WVU High Performance Computing Seminars

Guillermo Avendano-Franco

June 6, 2017

Contents

1 Linux, Command Line and HPC Environment (Newcomer)

1.1 Logging into remote systems

Currently WVU has two clusters for HPC, mountaineer and spruce. You can access them using SSH. SSH provides a secure channel over an unsecured network such as internet. Both Linux and macOS commonly include the SSH client by default. On Windows machines you can use a free application called PuTTY.

To connect to Mountaineer use:

```
ssh <username>@mountaineer.hpc.wvu.edu
```

For Spruce

```
ssh <username>@spruce.hpc.wvu.edu
```

Once you enter on the system, you can start typing commands. You can open several connections simultaneously. Each connection is independent of each other.

Power users can benefit from a terminal multiplexer such as tmux. tmux allows users to keep several virtual windows and panels open from a single connection. It offers also preserve the terminal status in case of disconnection from the server.

To use tmux, first connect to the server and execute the command

```
tmux
```

You can create new virtual windows with `CTRL-b c`, you move between windows with `CTRL-b n` and `CTRL-b p`. You can detach from your multiplexed terminals with `CTRL-b d`.

If for some reason you lost the connection to the server or you detached from the mulpiplexer all that you have to do to reconnect is to execute the command:

```
tmux a
```

There are many options for using tmux, see the cheat cheat for some of them.

1.2 Top 10 commands to learn

When you interact with a HPC cluster your interaction is basically by executing commands on a terminal and editing text files. For newcomers using command lines could be a frustrating experience knowing that there are literally hundreds of commands. Certainly

there are manuals for most of those commands, but they are of no use if you do not know which is the command you need to use for each situation. The good news is that you can do a lot of things with just a bunch of them and you can learn others in due time.

This is a selection of the 10 most essential commands you need to learn.

1.2.1 ls

List all the files in a directory. Linux as many Operating Systems organize files in files and directories (also called folders).

```
$ ls
file0a  file0b  folder1  folder2 link0a  link2a
```

Some terminal offer color output so you can differentiate normal files from folders. You can make the difference more clear with this

```
$ ls -aCF
./  ../  file0a  file0b  folder1/  folder2/ link0a@  link2a@
```

You will see a two extra directories `"."` and `".."`. Those are special folders that refer to the current folder and the folder up in the tree. Directories have the suffix `"/"`. Symbolic links, kind of shortcuts to other files or directories are indicated with the symbol `"@"`.

Another option to get more information about the files in the system is:

```
$ ls -al
total 36
drwxrwxr-x.  4 gufranco users     86 May 30 12:16 .
drwxr-xr-x. 82 gufranco users  12288 May 30 12:05 ..
-rw-rw-r--.  1 gufranco users      0 May 30 12:08 file0a
-rw-rw-r--.  1 gufranco users      0 May 30 12:08 file0b
drwxrwxr-x.  2 gufranco users     32 May 30 12:07 folder1
drwxrwxr-x.  2 gufranco users     32 May 30 12:07 folder2
lrwxrwxrwx.  1 gufranco users      6 May 30 12:16 link0a ->
   file0a
lrwxrwxrwx.  1 gufranco users     14 May 30 12:16 link2a ->
   folder2/file2a
```

Those characters on the first column indicate the permissions. The first character will be "d" for directories, "l" for symbolic links and "-" for normal files. The next 3 characters are the permissions for "read", "write" and "execute" for the owner. The next 3 are for the group, and the final 3 are for others. The meaning of "execute" for a file indicates that the file could be a script or binary executable. For a directory it means that you can see its contents.

1.2.2 cp

This command copies the contents of one file into another file. For example

```
$ cp file0b file0c
```

1.2.3 rm

This command deletes the contents of one file. For example

```
$ rm file0c
```

There is no such thing like a trash folder on a HPC system. Deleting a file should be consider an irreversible operation.

