Michigan Tech

UN5390: Scientific Computing I

Fall 2016

Contents

Introduction

This is a living document. It is subject to change based on feedback. If you printed it out, be sure to check for its latest version in the GitHub repository before starting/continuing to work on the enlisted tasks.

Notations

`john`	Username
`john@mtu.edu`	Email address
http://lmgtfy.com	URL
`colossus.it.mtu.edu`	Server name (i.e., hostname of a workstation)
`hello_world.cpp`	File (or folder or link) name
`hello_world()`	Function name
`# Prints "Hello, World"`	Comment
`print "Hello, World!";`	Code
`rm -rf *`	Command

Guidelines

1. Follow the numerical sequence of training camps, and the sequence of tasks therein. Each task usually tends to build upon results (from a previous task). Each training camp should take 2-3 hours to complete (including time for taking handwritten notes).

   There is no due date by which these need to be completed. However, doing so before the beginning of Fall instruction will help you get bootstrapped (i.e., *lacing up the shoes before beginning the journey*), and in turn, make a better use of time and opportunities.

2. Replace `john` with your ISO username, and John Sanderson with your real/full name. ISO credentials refer to your Michigan Tech ISO username and password (i.e., the ones you use to log into Banweb or Canvas or Michigan Tech Gmail).

3. Keeping detailed handwritten notes is a very good habit to cultivate/maintain. It can save time and frustration when things go wrong. Saving meaningfully named electronic screenshots of workflows (commands and their output) is another very useful habit.

   Notes should include date and time, location (building, floor and room #), and the hostname of the workstation being used. External help, from instructor or anyone else, will be very limited in their absence.

4. Set yourself up with a reliable data backup scheme, and use it.

5. Acknowledging that you need help and asking for it when necessary is not a sign of weakness. Be sure to appropriately cite the source of any such help.

Rationale

The following aspects impact the ability to learn and/or retain the material.

1. The (perceived) difficulty of the material
2. The (perceived) relevancy and usefulness of the material
3. The method used to learn the material
4. The method used to reinforce and/or recall the material
5. Sleep, stress and other physiological aspects (i.e., life)

#1. Pleasure vs Familiarity

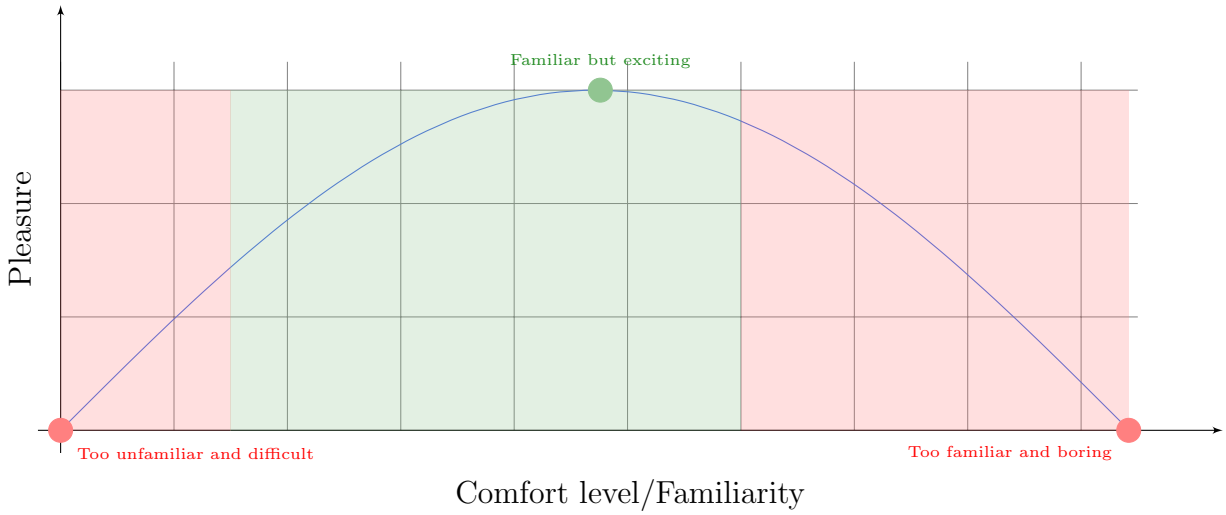

Figure 1: Schematic representation pleasure as a function of familiarity. New topics often take time and effort to become familiar. Summer provides a relatively relaxed atmosphere to become familiar with Training Camp material so that more exciting topics can be covered in Fall.

#2. The learning curve

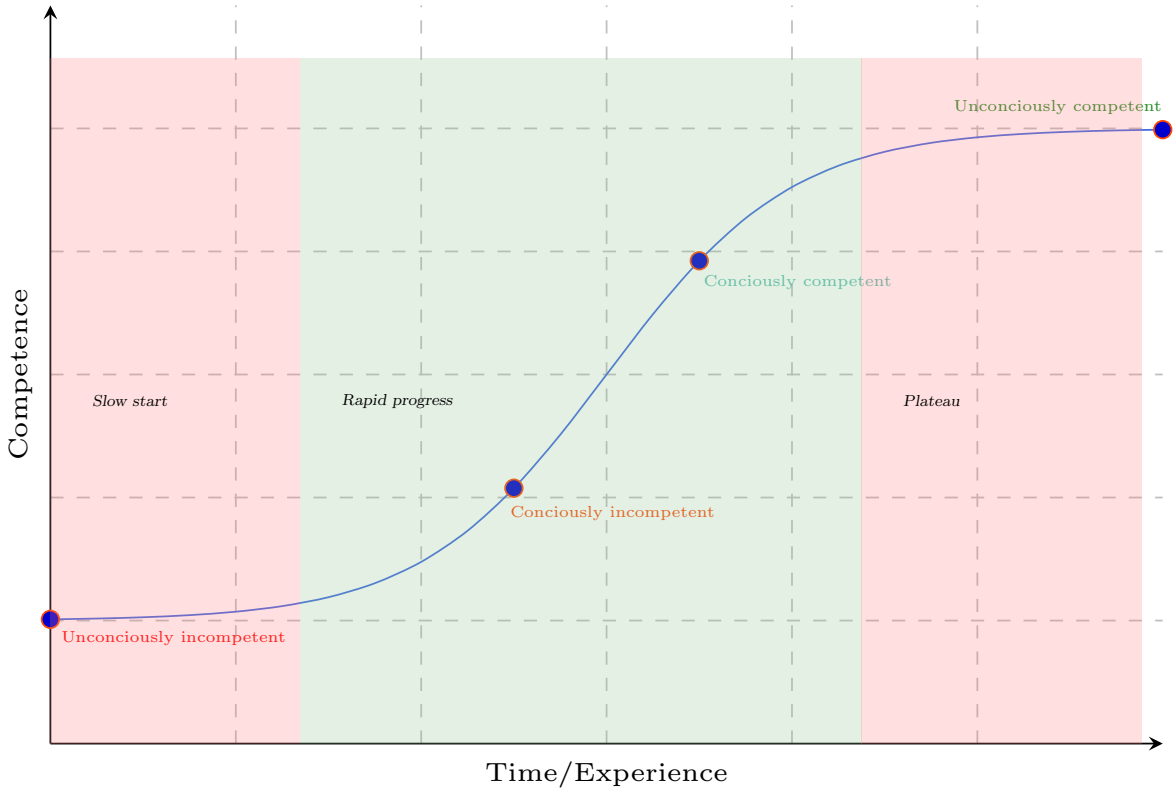

Figure 2: Schematic representation of the learning curve (i.e., competence as a function of time/experience). Summer provides a relatively relaxed atmosphere to accommodate slow start on Training Camp material and facilitate rapid progress (via their applications to scientific computing) in Fall.

#3. Spaced repetition/Forgetting curve

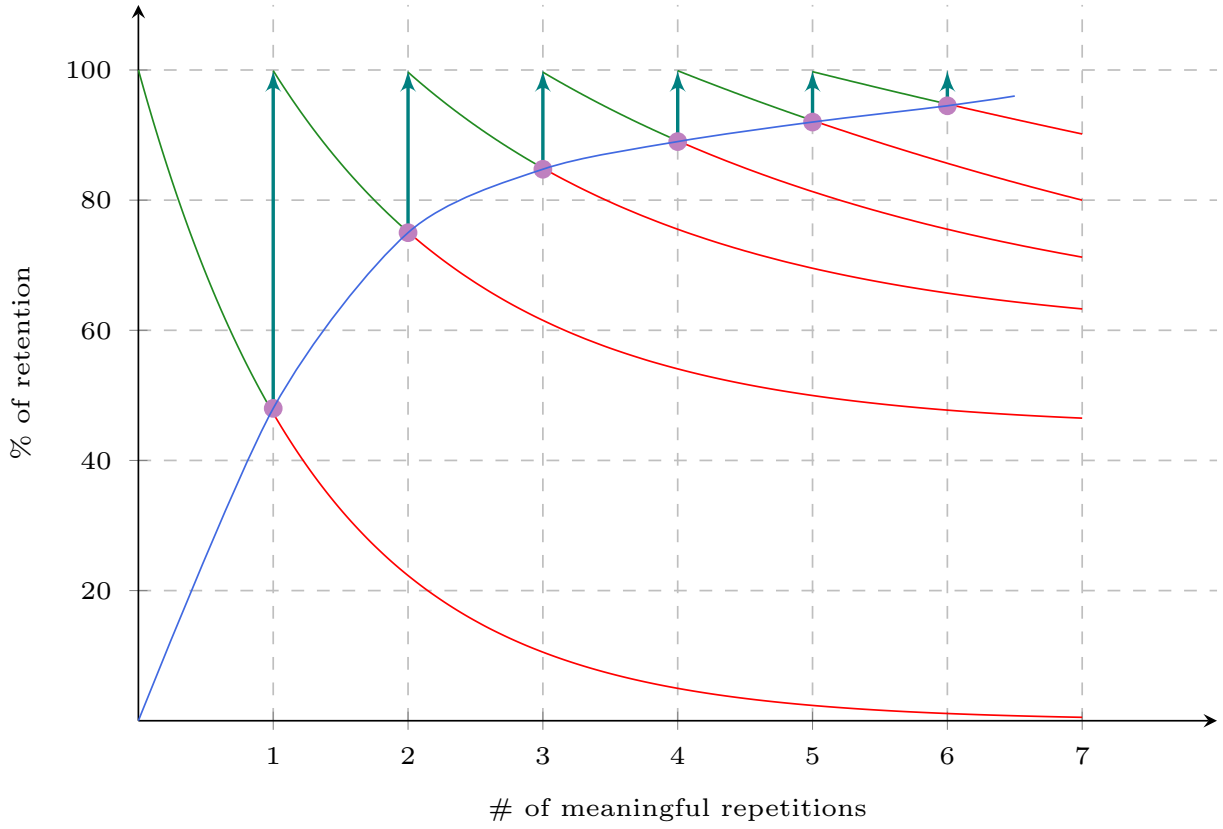

Figure 3: Schematic representation of the spaced repetition (green), forgetting (red), and the modified learning (blue) curves. The number and frequency of repetitions can vary by the individual. Each repetition should require less time compared to the previous one. Summer provides a relatively relaxed atmosphere to learn and review the Training Camp material at least once.

Training Camp #01

1. Do you know your Michigan Tech ISO username and password?
2. Do you know an IT-managed lab with a Linux workstation?
   https://www.it.mtu.edu/computer-labs.php
3. Can you log into one such workstation with your ISO credentials?
4. Do you have a notebook and pen/pencil to keep handwritten notes?
5. Do you have a reliable data backup scheme? Are you using it?

Task 1

[Changing the default shell to /bin/bash] Once successfully logged into an IT-managed Linux workstation with ISO credentials, open the URL below in a web browser.

https://mylogin.mtu.edu/

Log into the portal using the ISO credentials. If this is the first time using this portal, the terms of use will need to be accepted. Click on My Profile. Select `/bin/bash` from the drop-down list for NIS Shell. Click on Submit. It can take up to twenty minutes for the changes to take effect.

Task 2

[Basic commands] The `READ-EVALUATE-PRINT` loop (`REPL`) is a simple and interactive programming environment that take a single user input, evaluates it, and returns the result to the user. A program written in `REPL` is executed piecewise. Common examples include command line interface in Linux OS, and is particularly characteristic of scripting languages. `REPL` continues until the user decides to log out.

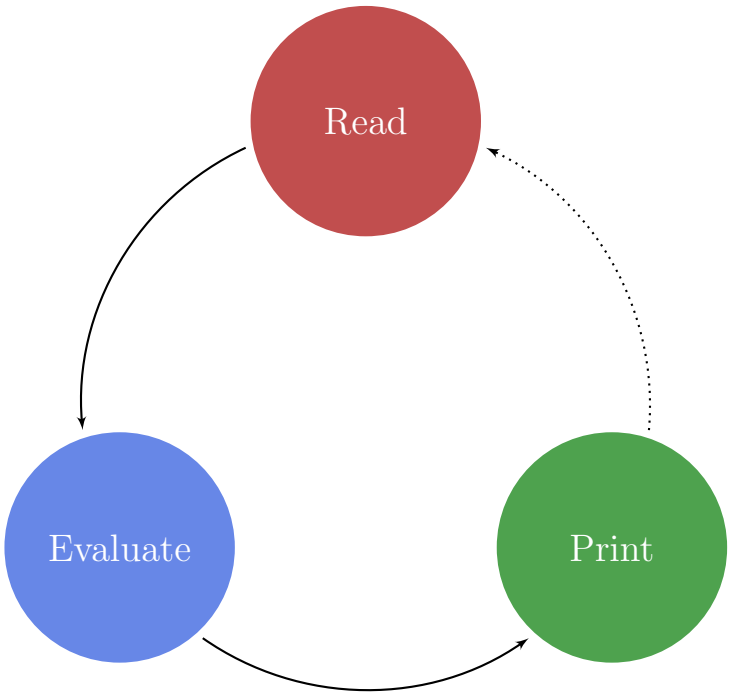

Figure 4: Schematic representation of the `READ-EVALUATE-PRINT` loop

Try these commands in a Terminal, and note down what they do. Make it a habit to use the `man` command to learn more instead of looking things up in Google. Meaning of an option for a command could be different based on the flavor of Linux OS.

```
hostname
whoami
pwd
date
date -R
date +"%Y%m%d_%H%M%S"
date +"%Y%m%d_%H%M%S_%N"
cal
cal -3
cal 06 2016
cal 2017
clear
lspci
who
w
df
df -h
du
du -sh
free
free -g
ps
top -u ${USER} -b -n 1
dingdong
```

Task 3

[Variables] Locate and open the Terminal application. Note down the output of the following commands.

```
echo "${HOSTNAME}"
echo "${SHELL}"
echo "${USER}"
echo "${HOME}"
echo "${PWD}"
echo "${PS1}"
```

It is an excellent practice to enclose the variables, system- or user-defined, in curly braces. Doing so lends itself for not only printing the value of variables correctly but also for their manipulation. Carefully observe the output of the following commands.

```
echo "${HOSTNAME}"
echo "$HOSTNAME"
echo "${#HOSTNAME}"
echo "$#HOSTNAME"
echo "${HOSTNAME:0:2}"
echo "$HOSTNAME:0:2"
echo "${HOSTNAME:3}"
echo "${HOSTNAME:3:5}"
echo "${HOSTNAME:(-4)}"
echo "${HOSTNAME:(-5):2}"

A_B="MyAwesomeVariable"
echo "${A_B}"
echo "${A_B}_001"
echo "$A_B_001"
echo "${A_B/Awesome/Poor}"

X_Y="MyOtherAwesomeVariable"
echo "${X_Y}"
echo "${A_B}_${X_Y}"
echo "$A_B_$X_Y"
echo "$A_B $X_Y"

PQR=""
echo "${PQR}"
echo "${PQR:-YourExcellentVariable}"
echo "${PQR}"
TUV=""
echo "${TUV}"
echo "${TUV:=YourExcellentVariable}"
echo "${TUV}"

VARIABLE="this is not a test"
echo "${VARIABLE}"
echo "${VARIABLE#t*is}"
echo "${VARIABLE##t*is}"
echo "${VARIABLE%t*is}"
echo "${VARIABLE%%t*is}"
```

Task 4

[Creating files, folders and symbolic links] Run `ls -latrh` after every command given below.

```
mkdir ${HOME}/UN5390_Sandbox
cd ${HOME}/UN5390_Sandbox
mkdir folder_01 folder_02
mkdir folder_{03,04,05}
mkdir folder_06/sub_folder_01
mkdir -p folder_06/sub_folder_01
touch file_01.txt file_02.txt file_{03,04,05}.txt
ln -s folder_01 folder
ln -s folder_02 folder
ln -sf folder_02 folder
ln -sf file_01.txt file.txt
file folder_01
file file_02.txt
file folder
file file.txt
file file_02.txt folder_01 file.txt file_99.txt folder_76 file.doc
```

Symbolic links can be very useful and time saving entities when designing/using computational workflows. For e.g., a file or folder name could be hard-coded into the workflow but multiple files/folders (10s if not 100s or more) need to be processed using that workflow.

