Basic guide to the command line

Outline

Introduction

To utilize many of the programs in NMRbox you will need to at least have a rudimentary understanding of how to navigate around the command line. With some practice you may even find that the command line is easier than using a GUI based file browser – it is certainly much more powerful.

In addition to navigating from the command line you will also need a rudimentary understanding of how to create and execute a shell script.

This Guide may take a little bit to get through, but if you know most of the contents in this document you will be well served.

(top)

Definitions

Terminal Emulator – A terminal emulator is a program that makes your computer act like an old fashion computer terminal. When running inside a graphical environment it is often called a "terminal window", "terminal", "term", or "shell" (although it is actually not a shell – read on).

- I will refer to a "Terminal Emulator" as "Terminal" in this document.

Shell – A shell is a program that acts as a command language interpreter that executes the commands sent to it from the keyboard or a file. It is not part of the operating system kernel, but acts to pass commands to the kernel and receives the output from those commands.

- When you run a Terminal Emulator from a graphical interface a shell is generally run from within the Terminal so your interaction is with the shell and the Terminal simply presents a place for the shell to run in the graphical interface.

Script (shell script) – When commands are written into a text file to be sent to the shell it is called a script.

- Scripts are very useful inside NMRbox because they give a written record of how you processed or analyzed your data.

BASH (Bourne again shell) – The most common shell in use today and the one primarily used in NMRbox. This document is written with the assumption that you are using bash version 4. Bash is both a command line interpreter and a programming language which makes it powerful as a scripting language.

CSH (C shell) – Another shell that was much more common in the past. The C shell and its related shell, tcsh, are installed in NMRbox, however, we encourage everyone to stay with the bash shell as their default shell and all the configurations inside NMRbox are designed for the bash shell.

- NMRPipe and CNS are two programs that are typically run in a C-shell. However, they will generally run fine in bash. In addition, if the first line of your NMRPipe or CNS script is "#! /bin/csh" than the script will run as a C-shell script even though it is launched from a bash shell.

Prompt (shell prompt) – Often called the command line is where one types commands in the shell. When a program is run from the shell prompt the prompt will be inaccessible until the program ends unless the program is run in the background.

- In this document examples are written in **`Courier New bold`** and prompts are shown as "**`$>`** "

(top)

Special Characters

There are many special characters in Bash and it is very important to have a strong understanding of how they work to be proficient at using Bash. In the table below column 1 is the character, column 2 is the description and example, and column 3 is whether the special character is typically used from the terminal (T), inside scripts (S), or both (T/S).

`;`	Separates commands entered on a single line.	T/S
	`$> ls; pwd; echo 'hello'`	
`&`	When placed at the end of a command will run the command in the background and returns prompt immediately.	T
	`$> ./process.com &` **`$>`**	
`|`	Called a "pipe" takes the output from one command and "pipes" it into the input for the next command	T/S
	**`$> cat file.txt	wc –l`**	
`||`	Runs the next command only if the first command fails. Nice to add to scripts when you want the script to exit if a command fails	S
	`$> cd nmrdata || echo 'change dir failed'; exit`	
`#`	Comment. Everything after # is ignored.	S
`\`	Indicates that the command is continued on the next line. Nice when writing scripts with very long lines to make them more readable	S
	`nmrPipe –in test.in \` **`| nmrPipe –fn FT \`** **`| nmrPipe –out test.out`** (in this example the whole things is considered a single line to the shell)	

~	Called a tilde and is a shortcut to your home folder.	T/S
	$> cd ~/nmrdata	
..	When used in a path, two dots means the parent of the current working directory.	T/S
	$> cd ../../nmrdata	
.	When used in a path a dot represents the current working directory	T/S
	./process.com [Runs process.com in the present working directory]	
ctrl-c	Stops a stuck program	T
ctrl-d	Marks the end of input when typing to a program. Acts to exit certain programs.	T
ctrl-z	Suspends a program that is currently running. Is often followed by "bg" which restarts the program in the background returning the prompt immediately.	T

(top)

Filenames

Filenames and directory names (I will just refer to both as filenames) in NMRbox are case sensitive – for the most part. The file systems that NMRbox utilize are designed to work with Linux, Windows, OSX, and other operating systems and therefore issues can arise sometimes when dealing with files that have the same name but different case (capitalization).

Filenames that start with a dot (e.g. .mozilla, .bashrc) are hidden files and are not shown when listing files unless you specify them to be visible. They typically contain configuration information.

Do's & Don'ts of Filenames

- Avoid spaces and tabs
- Use capitalization as you like, but avoid filenames that are identical except for case (capitalization) in the same directory.
- Never use: **~ ` ! @ # $ ^ & * ( ) + = ' " \ | ? / > < { } ; :**

(top)

Pathnames and the Path Variable (Search Path)

Present working directory (pwd) – The current location in the path

Root directory – In Linux the root directory is the highest-level directory and is located at "/".

- Example: **$> cd /** [would change to the root directory]

Relative path – Paths that are described relative to the present working directory

- Example: **$> cd ../nmrdata/hsqc.fid** [would move up one directory and then into the nmrdata/hsqc.fid directory relative to the present working directory]

Explicit path – Paths that are described explicitly starting from the root directory "/". Explict paths always start with a slash "/".

- Example: **`$> cd /usr/software/bin`**

Home path – Home paths for NMRbox accounts are in /home/nmrbox/username and a tilde "~" is a shortcut to your home path.

- Example: **`$> cd /home/nmrbox/username`** and **`cd ~`** are equivalent.

Path variable – The path variable defines the location on the filesystem that is searched when you enter a command. For instance, if you type "**`$> grep`**" the system will search and find it in "/usr/bin" because "/usr/bin" is in the path variable.