Recursive deletes can be done with

```
$ rm -rf folder_to_delete
```

Be extremely cautious deleting files recursively. You cannot damage the system as the files that you do not own you cannot delete. However, you can delete all your files forever.

1.2.4 mv

This command moves a files from one directory to another. It also can be used to rename files or directories.

```
$ mv file0b file0c
```

1.2.5 pwd

It is easy to get lost when you move in complex directory structures. pwd will tell you the current directory.

```
$ pwd
/home/gufranco/Dropbox/SummerHandsOn
```

1.2.6 cd

This command moves you to the directory indicated as an argument, if no argument is given, it returns to your home directory.

```
$ cd folder1
```

1.2.7 cat and tac

When you want to see the contents of a text file, the command cat displays the contents on the screen. It is also useful when you want to concatenate the contents of several files.

```
$ cat INCAR
system    =  LiAu
PREC      =  High
NELMIN    =  8
NELM      =  100
EDIFF     =  1E-07
...
```

To concatenate files you need to use the symbol `">"` indicating that you want to redirect the output of a command into a file

```
$ cat file1 file2 file3 > file_all
```

The command tac shows the files in reverse starting from the last line back to the first one.

1.2.8 more and less

Sometimes text files, as those created as product of simulations are too large to be seen in one screen, the command "more" shows the files one screen at a time. The command `"less"` offers more functionality and should be the tool of choice to see large text files.

```
$ less OUTCAR
```

1.2.9 ln

This command allow to create links between files. Used wisely could help you save time when traveling frequently to deep directories. By default it creates hard links. Hard links are like copies, but they make references to the same place in disk. Symbolic links are better in many cases because you can cross file systems and partitions. To create a symbolic link

```
$ ln -s file1 link_to_file1
```

1.2.10 grep

The grep command extract from its input the lines containing a specified string or regular expression. It is a powerful command for extracting specific information from large files. Consider for example

```
$ grep TOTEN OUTCAR
  free energy    TOTEN  =        68.29101273 eV
```

```
free energy    TOTEN  =       -13.46870926 eV
free energy    TOTEN  =       -18.78141268 eV
...
```

Regular expressions offers ways to specified text strings that could vary in several ways and allow commands such as grep to extract those strings efficiently. We will see more about regular expressions in third chapter.

1.2.11 More commands

The 10 commands above, will give you enough tools to move files around and travel the directory tree. There are more commands summarized

Output of entire files	
cat	Concatenate and write files
tac	Concatenate and write files in reverse
nl	Number lines and write files
od	Write files in octal or other formats
base64	Transform data into printable data

Formatting file contents	
fmt	Reformat paragraph text
numfmt	Reformat numbers
pr	Paginate or columnate files for printing
fold	Wrap input lines to fit in specified width

Output of parts of files	
head	Output the first part of files
tail	Output the last part of files
split	Split a file into fixed-size pieces
csplit	Split a file into context-determined pieces

Summarizing files	
wc	Print newline, word, and byte counts
sum	Print checksum and block counts
cksum	Print CRC checksum and byte counts
md5sum	Print or check MD5 digests
sha1sum	Print or check SHA-1 digests
sha2 utilities	Print or check SHA-2 digests

Operating on sorted files	
sort	Sort text files
shuf	Shuffle text files
uniq	Uniquify files
comm	Compare two sorted files line by line
ptx	Produce a permuted index of file contents
tsort	Topological sort

Operating on fields	
cut	Print selected parts of lines
paste	Merge lines of files
join	Join lines on a common field

Operating on characters	
tr	Translate, squeeze, and/or delete characters
expand	Convert tabs to spaces
unexpand	Convert spaces to tabs

Directory listing	
ls	List directory contents
dir	Briefly list directory contents
vdir	Verbosely list directory contents
dircolors	Color setup for 'ls'

Basic operations	
cp	Copy files and directories
dd	Convert and copy a file
install	Copy files and set attributes
mv	Move (rename) files
rm	Remove files or directories
shred	Remove files more securely