Task 5

[Navigating the file system] The path to an entity (i.e., a file or a folder or a symbolic link) is said to be absolute if it begins with a leading `/` or some variant of thereof (e.g., `~` or `${HOME}`, etc.). The path is said to be relative if it begins with no leading `/` or some variant thereof (e.g., `./` or `../`, etc.).

Both (absolute and relative) have advantages and disadvantages. Absolute path names can be useful when referencing system entities – they usually don't change for a given flavor of Linux OS. Relative path names can be useful when referencing user-developed workflows – such an approach makes the workflow easily portable.

Run `pwd` followed by `ls -latrh` after every following command. Doing so, as a habit, will ensure that one is indeed in the appropriate folder and that the folder contains necessary entities before attempting subsequent commands.

```
cd /
```

```
cd usr/local
cd -
cd
cd /usr/local/share
cd ~/
cd /root/
cd ${HOME}
cd UN5390_Sandbox/folder_06/sub_folder_01
cd ../../folder_01
cd ~/UN5390_Sandbox
ls -R
tree
```

What do the `cd` (without any option or argument), `cd -` and `tree` commands do? `dirs`, `pushd` and `popd` commands can be used when there is a need to frequently switch between multiple locations as part of a workflow. Run `pwd` after each command.

```
cd ~/
dirs
pushd /usr/local
pushd ${HOME}/UN5390_Sandbox/folder_06
popd
popd
popd
popd
```

Task 6

[Accessing a remote server] Open a Terminal, and type the following command.

```
ping -c 10 colossus.it.mtu.edu
```

If `colossus.it` responded favorably to the ping requests, then run

```
ssh colossus.it.mtu.edu
```

If this is the first time ever connecting to `colossus.it`, then the RSA key fingerprint will need to be accepted by typing `yes`. Enter the ISO password when prompted. Was the login attempt successful? Did the command prompt change indicating that you are now in `colossus.it`? Type `exit` to log out of `colossus.it` and return to the local workstation (the command prompt should have changed to indicate success).

Task 7

[SSH authentication keys - Part #01] Check if the `${HOME}/.ssh` folder already exists.

```
file ${HOME}/.ssh
```

If yes, then check its contents using the command `ls -latrh ${HOME}/.ssh`. Are there files named `id_rsa` and `id_rsa.pub`? If yes, skip Task 8.

How would one determine if a listed entity is a file or a folder or a symbolic link? Attempt to understand various options used in conjunction with `file` and `ls` commands by running the `man file` and `man ls`. Note down the meaning of each option. One can scroll down by using the down arrow key or the space bar. When done, one can type `q` to exit the manual.

Task 8

[SSH authentication keys - Part #02] By design, the campus home directory is mounted across all IT-managed workstations (unless explicitly indicated otherwise). As such, setting up the SSH keys once on any one such workstation using the following command will suffice.

```
ssh-keygen -t rsa -b 4096 -C "john@mtu.edu"
```

Accept the default location for `${HOME}/.ssh/id_rsa`, set a passphrase (must be different from the ISO password). These keys constitute the *campus identity*. Do not ever share or distribute `${HOME}/.ssh/id_rsa`. This identity will be used to facilitate passwordless secure login and secure data transfer attempts.

Task 9

[SSH authentication keys - Part #03] Run the following commands to set up authentication using the SSH public key.

```
ssh-copy-id colossus.it.mtu.edu
```

Enter the ISO password when prompted. Now, attempt to SSH into `colossus.it`.

```
ssh colossus.it.mtu.edu
```

Was there a prompt for the SSH key passphrase? Was there a prompt for the ISO password? How would one go about setting this up in a non IT-managed workstation or a personal laptop/workstation that does not mount the campus home directory?

Task 10

[Command history] If not all commands have been noted down in the book, `history` command can recall them. Does its output match the notes so far? `history` only shows the commands and not their output. The number of commands it can recall can also be very limited.

Training Camp #02

1. Have you completed the tasks in previous training camp?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 11

[Username and groups] Linux OS is a multi-user system. Every user has a unique username and belongs to one primary group (and potentially one or more additional groups). Similarly, every entity (i.e., file or folder or symbolic link) has one owner and one group associated with it. By default, the user that creates the entity is the owner, and her/his primary group is the group with which the entity is associated with.

Run the following commands to identify your username and the groups you belong to.

```
whoami
id
```

The primary group is identified by `gid` in the output of `id`. `groups` is another command that provides similar information.

Task 12

[Permissions and ownerships] Refresh your notes for Task 5 in Training Camp #01. One of the commands you tried was the following.

```
cd /root/
```

This should have resulted in the following error message (does it match your notes?)

```
-bash: cd: /root/: Permission denied
```

This is a simple yet powerful example of Linux OS's security features. Permissions are one way it protects against inadvertent/malicious tampering of files and/or folders.

The `ls` command with `-l` option (i.e., `ls -l`) provides a complete listing that includes the permissions and ownership information.

```
cd ~/UN5390_Sandbox
ls -l folder_02 file_01.txt
```

This results in two lines of output, and each line contains nine columns (separated by blank space). Linux OS has three types of permission for a given entity: read, write and execute. In alphabet mode, these are represented by the letters `r`, `w` and `x` respectively. The numeric mode counterparts are the octal digits, `4`, `2` and `1` respectively.

The first character in the very first column indicates the entity type:

- regular file
b block special device
c character special device
d folder
l symbolic link
p pipeline
s socket

The next nine characters in this column, in groups of three, indicate the permission information for the owner (identified by u for user), members of owner's primary group (identified by g group members) and for everyone else (identified by o for others). r indicates that the entity can be read, w indicates that the entity can be written to, x indicates that the entity can be executed (if it is script or a program), and - indicates a specific permission has not been assigned (or has been removed).

The third column indicates the owner of the entity, and the fourth column indicates the group with which the entity is associated with. Ninth column contains the name of the entity. Can you identify and understand the permissions and ownership for folder_02 and file_01.txt? Attempt to understand the meaning of other columns.

Task 13

[Changing permissions] Note that permissions are a security feature, and changing them (especially granting read, write and/or execute permission) increases the risk of an entity being tampered with – inadvertently or otherwise. As such, permissions should be granted if and only if there is a need. chmod command can be used to alter permissions – using alphabet or numeric mode.

Suppose that a certain entity, file_07.txt, has the following permissions.

```
-rwxr-x--x
```

1. This entity is a regular file.
2. The user (i.e., the owner of the entity) has read (r; 4), write (w; 2) and execute (x; 1) permissions. 4 + 2 + 1 yields a numerical score of 7 for the user.
3. Members of the group to which the entity belongs to has read (r; 4) and execute (x; 1) permissions but. no write (w; 2) permission. 4 + 0 + 1 yields a numerical score of 5 for the group.

4. Others (i.e., except the owner and members of the group to which the entity belongs to) have execute (`x`; `1`) permission but no read (`r`; `4`) or write (`w`; `2`) permission. `0 + 0 + 1` yields a numerical score of `1` for others.

5. `-rwxr-x--x` translates to `751` in numeric mode. Thus, the following two commands are equivalent.

   ```
   chmod u=rwx,g=rx,o=x file_07.txt
   chmod 751 file_07.txt
   ```

Try the alphabet mode given below, and run `ls -l` after each command.

```
cd ${HOME}/UN5390_Sandbox
chmod u-rwx file_01.txt
cat file_01.txt
chmod u=rwx file_01.txt
cat file_01.txt
chmod g+rw,o-rwx file_01.txt
chmod g+x,o-x folder_02
chmod u-rwx folder_02
cd folder_02
chmod 644 file_01.txt
chmod 755 folder_02
```

Try the numeric mode given below, and run `ls -l` after each command. Does it reproduce the results from alphabet mode?

```
cd ${HOME}/UN5390_Sandbox
chmod 044 file_01.txt
cat file_01.txt
chmod 744 file_01.txt
cat file_01.txt
chmod 660 file_01.txt
chmod 754 folder_02
chmod 055 folder_02
cd folder_02
chmod 644 file_01.txt
chmod 755 folder_02
```

Refer to `man chmod` for more information.

Task 14

[Copy and move] `cp` command can be used for copying entities Run `ls -ltrh` after each command.

```
cd ${HOME}/UN5390_Sandbox
cp file_01.txt file_06.txt
cp file_01.txt folder_02
cp file_02.txt file_03.txt folder_02
cp folder_02 folder_06
cp -r folder_02 folder_06
cp -r folder_02 folder_06
cp file_02.txt /usr/local/bin/
```

Check the contents of `folder_02` and `folder_06`, and understand what happened.

`mv` command can be used for moving entities. Run `ls -ltrh` after each command.

```
cd ${HOME}/UN5390_Sandbox
mv file_01.txt file_07.txt
mv file_07.txt folder_02
mv file_02.txt file_03.txt folder_02
mv folder_02 folder_07
mv folder_03 folder_07
mv file_02.txt /usr/local/bin/
mv file_04.txt /usr/local/bin/
```

Check the contents of `folder_07`, and understand what happened.

Refer to `man cp` and `man mv` for more information.

Task 15

[Transferring files/folders] Suppose that there is a need to transfer a file or a folder from one location (source) to another (destination). Depending on the nature and amount of data that needs to be transferred, using portable USB devices can be an option. However, if either the source or destination is physically inaccessible (e.g., a server in a secure data center or in a geographically distant location), portable USB devices may not be of much use.

If the source and destination reside within the same workstation, one can use the previously discussed `cp` command. If the source is in the local workstation and the destination is in a remote workstation (or vice versa), one can use the `scp` command.

```
cd ${HOME}/UN5390_Sandbox
truncate -s 25M file_${USER}.txt
mkdir folder_${USER}
cd folder_${USER}
truncate -s 10M data_file_01.txt
```

The general syntax for the `scp` command is as follows.

```
scp OPTIONS SOURCE DESTINATION
```

Suppose that the files and folders need to be transferred from the local workstation (source) to `colossus.it` (destination). `scp` requires the option `-r` to transfer folders. In the absence of SSH authentication keys setup in Training Camp #01, `scp` will prompt for password.

```
cd ${HOME}/UN5390_Sandbox
scp file_${USER}.txt john@colossus.it.mtu.edu:/tmp/
scp file_${USER}.txt john@colossus.it.mtu.edu:/tmp/
scp -r folder_${USER} john@colossus.it.mtu.edu:/tmp/
truncate -s 15M folder_${USER}/data_file_02.txt
scp -r folder_${USER} john@colossus.it.mtu.edu:/tmp/
```

Now, suppose that the files and folders need to be transferred from `colossus.it` (source) to the local workstation (destination).

```
cd ${HOME}/UN5390_Sandbox
scp john@colossus.it.mtu.edu:/tmp/file_${USER}.txt /tmp/
scp -r john@colossus.it.mtu.edu:/tmp/folder_${USER} /tmp/
```

SSH into `colossus.it` and navigate to `/tmp` folder. Both `file_${USER}.txt` and `folder_${USER}` should be present. Delete them and exit out of `colossus.it`. Also, navigate to `/tmp` folder in the local workstation, and delete the transferred entities.

One may have noticed that the `scp` command took approximately the same amount of time irrespective of whether the destination did not have the entity being transferred or not. `scp` works quite well for one time file/folder transfers but is not very well suited for subsequent transfers (or when the contents – of the file or the folder) have changed in the source. `cp` command also suffers from the same drawback for local transfers.

A very common scenario of growing number of files and folders, and changed content therein is the lifetime of a (research) project. In the very early stages, there could be very few of them and very little (changing) content in them. As such, transferring them using `cp` or `scp` takes very little time. But with time, both can change to an extent that time required to transfer them in their entirety every time can be prohibitively very expensive.

Linux OS provides a command, rsync, to overcome this problem. It can work for purely local transfers (where the source and the destination reside within the same workstation) or between a local and a remote workstation using ssh for security. The general syntax for the rsync command is as follows.

```
rsync OPTIONS SOURCE DESTINATION
```

In some sense, it combines the features of cp and scp commands, and goes a bit further by transferring only the changed content. As such, it is the transfer tool of choice in many data backup programs, and in setting up content mirroring websites.

```
cd ${HOME}
rm -f UN5390_Sandbox/folder_${USER}/data_file_02.txt
rsync -avhP UN5390_Sandbox/ UN5390_Sandbox2/
rsync -avhP UN5390_Sandbox/ UN5390_Sandbox2/
truncate -s 5M ${HOME}/UN5390_Sandbox/file_${USER}.txt
rsync -avhP UN5390_Sandbox/ UN5390_Sandbox2/
```

The second and subsequent runs of rsync should have taken considerably less time compared to the first run. rsync minimizes the transfer time by comparing the files and folders (and their size and time stamp) in source and destination, and transferring only the changed content. A similar approach is used even in case of remote transfers. In the absence of SSH authentication keys setup in Training Camp #01, rsync over ssh will prompt for password.

```
cd ${HOME}
\rm -r UN5390_Sandbox2
cd UN5390_Sandbox
rsync -ave ssh -hPz file_${USER}.txt john@colossus.it.mtu.edu:/tmp/
rsync -ave ssh -hPz file_${USER}.txt john@colossus.it.mtu.edu:/tmp/
rsync -ave ssh -hPz folder_${USER} john@colossus.it.mtu.edu:/tmp/
truncate -s 15M folder_${USER}/data_file_02.txt
rsync -ave ssh -hPz folder_${USER} john@colossus.it.mtu.edu:/tmp/
```

rsync, in previous examples of sending data to colossus.it from the local workstation, should have taken considerably less time for its second run compared to the first. Now, suppose that the files and folders need to be transferred from colossus.it (source) to the local workstation (destination).

```
rsync -ave ssh -hPz john@colossus.it.mtu.edu:/tmp/file_${USER}.txt /tmp/
rsync -ave ssh -hPz john@colossus.it.mtu.edu:/tmp/file_${USER}.txt /tmp/
rsync -ave ssh -hPz john@colossus.it.mtu.edu:/tmp/folder_${USER} /tmp/
rsync -ave ssh -hPz john@colossus.it.mtu.edu:/tmp/folder_${USER} /tmp/
```

Delete `file_${USER}.txt` and `folder_${USER}` in the local workstation as well as in `colossus.it`.

Both `scp` and `rsync` accept absolute and relative to paths to entities being transferred. If either the source or the destination is a remote worstation, then the path can either be absolute or relative to the user's home directory in the remote workstation. If either the source or the destination is a local workstation, then the path can either be absolute or relative to the present working directory. In itheir most commonly used form, only one – the source or the destination – can be a remote workstation.

Refer to `man scp` and `man rsync` for more information.

Training Camp #03

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 16

[Adding content via output redirection] Every single file created so far, in Training Camps #01 and #02, is empty. Since the files lack content, there is nothing in them to display.

```
cd ~/UN5390_Sandbox
ls -l
cat file_04.txt
```

What did the last command display? The `cat` command (short for concatenate) can be used to display the contents of a file on the screen. Output redirection (meaning: sending the output of a command to a file instead of the screen) is one way of adding content to a file. Run `cat file_04.txt` after each command.

```
ls -l > file_04.txt
date -R > file_04.txt
hostname >> file_04.txt
whoami >> file_04.txt
echo "JRVP Library, Room #244" >> file_04.txt
echo "HOME: ${HOME}; SHELL: ${SHELL}" >> file_04.txt
echo >> file_04.txt
pwd -P >> file_04.txt
ls -l >> file_04.txt
```

Do you notice any difference between using a `>` (i.e., single) and `>>` (i.e., double) when redirecting the output of a command to a file? The `cat` command can also be used to copy or combine multiple text files.

```
cat file_04.txt
cat file_04.txt > file_05.txt
cat file_05.txt
cat file_04.txt >> file_05.txt
cat file_05.txt
cat file_04.txt file_05.txt >> file_06.txt
cat file_06.txt
cat file_04.txt file_05.txt file_06.txt
cat folder_07
```

Refer to `man cat` for more information.

Task 17

[Opening a file in read-only mode] While opening a file in edit (i.e., read and write) mode is useful, doing so can often cause unintended damage. For e.g., a simulation can be updating a file in the background, and the act of saving the file (i.e., writing to it) could lead to loss of some information. As such, it is a really good practice to open a file in read-only mode unless there is a really really good reason to do otherwise.

For the most part, `cat` is one such command that displays the contents of a file in read-only mode. As you may have noted in the previous task, it displays all the contents of a file (or multiple files) to the screen at once. Linux OS provides a class of commands called PAGER – `less` and `more` – (i.e., display the contents one page at a time). Run the following commands (press `space bar` to move to the next page, and press `q` to exit out of it at any time).

```
cd ~/UN5390_Sandbox
less file_04.txt
more file_04.txt
less file_06.txt
more file_06.txt
```

Refer to `man less` and `man more` for more information.