- In NMRbox version 2 the ~/bin folder is in the search path. Therefore, if you create your own scripts that you want in the path you can simply create a folder called bin in your home folder and place your scripts in the bin folder.
- To view the search path use either:
  - $> echo $PATH
  - $> env | grep PATH
- To change the search path for your account add a line such as this to the end of the file ".profile" in your home folder:
  - **`$> cd ~; gedit .profile`** [then add the following line to the end of the file and save]
  - export PATH=$PATH:/home/nmrbox/username/new_path_to_add
  - **Notes**:
    - You can see if ".profile" exists by typing "**`$> cd ~; ls -la`**"
      - " If ".profile" does not exist you can create it with "**`$> touch .profile`**"
    - When you open a new terminal the new path should exist
    - The "$PATH" keeps the existing search path and the ":" separates the new path to add
- Placing the current working directory in the path.
  - When you create scripts to process or analyze data, such as nmrpipe.com, you might try to run the script by typing "$> nmrpipe.com" and get a "nmrpipe.com: command not found" error. This is because the current working directory is NOT in your path and the therefore the command nmrpipe.com is not found.
  - To resolve this issue always run your scripts with a "**`./`**" in front of them. As you recall the period represents the current working directory. Therefore "**`$> ./nmrpipe.com`**" says to run the nmrpipe.com script in the current working directory.
  - You may be tempted to add "." to the path. However, we strongly discourage this practice for several reasons:
    - It is a known security issue. Mischievous people may try to add a piece of malware and name it the same as a common command such as "ls". If you accidently download such a file and run the "ls" command from that location you will be executing the malicious software, which may even run the proper command so you don't even realize there is a problem.
    - It is common to accidently create files with names such as grep, ls, find, etc. If you do then you will be confused when you try to run these common commands from a directory where a filename with the same name exists. For example, if you create a file called "find" in a directory and you have the current working directory in your path then you will get a "command not found" error when trying to run "find" from that directory.
    - Lastly, it is good practice to run scripts with the "./" as it forces the correct script to execute and not some other script with the same name that is in your path.

(top)

Wildcards

Wildcards are characters that can be used to substitute for any other character in a search.

- **?** Matches any single character
- ***** Matches any number of characters
- **[ ]** Matches any characters inside the range
- **{}** Matches anything in the brackets separated by commas
- **[!]** Similar to **[]** except matches as long as it is not in the bracket

For some examples imagine a directory with the following filenames:

data1a, data1b, data2, data2a, data2c, data3, data3b, input1, input2, input3

Example command	Files that are found
$> ls data*	**data1a, data1b, data2, data2a, data2c, data3, data3b**
$> ls *3	**data3 and input3 (not data3b)**
$> ls *3*	**data3, data3b, and input3**
$> ls data?a	**data1a and data2a**
$> ls data[12]*	**data1a, data1b, data2, data2a, and data2c**
$> ls input[1-3]	**input1, input2, and input3**
$> ls input[1,3]	**input1 and input3**
$> ls in{put1,put2}	**input1 and input2**
$> ls input[!3]	**input1 and input2**

Of course, wildcards can be used with all commands, not just **ls** as in these examples.

(top)

Standard input, Standard output, Standard error, and Redirection

It is often the case the input to a program is easier to put into a text file rather than typing from a command line. It is also advantageous to capture screen output to files in order to have a log of what occurred and to more easily look back for errors. It is important to have a strong understanding of standard in, standard out, standard error, and redirection to accomplish this.

Standard input (*stdin*) – Reads input for a command from the console. For example:

```
$> cat filename
```

uses the command "**cat**" to show the contents of "**filename**". In this example the ***stdin*** comprises both the command "**cat**" and the argument to cat "**filename**". STDIN can also come from a redirection – see below.

Standard output (*stdout*) – Produces output from your command to the terminal (screen). STDOUT can also be redirected to a file – see below.

Standard error (*stderr*) – Produces "error" output from your command to the terminal (screen). STDERR can also be redirected to a file – see below.

STDIN Redirection – With redirection the ***stdin*** does not come from the keyboard. Instead the ***stdin*** comes from a file. As an example:

```
$> xplor < xplor_parameters.xpl
```

In this example the text file "**xplor_parameters.xpl**" contains the input for the command "**xplor**". Additionally ***stdin*** redirection can come from another command with the use of a pipe:

```
$> ps | grep username
```

In this example the output from the command "**ps**" is used as the ***stdin*** for the "**grep username**" command by use of the pipe "**|**".

STDOUT Redirection – With redirection the ***stdout*** is sent to a file or other device. The file will be created if it does not exist. If the file does not exist, or should be overwritten, use a single greater than sign:

```
$> ls > files.txt
```

If the file should be appended use two greater than signs:

```
$> ls –l >> files.txt
```

STDERR Redirection – With redirection the ***stderr*** is sent to a file or other device. The file will be created if it does not exist. If the file does not exist, or should be overwritten, use "**2>**". For example:

```
$> ls asdf 2> errors.txt
```

If the file should be appended use "**2>>**". For example:

```
$> ls asdf 2>> errors.txt
```

STDOUT and STDERR Redirection to separate files – In Bash the ***stdout*** and ***stderr*** can both be redirected to separate files (**$> command > log.txt 2> errors.txt**) or with appending (**$> command >> log.txt 2>> errors.txt**).