Special file types	
link	Make a hard link via the link syscall
ln	Make links between files
mkdir	Make directories
mkfifo	Make FIFOs (named pipes)
mknod	Make block or character special files
readlink	Print value of a symlink or canonical file name
rmdir	Remove empty directories
unlink	Remove files via unlink syscall

Changing file attributes	
chown	Change file owner and group
chgrp	Change group ownership
chmod	Change access permissions
touch	Change file timestamps

Disk usage	
df	Report file system disk space usage
du	Estimate file space usage
stat	Report file or file system status
sync	Synchronize data on disk with memory
truncate	Shrink or extend the size of a file

Printing text	
echo	Print a line of text
printf	Format and print data
yes	Print a string until interrupted

Conditions	
false	Do nothing, unsuccessfully
true	Do nothing, successfully
test	Check file types and compare values
expr	Evaluate expressions
tee	Redirect output to multiple files or processes

File name manipulation	
basename	Strip directory and suffix from a file name
dirname	Strip last file name component
pathchk	Check file name validity and portability
mktemp	Create temporary file or directory
realpath	Print resolved file names

Working context	
pwd	Print working directory
stty	Print or change terminal characteristics
printenv	Print all or some environment variables
tty	Print file name of terminal on standard input

User information	
id	Print user identity
logname	Print current login name
whoami	Print effective user ID
groups	Print group names a user is in
users	Print login names of users currently logged in
who	Print who is currently logged in

System context	
arch	Print machine hardware name
date	Print or set system date and time
nproc	Print the number of processors
uname	Print system information
hostname	Print or set system name
hostid	Print numeric host identifier
uptime	Print system uptime and load

Modified command	
chroot	Run a command with a different root directory
env	Run a command in a modified environment
nice	Run a command with modified niceness
nohup	Run a command immune to hangups
stdbuf	Run a command with modified I/O buffering
timeout	Run a command with a time limit

Process control	
kill	Sending a signal to processes

Delaying	
sleep	Delay for a specified time

Numeric operations	
factor	Print prime factors
seq	Print numeric sequences

1.3 Text Editors

1.4 Working with a HPC cluster

1.5 Managing inputs and outputs

1.6 Transferring files between systems

2 Scientific Workflows (Building, HPC Running, and post-processing)

2.1 Building/installing software

2.2 Using Python packages (pip and virtualenv)

2.3 Basic Scripting

2.4 Managing the 3 HPC variables (cores, memory and time)

2.5 Job submission, monitoring, debugging and optimization

2.6 Chaining Job with dependencies

2.7 Executing many jobs at one, job arrays

2.8 Parallel jobs (MPI)

2.9 Plotting (gnuplot, xmgrace and matplotlib)

3 A glimpse on advanced topics

3.1 Advanced Bash/Python Scripting

Here we will cover a few more advanced topics not covered on our introductory scripting session.

3.1.1 Regular Expressions with python

Consider the following challenge. We have the output from a simulation with some data that we would like to process, the problem now is that the data is not on a single like, so a simple grep will not work. The data we want to parse looks like this:

```
...
     14        7.7300       0.00000
     15        7.9145       0.00000
     16        8.7421       0.00000

 k-point 115 :       0.4444    0.3636    0.4286
  band No.  band energies     occupation
      1       -6.7076       1.00000
      2       -6.6256       1.00000
      3       -3.8932       1.00000
      4       -3.8031       1.00000
      5        0.1344       1.00000
      6        0.4871       1.00000
      7        0.7520       1.00000
      8        1.1131       1.00000
      9        3.2272       1.00000
     10        3.3574       1.00000
     11        7.4689       0.00000
     12        7.4905       0.00000
     13        7.7325       0.00000
     14        7.9343       0.00000
     15        8.3742       0.00000
     16        8.7648       0.00000

 k-point 116 :       0.0000    0.4545    0.4286
```

```
  band No.  band energies     occupation
      1        -7.0118       1.00000
      2        -6.8668       1.00000
      3        -4.4179       1.00000
...
```

This is quite complex set of data and we would like to take the different elements in such a way that we can manipulate them later on.