Task 18

[Adding content using gedit] Output redirection is one way of adding content to a file, and it has a very useful place in the development of computational workflows. However, it can be a very tedious, time-consuming and error-prone approach to add general content to a file (e.g., typesetting a weekly status report on your research activities OR source code for a computer program OR LaTeX document to typeset your solutions in an assignment).

`gedit` is a very Notepad-like graphical application that serves as a good starting point to typeset documents. Typeset the content in the following page.

```
cd ~/UN5390_Sandbox
gedit latex_01.tex
```

`gedit` may result in the following error message if one is accessing `colossus.it` from a local Microsoft Windows workstation (using PuTTY, Terminals, WinSCP or similar Terminal emulation application).

```
connect /tmp/.X11-unix/X0:  No such file or directory
(gedit:13595):  Gtk-WARNING **:  cannot open display:  localhost:10.0
```

```
% Type of document
\documentclass[letterpaper,12pt]{article}

% Packages
\usepackage[dvips]{graphicx}
\usepackage{amsmath,amssymb,color,fancybox,setspace}
\usepackage[numbers]{natbib}

% Page format
\setlength{\oddsidemargin}{0.00in}  % Margin on the odd numbered side
\setlength{\evensidemargin}{0.00in} % Margin on the even numbered side
\setlength{\topmargin}{-0.50in}     % Margin from the top
\setlength{\textwidth}{6.50in}      % Width of the text in a page
\setlength{\textheight}{9.00in}     % Height of the text in a page
\setlength{\parindent}{0.00in}      % New paragraph indentation
\setlength{\parskip}{0.20in}        % Spacing between two paragraphs

% Page style
\pagestyle{plain}

% Document begins
\begin{document}

Per Wikipedia, scientific computing is a multi-disciplinary field that
uses advanced computing capabilities to understand and solve complex
problems. It is the application of computer simulation to solve problems
in various arts, science and engineering disciplines -- to gain an
understanding mainly through the analysis of mathematical models
implemented on computers.

The field includes algorithms and modeling/simulation software developed
to solve the problems, computer and information science that develops and
optimizes the system (hardware, software, networking, and data management,
etc.) needed to solve the problems, and the computing infrastructure that
supports both the science and engineering problem solving and the
developmental computer and information science.

% Document ends
\end{document}
```

Task 19

[Improving command history] As you may have noticed, the `history` command – at least in its default setting – does not keep track of more than one thousand commands. It also does not include any date/time information.

```
echo "${HISTFILE}"
echo "${HISTFILESIZE}"
echo "${HISTSIZE}"
echo "${HISTCONTROL}"
```

Create `~/.bash_${USER}` using `gedit`. Add the following content, and close the file.

```
# History command settings
export HISTSIZE=25000
export HISTFILESIZE=25000
export HISTTIMEFORMAT="%Y-%m-%d %H:%M:%S "
export HISTCONTROL=ignoreboth
```

Now, edit `~/.bashrc`. Add the following content at the very end, and close the file.

```
# Source user's settings
if [ -f "${HOME}/.bash_${USER}" ]
then
  source "${HOME}/.bash_${USER}"
fi
```

Running the following command (generally known as *sourcing*)

```
source ~/.bashrc
```

will execute the contents of `~/.bashrc`, one line at a time, as if they were manually typed at the command line. This is necessary for the changes (made to `~/.bashrc` or `~/.bash_${USER}`) to come into effect in the same Terminal. `~/.bashrc` is automatically sourced every time a new Terminal is opened, and as a result, the settings will be in effect. Note that the following command

```
.  ~/.bashrc
```

is a shortcut for

```
source ~/.bashrc
```

Sourcing the file is different from *executing* it. When a file is sourced, the commands in it are being typed in the same/current shell. As a result, any changes made in the file will take effect and stay in the current shell. When a file executed, the commands are typed in a new shell and the new shell is closed after the output is copied back to the current shell. As a result, any changes made in the file will take effect only in the new shell and will be lost once the new shell is closed.

Task 20

[Adding content using vim] While `gedit` is an easy to use editor with GUI, `vi` (or `vim`) is the editor of choice in UN5390. The latter uses command line and can be extremely useful to typeset a document when the (remote) workstation does not provide graphical capabilities. Learn/Familiarize yourself with this editor using the resource listed below:

http://www.openvim.com/

Typeset `latex_01_vim.tex` with the same contents as in `latex_01.tex` using `vim`.

Training Camp #04

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 21

[Adding shortcuts via rc files] The final task in Training Camp #03 introduced the concept of customizing the shell via editing ~/.bashrc and ~/.bash_${USER} files. While creating and maintaining the latter seems added labor, it provides the following advantage: the default settings will remain intact as defined by the system or the administrator, and the user-defined settings will exist as a separate modular entity to be included on the fly. In the improbable event that ~/.bashrc is overwritten by the system (or the administrator), restoring user-defined settings is as easy as adding the following lines and *sourcing* it.

```
# Source user's settings
if [ -f "${HOME}/.bash_${USER}" ]
then
  source "${HOME}/.bash_${USER}"
fi
```

A rc file is a configuration file (also known as a *config* file) containing the parameters and initial settings to configure a given application. Such a file almost always is in ASCII (i.e., plain text) format and resides in the user's ${HOME} directory. Also, its name starts with a . making it invisible/unlisted for the usual ls -l command. For e.g., .bashrc contains such parameters and settings for the user's shell, .vimrc for vim editor, and so on.

The naming convention from the runcom files CTSS (*Compatible Time-Sharing System*), a predecessor of UNIX OS developed in MIT in the early 1960s. These plain text startup files contained commands that might have been invoked manually once the system was running but are to be executed automatically each time the system starts up. Various rc files with instructions for an application (or an entire OS) were found in /etc/rc/ folder. Thus, rc is short for runcom, and runcom, in turn, is short for *run commands*. This convention and practice is still used to this day in Linux OS. rc can also be, albeit unofficially, remembered as *runtime configuration*.

Remembering complete commands, with various options and arguments, can be difficult and typing them out in entirety can be time-consuming as well as an error-prone activity. The consequence of some such errors can be very costly. For e.g., note the following commands.

```
clear
bc -l
ls -l
ls -a
mkdir -p
ssh colossus.it.mtu.edu
```

Typing them out every once in a while might seem neither time-consuming nor error-prone but a general/computational workflow often involves many many repetitions of these and/or other commands. Create/Open ~/.bash_${USER} using vim and add the following content.

```
# User-edited variables
export TZ="America/Detroit"
export PATH="${PATH}:${HOME}/bin"
export MANPATH="${MANPATH}:${HOME}/man"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${HOME}/lib"
export CC="/usr/bin/gcc"
export CXX="/usr/bin/g++"
export FC="/usr/bin/gfortran"
export EDITOR="/usr/bin/vim"
export OMP_NUM_THREADS=1
export PAGER="/usr/bin/less"
export PS1="\[\e[;34m\][\u@\h \W]\$ \[\e[m\]"

# User-defined variables
export COLOSSUS="colossus.it.mtu.edu"
export UN5390="${HOME}/git_work/un5390_f2016_${USER}"

# User-defined aliases (i.e., shortcuts)
alias bc="bc -l"
alias bzip2="bzip2 -9"
alias cl="clear"
alias colossus="ssh -Y ${COLOSSUS}"
alias git_log='git log --date="iso8601" --pretty=format:"%h %an %ad : %s"'
alias gpull="git pull"
alias gzip="gzip -9"
alias la="ls -a"
alias ll="ls -lh"
alias lr="ls -ltrh"
alias mkdir="mkdir -p"
alias vi="vim"

# User-defined function to display home folder, shell, and hostname
hfsh() {
  printf "\n  %-24s  %-10s  %-s\n" "Home folder" "Shell" "Hostname"
  printf "  %-24s  %-10s  %-s\n\n" "${HOME}" "${SHELL}" "${HOSTNAME}"
}
```

Save and close the file, and *source* ~/.bashrc. Now, try the following and observe how each alias runs its definition.

```
cd ~/UN5390_Sandbox
cl
ll
la
lr
hfsh
ssh ${COLOSSUS}
exit
colossus
exit
```

If need be, an alias can be temporarily disabled using the unalias command.

```
mkdir ~/bin ~/lib ~/man ~/src ~/tmp
alias mkdir
unalias mkdir
alias mkdir
mkdir test1/test2/test3
alias mkdir="mkdir -p"
alias mkdir
mkdir test1/test2/test3
unalias mkdir
alias mkdir
mkdir test4/test5/test6
.  ~/.bashrc
alias mkdir
mkdir test4/test5/test6
\rm -r test1
\rm -r test4
```

One may add additional aliases, variables and functions as necessary to ~/.bash_${USER}. But note that overwriting/re-defining certain system variables can have unintended (or irreversible) consequences. It is a good practice to check the defined aliases, and functions and variables using the following commands respectively before attempting to (re)define them.

```
alias
set
```

env command can be used to list environment variables that are already defined.

Task 22

[Updating vim settings] Much like ~/.bashrc for a user's shell, ~/.vimrc can be used to control the configuration (i.e., appearance, shortcuts, etc.) for the vim editor. Create/Update ~/.vimrc with the following content using vim editor.

```
" General settings
syntax on
set expandtab tabstop=2
set history=1000
set ignorecase smartcase
set incsearch hlsearch
set noautoindent
set showcmd showmatch showmode
set pastetoggle=<F3>
set ruler
set wildmenu
filetype indent off

" Show/Hide line numbers
set nonumber numberwidth=4
nmap <C-N><C-N> :set invnumber<CR>
```

Save and close the file. To observe the effect of updated .vimrc settings, open ~/UN5390_Sandbox/latex_01_vim.tex using vim and do the following:

1. Move around the document using arrow keys. Do the line and column numbers change?
2. Place the cursor on the first opening brace (i.e., {). Is the corresponding closing brace (i.e., }) highlighted?
3. Search the following strings: The, the, and setlength. Make a note of the results.
4. Execute the key combination, CTRL + n + n (i.e., press the n key twice while pressing the CTRL key). Did the line numbers show up? Execute the same key combination again. Did the line numbers disappear?
5. While in ESCAPE mode, search for Wikipedia and replace it with Wikimedia.

Undo all changes. Save and close the file.

Task 23

[Comparing files] Computational workflows, while either in the testing phase or in the absence of a consistent file naming convention, can result in files with (nearly) identical content. Linux OS provides a set of commands to compare files and check for duplicate content.

```
cd ~/UN5390_Sandbox
diff latex_01.tex latex_01_vim.tex
```

Assuming you exercised care and caution when typesetting these two files, the `diff` command should have displayed no output. One can also run the `sdiff` command as shown below to compare the two files side by side (the Terminal may need to be widened).

```
sdiff latex_01.tex latex_01_vim.tex
```

If there are differences between the two files, fix them and make the two files look identical. To minimize the number of files and keep redundancy to a minimum, run

```
rm -f latex_01_vim.tex
```

Run the following commands to create two data files, and check the contents of `file_01.dat` and `file_02.dat`.

```
cd ~/UN5390_Sandbox
touch file_01.tmp file_01.dat file_02.tmp file_02.dat
truncate -s0 file_01.tmp file_01.dat file_02.tmp file_02.dat

for x in $(seq -w 0 1 100)
do
  echo "${x}" >> file_01.tmp
done
shuf file_01.tmp > file_01.dat
rm -f file_01.tmp

for x in $(seq -w 0 3 100)
do
  echo "${x}" >> file_02.tmp
done
shuf file_02.tmp > file_02.dat
rm -f file_02.tmp
```

These two data files can be thought as the result of a series of complete (`file_01.dat`) and incomplete (`file_02.dat`) computational experiments – each line representing the ID of a successfully completed experiment. Suppose that the task is to identify the extent of failure (i.e., the IDs of missing experiments in `file_02.dat`) so that only a portion of the said computational experiments can be performed to save time and other resources.

Linux OS provides a command, `comm`, that can compare two sorted files and print a three-column output: first column with lines unique to the first file, second column with lines unique to the second file, and third column with lines common to both files. Note down the result of following commands.

```
comm file_01.dat file_02.dat
sort file_01.dat -o file_01.dat
sort file_02.dat -o file_02.dat
comm file_01.dat file_02.dat
```

Why does the last command display only two columns instead of three as described? Which of the following produces the necessary information (i.e., the extent of failure)?

```
comm -12 file_01.dat file_02.dat
comm -13 file_01.dat file_02.dat
comm -23 file_01.dat file_02.dat
```

`cmp` is another command that can be used to compare files. Refer to `man cmp`, `man comm`, `man diff`, `man sdiff`, `man seq`, `man shuf` and `man sort` for more information.

Task 24

[Text manipulation without opening a file] `vim` (or any other interactive) editor easily provides a way to manipulate the text in a file. However, one of the main reasons for developing (general or computational) workflows is to infuse as much automation into and remove as much user interaction from the process as possible. `sed`, stream editor, is a very powerful command that can be used to accomplish this. In its simplest form, `sed` command has one of the following formats.

```
sed "s/SEARCH/REPLACE/" FILE
sed "s/SEARCH/REPLACE/2" FILE
sed "s/SEARCH/REPLACE/g" FILE
sed "s/SEARCH/REPLACE/3g" FILE
sed "/IF_THIS_IS_THERE/s/SEARCH/REPLACE/g" FILE
sed "/IF_THIS_IS_NOT_THERE/!s/SEARCH/REPLACE/g" FILE
sed "s/SEARCH_THIS/REPLACE_THIS/g;s/SEARCH_THAT/REPLACE_THAT/g" FILE
```

```
cd ~/UN5390_Sandbox
sed "s/article/report/" latex_01.tex > latex_02.tex
sed "s/setlength/setwidth/" latex_01.tex > latex_02.tex
sed "s/setlength/setwidth/2" latex_01.tex > latex_02.tex
sed "s/setlength/setwidth/3g" latex_01.tex > latex_02.tex
sed "s/setwidth/setlength/g" latex_01.tex > latex_02.tex
sed "/Per/s/Wikipedia/Wikimedia/g" latex_01.tex > latex_02.tex
sed "/begin/!s/document/nodocument/g" latex_01.tex > latex_02.tex
sed "s/oddside/sideodd/g;s/evenside/sideeven/g" latex_01.tex > latex_02.tex
```

Open `latex_02.tex` and verify the impact of each `sed` commands. Delete it to avoid duplication. To further minimize the number of (temporary) files, `-i` option can be used (with care and caution) to perform live edits as follows. Open `latex_01.tex` and verify the impact of each `sed` command.

```
sed -i "s/article/report/" latex_01.tex
sed -i "s/report/article/" latex_01.tex
sed -i "s/setlength/setwidth/g" latex_01.tex
sed -i "s/setwidth/setlength/g" latex_01.tex
sed -i "/Per/s/Wikipedia/Wikimedia/" latex_01.tex
sed -i "/Per/s/Wikimedia/Wikipedia/" latex_01.tex
sed -i "/begin/!s/document/nodocument/g" latex_01.tex
sed -i "/begin/!s/nodocument/document/g" latex_01.tex
sed -i "s/oddside/sideodd/g;s/evenside/sideeven/g" latex_01.tex
sed -i "s/sideodd/oddside/g;s/sideeven/evenside/g" latex_01.tex
```

In the event that the search string does contain the slash (`/`), it needs to escaped as follows (often known as the *tooth-saw effect*). Open `latex_01.tex` and verify the impact of each `sed` command.

```
sed -i "s/modeling\/simulation/simulation\/modeling/g" latex_01.tex
sed -i "s/simulation\/modeling/modeling\/simulation/g" latex_01.tex
```

Tooth-saw effect works but becomes increasingly difficult to keep track of (and error prone) when the search string involves more than one slash. `sed` provides the option of choosing a custom delimiter other than the slash. In the following examples, an underscore (`_`) is used.

```
sed -i "s_modeling/simulation_simulation/modeling_g" latex_01.tex
sed -i "s_simulation/modeling_modeling/simulation_g" latex_01.tex
```

Refer to `man sed` (or http://sed.sourceforge.net/sed1line.txt) for more practical examples.