STDOUT and STDERR Redirection to the same file – In Bash the ***stdout*** and ***sterr*** can both be redirected into a single file as such:

```
$> command > log.txt 2>&1
```

or with appending:

```
$> command >> log.txt 2>&1
```

What is happening here is the ***stdout*** from the command is being redirected to the file **log.txt**. In addition the **2>&1** tells the ***stderr*** to be redirected to ***stdout*** so the ***stderr*** ends up in the **log.txt** file along with the ***stdout***.

Note – In Bash 4

```
$> command >> log.txt 2>&1
```

can be written in a shortcut method as

```
$> command &>> log.txt
```

I would suggest sticking with the long form as it is guaranteed to work in other shells you may use, such as C-shell scripts for NMRPipe, but I mention it here in case you are confused by other examples where they use the shorthand notation.

(top)

Here document (heredoc)

A heredoc is a section of a script that is treated as if it were a separate file. It is similar to redirection of ***stdin*** from a file, but rather than a separate file it is simply embedded in the script itself. Heredoc's are used with several programs installed in NMRbox, such as rnmrtk scripts.

The best way to describe how heredocs work is by example. In this example we utilize the NMR data processing program rnmrtk. When you enter rnmrtk from a command prompt the program starts and the prompt changes to a rnmrtk prompt for the user to enter commands. This is very inconvenient for day to day processing as it is error prone and leaves no written record of how your data was processed. In this case, rather than entering commands manually at the command prompt we can use a heredoc.

```
rnmrtk << EOF
loadvnmr ./fid
seepar
EOF
rnmrtk << EOF
sstdc
fft
phase 47.4 0.0
realpart
save spectrum.sec
EOF
```

In this example the command "**rnmrtk**" is started with a "**<< EOF**" afterwards. The "**<< EOF**" says pass the next lines as arguments to "**rnmrtk**" one line at a time until another "**EOF**" is encountered which then exists the "**rnmrtk**" program. In this example another "**rnmrtk << EOF**" line is used showing that multiple heredocs can be used together in the same script.

A few notes:

- While **EOF** is a common delimiting identifier you can use any identifier you like as long as the starting and ending identifier match.
- In a heredoc **don't** use any spaces or tabs at the beginning of the lines between the identifiers. The script below will fail due to the indentation spaces in front of the loadvnmr and seepar commands.

```
rnmrtk << EOF
   loadvnmr ./fid
   seepar
EOF
```

(top)

Owners, Groups, and Permissions

The file systems used in NMRbox utilize a standard Linux model where each filename has an owner, a group, and three groups of three permissions for privileges for the owner, group, and other.

The best way to explain owners, groups, and permissions is by example from the "**ls -l**" command. Here is the ***stdout*** from the command "**ls -l**" from a NMR experimental directory.

```
$> ls -l
total 5352
```

```
drwxr-xr-x 3 mark mark-group    1024 Aug 13  2016 Data
-rw-r--r-- 1 mark mark-group     487 Jun 11  2014 f2_proc.par
-rw-r--r-- 1 mark mark-group  393216 Jun 11  2014 f2_proc.sec
-rw-r--r-- 1 mark mark-group  395936 Oct  9  2009 fid
-rw-r--r-- 1 mark mark-group 2099200 Jun 11  2014 final.ft2
-rw-r--r-- 1 mark mark-group     488 Jun 11  2014 final.par
-rw-r--r-- 1 mark mark-group 2097152 Jun 11  2014 final.sec
-rw-r--r-- 1 mark mark-group      92 Oct  9  2009 log
-rwxr--r-- 1 mark mark-group     377 Jan 24  2010 proc.com
-rwxr--r-- 1 mark mark-group    2049 Oct  9  2009 process.com
-rw-r--r-- 1 mark mark-group   29630 Oct  9  2009 procpar
-rw-r--r-- 1 mark mark-group    1307 Oct  9  2009 procpar.txt
-rw-r--r-- 1 mark mark-group       5 Oct  9  2009 text
-rw-r--r-- 1 mark mark-group     487 Jun 11  2014 time.par
-rw-r--r-- 1 mark mark-group  393216 Jun 11  2014 time.sec
```

Information for the directory **Data** from the example above

Column	Element	Description
1	**drwxr-xr-x**	Permissions; r=read, w=write, x=execute. The first character is typically – for files, d for directories, and l for symbolic links.
2	**3**	Links to the file or directory. For directories there is a link for the current directory and the directory itself, for a minimum of two. In this case the value is 3 meaning that there must be one subdirectory inside Data.
3	**mark**	owner
4	**mark-group**	group
5	**1024**	Size (bytes)
6	**Aug**	Modification time (Month)
7	**13**	Modification time (Day)
8	**2016**	Modification time (Year)
9	**Data**	Filename

The permissions are broken into 4 groups **(-)(---)(---)(---)** with a single character in the first group and then there are three groups of three. In the example above for the filename Data **(d)(rwx)(r-x)(r-x)**

Group 1: Is generally a "**-**" when the filename is a file, "**d**" when the filename is a directory, or a "**l**" if the filename is a symbolic link to another filename.

Group 2: Read, Write, Execute permissions for the owner. In this example above the owner has the ability to read and write to all the filenames and for the directory, Data, process.com, and proc.com the owner has the ability to execute the filename.

Note: Directories must be executable in order to change into them.

Group 3: Read, Write, Execute permissions for the group. In this example any user who belongs to the mark-group will have either **(r--)** or **(r-x)** permissions, which means they can read the file and in the case of the directory, Data, they can change into the directory.

Group 4: Read, Write, Execute permissions for other. In this example any user who is not the owner or is not in the group mark-group will have either **(r--)** or **(r-x)** permissions, which means they can read the file and in the case of the directory, Data, they can change into the directory.

Note: See the commands below chown, chgrp, chmod for information on changing the owner, group, and permissions of files.

