage: **export ENVIRONMENT_VARIABLE= [VALUE]**
  - ex: `user@localhost ]$ export MYVAR= 100`

3.4 Filter commands

- **awk** *"pattern scanning and processing language"*
  - usage: **awk [OPTION] ... [REGULAR EXPRESSION] [ARG]**
  - ex: `user@localhost ]$ awk '{printf $NF\n}' MyFile`

- **sed** *"stream editor for filtering and transforming text"*
  - usage: **sed [OPTION] ... [REGULAR EXPRESSION] [ARG]**
  - ex: `user@localhost ]$ sed 's;MYPATH;$HOME;g'`

- **grep** *"print lines matching a pattern"*
  - usage: **grep [OPTION] ... [REGULAR EXPRESSION] [ARG]**
  - ex: `user@localhost ]$ grep "(K+E1+L+N+X) TOTAL ENERGY =" cpmd.out`
  - options: `-n` (print line number), `-A NUM` (print `NUM` lines after pattern), `-B NUM` (print `NUM` lines before pattern), `-v` (invert correspondence = non-matching lines)

Bibliography

[1] http://www.gnu.org/software/bash/.

[2] http://tldp.org/LDP/abs/html/.

This document has been prepared using the Linux operating system and free softwares:
The text editor "gVim"
And the document preparation system "LaTeX 2ε".