Task 25

[Patching files] The life cycle of a source code (or a subset of source codes for a larger program) often involves multiple revisions and sharing it with recipients or other users. Suppose that `HelloWorld.c`, included below, is one such file in its first revision (typeset it using `vim` and save it in `~/UN5390_Sandbox`).

```
// HelloWorld.c

#include <stdio.h>

int main() {
  printf("  Hello, World!");
}
```

Now suppose that, after sharing the above program with several recipients, it is updated to include meaningful comments, compilation and execution instructions and give it an aesthetically pleasing appearance, etc. in the second revision (`HelloWorld1.c`; typeset it using `vim` and save it in `~/UN5390_Sandbox`).

```
// HelloWorld.c
// C program to display 'Hello, World!' to the screen/terminal.
// Compilation and execution, under normal circumstances in most modern
// Linux workstations, takes less than one second.
//
// Compilation and execution:
// gcc -g -Wall HelloWorld.c -o HelloWorld.x -lm
// ./HelloWorld.x

// Headers
#include <stdio.h>

// main()
int main(int argc, char *argv[]) {
  // Print the message to the screen
  printf("\n");
  printf("  Hello, World!\n\n");

  // Indicate termination
  return 0;
}
```

One way to ensure that the recipients have the latest version is to have them manually update their copy of `HelloWorld.c` using `vim` (or any other interactive) editor. It might be easier and more economical to distribute `HelloWorld1.c`, and have the recipients rename it as `HelloWorld.c` so that they all have and use the updated version.

Linux OS provides the `patch` command, to be used in conjunction with the previously learned `diff` command, to apply patches from one version to the next. This command also works (and is better suited) when working with a larger number of files in the distribution. The workflow is as follows.

```
cd ~/UN5390_Sandbox
diff -u HelloWorld.c HelloWorld1.c > HelloWorld.patch
```

Check the contents of `HelloWorld.patch` using `cat` or `less` command. Now, instead of distributing an entirely new version of the code, it will suffice to distribute just the patch file. This patch file contains all the necessary information to update (or in other words, *apply patches*) with a fairly simple command – without even having to explicitly open `HelloWorld.c`. Run the following commands mimic a recipient's workflow.

```
cd ~/UN5390_Sandbox
patch --dry-run -i HelloWorld.patch
cat HelloWorld.c
patch -b -i HelloWorld.patch
cat HelloWorld.c
```

The `-b` option to `patch` command will make a backup of the original file (with `.orig` extension; `HelloWorld.c.orig` in this particular case), and then apply the patches to update `HelloWorld.c`. `-V numbered` option can be used to suffix the backup filename with numbers instead of `.orig` extension. One may check the after-effects of patching as follows.

```
diff HelloWorld.c HelloWorld.c.orig
diff HelloWorld.c HelloWorld1.c
```

The `patch` command also provides an elegant way to reverse the applied changes (useful if/when the applied patches break the workflow or lead to unpleasant results) as follows.

```
patch -R -i HelloWorld.patch
cat HelloWorld.c
diff HelloWorld.c HelloWorld.c.orig
diff HelloWorld.c HelloWorld1.c
```

Refer to `man diff` and `man patch` for more information.

Training Camp #05

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 26

[Finding files, folders and links] It often becomes necessary to find a file or a folder (or a set of them) that matches a given criteria. For e.g., *all entities that have been modified in the last three days in the current working folder* or *all entities that have a 644 permission* or *all files that are at least 5 MB in ~/UN5390_Sandbox*, etc. Linux OS provides the `find` command to accomplish such tasks.

```
cd ~/UN5390_Sandbox
find .  -type d
find .  -type f
find .  -type l
find .  -name "folder_01"
find .  -iname "file_07.txt"
find / -maxdepth 3 -type d
find /etc -name passwd
find ${HOME} -name "folder_01"
find ~/UN5390_Sandbox -iname "file_07.txt"
find .  -type d -iname "folder_??"
find .  -type f -iname "file_??"
find .  -type f -iname "*.tex"
find .  -perm 644
find .  -type f !  -perm 644
find ${HOME} !  -user ${USER} -exec ls -lh {} \;
find .  -type f -name "file_07.txt" -exec rm -f {} \;
find .  -type f -perm 777 -print -exec chmod 644 {} \;
find .  -type d -perm 777 -print -exec chmod 755 {} \;
find .  -empty
find .  -type f -empty -exec rm -f {} \;
find /usr/lib64 -atime 7
find .  -mtime 3
find ~ -mtime +3 -mtime -7
find /tmp -amin -90
find /var/log -mmin -60
find .  -size 2k
find .  -size +5M -size -10M
find .  -mtime 2 \( -iname "*.tex" -o -iname "*.dat" \) -exec ls -lh {} \;
find .  -type f -user ${USER} -iname "core.*" -size +5M -exec rm -f {} \;
```

`locate` is another useful command that can help find files by name. Refer to `man find` and `man locate` for more information.

Task 27

[Combining files] General/Computational workflows often include the task of combining multiple ASCII (i.e., plain text) files into one – one below the other OR side by side. The first task in Training Camp #03 demonstrated how the `cat` command can be used to combine multiple files into one – one below the other. Run the following commands to create two data files, and check the contents of `file_03.dat` and `file_04.dat`.

```
cd ~/UN5390_Sandbox
touch file_03.dat file_04.tmp file_04.dat
truncate -s0 file_03.dat file_04.tmp file_04.dat

for x in $(seq 1 1 100)
do
  y=$((x + 100))
  z=$(( ( RANDOM % 32000 ) + 1 ))
  printf "%03d\n" "${x}" >> file_03.dat
  printf "%3d.%05d\n" "${y}" "${z}" >> file_04.tmp
done

shuf file_04.tmp > file_04.dat
rm -f file_04.tmp
```

Each line in `file_03.dat` can be thought as the ID of a successfully completed computational experiment, and each line in `file_04.dat` can be thought of as the value of some measured quantity in such an experiment. Each file should have 100 lines in them, with a one-to-one mapping between ID of the experiment (in `file_03.dat`) and the measured value from that experiment (in `file_04.dat`).

```
wc -l file_03.dat
wc -l file_04.dat
```

Did the `wc` command return 100 lines for both files? Suppose that the contents of two files need to be combined side by side to for further analysis.

```
echo "# " > file_05.dat
echo "# Generated on " $(date -R) >> file_05.dat
paste file_03.dat file_04.dat >> file_05.dat
```

Open `file_05.dat` using `less` and make sure it has the appropriate content. How many lines does the new file have? Refer to `man cat`, `man paste` and `man wc` for more information.

Task 28

[Finding strings in files] Workflows often involve searching for strings in a file – either manually typeset or generated by computational experiments. For e.g., *search for the string 'setlength' in latex_01.tex* or *print the lines that start with 01 in file_05.dat* or *search for the lines that have either 'algorithm' or 'information' in latex_01.tex* or *print a specified number of lines before/after the matching string* or *print the lines that end with 06 in file_05.dat*, etc.

While one can always open a file in an interactive editor and look for desired strings, Linux OS provides the `grep` command (along with `egrep` and `fgrep`) to accomplish the same without opening the file. This can assist in developing automated computational workflows.

```
grep "setlength" latex_01.tex
grep -n "setlength" latex_01.tex
grep "the" latex_01.tex
grep -i "the" latex_01.tex
grep "document" latex_01.tex
grep -w "document" latex_01.tex
grep -w "documentclass" latex_01.tex
grep -iw "document" latex_01.tex
grep -A2 "documentclass" latex_01.tex
grep -B5 "evensidemargin" latex_01.tex
grep -E "^01" file_05.dat
egrep "^01" file_05.dat
grep "(algorithm|information)" latex_01.tex
grep -E "(algorithm|information)" latex_01.tex
grep -E "06" file_05.dat
grep -E "06$" file_05.dat
```

Suppose that *algorithm* and *information* are two of the many strings that need to be searched in `latex_01.tex`. One could save such strings in a separate ASCII file, and search for them all at once.

```
touch strings2search.txt; truncate -s0 strings2search.txt
echo "algorithm" >> strings2search.txt
echo "information" >> strings2search.txt
echo "scientific" >> strings2search.txt
grep -F -f strings2search.txt latex_01.tex
fgrep -f strings2search.txt latex_01.tex
```

Refer to `man grep`, `man egrep` and `man fgrep` for more information.

Task 29

[Pattern recognition] Many, if not most, files used in general/computational workflows store data in some pre-defined pattern – defined either by the system or the user that created it.

One example is an input file with atomic labels and Cartesian coordinates. It can be thought of as having four columns: the first column being the atomic labels, the next three being x, y and z coordinates corresponding to each atomic label. The columns are usually separated by space (either the regular blank space or a `TAB`). Another example is the `/etc/passwd` file. It can be thought of as having seven columns: username, password, UID, GID, user information (e.g., real name), home directory and default shell. The columns are separated by colon (`:`).

The concept of pattern is not limited just to the data within files but can be easily generalized to the file (and/or folder) naming convention. Suppose that a project involves running the a given CS&E software suite (say, `program.x`) on one hundred different input files (say, `filename_001.input`, `filename_002.input`, ..., `filename_100.input`). Further suppose that each input file has a corresponding output file (say, `filename_001.output`, `filename_002.output`, ..., `filename_100.output`). One could then treat the period (`.`) as a field separator to get the basename as the first column, and subsequently use the underscore (`_`) as the field separator to get the ID as the second column.

Taking time to design a consistent naming convention before running the software – using padded/leading zeros as appropriate (i.e., one for up to 10 cases, two for up to 100 cases, three for up to 1000 cases, etc.) – not only makes their listing more aesthetic but also lends for their programmatic execution and/or analysis.

One can use `vim` (or any other interactive) editor to open the file, identify the pattern and manually extract the necessary information from it. However, as noted previously, one of the main reasons for developing (general or computational) workflows is to infuse as much automation into and remove as much user interaction from the process as possible. `awk` is a pattern scanning and processing language (yes, a language). It also serves as a very powerful command to recognize a specified pattern in and extract necessary information, manipulate/alter it, if necessary, from a file.

Run the following commands to produce `file_06.dat` with two columns and `file_07.dat` with four columns.

```
cd ~/UN5390_Sandbox
paste file_03.dat file_04.dat > file_06.dat
paste file_06.dat file_06.dat > file_07.dat
```

Now, run the following commands.

```
awk '{ print $0 }' file_03.dat
awk '{ print $1 }' file_03.dat
awk '{ print $NF }' file_03.dat
awk '{ print $0 }' file_06.dat
awk '{ print $1 }' file_06.dat
awk '{ print $2 }' file_06.dat
awk '{ print $NF }' file_06.dat
awk '{ print $0 }' file_07.dat
awk '{ print $1 }' file_07.dat
awk '{ print $2 }' file_07.dat
awk '{ print $3 }' file_07.dat
awk '{ print $4 }' file_07.dat
awk '{ print $NF }' file_07.dat
awk '{ print NR }' file_04.dat
awk 'NR % 3 == 0' file_07.dat
awk 'END { print NR }' file_04.dat
awk '{ print $1 "\t" $2 }' file_07.dat
awk '{ print $4 "\t" $2 "\t" $3 }' file_07.dat
awk '{ print $4 "\t" $2 "\t" $3 }' file_07.dat > file_08.dat
awk '{ if (NR > 50) { print $2 "\t" $1 } }' file_06.dat
awk '{ if ((NR > 50) || (NR < 75)) { print $2 "\t" $1 } }' file_06.dat
awk 'BEGIN { sum = 0 } { sum = sum + $3 } END { print sum }' file_07.dat
awk -F ':' '{ print $1 "\t" $6 "\t" $7 }' /etc/passwd
awk -F ':' '{ printf "%15s \t %25s \t %15s\n", $1, $6, $7 }' /etc/passwd
awk -F ':' '{ printf "%-15s \t %-25s \t %-15s\n", $1, $6, $7 }' /etc/passwd
awk -F ':' 'length($1) == 8' /etc/passwd
awk -F ':' 'length($1) == 8 { print $1 }' /etc/passwd
awk -F ':' '$3 == $4' /etc/passwd
awk -F ':' '$3 == $4 { print $NF }' /etc/passwd
awk -F ':' '/\/bin\/bash/ { print $1 }' /etc/passwd
awk -F ':' '$NF ~ /\/bin\/bash/ { user++ } END { print user }' /etc/passwd
awk '{ x = sqrt($1) ; print $1 "\t" x }' file_03.dat
awk '{ x = sqrt($1) ; printf "%03d \t %9.6f\n", $1, x }' file_03.dat
awk -v d="5" 'BEGIN { print strftime("%Y-%m-%d", systime() + (d * 86400))}'
awk '{ gsub("Wikipedia", "Wikimedia", $0); print $0 }' latex_01.tex
```

`cut` is another useful command that can be used for extracting information from a file. Refer to `man awk` and `man cut` (or http://www.pement.org/awk/awk1line.txt and http://www.gnu.org/software/gawk/manual/gawk.html) for more practical examples.

Task 30

[Output of one command as input for the next] Suppose that one needs to list all users that have logged into the current workstation in descending order of the number of login attempts. A traditional workflow to accomplish this would be as follows. Observe the contents of each temporary file using the `cat` command.

```
# Run the 'last' command
last > login_count_01.tmp

# Remove empty lines
sed "/^\s*$/d" login_count_01.tmp > login_count_02.tmp

# Extract the first column using space as the field separator
# If the -F option and field separator is absent, space is assumed
awk -F ' ' '{ print $1 }' login_count_02.tmp > login_count_03.tmp
awk '{ print $1 }' login_count_02.tmp > login_count_03.tmp

# Sort the first column alphabetically
sort login_count_03.tmp > login_count_04.tmp

# Count the number of duplicates
uniq -c login_count_04.tmp > login_count_05.tmp

# Sort in descending order
sort -nr login_count_05.tmp
```

The workflow, as can be seen, involves creating five temporary files. The number can be reduced to four if first of the `sort` commands is run with the `-o` option and re-purposed `login_count_03.tmp`. Even simplest of general/computational workflows often include multitude of commands. As a result, saving the output of one command in a temporary file to be used as input for a subsequent command can create an overwhelmingly large number of such temporary files. This also creates the additional burden of carefully managing and deleting them at the end of the workflow, if not immediately.

Piping, *the act/practice of treating the output of one command as input for a subsequent command*, is an elegant solution to this burden. The pipe (or *the connector between two commands*) is represented by the `|` character.

Run the following command incrementally, compare the final output with that of `sort -nr login_count_05.tmp`, and observe how it does not create temporary files.

```
last | sed "/^\s*$/d" | awk '{ print $1 }' | sort | uniq -c | sort -nr
```

Note that *piping* is not a substitute for a step by step approach with careful creation, management and deletion of temporary files while the workflow is being designed, developed and tested. Replace temporary files with pipe iff the workflow can be successfully reproduced.

Run each set of commands incrementally and observe the output.

```
du | sort -nr
ls -l | tail -n +2 > file.txt
ls -l | sed "1d" >> file.txt
cat file.txt | wc -l
ps aux | grep "${USER}"
echo "scale=15; 4*a(1)" | bc
seq 1 1 100 | awk 'BEGIN { sum = 0 } { sum += $1 } END { print sum }'
ls -l | grep "^.....w"
ls -l | tail -n +2 | sed "s/\s\s*/ /g" | cut -d ' ' -f 3 | sort | uniq -c
```

\ is the continuation character and indicates that what follows in the next line is still a part of the same command. It helps keep one command per line (especially in a script), and makes editing/debugging easier. It is, however, very important to ensure that there is no space after the \ character.

```
echo "filename_001.input" | \
  awk -F '.' '{ print $NF }'

echo "filename_001.input" | \
  awk -F '.' '{ print $1 }'

echo "filename_001.input" | \
  awk -F '.' '{ print $1 }' | \
  awk -F '_' '{ print $1 }'

echo "filename_001.input" | \
  awk -F '.' '{ print $1 }' | \
  awk -F '_' '{ print $NF }'
```

Training Camp #06

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 31

[Compressing and uncompressing entities] Computational workflows can often result in *large* files. Should there be a need to transfer such files between two workstations (or just archiving for long term storage), compressing them helps to speed up such transfers (or their storage consumes less space). Linux OS provides a class of commands to accomplish this task: `bzip2`, `gzip` and `zip`. The following commands will create additional files to test (de)compression.

```
cd ~/UN5390_Sandbox
mkdir folder_20
for x in $(seq 10 1 19)
do
  y=$(shuf -i 2-8 -n 1)
  base64 /dev/urandom | head -c ${y}M > file_${x}.dat
  cp file_${x}.dat folder_20/
done
```

`zip` is a commonly used compression and file packaging utility. It can be used to compress one or more entities. `unzip` is the corresponding decompression utility. An important thing to note is that `zip` leaves behind the uncompressed entities after compression, and `unzip` leaves behind the compressed entities after decompression. The user, as a result, is expected to delete the entities as necessary. Run `ls -lh` after each command, and compare the sizes of compressed files with their uncompressed counterparts.

```
cd ~/UN5390_Sandbox
zip file_10.dat.zip file_10.dat
unzip -l file_10.dat.zip
unzip file_10.dat.zip
unzip file_10.dat.zip file_20.dat
zip file_10_11.dat.zip file_10.dat file_11.dat
unzip -l file_10_11.dat.zip
unzip file_10_11.dat.zip
zip file_10_19.dat.zip file_1?.dat
unzip file_10_19.dat.zip
zip -r folder_20.zip folder_20
unzip folder_20.zip
zip -r files_folder.zip file_15.dat file_17.dat folder_20
unzip -l files_folder.zip
unzip files_folder.zip
```