(top)

Commands

Below is an alphabetical list of common commands in Bash and other shells along with their descriptions and arguments. The "**$>**" represents the prompt where command lines would be entered. The prompt, commands, and arguments are in **Courier bold font**.

Arguments in **[brackets]** are optional. There are often many more arguments than listed in this document. Use "**$> man command**" for more detailed information about any command.

Many of the command have a filename as required input, but the filename can often be omitted when the output from one command is piped "**|**" into the command. For example:

```
$> wc –l file.txt
$> cat file.txt | wc –l
```

both do the same thing. In the first case the command **wc** is given a filename directly and in the second case the contents for the **wc** command come from the ***stdout*** of the **cat** command.

alias

Creates an alias to a command or when run without arguments lists the current aliases. Can be helpful when you repeatedly need to run a longer command.

```
alias [name[='command']]
   $> alias
   $> alias ls
   $> alias u='touch proc.com; chmod u+x proc.com'
```

Notes

- Aliases can be placed in your .bashrc file so they are persistent every time you login.

awk

awk is a programming language for processing text and is beyond the scope of this document.

cal

Prints the calendar for the current month or any month or year specified

```
cal [month year][year]
   $> cal
   $> cal march 2017
   $> cal 2016
```

cat

Displays a file to the screen

```
cat filename
   $> cat proc.com
   $> cat file1 file2 file3 > combined
```

cd

Changes to a new directory. Takes you to your home folder if run with no arguments.

cd [*directory*]

```
$> cd hsqc.fid
$> cd /usr/software/bin
$> cd ~/NMRdata
$> cd
```

chgrp

Changes the group ownership of a file

chgrp [-R] *newgroup filenames*

-R	Recursively changes the group for files in subdirectories

```
$> chgrp mark-group proc.com
$> chgrp –R mark-group hsqc.fid
$> chgrp mark-group proc.com hsqc.ft2
```

chmod

Changes the permissions for a file

chmod [-R] *permissions filenames*

-R	Recursively changes the permissions of files in subdirectories too
Who gets the permission setting	
u	User who owns the file
g	Group that owns the file
o	Other. Everyone that is not covered by u and g
a	All
Permissions being granted (+), removed (-), or set equal to (=)	
+r, -r, =r	Read the file
+w, -w, =w	Write, edit, or delete the file
+x, -x, =x	Execute or run the file
+X, -X, =X	The uppercase X works on two rules: 1. If the file is a directory it will add execute permission 2. If the file already has execute permission it will be added for the whoever is specified. This is almost always used in conjunction with -R when changing the permissions on all files inside a directory and subdirectories.

```
$> chmod u+x *.com
$> chmod –R a+X NMRdata
$> chmod –R u=rwX,g=rX,o=rX Data
$> chmod –R g+rX Data
```

Notes

- When using = to set permission all existing permissions will be removed and only the permissions being set with the = will be set. This is unlike + and - where permissions are left and only added or subtracted as defined in the arguments.
- Your home folder has default permissions of **drwx------**. We strongly advise to leave those permissions to protect your data from other users. Even if a file has group or other read, write, or execute permissions inside your home folder, if the home folder itself is **drwx------** no other user can get into your directory in the first place.

chown

Changes the owner of a file

chown [-R] *newowner filenames*

-R	Recursively changes the ownership for files in subdirectories

```
$> chown mark proc.com
$> chown –R mark hsqc.fid
$> chown mark proc.com hsqc.ft2
```

clear

Clears the terminal window screen

```
clear
   $> clear
```

cp

Copies one or more files or directories and their contents.

```
cp [-i] original-file new-file
cp [-i][-r] original-directory new-directory
```

-i	Interactive mode will ask before overwriting any existing files.
-r	When you copy a directory the –r argument will copy all the files inside the directory including the subdirectories and their contents.

```
$> cp –i proc.com process.com
$> cp –r hsqc.fid NMRdata
$> cp proc.com hsqc.fid/process.com
$> cp xplor.xpl ~/Projects/Cofilin/xplor
$> cp ../hsqc.fid/process.com .
```

csh

Change the shell type to the C-shell

```
csh
   $> csh
```

Notes:

- There are some programs in NMRbox that are generally run from the C-shell. If you enter the command csh your will switch to the C-shell and all the environmental variables that are set in Bash will be migrated to the C-shell.
- Typically this should not be needed, as programs that rely on the C-shell will more likely be run as a script. To run a script as the C-shell enter "#! /bin/csh" as the first line of your script.

date

Shows the current data and time

```
date
   $> date
```

df

Displays how much space is free on the filesystems.

```
df [-h][directory]
```

-h	Makes the output human readable by converting bytes to KB, MG, GB, TB, PB as appropriate.
directory	If a directory is specified it will show only the filesystem where the directory exists, otherwise all filesystems will be shown.

```
$> df
$> df –h
$> df –h ~
```

diff

Shows the differences between two files or the differences in what files exist between two directories.

```
diff [-i][-b][-B][-w] file1 file2
diff directory1 directory2
```

-i	Ignore case
-b	Ignore changes in the amount of white space
-B	Ignore changes whose lines are all blank
-w	Ignore all white space

```
$> diff proc.com process.com
$> diff –w process.com ../hsqc.fid/process.com
$> diff Data /nmr/archive/mark/Backup-Data
```

du

Shows the differences between two files or the differences in what files exist between two directories.

```
du [-s][-a][-h][-S] directories
```