There are several ways of solving this problem, for example, knowing that each block of data starts with "k-point" and spans 17 rows. Such task could be done using just grep

```
grep -A 17 k-point OUTCAR
```

The argument "-A 17" will tell grep to show 17 rows after each occurrence of the line k-point. We extract the line of information that we need but still is just a piece of text that is not so simple to manipulate.

With python we can achieve this task with just 4 lines, using the so called regular expressions, a way to explain a computer that we want extract text with some format by indicating the kind of data that we expect on the text.

This following script will extract the pieces still as text but, we will work on the conversion to text later.

```
import re
rf = open('OUTCAR')
data = rf.read()
kp = re.findall('k-point([\d\s]*):([\d\s.]*)band[\.\s\w]*
    occupation([\s\d:\-\.]*)\n\n', data)
```

The most cryptic part of this small script is understanding the meaning of all those symbols used as arguments for the findall function. Lets start with a simpler version of the findall line an we will understand it piece by piece.

Using IPython lets start with executing the first 3 lines and we will explore the findall function step by step

```
import re
rf = open('OUTCAR')
data = rf.read()
```

Now, we start exploring this line

```
re.findall('k-point[\d\s]*:', data)
```

The output will look like this:

```
['k-point   1 :',
 'k-point   2 :',
 'k-point   3 :',
 'k-point   4 :',
```

The line in findall can be read like this: Search for text that start with “k-point” followed by 0 or more (that is the meaning of ’*’) groups of characters (what is enclose by “[” and “]”) that can be either numbers ”\d” or characters that looks like spaces “\s”

Now, if we just need the number, we can enclose the information that findall will return by enclosing it in parenthesis, like this:

```
re.findall('k-point([\d\s]*):', data)
```

At this point could be interesting to show how we can convert the list of strings returned by findall into actual numbers. This could be done like this:

```
[int(x) for x in re.findall('k-point([\d\s]*):', data)]
```

Now lets move forward and get the next piece of information, the three numbers after colon, the numbers before the word ’band’

```
re.findall('k-point([\d\s]*):([\d\s.]*)band', data)
```

As you can see we are getting more information this time

```
[('   1 ', '       0.0000     0.0000     0.0000\n  '),
 ('   2 ', '       0.1111     0.0000     0.0000\n  '),
 ('   3 ', '       0.2222     0.0000     0.0000\n  '),
 ('   4 ', '       0.3333     0.0000     0.0000\n  '),
 ('   5 ', '       0.4444     0.0000     0.0000\n  '),
 ('   6 ', '       0.0000     0.0909     0.0000\n  '),
 ('   7 ', '       0.1111     0.0909     0.0000\n  '),
 ('   8 ', '       0.2222     0.0909     0.0000\n  '),
...
```

The output is a list of tuples, each tuple consisting of two strings. There is just one extra character on the regular expression, dot “.” is added to cover the existence of that character in the 3 numbers after colon. In regular expressions ”dot” is used to match any character except a newline. Inside the “[]” special characters lose their special meaning, so “dot” here means just a “.”.

Now we can go to our final version of the findall function

```
re.findall('k-point([\d\s]*):([\d\s.]*)band[\s\w.]*occupation([\s
    \d:.\-]*)\n\n', data)
```

The meaning of all this cryptic code become far more clear now, the only notice here is that the character minus “-” needs still to be escaped like “\-” because it has a meaning for ranges inside “[]”. We close the regular expression with a double \n\n, indicating that each block is separated by a double newline.

Out final task is to convert the output from findall into actual numbers such that we can manipulate them for whatever purpose we need.