`gzip` is another commonly used command to compress one or more files using the Lempel-Ziv coding (LZ77). The compressed file can be identified by the extension, `.gz`. `gunzip` is the corresponding decompression utility. `gzip` does not retain the uncompressed file(s) after compression. Likewise, `gunzip` does not retain the compressed file(s) after decompression. Run `ls -lh` after each command.

```
cd ~/UN5390_Sandbox
gzip file_10.dat
gzip -l file_10.dat.gz
gunzip file_10.dat.gz
gzip file_10.dat file_11.dat
gzip -l file_10.dat file_11.dat
gunzip file_10.dat.gz file_11.dat.gz
gzip file_1?.dat
gzip -l file_1?.dat.gz
gunzip file_1?.dat.gz
gzip -r folder_20
gzip -rl folder_20
gunzip -r folder_20
gzip file_10.dat
gzip -d file_10.dat.gz
gzip -r file_15.dat file_17.dat folder_20
gzip -rl folder_20 *.dat.gz
gunzip -r *.gz folder_20
```

`gzip` facilitates concatenation of multiple compressed files. Run the following commands.

```
gzip -c file_10.dat > file.dat.gz
gzip -c file_11.dat >> file.dat.gz
```

The same result can be achieved via

```
cat file_10.dat file_11.dat | gzip > file.dat.gz
```

Now, the following three commands produce identical results.

```
gunzip -c file.dat.gz
cat file_10.dat file_11.dat
zcat file.dat.gz
```

bzip2 is another commonly used command to compress one or more files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. The compressed file can be identified by the extension, .bz2. bunzip2 is the corresponding decompression utility. Without the -k option, bzip2 does not retain the uncompressed file(s) after compression and bunzip2 does not retain the compressed file(s) after decompression. bzip2 and bunzip2 are quite often used as the traditional alternative to gzip and gunzip. bzip2 has a higher memory requirement than gzip. Run ls -lh after each command.

```
cd ~/UN5390_Sandbox
bzip2 file_10.dat
bunzip2 file_10.dat.bz2
bzip2 file_10.dat file_11.dat
bunzip2 file_10.dat.gz file_11.dat.bz2
bzip2 file_1?.dat
bunzip2 file_1?.dat.bz2
bzip2 file_10.dat
bzip2 -d file_10.dat.bz2
bzip2 -k file_13.dat
bzip2 file_15.dat
bunzip2 -k file_15.dat.bz2
```

bzip2 facilitates concatenation of multiple compressed files. Run the following commands.

```
bzip2 -c file_10.dat > file.dat.bz2
bzip2 -c file_11.dat >> file.dat.bz2
```

The same result can be achieved via

```
cat file_10.dat file_11.dat | bzip2 > file.dat.bz2
```

Now, the following three commands produce identical results.

```
bunzip2 -c file.dat.bz2
cat file_10.dat file_11.dat
bzcat file.dat.bz2
```

The -1 option for bzip2 and gzip indicates faster completion times at the expense of a least thorough compression, and -9 indicates the most thorough compression at the expense of slower completion times. Run the following commands to clean up the files.

```
cd ~/UN5390_Sandbox
find .  -size +1M -size -9M -iname "file_??.dat*" -exec rm -f {} \;
\rm -r folder_20
```

In order to compare different compression algorithms, create a sufficiently *large* data file.

```
cd ~/UN5390_Sandbox
touch file.tmp file.dat
truncate -s0 file.tmp file.dat

for x in $(ls /var/log/cron*.gz)
do
  gzip -c ${x} >> file.tmp
done

for x in $(seq 1 1 100)
do
  cat file.tmp >> file.dat
done

rm -f file.tmp
```

Now, compress each file with a different compression utility and compare their sizes using the following commands.

```
zip file.dat.zip file.dat
gzip -c file.dat > file.dat.gz
bzip2 -c file.dat > file.dat.bz2
du -sh file.dat*
```

Run the following command to clean up these files.

```
rm -f file.dat*
```

Refer to `man bzip2`, `man bunzip2`, `man gzip`, `man gunzip`, `man zip` and `man unzip` for more information.

Task 32

[Archiving files and folders] The previous task detailed various compression and decompression utilities. Often the workflows come with a need to compress an entire folder – for saving space or for later usage or purely for archival purposes. Linux OS provides the `tar` command (short for *tape archive* – derived from a time when entities were commonly backed up on and occasionally restored from magnetic tape as a permanent storage device) to prepare archives of files and folders. Run the following commands to prepare the necessary files and folder.

```
cd ~/UN5390_Sandbox
mkdir folder_20
p=0
for x in $(ls /var/log/cron*.gz)
do
  p=$((p + 1))
  q=$((p + 100))
  r=$(printf "%03d" ${q})
  gzip -c ${x} > file_${p}.dat
  gzip -c ${x} > folder_20/file_${r}.dat
done
```

Run the following commands to create an archive of `folder_20` (the act is commonly known as *tarring*), check its type and list its contents.

```
tar -cvf folder_20.tar folder_20
file folder_20.tar
tar -tvf folder_20.tar
```

The following commands can be use extract the archive (the act is commonly known as *untarring*) and verify the contents. Note that the `tar` command leaves behind the original entity. As a result, it is important to exercise caution so as not to overwrite the existing contents of a file or a folder from its archive. Run `ls -ltrh` after each command.

```
tar -xvf folder_20.tar
cd folder_20/
touch *.dat
cd ../
cp file_01.dat folder_20/
tar -xvf folder_20.tar
cd folder_20/
cd ../
```

Files or folders can be added to an existing archive as follows.

```
tar -rvf folder_20.tar file_02.dat
tar -tvf folder_20.tar
cp -r folder_20 folder_30
tar -rvf folder_20.tar folder_30
tar -tvf folder_20.tar
```

The archive can be further compressed using `zip`, `gzip` or `bzip2` utilities. The latter two are more commonly used.

```
gzip folder_20.tar
file folder_20.tar.gz
gunzip folder_20.tar.gz
bzip2 folder_20.tar
file folder_20.tar.bz2
rm -f folder_20.tar.bz2
```

The `tar` command includes an option to compress the archive using `gzip` or `bzip2` utilities. The filename extension for such compressed archives is usually `.tar.gz` (or `.tgz`) or `.tar.bz2` (or `.tbz2`) – depending on the compression technique used. It also includes an option to directly decompress such compressed archives as well.

```
tar -cvzf folder_20.tar.gz folder_20
tar -cvzf folder_20.tgz folder_20
tar -cvjf folder_30.tar.bz2 folder_30
tar -cvjf folder_30.tbz2 folder_30
tar -tvzf folder_20.tar.gz
tar -tvzf folder_20.tgz
tar -tvjf folder_30.tar.bz2
tar -tvjf folder_30.tbz2
rm -f folder_20.tgz folder_30.tbz2
tar -xvzf folder_20.tar.gz
tar -xvjf folder_30.tar.bz2
\rm -rf folder_20* folder_30* file*.dat
```

Note that a compressed archive does not facilitate further addition of files or folders directly. Refer to `man tar` for more information.

Task 33

[Reading compressed files] Suppose that a general/computational workflow has been successfully completed. Further suppose that all unnecessary files and folders have been deleted, and the necessary files have been compressed (using `gzip` or `bzip2`) to either save space on the storage device and/or to facilitate faster transfer between workstations.

Suppose that now there is a need to view/read the contents of one such compressed file. Linux OS provides a class of commands that can accomplish this without explicitly having to first uncompress the file. The first task in this Training Camp (i.e., #06) gave a sneak preview of two such commands: `zcat` and `bzcat`. Run the following commands to to generate the compressed files.

```
cd ~/UN5390_Sandbox
gzip -c latex_01.tex > latex_01.tex.gz
bzip2 -c latex_01.tex > latex_01.tex.bz2
```

The following commands can now be used to view the contents of `latex_01.tex.gz` and `latex_01.tex.bz2` without explicitly uncompressing them.

```
zcat latex_01.tex.gz
zcat latex_01.tex.gz | more
zless latex_01.tex.gz
zmore latex_01.tex.gz
bzcat latex_01.tex.bz2
bzcat latex_01.tex.bz2 | more
bzless latex_01.tex.bz2
bzmore latex_01.tex.bz2
rm -f *.gz *.bz2
```

Refer to `man zcat`, `man bzcat`, `man zless`, `man zmore`, `man bzless` and `man bzmore` for more information.

Task 34

[Integrity of a file] When sharing programs, Makefiles, source code, etc. with collaborators or third-party users over the network (e.g., personal or institutional website, GitHub repository), it can be of importance to ensure their integrity. Linux OS provides a class of commands to generate the *hash* (or *checksum*) that can serve as a digital fingerprint of a file. Any change due to a faulty file transfer, disk error or tampering – malicious or otherwise – will cause the hash to change. Run the following commands to prepare two additional files.

```
cd ~/UN5390_Sandbox
gzip -c latex_01.tex > latex_01.tex.gz
bzip2 -c latex_01.tex > latex_01.tex.bz2
```

Now, run the `md5sum` command as follows.

```
echo -n "${USER}" | md5sum
md5sum latex_01.tex
md5sum latex_01.tex.gz latex_01.tex.bz2
```

The result should have been a string of alphanumeric characters that represent the hash of each file. Run the following commands to prepare the hash for all `latex_01.tex*` and save them in `md5sum.txt`.

```
cd ~/UN5390_Sandbox
touch md5sum.txt
truncate -s0 md5sum.txt
md5sum latex_01.tex* > md5sum.txt
```

The following command can then be used to verify the integrity of all `latex_01.tex*`.

```
md5sum -c md5sum.txt
```

Run the following commands to mimic editing the file, and run the integrity check again.

```
sed -i "s/Wikipedia/Wikimedia/g" latex_01.tex
md5sum -c md5sum.txt
sed -i "s/Wikimedia/Wikipedia/g" latex_01.tex
md5sum -c md5sum.txt
rm -f latex_01.tex.*
```

`SHA1`, `SHA256` and `SHA512` are stronger alternatives to `MD5` algorithm. Refer to `man md5sum`, `man sha1sum`, `man sha256sum`, `man sha512sum` for more information.

Task 35

[Detaching the Terminal] Suppose that a general/computational workflow has grown more complex and/or involves very time consuming steps (e.g., `configure`, `make`, etc.). Further suppose that after starting the workflow in one workstation, there comes a need to go elsewhere as a result of an unforeseen situation. Such a scenario provides two options: terminate the workflow and re-start it from a different workstation later OR return to the same workstation later to monitor its progress and/or continue.

Linux OS provides the `screen` utility to overcome this problem: start the workflow in one location and continue it seamlessly from another. To mimic a time consuming workflow, `top` command is used in the following example. It can be replaced with any other command to meet a practical need. Similarly, `SESSION_NAME` can be replaced with a more meaningful and easy to remember name for the session.

```
colossus
screen -S SESSION_NAME
top
CTRL+A+d
exit
```

`CTRL+A+d` implies pressing the `A` and `d` keys while holding down the `CTRL` key, and it will detach the screen from the Terminal. To mimic continuing the work from a different location, re-login to `colossus.it`.

```
colossus
screen -ls
screen -d -R SESSION_NAME
CTRL+A+d
exit
```

The detach and re-attach process can be repeated as many times as necessary as long as the workflow is still in progress. Replace `CTRL+A+d` with `CTRL+A+K` to terminate the session. Any number of sessions can be active at a given time, and can be used to emulate multiple Terminals in a single one – especially useful when certain workstations could have a limit on the number of simultaneously active Terminals.

`tmux` is a feature-filled alternative to `screen`. Refer to `man screen`, `man tmux` and `man top` for more information.

Training Camp #07

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 36

[Splitting files] Many institutions and/or service providers could impose an upper limit on the size of a single file. For e.g., GitHub has a soft limit of 50 MB (warning) and a hard limit of 100 MB (rejection) per file in any given repository. Linux OS provides the `split` command to decompose such a *large* file into two or more *smaller* files – either by size or by number of lines – that can later be recomposed to form the original file, if/when necessary. Run the following command to create a ∼100 MB data file.

```
cd ~/UN5390_Sandbox
base64 /dev/urandom | head -c 100M > file.dat
```

Check the file size and number of lines using the following commands.

```
du -sh file.dat
wc -l file.dat
```

As an example of splitting the *large* file by size (say, 10 MB per file) and checking the size of resulting *smaller* files, run the following commands.

```
split -d -a 4 -b 10M file.dat file.dat_size_
du -sh file.dat_size_*
```

As an example of splitting the *large* file by number of lines (say, 100 thousand per file) and checking the number of lines in the resulting *smaller* files, run the following commands.

```
split -d -a 4 -l 100000 file.dat file.dat_lines_
wc -l file.dat_lines_*
```

Whether splitting by size or by number of lines, `cat` command can be used to recompose the original *large* file as follows.

```
cat file.dat_lines_* > file.dat_lines
cat file.dat_size_* > file.dat_size
```

One can compare the files – `file.dat`, `file.dat_size` and `file.dat_lines` – by using `md5sum` and other commands introduced in previous Training Camps.

```
md5sum file.dat file.dat_size file.dat_lines
du -sh file.dat file.dat_size file.dat_lines
wc -l file.dat file.dat_size file.dat_lines
rm -f file.dat*
```

Refer to `man split` for more information.

Task 37

[Downloading content from a website] While opening a web browser to download a file from a remote destination works well, it does not lend itself to automating the workflows that involve such tasks. Also, opening a web browser might not be a possibility at all if/when the remote workstation does not provide a graphical user interface.

Suppose that the workflow involves downloading a specific version (say, 2.2) of the HPL source code.

```
cd ~/UN5390_Sandbox
curl -O http://www.netlib.org/benchmark/hpl/hpl-2.2.tar.gz
```

This should have resulted in a tar-gzipped entity, `hpl-2.2.tar.gz`. Such an approach has an additional advantage: the version necessary can be saved in a variable and programmatically downloaded as necessary.

```
cd ~/UN5390_Sandbox
HPL_URL="http://www.netlib.org/benchmark/hpl"
HPL_VERSION="2.2"
curl -O ${HPL_URL}/hpl-${HPL_VERSION}.tar.gz
```

Suppose that versions 2.1 needs to be downloaded, then only `HPL_VERSION` variable needs to be updated while the `curl` command remains unchanged.

```
HPL_VERSION="2.1"
curl -O ${HPL_URL}/hpl-${HPL_VERSION}.tar.gz
rm -f hpl-*.tar.gz
```

Alternatively, suppose that the workflow involves downloading a specific version (say, 5.0.3) of the Gnuplot source code. The `-L` option tells `curl` to follow the URL redirects.

```
cd ~/UN5390_Sandbox
GNUPLOT_URL="http://downloads.sourceforge.net/project/gnuplot/gnuplot"
GNUPLOT_VERSION="5.0.3"
curl -L -o gnuplot-${GNUPLOT_VERSION}.tar.gz \
  ${GNUPLOT_URL}/${GNUPLOT_VERSION}/gnuplot-${GNUPLOT_VERSION}.tar.gz
rm -f gnuplot-*.tar.gz
```

Refer to `man curl` for more information.

Task 38

[Spell check a file] Linux OS provides the `aspell` command to spell check a file and perform other grammatical activities. It includes an option to replace an incorrectly spelled word with one or more alternatives or ignore or add to the dictionary as correctly spelled word. `aspell` may not, however, detect (programming) language-specific constructs and fix mistakes such as incorrectly placed parenthesis. Refer to `man aspell` for more information.

```
cd ~/UN5390_Sandbox
aspell check latex_01.tex
```

If an incorrectly spelled phrase was inadvertently added to `aspell`, one can manually remove it from `~/.aspell.en.pws`.

Task 39

[LaTeX compilation] A document, `latex_01.tex`, was typeset and edited in previous training camps. Unlike Adobe InDesign or Microsoft Word, LaTeX is not a WYSIWYG (*what you see is what you get*) document preparation system. A document typeset using LaTeX (i.e., `.tex`) needs to be successfully compiled before it results in an easily readable product – a `.dvi` or `.ps` or `.pdf`.

For all intents and purposes, the `.tex` can be thought of as the source code, and the `.dvi` or `.ps` or `.pdf` as the end product. LaTeX documents lend themselves for modularization, optional/programmatic inclusion of files, editing without being explicitly opened, and in turn, make document preparation a part of the automated workflow. LaTeX is the only accepted system for typesetting assignments, weekly status reports and project work in UN5390. The sequence of commands for generating a `.pdf` file from a corresponding `.tex` file (with no floats and/or no cross-references) are as follows. Run `ls -ltrh` after each command.

```
cd ~/UN5390_Sandbox
latex latex_01.tex
dvips -Ppdf -o latex_01.ps latex_01.dvi
ps2pdf latex_01.ps latex_01.pdf
```

Alternatively, one can use the following sequence of commands to directly generate the `.pdf`.

```
pdflatex latex_01.tex
```

Refer to https://github.com/MichiganTech/LaTeX_GettingStarted and learn more about installing and using LaTeX. This resource includes several different sample files and corresponding PDF with expected output.