-s	Summarize results into a single size
-a	Show the size of every file and not just the directories
-h	Make the results human readable by presenting the results as KB, MB, GB, TB as appropriate.
-S	Show the results for each directory individually rather than summing the size of subdirectories as well.

```
$> du –sh Data
$> du –sh ~/Data/cofilin ~/Data/ubiquitin
$> du –ah NMRData
```

echo

Echoes back whatever is typed to the screen. If the text is in single quotes the result is exactly as typed. If the text is in double quotes then variables are expanded in the output. Echo statements are often very helpful in shell scripts.

```
echo 'text'
echo "text"
   $> echo 'The cost is $5.55'
   $> echo "The current path is $PATH"
```

env

Shows information about the currently set environmental variables

```
env
   $> env
   $> env | grep Home
```

exit

Exits the process. If typed from a command line it will exit the shell and close the terminal. In a shell script will exit the script.

```
exit
   $> exit
```

export

Sets a variable for the current shell and any sub-process spawned from that shell

```
export name=value
   $> export PATH=$PATH:/home/nmrbox/markm/bin
```

Notes:

- If you want to set a variable to always be set for all shells you can add the export command to your .profile file.

file

Shows whether a filename is a file, directory, link, or something else. If it is a file it attempts to guess the file type. Can be usefult to know if a program is 32 or 64 bit and whether it has dynamically linked libraries.

file *filenames*

```
$> file process.com
$> file /usr/software/rnmrtk/rnmrtk
```

find

Finds one or more files from the directories that you specify and either prints the output to the screen or performs a command on each of them.

find *directories* [-name *filename*][-user *username*][-group *groupname*][-print][-exec *command* {} \;][-ok *command* {} \;]

-name	Will find files in the specified directories with the filename or part of the filename you are searching if wildcards are added. If wildcards are used the filename must be in quotes. It is just best to always use quotes.
-user	Will find files in the specified directories owned by the specified username
-group	Will find files in the specified directories with the group as specified by groupname
-print	Will print the result to the screen. This is now the default.
-exec	Will execute the command on each file as they are found. Note that the syntax must be followed exactly as shown above.
-ok	Identical to –exec except the user is prompted if the command should be executed as the files are found.

```
$> find . -name 'process.com' -print
$> find Ubiquitin Cofilin -name 'hsqc.ft2' -print
$> find . -name '*.com' -exec chmod u+x {} \;
$> find . -name '*.bak' -ok rm {} \;
```

grep

Searches the contents of files in the present working directory for a word or phrase and displays the filename and line where a match is found.

grep [-i][-l][-L][-A *num*][-B *num*][-r][-s] *text filenames*

-i	Ignore case
-l	Only the filename where a match is found is sent to stdout
-L	Only print filenames where a match is NOT found.
-A num	Print the line where the match is found and ***num*** number of lines **A**fter the match
-B num	Print the line where the match is found and ***num*** number of lines **B**efore the match.
-r	Search the current directory and recursively into the subdirectories as well.
-s	Suppress messages about directories and unreadable files.

```
$> grep -rs rnmrtk *.com
$> grep -rl '#! /bin/bash' *.sh
$> grep tof -A2 procpar
$> ps -ef | grep mark
```

head

Displays the first few lines in a file.

head [-*lines*] *filename*

-lines	Specify the number of lines to show

```
$> head procpar
```

```
$> head -25 procpar
```

help

Displays on-line help for the shell.

help [*command*]

command	Get more detailed information about a specific command

```
$> help procpar
$> help hash
```

history

List a history of the latest commands that were run proceeded by an ordered numbered list.

history

!!	Runs the last command again
!n	Where **n** is the number of line from the **history** command.

```
$> history
$> !!
$> !32
```

Notes:

- It is common to alias history to h by placing the following line at the end of your ~/.bashrc file

```
alias h='history'
```

hostname

Displays the name of the current host system

hostname

```
$> hostname
```

id

Shows you what your user ID (uid) and group ID (gid) are. Also shows what groups you belong to.

id

```
$> id
```

kill

Kills a job that you don't want to continue or if frozen and unresponsive.

kill [-9] *pid*

-9	Show no mercy when killing the program. Used when kill by itself does not kill the process. Must be used with **ps -ef** to determine the pid.

Example. Let's say ***nmrDraw*** is unresponsive and you want to kill it. Run $> **ps -ef** to find the ***pid*** and then **kill** to kill ***nmrDraw***.

```
$> ps -ef | grep nmrDraw
mark 13032  4647  /bin/csh -f /usr/software/nmrpipe/nmrbin.linux212_64/nmrDraw
mark 13166 15746   grep --color=auto nmrDraw
```

Notes:

- The output from the **ps -ef | grep nmrDraw** shows the ***pid (13032)*** for ***nmrDraw*** and a ***pid (13166)*** for the **grep nmrDraw** command itself. In this case the ***pid*** that we would want to kill is ***13032***

```
$> kill 13032
```

ln

Creates a link to a file or directory allowing a single file to have more than one name or reside in multiple locations.

ln [-s] *filename linkname*

-s	Makes the link a symbolic link which is the most common type of link

```
$> ln –s /usr/software/rnmrtk/section section
$> ln –s nmrPipe-version1.2345_64bit_linux.exe nmrPipe
```

locate

Locates files by looking up the files in a database. Unfortunately for NMRbox your files are stored on remote file systems so locate is only good for finding system files.

locate [-i][-A][-c] *patterns*

-i	Ignore case
-A	Match all patterns together and not just a single pattern
-c	Report the number of files found and not the pattern match itself

```
$> locate nmrPipe
$> locate –i nmrpipe
$> locate –Ac nmrtxt nmrPipe
```

lpr

Prints a file to the default printer

lpr *filename*

```
$> lpr process.com
$> ls | lpr
$> lpr document.ps
```

Note: Depending on your printer it may interpret ASCII text files and postscript files and print them properly. However, if you print a postscript file or a binary file to a printer that does not know how to interpret the file you may print the code of the file which can be gibberish and very large.