Lets do first a more simple exercise by storing correctly the k-point number and the three numbers after colon, they are the positions but their actual meaning is not important here.

```
ret=[]
for ikp in kp:
    entry={}
    entry['number']= int(ikp[0])
    entry['position']= [float(x) for x in ikp[1].split()]
    entry['values']= len(ikp[2].split())
    ret.append(entry)
```

What we are doing here is creating a list called **ret** and for each element in our list kp we will create a python dictionary, converting the elements from the tuple into numbers, the first one will be integer, the second one is a set of three floating point numbers and for the third one we will just split the string into words and count the elements.

```
[{'number': 1, 'position': [0.0, 0.0, 0.0], 'values': 48},
 {'number': 2, 'position': [0.1111, 0.0, 0.0], 'values': 48},
 {'number': 3, 'position': [0.2222, 0.0, 0.0], 'values': 48},
 {'number': 4, 'position': [0.3333, 0.0, 0.0], 'values': 48},
 {'number': 5, 'position': [0.4444, 0.0, 0.0], 'values': 48},
 {'number': 6, 'position': [0.0, 0.0909, 0.0], 'values': 48},
...
```

For the position we use a list comprehension, a syntactic construct available in some programming languages for creating a list based on existing lists.

The conversion of values is a bit more elaborated. First, notice that the final element contain 49 elements due to a final string with several dashes. We would like to extract the numbers that are really relevant the floating point numbers. Lets consider just the final element from kp

```
In [44]: kp[-1]
Out[44]:
(' 120 ',
 '       0.4444     0.4545     0.4286\n  ',
 ' \n       1       -6.6226       1.00000\n       2       -6.5937
    1.00000\n       3       -3.7536       1.00000\n       4
   -3.7218       1.00000\n       5       -0.0498       1.00000\n
    6       0.0803       1.00000\n       7       0.5565
   1.00000\n       8       0.6896       1.00000\n       9
   3.3146       1.00000\n       10       3.3603       1.00000\n
   11       7.6585       0.00000\n       12       7.6740
   0.00000\n       13       7.9721       0.00000\n       14
   8.0356       0.00000\n       15       8.8014       0.00000\n
   16       8.9382       0.00000\n\n\n
   ------------------------------------------------------------------------------------------------------------------------
   n')
```

Using Numpy we can easily get the information converted ready easily. Consider this line

```
import numpy
np.array(kp[-1][2].split()[:48], dtype=float).reshape(-1,3)
```

This line can be readed like this. Take the last element in kp (kp[-1]). Now take the third element of the tuple (kp[-1][2]). Split the string in words and make a list with the first 48 words encountered

```
kp[-1][2].split()[:48]
```

The final step is to convert those 48 strings into numbers as floating point numbers and reshape the whole array in 3 columns.

The final version of this part of the script will look like this:

```
ret=[]
for ikp in kp:
    entry={}
    entry['number']= int(ikp[0])
    entry['position']= [float(x) for x in ikp[1].split()]
    entry['values']= np.array(ikp[2].split()[:48], dtype=float).
   reshape(-1,3)
    ret.append(entry)
```

For reasons that will become clearer later we would like to keep everything as simple lists of numbers rather than numpy arrays. So we will serialize the numpy array into a list of lists

```
ret=[]
for ikp in kp:
    entry={}
    entry['number']= int(ikp[0])
    entry['position']= [float(x) for x in ikp[1].split()]
    entry['values']= np.array(ikp[2].split()[:48], dtype=float).
   reshape(-1,3).tolist()
    ret.append(entry)
```

It is time for us to save the data that we parse in something that allow us to recover later. There are several ways to store python objects into files. One way is using JSON, another is using pickle

Right now, the variable ret is a list of dictionaries where each of them contains either single numbers, lists or lists of lists. We can store that in a JSON file such that we can recover that information easily.

JSON is a lightweight data interchange format inspired by JavaScript object literal syntax. The JSON module in python offers convenient functions to convert simple variables

such as list and dictionaries into strings that could be stored in text files such that their contents could be easily retrieved.