Task 40

[Data visualization] A visualization often makes for easier observance of trends and results than raw data. In other words, *a picture is still worth a thousand words*. There are any number of plotting utilities that can be used to visualize raw data, and many have a command line mode for using them.

`gnuplot` is one such utility that can be used in interactive/graphical as well as command line mode. As a result, it is possible to extend the computational workflow to include automated visualization tasks. `gnuplot` not only generates the usual 2D/3D/parametric plots but also facilitates reading stored data from files and live data from C/C++/Python programs. It also supports LaTeX syntax for title, axis label and legend when necessary. Data manipulation and curve fitting can also be performed using `gnuplot` on the fly. Images generated from such a process can further be used to make movies/animations with `convert` utility. Gnuplot and MATLAB are the plotting/visualization tools of choice in UN5390.

Suppose that a computational workflow has been successfully completed, and has resulted in data that needs to be plotted. Further suppose that this data is stored in `~/UN5390_Sandbox/performance.dat` in the following format: N being the problem size, and the remaining four columns being time (in minutes) to solve the problem using a given number of processors.

```
# performance.dat
#   ---------------- time ------------------
# N NP001  NP002  NP004  NP008  NP016  NP032
04  00126  00062  00030  -----  -----  -----
06  00252  00124  00060  -----  -----  -----
08  00378  00186  00090  00042  -----  -----
10  00630  00310  00150  00070  00030  -----
12  01008  00496  00240  00112  00048  -----
14  01638  00806  00390  00182  00078  00026
16  02583  01271  00615  00287  00123  00041
18  04158  02046  00990  00462  00198  00066
20  06678  03286  01590  00742  00318  00106
22  10647  05239  02535  01183  00507  00169
24  17010  08370  04050  01890  00810  00270
```

This data can be interactively plotted within the `gnuplot` program. But in the interest of infusing automation into computational workflows, it serves well to have all the plotting instructions in a separate file (`.gnu` or `.gnuplot` or `.plt`). To that effect, create the file – `~/UN5390_Sandbox/performance.gnu` – using `vim` and add the following content.

```
# performance.gnu
#
# Gnuplot instructions to plot the data in performance.dat.
#
# Usage:
# gnuplot performance.gnu

# Gnuplot terminal and output settings
set term postscript landscape enhanced color dashed 15
set output "performance.eps"

# Title, axis label, range and ticks
set title  "Log of performance vs Problem size"
set xlabel "Problem size, N"
set ylabel "Log of time, t (minutes)"
set xrange [2:26]
set xtics 4, 4, 28
set mxtics 2

# Line styles
set style line 1 lt 1 lc rgb "#A00000" lw 2 pt 7  ps 1.5
set style line 2 lt 1 lc rgb "#00A000" lw 2 pt 11 ps 1.5
set style line 3 lt 1 lc rgb "#5060D0" lw 2 pt 9  ps 1.5
set style line 4 lt 1 lc rgb "#B200B2" lw 2 pt 5  ps 1.5
set style line 5 lt 1 lc rgb "#D0D000" lw 2 pt 13 ps 1.5
set style line 6 lt 1 lc rgb "#00D0D0" lw 2 pt 12 ps 1.5
set style line 7 lt 0 lc rgb '#808080' lw 1

# Legend location and grid
set key top left
set grid back ls 7

# String to identify missing data points, and plot the data
set datafile missing "-----"
plot "performance.dat" using 1:(log($2)) title "NP=01" with linesp ls 1, \
     "performance.dat" using 1:(log($3)) title "NP=02" with linesp ls 2, \
     "performance.dat" using 1:(log($4)) title "NP=04" with linesp ls 3, \
     "performance.dat" using 1:(log($5)) title "NP=08" with linesp ls 4, \
     "performance.dat" using 1:(log($6)) title "NP=16" with linesp ls 5, \
     "performance.dat" using 1:(log($7)) title "NP=32" with linesp ls 6
```

Run the following commands to execute the instructions in `performance.gnu`.

```
cd ~/UN5390_Sandbox
gnuplot performance.gnu
```

The generated plot, saved as `performance.eps`, should look as follows.

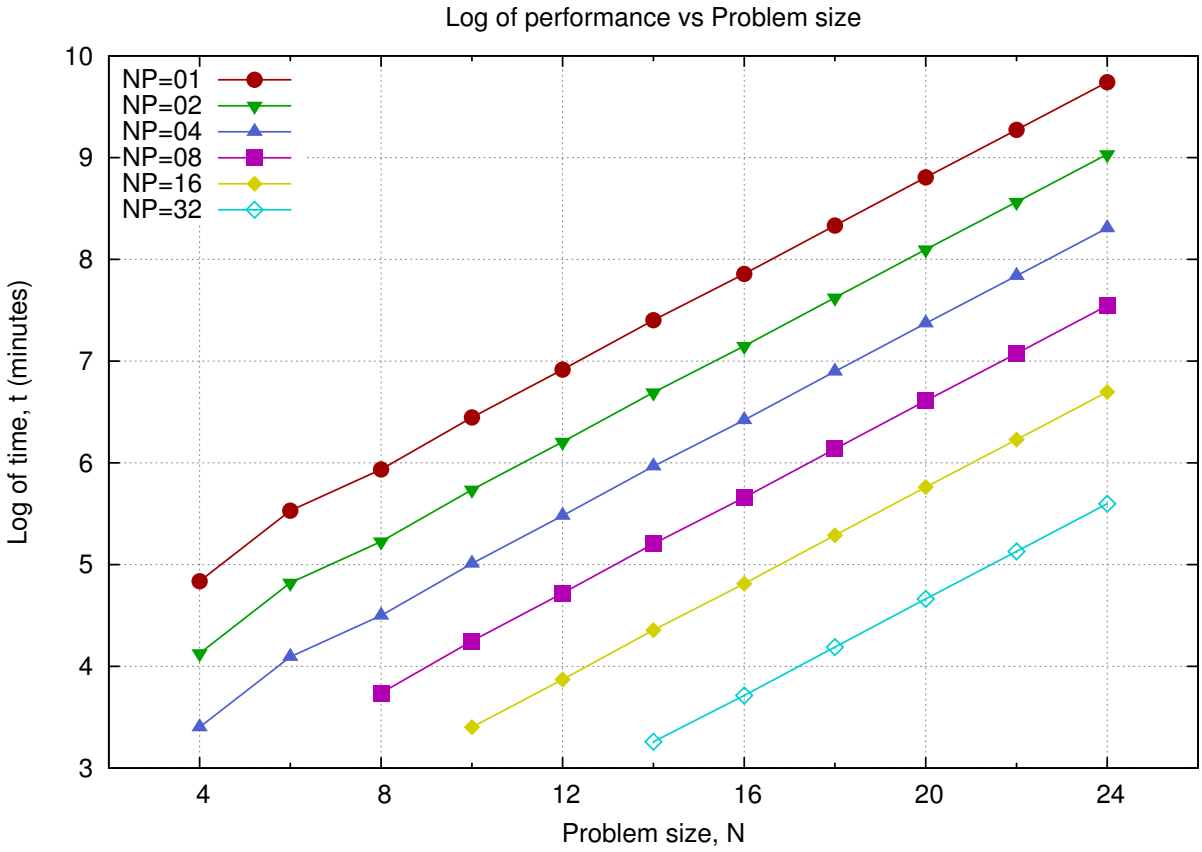

Figure 5: A meaningful caption for the Gnuplot graph

Refer to `man convert` and `man gnuplot` for more information, and http://www.gnuplot.info/, http://lowrank.net/gnuplot/index-e.html and http://www.gnuplotting.org/ for more practical examples.

Training Camp #08

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 41

[Functions - Introduction] Though it is easier to explicitly type three commands, doing so repeatedly – even for mundane entities such as home folder, shell, and the hostname – can become very tedious.

```
echo "${HOME}"
echo "${SHELL}"
echo "${HOSTNAME}"
```

Training Camp #04 included creating the ~/.bash_${USER} file. Not only it included aliases (i.e., shortcuts for long commands) but a user-defined function to display home folder, shell and hostname called hfsh. The following command will execute the function.

```
hfsh
```

Functions are one way of executing a series of commands without having to remember every one of them and/or their options. Given a meaningful name for easier remembrance, functions (along with scripts, discussed in a subsequent Training Camp) facilitate design and development of a workflow, and add to the user's arsenal of general/computational tools.

Task 42

[Functions - Count the number of login attempts in a workstation] The previous task introduced the concept of functions. Training Camp #05 introduced the concept and practice of piping (i.e., using output of one command as an input for the next without creating temporary files). Update ~/.bash_${USER} with the following function definition, and source ~/.bashrc.

```
# User-defined function to display a count of successful login attempts
login_counter() {
  echo
  echo "  Login counter as of $(date -R)"
  echo
  last | sed "/^\s*$/d" | awk '{ print $1 }' | sort | uniq -c | sort -nr
  echo
}
```

The following command will execute the function.

```
login_counter
```

Task 43

[Functions - Hello, User!] Functions, much like their portable counterparts – scripts and programs, have the capacity to accept one or more arguments (i.e., a string or a file/folder name, etc.) and act on them producing desired output. The first argument is contained in `$1`, the second in `$2`, ..., and all of them in `$*`. Update `~/.bash_${USER}` with the following function definition, and source `~/.bashrc`.

```
# User-defined function to display "Hello, SOME_NAME"
hello_user() {
  echo
  echo "  Hello, $1"
  echo
}
```

The following commands will execute the function for various number of arguments.

```
hello_user
hello_user Jane
hello_user Ada Steve
```

The function, `hello_user()`, works but does not include a mechanism to check if an argument was actually supplied. Update definition of this function in `~/.bash_${USER}` with an `if` construct to check the number of arguments (contained in `$#`), and source `~/.bashrc`.

```
# User-defined function to display "Hello, SOME_NAME"
hello_user() {
  echo
  if [ $# -ne 1 ]
  then
    echo "  Usage: $FUNCNAME STRING"
  else
    echo "  Hello, $1"
  fi
  echo
}
```

Now, the previous function calls will behave differently based on the number of arguments.

Task 44

[Functions - Convert seconds to human readable format] Suppose that a workflow requires that a given number of seconds (an integer) be converted to human readable format such as `hh:mm:ss` (hours, minutes and seconds). Update `~/.bash_${USER}` with the following function definitions, and source `~/.bashrc`. It is a common practice to use lower case alphabets to denote variables local to a function.

```
# User-defined function to validate non-negative integer input
validate_integer() {
  # Strip off all non-integer characters from original_string
  local original_string="$1"
  sanitized_string=$(echo "${original_string}" | sed 's/[^0-9]//g')

  # Compare original_string and sanitized_string, and return a value
  if [ "${original_string}" != "${sanitized_string}" ]
  then
    return 1 # Invalid user input
  else
    return 0 # Valid user input
  fi
}


# User-defined function to convert seconds to human readable format
# Incorporate a check for the number of arguments
seconds2hms() {
  # Save SECONDS in a local variable and validate it
  local seconds="$1"
  if ! validate_integer "${seconds}"
  then
    echo "  Invalid input. SECONDS must be a non-negative integer."
  else
    # Convert SECONDS to hh:mm:ss format and print the result
    ((hours=seconds/3600))
    ((minutes=seconds%3600/60))
    ((seconds=seconds%60))
    printf "%d:%02d:%02d\n" ${hours} ${minutes} ${seconds}
  fi
}
```

The following commands will execute the function for various number of arguments.

```
seconds2hms
seconds2hms -2 ; seconds2hms 3.14159 ; seconds2hms 35/44
seconds2hms CD ; seconds2hms a8p bgW-Sa823_KADfqs
seconds2hms 0 ; seconds2hms 1 ; seconds2hms 61 ; seconds2hms 3661
```

Task 45

[Functions - Smart extraction based on file name extension] Suppose that the workflow involves repetitive extraction of differently compressed entities. Keeping track of the necessary commands and their respective options can sometimes lead to an error. Update ~/.bash_${USER} with the following function definition, and source ~/.bashrc. Create a compressed entity of each kind using previous Training Camps, and use this function to perform smart extraction.

```
# User-defined function to perform smart extraction of compressed entity
extract () {
  if [ $# != 1 ]
  then
    echo "  Usage: extract COMPRESSED_ENTITY"
    exit
  fi

  # Include a check to see if the COMPRESSED_ENTITY exists before
  # attempting extraction. Switch-case construct (prevents deeply
  # nested if statements)
  case $1 in
    *.tar)       tar -xvf  $1  ;;
    *.tgz)       tar -xvzf $1  ;;
    *.tar.gz)    tar -xvzf $1  ;;
    *.tbz2)      tar -xvjf $1  ;;
    *.tar.bz2)   tar -xvjf $1  ;;
    *.gz)        gunzip    $1  ;;
    *.bz2)       bunzip2   $1  ;;
    *.zip)       unzip     $1  ;;
    *)           echo "  '$1' file type unknown" ;;
  esac
}
```

Training Camp #09

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 46

[Scripts - System details] The previous Training Camp introduced the concept of functions to minimize the need for remembering a series of commands (and their options) that constitute a workflow.

Scripts are different from functions in that they are independent (or standalone) entities and don't reside within another entity (for e.g., ~/.bash_${USER} or another script). Phrased alternatively, a function can reside within a script but not vice versa. Being standalone entities, scripts are more portable and easier to share.

Create the script, ~/UN5390_Sandbox/system_details.sh, with following content.

```
#! /bin/bash
#
# BASH script to extract and (formatted) display of system information.
#
# Usage:
# system_details.sh

# Collect and print system-specific information
OFFSET="25"
LINUX_OS=$(cat /etc/redhat-release | sed "s/^[ \t]*//;s/[ \t]*$//")
LINUX_KERNEL=$(uname -r)
CPU_MODEL=$(lscpu | grep "^Model name" | awk -F ':' '{ print $2 }' | \
            tr -s " " | sed "s/^[ \t]*//;s/[ \t]*$//")
CPU_COUNT=$(lscpu | grep "^CPU(s)" | awk -F ':' '{ print $2 }' | \
            sed "s/^[ \t]*//;s/[ \t]*$//")
RAM_TOTAL=$(free -g | sed "1d" | sed '$d' | awk '{ print $2 }' | \
             sed "s/^[ \t]*//;s/[ \t]*$//")

printf "%"${OFFSET}"s %s\n"  "Hostname :"     "${HOSTNAME}"
printf "%"${OFFSET}"s %s\n"  "Linux OS :"     "${LINUX_OS}"
printf "%"${OFFSET}"s %s\n"  "Linux kernel :" "${LINUX_KERNEL}"
printf "%"${OFFSET}"s %s\n"  "CPU model :"    "${CPU_MODEL}"
printf "%"${OFFSET}"s %s\n"  "CPU count :"    "${CPU_COUNT}"
printf "%"${OFFSET}"s %s\n"  "RAM :"          "${RAM_TOTAL} GB"
```

Running a script is very akin to running a command (or a program). Scripts, like commands (or programs), ought to have at least 700 permission (if not 750 or 755) to be executable.

```
cd ~/UN5390_Sandbox
chmod 700 system_details.sh

cd
pwd
system_details.sh

cd /tmp
pwd
system_details.sh

cd ~/UN5390_Sandbox
pwd
system_details.sh
```

Attempts to run `system_details.sh` should have resulted in the following error message.

```
-bash: system_details.sh: command not found
```

Now, try the following.

```
cd
pwd
~/UN5390_Sandbox/system_details.sh

cd /tmp
pwd
~/UN5390_Sandbox/system_details.sh

cd ~/UN5390_Sandbox
pwd
./system_details.sh
```

Did it work? If yes, understand why it worked.

Just about every command executed so far (e.g., `ls`, `pwd`, `shuf`, etc.) did not require explicit specification of the full path (absolute or relative). Linux OS looks for commands in the locations contained in the definition of `PATH` variable. If a similarly named command exists in two different locations, the one that's found first is used.

A normal user without `root` (or `sudo`) privileges cannot usually write outside of her/his `${HOME}` directory. But the ability to update the definition of `PATH` variable (carefully) to include custom locations is well within the reach of a normal user. In an attempt to accomplish this, `~/.bash_${USER}` was created in Training Camp #04 and the definition of

PATH variable was updated to include ${HOME}/bin. Now, move the script to ${HOME}/bin, and give it another try.

```
cd ~/
mkdir bin lib man src tmp
cd UN5390_Sandbox
mv system_details.sh ../bin/

cd
pwd
system_details.sh

cd /tmp
cd pwd
system_details.sh

cd ~/UN5390_Sandbox
pwd
system_details.sh

cd /usr/local/bin
pwd
system_details.sh
```

The script should have worked successfully even though its path (absolute or relative) was not explicitly specified. Storing the meaningfully named user-created scripts in ${HOME}/bin is a very good practice – it prevents duplication, and saves time and effort.