ls

Lists the files in a directory

ls [-a][-l][h][-r][-R][-t][-S][*pathname*]

-a	Displays all files including hidden files
-l	Displays detailed information including owner, group, size, permissions, and when the file was last modified.
-h	When used with -l reports the size in a human readable manner
-r	Displays files in reverse order
-R	Displays files in subdirectories too
-t	Displays files in order of modification data
-S	Sorts by files size with largest first
pathname	Displays files in the pathname rather than the present working directory

```
$> ls
$> ls –la
$> ls –ltr
$> ls -lhSr
$> ls /usr/software/bin
$> ls hsqc.fid/*.ft2
$> ls –R project_directory
```

man

Displays an internal manual for commands and programs

```
man command
   $> man find
   $> man nmrPipe
```

mkdir

Create a new directory

```
mkdir [-p] directory
```

-p	Creates parent directories if needed. Also, will skip the error message if you try to create a directory that already exists – it will simply not be created.

```
   $> mkdir Data
   $> mkdir –p Data
   $> mkdir –p NMRdata/cofilin/Experiments
   $> mkdir ../Results
```

more

Displays information on the screen one page at a time so you can read it

```
more filename
```

After typing more filename use these keys to navigate	
Key	***Definition***
h	Displays a comprehensive list of ***Keys*** and what they do
Space	Scrolls to show the next page of text
Enter	Scrolls down a single line
/*pattern*	Will search for the ***pattern*** in the file
q	Quits the display

```
   $> more procpar
   $> cat logfile | more
```

mv

Move a file to a new location or rename the file if in the same directory

```
mv [-i] oldfilename newfilename
```

-i	Prompts the user before overwriting an existing file

```
   $> mv proc.com process.com
   $> mv process.com ../hncacb.fid
   $> mv -i hsqc.ft2 NMRdata/cofilin/spectra
```

Note: mv acts on directories in the same way that it acts on files.

passwd

Changes your password

```
passwd
   $> passwd
```

Notes:

- After typing **passwd** you will be prompted to enter your current password and then your new password twice. Be careful to be sure that your password changed – it will not change if the new password does not meet the password complexity rules or you are changing to a password that you used recently.

ps

Displays information on currently running processes such as programs.

```
ps [-e][-f][-F][-l][-y]
```

-e	Displays all processes on the system. Typically should **ALWAYS** be used.

-f	Shows a full-format listing. This is typically used along with **-e**
-F	Extra full format
-l	Displays long format
-y	Cleans up the formatting when using **-l**

```
$> ps -e
$> ps -ef
$> ps -eF
$> ps -ely
```

Notes:

- For most users the four examples shown here are the typical way the **ps** command is used.
- The **ps** command is often used in conjunction with **grep** to locate processes that need to be killed. For example, lets say you need to kill your VNC server session and your username is ***mark***. This command can be used to find the ***pid*** for the VNC server session "**ps -ef | grep mark | grep Xvnc**"

pwd

Displays the current working directory.

```
pwd
   $> pwd
```

rm

Deletes a file or directory **permanently.**

```
rm [-i][-I][-f][-v][-r]
```

-i	Asks the user to confirm the deletion
-I	Prompts the users once when deleting more than 3 files. Less intrusive than **-i**, which will prompt for each file individually, but still, gives the user some safety when deleting files.
-f	Forces the delete by ignoring errors and turning off ALL prompts even if **-i** or **-I** were used. ***BE CAREFUL – generally only use –f in special cases.***
-v	Verbose. Shows what is being done.
-r	Recursively deletes the contents of a directory and **ALL** its subdirectories. ***BE CAREFUL – if used incorrectly a large number of files can be deleted.***

```
$> rm log.txt
$> rm -I *.ft3
$> rm -ivr FT
$> find . -name '*.bak' exec rm -i {} \;
```

script

Records a shell session complete with commands and output**.**

```
script [-a filename]
```

-a *filename*	Defines the filename where the output is stored. If the file already exists the results are appended. If the **-a** filename argument is not entered the filename "***typescript***" is used by default.

How to use: Type script **-a** filename and hit ***Enter***. Then perform whatever shell commands you want to run. When you are finished and want to exit capturing your commands and their output type **exit**.

```
$> script -a log.txt
$> ls
$> du -sh
$> exit
```

sdiff

Compares two files by showing them in a side-by-side fashion.

rm [-s][-w][-i][-b][-B][-W]

-s	Suppress lines that are the same
-w *size*	Set the size of your screen.
-i	Ignore case
-b	Ignore changes in the amount of whitespace
-B	Ignore blank lines
-W	Ignore all white space

```
$> sdiff -W proc.com process.com
$> sdiff -i rnmrtk.sh ../hsqc.fid/rnmrtk.sh
$> sdiff -w 150 log1.txt
```

sed

sed is a powerful streamline editor that can be used to modify files on the fly. How to use sed would take a full document itself. Maybe I will add basic sed functionality in the future, but for now I leave you to "Google It"

setenv

Sets a variable in the C-shell only. Does not work in Bash!