Try first executing something like this

```
import json
json.dumps(ret)
```

Not easy to read for a human but that long string can be easily understood by a computer to recover the information you stored in it. Try this version for something clearer to read

```
import json
json.dumps(ret, sort_keys=True, indent=4, separators=(',', ': '))
```

Lets now store ret into a file and read it again to test we can recover the file.

```
wf = open('k-points.json','w')
dp = json.dump(ret, wf, sort_keys=True, indent=4, separators
   =(',', ': '))
wf.close()
```

Now lets test recovering the data from the file.

```
rf2=open('k-points.json')
json.load(rf2)
```

Finally, lets summarize all that we learn with this example. The whole script will be listed here:

```
1  import re
2  import numpy as np
3  import json
4
5  rf = open('OUTCAR')
6  data = rf.read()
7
8  # Parsing of the data
9  kp=re.findall('k-point([\d\s]*):([\d\s.]*)band[\s\w.]*occupation
       ([\s\d:.\-]*)\n\n', data)
10
11 # Giving structure to the data
12 ret=[]
13 for ikp in kp:
14     entry={}
15     entry['number']= int(ikp[0])
16     entry['position']= [float(x) for x in ikp[1].split()]
17     entry['values']= np.array(ikp[2].split()[:48], dtype=float).
       reshape(-1,3).tolist()
```

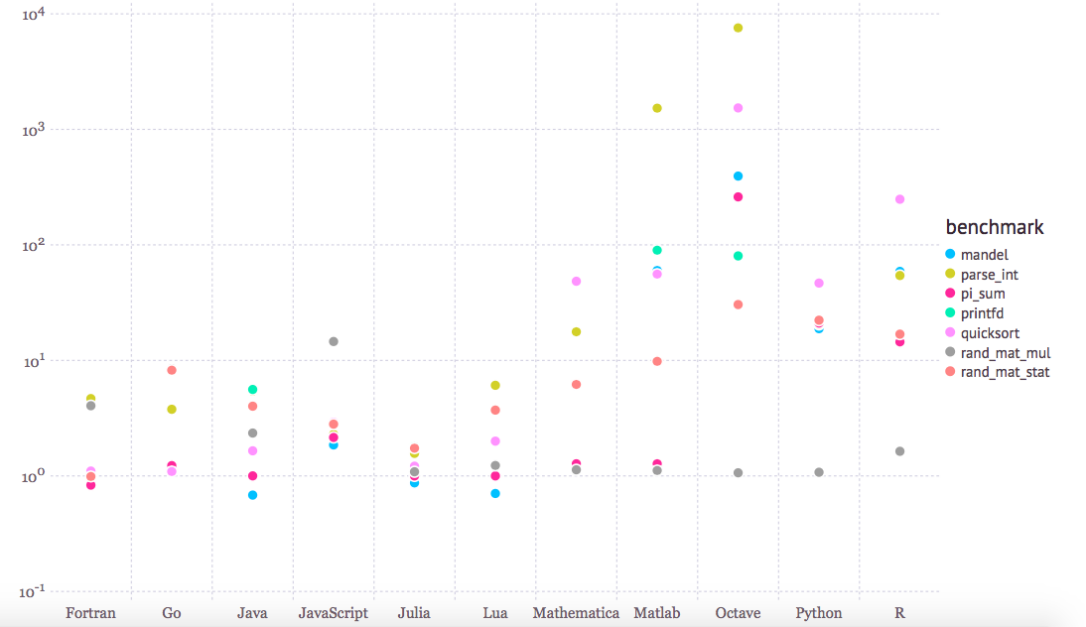

Figure 3.1: Comparison of performance for several computing languages. Benchmark times relative to C (smaller is better, C performance = 1.0). Source: https://julialang.org/benchmarks/.

```
18      ret.append(entry)
19
20  # Storing the results into a JSON file
21  wf = open('k-points.json','w')
22  dp = json.dump(ret, wf, sort_keys=True, indent=4, separators=(','
        , ': '))
23  wf.close()
```

3.2 Programming in C, Fortran and Python

Learn a new programming language takes time and goes beyond we can pretend here. It is not only the actual knowledge of the syntax and grammar of the language is also an understanding of the expressiveness of each language for describing operations to a computer. There are 3 full featured programming languages in use today for scientific computing. Fortran, C and Python.