If several scripts share a given set of functions (e.g., input validation), it serves better to store them all in a separate file to avoid errors due to duplication (say, ~/bin/functions.sh). Sourcing it will automatically make all the functions available to the parent script.

```
source ~/bin/functions.sh
```

Task 47

[Scripts - Sum numbers] As also noted in the previous Training Camp, scripts have the ability to accept one or more arguments (i.e., a string of alpha-numeric characters) and act on them to produce desired output. The first argument is contained in `$1`, the second in `$2`, ..., and all of them in `$*`. `$#` contains the total number of arguments. Much like functions, one can ensure that a script is executed with exactly the appropriate number of arguments.

This script, `~/bin/sum_numbers.sh`, can be used to add all numbers in a given sequence (denoted by a starting number, increment and an ending number) and print the sum. It should work for integers and non-integers alike. How would one go about incorporating input validation for any number (and not just an integer)?

```
#! /bin/bash
#
# BASH script to add a series of given numbers and print the sum.
#
# Usage:
# sum_numbers.sh BEGIN STEP END

# Check the number of arguments
if [ $# -ne 3 ]
then
  echo "  Usage: $(basename $0) BEGIN STEP END"
  echo "   e.g.: $(basename $0) 1 1 10"
  echo "         $(basename $0) -6.50 0.10 5.50"
  exit
fi

# Save the arguments in variables and compute the sum
BEGIN="$1"
STEP="$2"
END="$3"
SUM=$(seq ${BEGIN} ${STEP} ${END} | \
      awk 'BEGIN { sum = 0} { sum += $1 } END { print sum }')

# Print the result
echo "  Begin : ${BEGIN}"
echo "  Step  : ${STEP}"
echo "  End   : ${END}"
echo "  Sum   : ${SUM}"
```

Task 48

[Scripts - Parse a file one line at a time] The following script, ~/bin/parse_line_by_line.sh, will read a given ASCII file one line at a time and process it for extracting meaningful results. Incorporate input validation (i.e., existence, non-zero size, commented lines and usefulness) of the ASCII file before processing it.

```
#! /bin/bash
#
# BASH script to read a given file, one line at a time, and parse the data.
# File format: STUDENT_NAME|TEST1_SCORE|TEST2_SCORE|TEST3_SCORE
#
# Usage:
# parse_line_by_line.sh FILENAME

# Check the number of arguments
if [ $# -ne 1 ]
then
  echo "  Usage: $(basename $0) FILENAME"
  echo "   e.g.: $(basename $0) student_scores.txt"
  exit
fi

# Read through the file one line at a time
echo "  ----------------------------------------"
printf "  %-20s  %3s  %3s  %3s  %6s\n" "Student" "#1" "#2" "#3" "Avg"
echo "  ----------------------------------------"

exec<$1
while read student_info
do
  NAME=$(echo ${student_info} | awk -F '|' '{ print $1 }')
  TEST1=$(echo ${student_info} | awk -F '|' '{ print $2 }')
  TEST2=$(echo ${student_info} | awk -F '|' '{ print $3 }')
  TEST3=$(echo ${student_info} | awk -F '|' '{ print $4 }')
  AVG=$(echo "scale=2; (${TEST1} + ${TEST2} + ${TEST3})/3" | bc)
  printf "  %-20s  %3d  %3d  %3d  %6.2f\n" "${NAME}" "${TEST1}" \
         "${TEST2}" "${TEST3}" "${AVG}"
done
echo "  ----------------------------------------"
```

Task 49

[Scripts - LaTeX compilation] Training Camp #07 demonstrated the act of compiling a LaTeX (`.tex`) to generate a PDF (`.pdf`). The following script, `~/bin/latex.sh`, will compile the `.tex` into a corresponding `.pdf` and delete the temporary files. Incorporate the previously learned input validation techniques before starting to attempt LaTeX compilation.

```
#! /bin/bash
#
# BASH script to run LaTeX and convert the DVI to PS, and PS to PDF.
#
# Usage:
# latex.sh LATEX_FILENAME

# Check the number of arguments
if [ $# -ne 1 ]
then
  echo "  Usage: $(basename $0) LATEX_FILENAME"
  echo "   e.g.: $(basename $0) latex_01.tex"
  exit
fi

# Extension for temporary files that need deletion
declare -a TMP_FILES=("acn" "acr" "alg" "aux" "bbl" "blg" "dvi"
  "fdb_latexmk" "glg" "glo" "gls" "idx" "ilg" "ind" "ist" "lof" "log" "lot"
  "maf" "mtc" "mtc0" "nav" "nlo" "out" "pdfsync" "ps" "snm" "synctex.gz"
  "toc" "vrb" "xdy" "tdo")

# Save the filename and extract the basename (i.e., without .tex)
# Run LaTeX, convert DVI to PS, and PS to PDF
BASENAME=$(echo "$1"  | awk -F '.' '{ print $1 }')
latex $1
dvips -Ppdf -o ${BASENAME}.ps ${BASENAME}.dvi
ps2pdf ${BASENAME}.ps ${BASENAME}.pdf

# Delete unnecessary files
for tmp in ${TMP_FILES[@]}
do
  rm -f ${BASENAME}.${tmp}
done
```

Task 50

[Scripts - Data backup and restoration] Suppose that a project is progressing at a consistently good pace. Further suppose that the workflow involves a periodic and thorough checking of all necessary entities – source code, scripts, functions, input/output/temporary files and folders, etc. – once per week (say, Saturday afternoon). It is important to ensure that a necessary entity isn't accidentally deleted from the workflow, or at least have a way of restoring it with minimal effort. Many institutions (including Michigan Tech) provide backup for the home folder by default. In the absence of such a service, developing and using a reliable data backup scheme can be very useful.

The following script, `~/bin/data_backup.sh`, performs a full (or *level 0*) backup of a specified folder (say, `~/UN5390_Sandbox`) to a specified location (say, `~/backup`) on Sunday, and an incremental backup on all other days. It is impractical to run this script manually. Explore the `cron` utility to schedule automatic backups at specified times.

```
#! /bin/bash
#
# BASH script to perform incremental and full backup of a specific folder.
#
# Usage:
# data_backup.sh

# Necessary variables
DAY_OF_WEEK=$(date +"%w")
DATA_SOURCE="${HOME}"           # Location that contains DATA_FOLDER
DATA_FOLDER="UN5390_Sandbox"
DATA_BACKUP="${HOME}/backup"    # Add a check if necessary

# Delete the existing full/incremental backup and create a new one
mkdir -p ${DATA_BACKUP}
cd ${DATA_SOURCE}
rm -f ${DATA_BACKUP}/${DATA_FOLDER}_${DAY_OF_WEEK}.tar.bz2
if [ ${DAY_OF_WEEK} -eq 0 ]
then
  tar --listed-incremental=${DATA_BACKUP}/snapshot.txt --level=0 -cvjf \
    ${DATA_BACKUP}/${DATA_FOLDER}_${DAY_OF_WEEK}.tar.bz2 ${DATA_FOLDER}
else
  tar --listed-incremental=${DATA_BACKUP}/snapshot.txt -cvjf \
    ${DATA_BACKUP}/${DATA_FOLDER}_${DAY_OF_WEEK}.tar.bz2 ${DATA_FOLDER}
fi
```

In order to test the script, one may explicitly set the DAY_OF_WEEK variable (to 0, 1, 2, ..., 6) and run the script after adding new content or editing the existing content. Once successfully completed, the following commands can be used to list the contents of full and incremental backups.

```
DATA_FOLDER="UN5390_Sandbox"
DATA_BACKUP="${HOME}/backup"
for dow in $(seq 0 1 6)
do
  echo
  echo "  Displaying the contents of backup level ${dow}"
  tar -tvjf ${DATA_BACKUP}/${DATA_FOLDER}_${dow}.tar.bz2
done
```

If/When there is a need to restore from the backup, the following script – ~/bin/data_restore.sh – can be used. Once the necessary missing/deleted entities have been successfully copied over to the working folder, delete the ${HOME}/restore folder.

```
#! /bin/bash
#
# BASH script to restore full and incremental backups of a specific folder.
#
# Usage:
# data_restore.sh

# Necessary variables
DAY_OF_WEEK=$(date +"%w")
DATA_FOLDER="UN5390_Sandbox"
DATA_RESTORE="${HOME}/restore"
DATA_BACKUP="${HOME}/backup" # Add a check if necessary

# Restore the full backup first, and then incremental backups, if any
mkdir -p ${DATA_RESTORE}
cd ${DATA_RESTORE}
for dow in $(seq 0 1 ${DAY_OF_WEEK})
do
  tar --listed-incremental=/dev/null -xvjf \
    ${DATA_BACKUP}/${DATA_FOLDER}_${dow}.tar.bz2
done
```

Training Camp #10

1. Have you completed the tasks in previous training camps?
2. Is your handwritten notes as complete as possible?
3. Are you using the same notebook for this training camp?

Task 51

[Git - Introduction, SSH keys and identity] There is almost always a need for a logical and consistent way to organize and track the revisions of documents of all kinds. The most commonly used approach to organize but not necessarily to track the changes is to keep multiple physical copies of a given document – named in some convenient way to uniquely identify a revision.

As an example, such an approach could often take one of the following forms over a period of time for a plain text document: `filename.txt`, `filename_1.txt`, `filename.txt.bak`, `filename.txt.YYYYMMDD`, `filename-john-edited.txt`, `filename-john-revised.txt`, `filename-final.txt`, `filename-published.txt` and so on. The lack of a simple way to not only revert back and forth between such revisions but compare them, and the potential of publishing an incorrect version makes it extremely unreliable.

Revision Control System (RCS) comes in two kinds: Concurrent Versions System (CVS) and Subversion (SVN) are examples of the centralized flavor while Bazaar, Darcs, Git and Mercurial are examples of the distributed flavor. Centralized flavors contain a project's history in one location while contributors' copy contains a single snapshot. As such, these come with the drawback of having a single point of failure. In distributed flavors, every copy of the project contains the full history. So, if the *central* copy were to go missing for some reason, any of the copies can be used to restore it. Designed with simplicity, speed, integrity, ability to handle large projects and with support for non-linear development, Git will be the distributed RCS of choice for UN5390.

Everything in Git is SHA1 check-summed (computed based on the contents of a file or folder structure) before it is stored and is then referred to by that checksum. As such, it is impossible to change the contents of any file or folder without Git knowing about it. The copy of the project (hereafter referred to as the *repository* or the *repo*) contained in GitHub will be treated as the *central* copy (or in other words, *the holiest of the holies*).

So far, the repository in GitHub is being accessed as if it is just any other website. In order to start using it as a repository from the command line, visit

https://github.com/settings/keys
⟶ New SSH key
⟶ Title Colossus (Michigan Tech)
⟶ Key Paste the contents of `${HOME}/.ssh/id_rsa.pub` in `colossus.it`
⟶ Add SSH key

Once successfully completed, run the following commands in a Terminal to setup the identity that will be used in every Git commit.

```
git config --global user.name "John Sanderson"
git config --global user.email "john@mtu.edu"
git config --global core.editor vim
git config --list
```

Run the following command to download `git-completion.bash`.

```
GURL="https://raw.githubusercontent.com/git/git/master/contrib/completion"
curl -o ~/.git-completion.bash ${GURL}/git-completion.bash
```

Add the following content at the very end of `~/.bash_${USER}` and source it.

```
# SSH agent
if [ -f "${HOME}/.ssh_agent.env" ]
then
  source "${HOME}/.ssh_agent.env" > /dev/null
  if ! kill -0 ${SSH_AGENT_PID} > /dev/null 2>&1
  then
    eval $(ssh-agent | tee "${HOME}/.ssh_agent.env")
    ssh-add
  fi
else
  eval $(ssh-agent | tee "${HOME}/.ssh_agent.env")
  ssh-add
fi

# git tab-enabled completion
if [ -f "${HOME}/.git-completion.bash" ]
then
  source "${HOME}/.git-completion.bash"
fi
```

Doing so (i.e., from setting up SSH keys to updating `~/.bash_${USER}`) once per all IT-managed workstations will suffice. The process needs to be repeated once per every other machine (that does not mount the campus home directory or a personal workstation/laptop) that will access GitHub.

Task 52

[Git - Cloning a repository and committing changes] Every registered student in UN5390 has read and write access to a repository named `un5390_public` in GitHub. Run the following commands in `colossus.it` to clone (i.e., *make a local copy*) this repository.

```
mkdir -p ~/git_work
cd ~/git_work
git clone git@github.com:MichiganTech/un5390_public.git
cd un5390_public
```

The following command will keep the local copy of the repository up to date.

```
git pull
```

It is an excellent practice to run this command before attempting to edit files and/or commit any changes back to the repository. Once a repository is cloned to a local workstation, adding/editing contents is usually no different than adding/editing contents in any other folder. The last command should have resulted in the following message. If not, repeat it.

```
Already up-to-date.
```

`git status` command should result in the following message.

```
# On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit (working directory clean)
```

This indicates that the working repository is clean. In other words, there are no tracked and modified files. Also, that Git doesn't notice any untracked files. Use the following commands to add content to the repository.

```
mkdir test_${USER}
touch test_${USER}.txt
```

`git status` will now result in the following message.

```
On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# test_${USER}.txt
nothing added to commit but untracked files present (use "git add" to track)
```

In order to start tracking the newly added content, the following commands can be used.

```
git add test_${USER} test_${USER}.txt
```

`git status` will now result in the following message indicating that `test_${USER}.txt` is now tracked and staged to be committed to the repository.

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   test_${USER}.txt
```

However, one may have noticed that the folder `test_${USER}` was not added or being tracked or staged to be committed. Git, by design, does not add or track empty folders. Should there be a need to add an empty folder, it is a common practice to add a hidden but empty file in that folder to fool Git into tracking it.

```
touch test_${USER}/.fooling_git
git add test_${USER}
```

Now, `git status` will show

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   test_${USER}.txt
# new file:   test_${USER}/.fooling_git
```

It is important to remember that Git adds entities to be tracked, staged and eventually committed to the repository in compliance with the contents of `.gitignore` file. Its template for various programming languages is available at https://github.com/github/gitignore. In order to commit the changes to the repository, run

```
git commit -m "Adding test_${USER} stuff"
```

In presence of a carefully crafted and maintained `.gitignore` file, one may combine staging and committing of tracked entities as follows.

```
git commit -am "Adding test_${USER} stuff"
```

Finally, the following command will push the commits made in the local workstation to the remote repository (i.e., on GitHub).

```
git push origin master
```

One may view these changes in GitHub. Now, delete the recently added content, commit changes to the repository, and view them in GitHub.

```
git pull
\rm -rf test_${USER}*
git status
git commit -am "Removing test_${USER} stuff"
git push origin master
```

Suppose that a *read only* file (agreed upon by honor system) was inadvertently edited. Git provides a quick and easy way to undo all changes prior to committing it to the repository.

```
git checkout -- FILENAME
```

In summary, the workflow involves following.

1. `cd FULL_PATH_TO_THE_REPOSITORY`
2. `git pull`
3. Add/Edit/Remove content or undo changes as necessary
4. Track previously untracked entities via `git add` or `git add -f`
   Stage and commit all tracked entities via `git commit -am "MESSAGE"`
5. Push to the remote repository via `git push origin master`

Git makes it extremely convenient to continue working and keep committing changes to the repository even when there is no active internet connection (e.g., traveling by car, bus, train, ship and plane OR in a park or a forest), and push to the remote repository once the connection does become available.