Notes:

- Many NMR users who are familiar with the C-shell may try to use the command **setenv**. In Bash a variable can be set as shown in the example.

```
$> export NUM_THRDS=20
$> export MESSAGE='A message stored as an EVN variable'
```

sleep

Waits a time, in seconds, before proceeding

sleep time

```
$> sleep 5
```

sort

Sorts a text file

sort [-g][-n][-k key][-r][-f][-u][-h][-R][-b][-d][-M][-o *output*]

-g	Compare as a general numerical value
-n	Compare as a numerical value – similar to **-g**
-k*key* or **-k*key,ktype***	Sorts by a ***key***; typically a column. For example, "**sort -g -k5 text**" sorts by the fifth column in general numerical sort. **Note.** You can have multiple **-k** arguments if the sort should work on one column first and then another. In the case of multiple columns the syntax should define the type of sort for each column explicitly. For example, "**sort -k5,5g -k9,9n text**"
-r	Reverses the sort
-f	Ignores capitalization when sorting
-d	Uses a dictionary sort where only blanks and alphanumeric charachers
-u	Unique. Only output a single line even if multiple identical lines exist
-h	Sort numbers in that are human readable 1K, 1M, 1B, 1G, 1T, etc.
-R	Randomize the sort.
-b	Ignores spaces at the beginning of lines

-M	Treats the first three letters as a abbreviation for month and sorts on Month
-o *output*	Can be used to send the output to a file.

```
$> sort –n list.txt
$> ls –l | sort –k6,6d –k5,5n
```

Sort the output of the ls command by column six as a dictionary word and then column 5 as a number

```
$> ls -lh | sort –h -r -k5 –o sorted-by-size.txt
```

tail

Shows the tail-end of a file

tail [-r][-*lines*] *filename*

-r	Reverse the order of the output
-*lines*	Display the number of lines specified by ***lines***.

```
$> tail process.log
$> tail -25 process.log
$> tail –r process.log
```

tar

Copies files to a tar archive or extracts files from a tar archive.

tar [-x][-v][-t][-c][-r][-w]{compression} –f *tarfile* –C *extract_dir*

-x	Extract a tar archive
-v	Verbose mode
-t	Just lists the contents of a tar archive. Do not use with –c or -x
-c	Create a tar archive
-r	Append files to a tar archive. Archive cannot be compressed when appending.
-w	Interactive mode. Prompt before any file that is added or extracted to/from an archive
Compression types	
-z	Compression (**gzip**), Extensions **(*.tar.gz *.tgz *.taz**)
-Z	Compression (**compress**), Extensions **(*.tar.Z *.tZ *.taZ)**. Generally not used to compress tar archives anymore, but it may be useful for extracting older tar archives.
-J	Compression (**xz**), Extensions **(*.tar.xz *.txz)**
-j	Compression (**bzip2**), Extensions **(*.tar.bz2 *.tb2 *.tbz *.tbz2)**
--lzma	Compression (**LZMA**), Extensions **(*tar.lzma *.tlz)**
--lzip	Compression (**lzip**), Extensions **(*.tar.lz)**
--lzop	Compression (**lzop**), Extensions **(*.tar.lzo *.lzo)**
Historically tar was used with tape systems. Generally now you always want the **-f** argument and it should come directly before the tar filename	
-f *tarfile*	Always use the **-f** argument and always directly before the tar archive filename
-C *extract_dir*	At the end of the tar command when extracting tar archives allows the tar archive to be extracted to a different location than the current directory.

```
$> tar –xvf backup.tar
```

Extract archive backup.tar

```
$> tar –cvf backup.tar Data
```

Create archive backup.tar from Data

```
$> tar –tvf backup.tar
```

Show the contents of archive backup.tar

```
$> tar –rvf backup.tar hsqc.ft2
```

Append to archive backup.tar with hsqc.ft2

```
$> tar –xvjwf backup.tar.bz2
```

Extract bzip2 compressed archive backup.tar.bz2 prompting for each file extracted

$> tar –cvzf backup.tgz hncacb.fid

Create archive backup.tar from hncacb.fid directory with gzip compression

$> tar –cv –lzip –f backup.tar.lz hncacb.fid

Create lzip compressed archive backup.tar from hncacb.fid

$> tar –xvzf backup.tgz –C ~/test_data

Extract gzip archive backup.tgz to ~/test_data directory

Note: When extracting tar archives the type of compression can generally be skipped as the tar command does a good job of guessing if the archive is compressed and what type of compression and then sets the appropriate arguments automatically.

tee

Is used to log output to a file while still having the output go to the screen. Usually used in cases where you want to log the output, but still monitor the progress on the screen.

tee [-a] ***output_filename***

-a	Append to the ***output_filename*** rather than overwriting

The command tee is almost always used with a pipe in front.

$> ./nmrPipe.sh | tee process.log
$> ./xplor_refine.sh | tee –a structure.log

time

Reports the time a command took to run; from the time you hit ***Enter*** till the time the shell prompt is returned.

time ***command [arguments]***

$> time nmrPipe.sh
$> time find . –name '*.log' –print
$> time sleep 5

top

Reports the processes that are running on the system and updates every few seconds with the most resources intensive processes on the top by default. Also shows information about total system resources. Often used when trying to see if a process is consuming a significant amount of the system resources.

top

$> top

Note: While top is running type "**h**" to get a list of manipulations that can be performed and type "**q**" to quit

touch

Touch has two purposes. Creating a file if it does not exist and changing the date and time the file was last accessed and/or modified.

touch [-a][-c][-m] ***date filenames***
touch ***filename***

-a	Only change the data and time the file was last ***accessed*** (not modified)
-m	Only change the data and time the file was last ***modified*** (not accessed)
-c	Do not create the file if it does not already exist. The default behavior is to create a file if it does not exist.
date	Specify the date in **mmddhhnn** format where **mm**=month, **dd**=day, **hh**=hour (24 hour clock), and **nn**=minute

The command touch, run without arguments, will create a file if it does not exist.