There are more specialized languages, such as R, Julia. But we will try to keep the things simple here by showing how the same task is programmed in the 3 languages we have selected. See for example 3.2 for a comparison on the performance of those different

languages.

I am taking these examples from http://rosettacode.org. To give you a flavor of what is the feeling writing code in those 3 languages I have selected 2 tasks and showing how the solution is express in those languages.

3.2.1 Sieve of Eratosthenes

The Sieve of Eratosthenes is a simple algorithm that finds the prime numbers up to a given integer.

Lets start with the implementation in C. I am selecting not the most optimized version, but the simplest implementation for pedagogical purposes. The idea is to get a flavor of the language.

```c
#include <stdio.h>
#include <stdlib.h>

void sieve(int *, int);

int main(int argc, char *argv[])
{
  int *array, n;

  if ( argc != 2 ) /* argc should be 2 for correct execution */
    {
      /* We print argv[0] assuming it is the program name */
      printf( "usage: %s max_number\n", argv[0] );
    }
  else
    {
      n=atoi(argv[1]);
      array =(int *)malloc(sizeof(int));
      sieve(array,n);
    }
  return 0;
}

void sieve(int *a, int n)
{
  int i=0, j=0;

  for(i=2; i<=n; i++) {
    a[i] = 1;
  }
```

```
  for(i=2; i<=n; i++) {
    printf("\ni:%d", i);
    if(a[i] == 1) {
      for(j=i; (i*j)<=n; j++) {
  printf ("\nj:%d", j);
  printf("\nBefore a[%d*%d]: %d", i, j, a[i*j]);
  a[(i*j)] = 0;
  printf("\nAfter a[%d*%d]: %d", i, j, a[i*j]);
      }
    }
  }

  printf("\nPrimes numbers from 1 to %d are : ", n);
  for(i=2; i<=n; i++) {
    if(a[i] == 1)
      printf("%d, ", i);
  }
  printf("\n\n");
}
```

This example shows the basic elements from the c language, the creation of variables, loops and conditionals. The inclusion of libraries and the printing on screen.

You can compile this code using the code at

```
Day3_AdvancedTopics/2.Programming
```

```
gcc sieve.c -o sieve
```

and execute like this

```
./sieve 100
```

Now lets consider the Fortran version of the same problem.

```
module str2int_mod
contains

  elemental subroutine str2int(str,int,stat)
    implicit none
    ! Arguments
    character(len=*),intent(in) :: str
    integer,intent(out)         :: int
    integer,intent(out)         :: stat
```

```
    read(str,*,iostat=stat)  int
  end subroutine str2int

end module

program sieve

  use str2int_mod
  implicit none

  integer :: i, stat, i_max=0
  logical, dimension(:), allocatable :: is_prime
  character(len=32) :: arg

  i = 0
  do
    call get_command_argument(i, arg)
    if (len_trim(arg) == 0) exit

    i = i+1
    if ( i == 2 ) then
       call str2int(trim(arg), i_max, stat)
       write(*,*) "Sieve for prime numbers up to", i_max
    end if

  end do

  if (i_max .lt. 1) then
     write (*,*) "Enter the maximum number to search for primes"
     call exit(1)
  end if

  allocate(is_prime(i_max))

  is_prime = .true.
  is_prime (1) = .false.
  do i = 2, int (sqrt (real (i_max)))
    if (is_prime (i)) is_prime (i * i : i_max : i) = .false.
  end do
  do i = 1, i_max
    if (is_prime (i)) write (*, '(i0, 1x)', advance = 'no') i
  end do
```

```
  write (*, *)

end program sieve
```

You can notice the particular differences of this language compared with C, working with arrays is in general easier with Fortran.