Task 53

[Git - Tagging a commit] Git provides the ability to mark (or *tag*) specific points in history as being important. This is quite often used to denote specific versions or releases of software (e.g., `vBeta`, `v1.0RC`, `v1.0`, `v1.2`, etc.). To list the available tags, one can use

```
git tag
```

In UN5390, assignments – when ready for final submission for grading – will need to be tagged as such. Specific instructions will accompany each assignment at an appropriate time but a general sequence of commands to create an annotated tag is as follows.

```
cd ~/git_work/un5390_public
git pull
git tag -a v${USER}_$(date +"%Y%m%d") -m "Test tag by ${USER}"
git push origin v${USER}_$(date +"%Y%m%d")
```

If there are several tags that need to be pushed to the remote repository at once, one can use the following command instead in lieu of the last one.

```
git push origin --tags
```

Task 54

[Git - Viewing the commit history] In the lifetime of a repository, there often comes a time to explicitly view the history of commits. One potential instance is to include the history as a part of the periodic progress report (to a funding agency) that is sponsoring the project. Within UN5390, the commit history is used to track students' progress and the timeliness with which assignments/projects are being worked on. Git provides a command to list the commit history, and `~/.bash_${USER}` includes an alias (from Training Camp #04).

```
cd ~/git_work/un5390_public
alias git_log
git_log
```

One can extract customized portions of the commit history (e.g., between two specified dates for a periodic progress report). It will serve well to create aliases (in `~/.bash_${USER}` or in Git) or functions to accomplish such tasks with ease and regularity.

```
git log --pretty=format:"%h %an %ad :  %s" --since='2016-05-01' \
  --before='$(date +"%Y-%m-%d")'
```

Task 55

[Git - Permanently deleting entities] Suppose that a large data file (or one with sensitive information) was accidentally added, tracked and committed to the repository.

```
cd ~/git_work/un5390_public
git pull
base64 /dev/urandom | head -c 25M > BigData_${USER}.dat
git add -f BigData_${USER}.dat
git commit -am "Adding BigData_${USER}.dat"
git push origin master
```

While deleting and committing changes to the repository will remove it from the repository, the document still remains in repository's history. The following workflow will be necessary to completely purge it (replace `john` with ISO username).

```
git filter-branch --force --index-filter \
  'git rm --cached --ignore-unmatch BigData_john.dat' \
  --prune-empty --tag-name-filter cat -- --all
git push origin master --force
rm -rf .git/refs/original/
git reflog expire --expire=now --all
git gc --prune=now
git gc --aggressive --prune=now
```

Task 56

[Git - Advanced concepts] Git includes additional features such as forking, branching and merging, creating and applying patches but are beyond the scope of this Training Camp, and will not be required in UN5390. In the absence of access to a Linux workstation, the following online resources may be used to learn more about and practice Git.

https://git-scm.com/doc | https://try.github.io/

It could be of interest to write thoughtful and well-commented BASH scripts that accept a specified number of arguments, if necessary, to save time and prevent potential mistakes while accomplishing frequent tasks such as adding, staging, committing and pushing to a remote repository (e.g., `~/bin/gpush.sh`), viewing the commit history (e.g., `~/bin/glog.sh`), tagging a commit (e.g., `~/bin/gtag.sh`), etc.

Task 57

[Exit codes] Suppose that a computational workflow (specifically, a function or a shell script) should perform a subsequent task/command iff the previous task/command was successfully completed. Such a scenario is not uncommon in even simplest of workflows. For e.g., LaTeX compilation – run `latex` once on `.tex` to produce `.dvi`; if successful, then run `dvips` once on `.dvi` to produce `.ps`; if successful, then run `ps2pdf` once on `.ps` to produce `.pdf`. One way to accomplish this would be write the output of a task to a file and check its contents before proceeding to the subsequent one.

Linux OS, however, provides a concept called *exit code* (or also known as *return value* or *exit status*). Every command upon execution returns an exit code: a number in the range `0` – `255`, and is stored in `$?`. A successful execution is associated with an exit code of 0 while an unsuccessful execution is associated with a non-zero number.

`0`	Success
`1`	Catchall for general errors (e.g., divide by zero)
`2`	Misuse of shell built-ins
`126`	Command invoked cannot execute (e.g., permission issue)
`127`	Command not found
`128`	Invalid argument to `exit` (e.g., `exit 355` or `exit 6.28`)
`128+n`	Fatal error signal `n` (e.g., `137` for `kill -9`)
`130`	Script terminated by `CTRL+C`
`64-113`	User-defined exit codes (by honor system)
`255`	Exit code out of range

C/C++ has systematized the exit status codes. `64` – `113`, reserved for user-defined codes, are usually sufficient to cover various possible scenarios. Though BASH does not yet impose such a restriction, it is a good practice to use it in scripts and functions. By extension, a function/script will have an exit status as well: that of the last command in it. Run `echo $?` after each of the following commands:

```
whoami
let "abc = 100/0"
null_function() {}
/dev/null
chmod 644 ~/bin/login_counter.sh ; ~/bin/login_counter.sh
chmod 755 ~/bin/login_counter.sh ; ~/bin/login_counter.sh
whoareyou
```

Create ~/bin/infinite_loop.sh, set its permission to 755 and run it.

```
#! /bin/bash
#
# BASH script with an infinite while loop.
#
# Usage:
# infinite_loop.sh

while [ 1 -lt 2 ]
do
  echo "  1 is less than 2"
  sleep 5
done
```

While it's running, press CTRL+C to terminate it. Run echo $? immediately afterward to identify the exit code. Run the script once more. Open another terminal and run the following commands to identify the process ID (PID) and terminate it.

```
PID=$(ps aux | grep "infinite_loop" | grep -v "grep" | awk '{ print $2 }')
kill -9 ${PID}
```

The last command should have terminated the infinite_loop.sh with the message, Killed. Run echo $? in that terminal and note that the error code corresponds to the fatal error signal as described previously.

Typeset the following as ~/bin/invalid_exitcode.sh, and set its permission to 755.

```
#! /bin/bash
#
# BASH script with an invalid exit code.
#
# Usage:
# invalid_exitcode.sh

date -R
exit 3.14159
```

Run the script, and echo $? immediately afterward should indicate that the exit code is out of range.

Typeset the following as ~/bin/return_value.sh, and set its permission to 755.

```
#! /bin/bash
#
# BASH script to demonstrate the concept of return value -- for a command as
# well as for a function and the script itself.
#
# Usage:
# return_value.sh

# Necessary variables
EXIT_SUCCESS=0           # Success
EXIT_CODE_ARG=64         # Incorrect number of arguments
EXIT_CODE_DATE=65        # Date command error
EXIT_CODE_HELLO_WORLD=66 # Hello_World() function error

# Hello_World() function
Hello_World() {
  echo "  Executing the function, ${FUNCNAME}"
  echo "  Hello, ${USER}!"
  # Simulate failure by uncommenting the following line
  # whoareyou
}
# Check the number of arguments
if [ $# -ne 0 ]
then
  echo "  Usage: $(basename $0)"
  exit ${EXIT_CODE_ARG}
fi

# Run the 'date' command. If successful, move forward
echo
date -R
if [ $? -eq 0 ]
then
  echo "  date command execution was successful."
  echo "  Moving on to Hello_World() function."
else
  echo "  date command execution failed. Exiting the script."
  exit ${EXIT_CODE_DATE}
fi
```

```
# Run Hello_World() function
Hello_World
if [ $? -eq 0 ]
then
  echo "  Hello_World() execution was successful."
  echo "  Nothing more to do."
  exit ${EXIT_SUCCESS}
else
  echo "  Hello_World() execution failed. Exiting the script."
  exit ${EXIT_CODE_HELLO_WORLD}
fi
```

Run the script with and without suggested edits in the comments, and run `echo $?` immediately following the script to observe the value of exit code.

Repeat the process for the following script – a modified version of ~/bin/latex.sh.

```
#! /bin/bash
#
# BASH script to run LaTeX and convert the DVI to PS, and PS to PDF.
#
# Usage:
# latex.sh LATEX_FILENAME

# Necessary variables
EXIT_CODE_ARG=64      # Incorrect number of arguments
EXIT_CODE_LATEX=65    # LaTeX error
EXIT_CODE_DVIPS=66    # DVIPS error
EXIT_CODE_PS2PDF=66   # PS2PDF error
EXIT_CODE_TMPDEL=67   # Temporary file deletion error

# Check the number of arguments
if [ $# -ne 1 ]
then
  echo "  Usage: $(basename $0) LATEX_FILENAME"
  echo "   e.g.: $(basename $0) latex_01.tex"
  exit ${EXIT_CODE_ARG}
fi

# Extension for temporary files that need deletion
declare -a TMP_FILES=("acn" "acr" "alg" "aux" "bbl" "blg" "dvi"
  "fdb_latexmk" "glg" "glo" "gls" "idx" "ilg" "ind" "ist" "lof" "log" "lot"
  "maf" "mtc" "mtc0" "nav" "nlo" "out" "pdfsync" "ps" "snm" "synctex.gz"
  "toc" "vrb" "xdy" "tdo")

# Save the filename and extract the basename (i.e., without .tex)
BASENAME=$(echo "$1"  | awk -F '.' '{ print $1 }')

# Run LaTeX
latex $1
if [ $? -gt 0 ]
then
  echo "  latex command failed. Exiting the script."
  exit ${EXIT_CODE_LATEX}
fi
```

```
# Convert DVI to PS
dvips -Ppdf -o ${BASENAME}.ps ${BASENAME}.dvi
if [ $? -gt 0 ]
then
  echo "  dvips command failed. Exiting the script."
  exit ${EXIT_CODE_DVIPS}
fi

# Convert PS to PDF
ps2pdf ${BASENAME}.ps ${BASENAME}.pdf
if [ $? -gt 0 ]
then
  echo "  ps2pdf command failed. Exiting the script."
  exit ${EXIT_CODE_PS2PDF}
fi

# Delete unnecessary files
for tmp in ${TMP_FILES[@]}
do
  rm -f ${BASENAME}.${tmp}
done
if [ $? -gt 0 ]
then
  echo "  Temporary file deletion failed. Exiting the script."
  exit ${EXIT_CODE_TMPDEL}
fi
```

Task 58

[Capturing commands and their output] `history` command, as noted so far, only keeps track of the commands (and when they were executed) and not the output of such commands. Linux OS provides a class of commands that can capture the output of a command and save it in a file. This can be especially useful when commands produce very long or verbose output and such output needs to be communicated with developers or a discussion forum.

`latex` and `dvips` commands should display verbose outputs en route to producing the PDF. One may use the `tee` command instead of explicitly copying and pasting the output in a file.

```
cd ${HOME}/UN5390_Sandbox
DATETIME=$(date +"%Y%m%d_%H%M%S")
latex latex_01.tex 2>&1 | tee latex_${DATETIME}.txt
dvips -Ppdf -o latex_01.ps latex_01.dvi 2>&1 | tee -a latex_${DATETIME}.txt
ps2pdf latex_01.ps latex_01.pdf 2>&1 | tee -a latex_${DATETIME}.txt
```

When used with the `-a` option, `tee` command appends the output to an existing file instead of overwriting it. `script` is another command that can be used for such a purpose.

```
cd ${HOME}/UN5390_Sandbox
DATETIME=$(date +"%Y%m%d_%H%M%S")
script -c 'uptime' script_${DATETIME}.txt
script -c 'who' -a script_${DATETIME}.txt
```

The above examples demonstrate using `script` to capture the output of a single command. It can also be used to capture the commands and their respective output in an entire session.

```
script --timing=script_timing.txt ~/UN5390_Sandbox/script_commands.txt
cd ~/UN5390_Sandbox
date -R
latex latex_01.tex
dvips -Ppdf -o latex_01.ps latex_01.dvi
ps2pdf latex_01.ps latex_01.pdf
exit
```

`scriptreplay` command can be used in conjunction with `script_timing.txt` can be used to replay the above sequence of commands.

```
scriptreplay --timing=script_timing.txt ~/UN5390_Sandbox/script_commands.txt
```

Refer to `man tee`, `man script` and `man scriptreplay` for more information. These commands, however, are not a substitute for keeping meticulous handwritten notes.

Task 59

[Miscellaneous commands] Explore the following commands if/when necessary, and refer to the respective `man` page for more information.

`COMMAND`	A short description
`ar`	Create, modify, and extract archives
`at`	Queue, examine or delete jobs for later execution
`column`	Columnate lists
`diff3`	Compare three files line by line
`dig`	DNS lookup utility
`dstat`	Generate system resource statistics
`expect`	Automate responses to interactive commands
`factor`	Display prime factors of a given integer
`host`	DNS lookup utility
`ip`	Show (or manipulate) networking information
`join`	Join lines of two files on a common field
`kill`	Terminate a process
`ldd`	Display shared library dependencies
`logsave`	Save the output of a command in a file
`look`	Display lines beginning with a given string
`lsof`	List open files
`nl`	Display a file with number of lines
`popd`	Return the path to the top of the directory stack
`ps`	Snapshot of current processes
`pstree`	Display a tree of processes
`pushd`	Save the path to current working directory
`rename`	Rename files
`stat`	Display file or file system status
`tac`	Concatenate and print files in reverse
`time`	Time a command or display resource usage
`timeout`	Run a command with a time limit
`uname`	Display system information
`uptime`	Display how long the system has been running
`watch`	Execute a command periodically
`zcmp`	Compare compressed files
`zdiff`	Compare compressed files
`zgrep`	Search a compressed file for a regular expression
`znew`	Recompress .Z files to .gz files

Task 60

[Computational workflow] While the definition can vary from one researcher to another, a general schematic representation of a computational workflow is depicted in Figure 6. Specific examples will be discussed throughout UN5390.

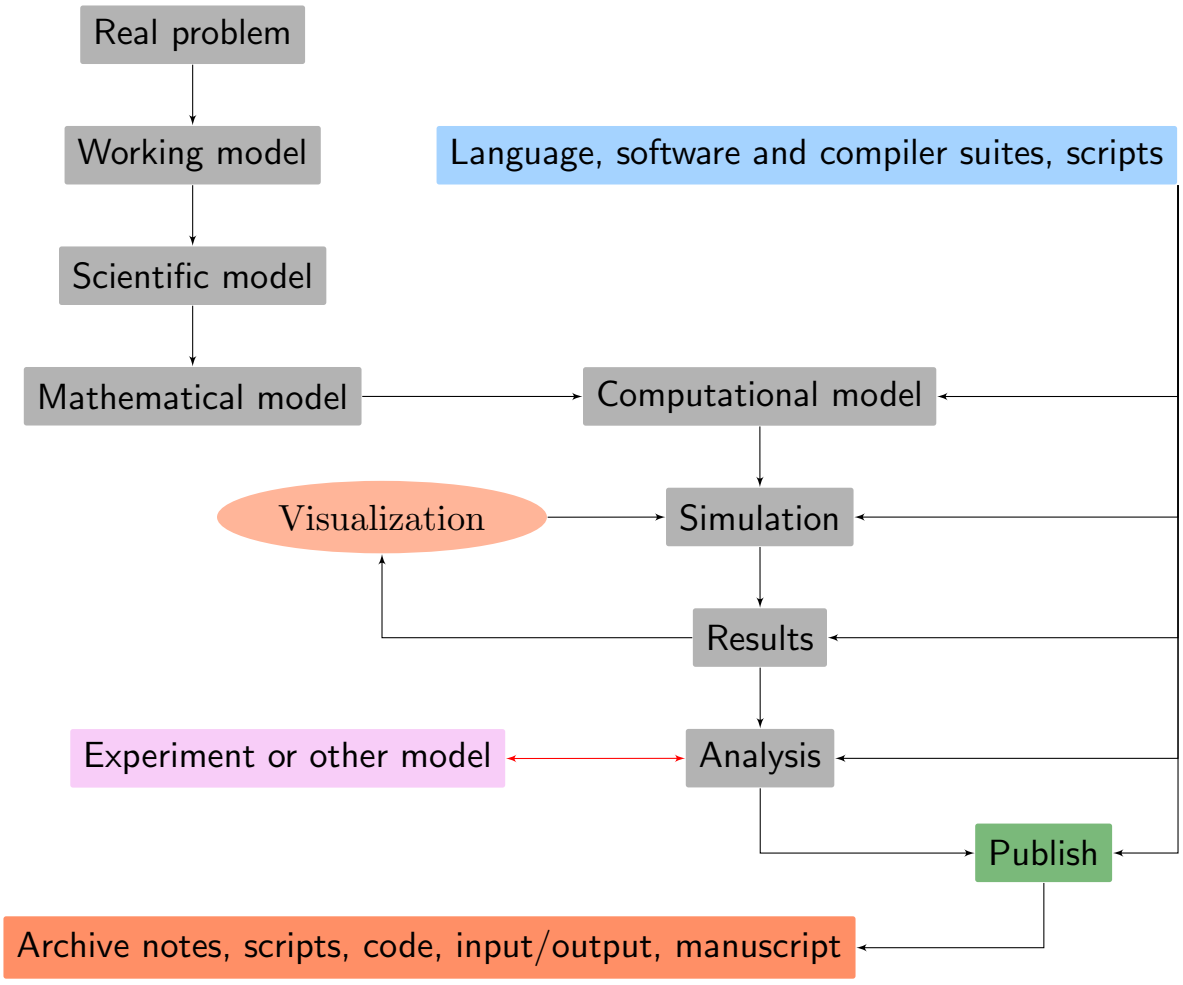

Figure 6: Schematic representation of a computational workflow

A proficiency in the Linux command line, one or more programming languages, LaTeX document preparation system and Git revision control system coupled with good written and verbal communication skills as well as file management techniques can help automate most of the routine and some not so routine tasks. They can also go a long way in enhancing one's mathematical/scientific/engineering prowess by creating time for such (and personal) activities in an otherwise hectic schedule.