$> touch process.com | chmod u+x process.com
$> touch 06271200 Data/*

Changes the accessed and modified time stamps on all files in the Data directory to June 27 at noon. You cannot specify the year.

umask

The umask sets the permissions by which **NEW** files and directories are created – it has no bearing on the permissions of existing files. For NMRbox the default umask is 0022. The first digit can be assumed to be zero and will not be discussed here, so in this document we will call the default umask 022.

The three digits set the default file permission for the owner, group, and others respectively. The system sets the default file permissions by subtracting the umask from 666 (e.g. 666 – 022 = 644) and the default directory permissions by subtracting the umask from 777 (e.g. 777 – 022 = 755). **However**, for safety reasons Ubuntu Linux will not allow a newly created file to have a default execute permission. This is reflected in the table below where the directory permissions are as expected, but the file permission never get an execute permission.

Notes:

- While the default umask is set to allow other users to have read access to your files your NMRbox home folder is set to not allow access to your files by default – by default they cannot get inside your directory.
- Some programs will override the default umask settings themselves.

umask *permissions*

umask digit	Resulting default file permissions	Resulting default directory permissions
0	rw	rwx
1	rw	rw
2	r	rx
3	r	r
4	w	wx
5	w	w
6	-	x
7	-	-

```
$> umask 0077
$> touch test-permissions; ls -l
```

unalias

Remove an alias from the current shell

unalias *alias*

```
$> unalias ls
```

Notes:

Sometimes the coloring from the **ls** output can be hard to see if the background terminal color is not set appropriately for the color options of the **ls** command. In this case typing "**unalias ls**" will make the **ls** output easier to view.

uname

Give information about the system hardware and kernel

uname [-a] [-s] [-r] [-v] [-o] [-m] [-p] [-n]

-a	Shows all. No other argument matters.
-s	kernel name
-r	kernel release
-v	kernel version

-o	operating system
-m	machine hardware
-p	processor type
-n	network hostname

```
$> uname -a
$> uname -n
```

uniq

Removes identical lines from a file **that occur adjacently**. Can also be used to report unique lines, lines that are not unique, and how many times a line is repeated. The **uniq** command is often mated with **sort** to put identical lines adjacent to each other.

uniq [-c] [-d] [-u] [-*fields*] [+*chars*] [*filename* [*new_filename*]]

-c	Displays each line along with how many times it occurs
-d	Only shows lines that occur multiple times
-u	Only shows lines that occur once
-*fields*	Skips the first ***fields*** of fields from the beginning of each line. Fields are either spaces or tabs
+*chars*	Skips the first ***chars*** number of characters from the beginning of each line
filename	The ***filename*** to check. Results can also be piped into **uniq** so that no filename is necessary.
new_filename	Redirect output to a file called ***new_filename***

```
$> uniq -c process.log
$> uniq results.txt uniq-results.txt
$> cat results.txt | sort -n | uniq > uniq-results.txt
```

uptime

Reports how long the system has been up, how many users are logged in, and average load

uptime

```
$> uptime
```

wc

Word count. Counts the number of words, characters, or lines in a file or from standard out.

wc [-l] [-w] [-c] *filename*

-l	Displays the number of lines
-w	Displays the number of words
-c	Displays the number of characters

```
$> wc -l results.txt
$> ls -l | grep mark | wc -l
```

who

Reports who is logged into the computer or the time since the last boot.

who [-aH] [-b]

-aH	Shows detailed information. The **H** puts a heading on the information
-b	Shows the time since the last boot.

```
$> who
$> who -aH
$> who -b
```

whoami

Reports what your username is

whoami

$> whoami

Regular Expressions

Expressions are a string of characters and those characters can be interpreted beyond their literal interpretation. The characters that are not literally interpreted are called metacharacters. For example, in the expression "NMR is great" the quote are not literally interpreted but are used to identify the quoted text. Regular expressions are used for searching text and consist of a set of characters and metacharacters that match a specific pattern that can be interpreted. They are a bit complicated to understand, but have tremendous power.

Regular Expressions contain one or more of the following:

- ***character*** – Characters that retain their literal meaning.
- ***anchor*** – Anchors designate the position in the line of text. For example, **^** represents the beginning of the line and **$** matches the end of a line.
- ***modifier*** – Modifiers expand or narrow the range of text to be matched. Examples include *****, **?**, **[]**, ****.

Below are some common anchors and modifiers used in regular expressions. There are additional extended regular expressions anchors and modifiers such as **?, + {n}, {n,}, {n,m}, \b, \B**. Feel free to search for their meanings and use cases if you want to become a regular expression master.

*****	Match zero or more characters
	$> ls *.txt Will find all files that end with .txt
[…]	Matches any one of the enclosed characters or if using a Hyphen any character in the range.
	$> grep 'B[oO][bB]' Will find Bob, BOB, BoB, and Bob
[^…]	Matches any single character EXCEPT those in the bracket or if using a Hyphen any character EXCEPT those in the range.
	$> ls results[^1-9].txt Will find resultsA.txt, but not results1.txt
	$> ls results[1-3].txt Will find results1.txt, results2.txt, and results3.txt
^	Matches an expression that is at the beginning of a line
	$> grep '^echo' Will find any line that starts with "echo"
$	Matches an expression at the end of the line
	$> grep 'hsqc.ft2$' Will find any line that ends with "hsqc.ft2"
****	Escape the next character and treat it as a literal
	$> grep '\^hello' Will find any line starting with "^hello". The ^ is escaped by the \
.	Will match any single character of any value except the end of a line
	$> grep 'B.B' Will find BaB, BbB, B1B, B%B, etc.
\<…\>	Escaped angle brackets define a word boundary
	$> grep '\<the\>' Will match "the", but not "them", "their", "soothe", etc.