You can compile this code using the code at

```
Day3_AdvancedTopics/2.Programming
```

```
gfortran sieve.f90 -o sieve
```

and execute like this

```
./sieve 100
```

Finally, this is the version of the Sieve written in python

```
#!/usr/bin/env python

from __future__ import print_function
import sys

def primes_upto(limit):
    is_prime = [False] * 2 + [True] * (limit - 1)
    for n in range(int(limit**0.5 + 1.5)): # stop at ``sqrt(limit
    )``
        if is_prime[n]:
            for i in range(n*n, limit+1, n):
                is_prime[i] = False
    return [i for i, prime in enumerate(is_prime) if prime]

if __name__=='__main__':

    if len(sys.argv)==1:
        print("Enter the maximum number to search for primes")
        sys.exit(1)
    limit = int(sys.argv[1])
    primes = primes_upto(limit)
    for i in primes:
        print(i, end=' ')
    print()
```

Python is an interpreted language so you do not need to compile it, instead directly execute the code at:

```
Day3_AdvancedTopics/2.Programming
```

using the command line:

```
python sieve.py 100
```

3.3 Version control with Github

The tutorial about Git will be based on the repository created at:

```
https://github.com/guilleaf/TutorialGitAutotools
```

3.4 Optimization, Profiling and Debugging

3.5 Parallel programming with MPI and OpenMP

Parallel programming is essential in High-Performance Computing. Computers nowadays are not increasing speed as they use to years ago. Instead, they increase the number of cores. Modern HPC clusters are now build from several nodes, with several processors each and with several cores each processor. Those processing capabilities are complemented by adding GPU and Co-processors such as Xeon Phi.

For this tutorial we will consider two popular alternatives for parallel computing, both OpenMP and MPI offers ways of execute calculations concurrently on several cores. An application can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes.

We will explore those two kinds of parallel programming alternatives with a few examples each.

3.5.1 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.

The basic idea is to write special comments called “pragmas” that will be interpreted by the compiler when you compile the code with some special argument. In most cases the code can compile just fine without the extra argument and it will work as a serial code, using just one core.

Lets start with the usual hello program. This is the implementation in C

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {
```

```
  int nthreads, tid;

  /* Fork a team of threads with each thread having a private tid
    variable */
#pragma omp parallel private(tid)
  {

    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0)
      {
  nthreads = omp_get_num_threads();
  printf("Number of threads = %d\n", nthreads);
      }

  }  /* All threads join master thread and terminate */

}
```

You compile it with the command

```
gcc -fopenmp omp_hello.c -o hello
```

The version in Fortran is:

```
program hello

  integer nthreads, tid, omp_get_num_threads, omp_get_thread_num

  !fork a team of threads with each thread having a private tid
   variable
  !$omp parallel private(tid)

  !obtain and print thread id
  tid = omp_get_thread_num()
  print *, 'Hello world from thread = ', tid

  !only master thread does this
  if (tid .eq. 0) then
     nthreads = omp_get_num_threads()
```

```
    print *, 'Number of threads = ', nthreads
  end if

  !all threads join master thread and disband
  !$omp end parallel

end program hello
```

You compile it with the command

```
gcc -fopenmp omp_hello.f90 -o hello
```

When you execute the program you will see messages comming from the different threads, all the program runs on the same machine but it creates threads to concurrently execute the section enclosed by the `"#pragma"` or `"!$omp"` blocks.

You can control the number of threads using

```
export OMP_NUM_THREADS=3
```

3.5.2 MPI

3.6 Creating Python Modules

3.7 Test Driven Development

3.8 Continuous Integration