WVU Summer Hands-on Introduction to HPC

Guillermo Avendano-Franco

June 12, 2017

Contents

Schedule

Day 1 - June 12, 2017	
09:00 - 10:00	Login into remote systems (SSH, PuTTY and tmux)
10:00 - 11:00	Command Line Interface
11:00 - 12:00	Text Editors (nano, emacs and vi)
12:00 - 13:00	Lunch Break
13:00 - 13:30	Using Environmental Modules (module)
13:30 - 15:30	Torque and Moab (qsub, qstat, qdel, checkjob)
15:30 - 16:00	Transfering files (rsync and globus)

Day 2 - June 13, 2017	
09:00 - 10:00	Shell Scripting (including grep)
10:00 - 11:00	Python Scripting
11:00 - 12:00	Using pip and virtualenv
12:00 - 13:00	Lunch Break
13:00 - 14:00	Plotting (gnuplot and matplotlib)
14:00 - 15:00	Building Software (example with fftw)
15:00 - 16:00	Creating Environmental Modules

Day 3 - June 14, 2017	
09:00 - 10:00	Advanced Scripting (regular expressions)
10:00 - 11:00	Programming in C, Fortran and Python I
11:00 - 12:00	Programming in C, Fortran and Python II
12:00 - 13:00	Lunch Break
13:00 - 14:00	Parallel Programming (OpenMP)
14:00 - 15:00	Parallel Programming (MPI)
15:00 - 15:30	Test Driven Development (Python nose)
15:30 - 16:00	Version Control with Git

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

1.1 Shell Scripting

1.2 Python Scripting

1.3 Using Python pip and virtualenv

1.4 Plotting (gnuplot, xmgrace and matplotlib)

1.5 Building/installing software

Sometimes, the modules available on the system are not enough or you need to compile the code by yourself to get some extra functionality not present on the current modules.

For this tutorial we will go on the whole process of compiling a code by yourself. We have selected fftw a well known library for computing Fast Fourier Transforms.

First, we go to the webpage of the FFTW code `http://www.fftw.org`

Before downloading the code, create a directory on your home folder suitable for compiling codes, for example `"$HOME/local/src"` and go into such directory

Go to downloads and copy the link to the code that we will download and copy the link.

On the terminal execute:

```
wget http://www.fftw.org/fftw-3.3.6-pl2.tar.gz
```

Now that you have downloaded the code uncompress it using the command line

```
tar -zxvf fftw-3.3.6-pl2.tar.gz
```

Now go into that directory. There is usually a file `README` or `INSTALL`. Those files will give you instructions on how to compile the code.

It is a good idea to create a directory for building the code. Here we will use `build_gcc`. Go into that directory and execute:

```
../configure --help
```

You will see all the options available for configure the code. System administrators are usually conservative when choosing options for compilation, usually shifting the preference towards stability rather than performance. Consider for example the case where we want the long-double precision library rather than the original double precision. The configuration line will be like this:

```
../configure --prefix=$HOME/local --enable-long-double
```

The next step is compile the code with

```
make
```

It is always good practice to test the compilation. Good software comes with tests that compare results with known results.

```
make check
```

Finally, the installation is done with:

```
make install
```

Now, we have fftw for long-double precision compiled and installed at `$HOME/local`. Check by yourself the existence of folders such as lib and include, they contain both the libraries and headers needed to compile other programs using the library you just compiled.

We have a small program to test the FFT library we just compiled. The code is as follows:

```
#include <stdio.h>
#include <math.h>
#include <fftw3.h>

#define NUM_POINTS 10000
#define REAL 0
#define IMAG 1

void create_input(fftwl_complex* signal) {
  /* The input is a sum of several cosines and sines with
   different frequencies
   * and amplitudes
   */
  int i;

  printf("Creating a signal with precision LONG DOUBLE (sizeof
   =%lu Bytes)\n", sizeof(long double));

  for (i = 0; i < NUM_POINTS; ++i) {
    long double theta = (long double)i / (long double)
   NUM_POINTS * 2 * M_PI;

    signal[i][REAL] = 1.0 * cos(10.0 * theta) +
      2.0 * cos(20.0 * theta) +
      3.0 * cos(30.0 * theta) +
      4.0 * cos(40.0 * theta) +
      5.0 * cos(50.0 * theta);

    signal[i][IMAG] = 1.0 * sin(10.0 * theta) +
      2.0 * sin(20.0 * theta) +
      3.0 * sin(30.0 * theta) +
      4.0 * sin(40.0 * theta) +
      5.0 * sin(50.0 * theta);
  }
}

void print_magnitude(fftwl_complex* result, FILE *fp) {
  int i;
  for (i = 0; i < NUM_POINTS; ++i) {
    long double mag = sqrt(result[i][REAL] * result[i][REAL] +
         result[i][IMAG] * result[i][IMAG]);
    fprintf(fp,"%34.25Le %34.25Le %34.25Le\n", result[i][REAL],
    result[i][IMAG], mag);
  }
}

int main() {
  FILE *fp;
  fftwl_complex signal[NUM_POINTS];
  fftwl_complex result[NUM_POINTS];

  fftwl_plan plan = fftwl_plan_dft_1d(NUM_POINTS,
              signal,
              result,
              FFTW_FORWARD,
              FFTW_ESTIMATE);

  create_input(signal);
  printf("Saving input signal...\n");
  fp = fopen("Input_FFT.dat", "w");
  print_magnitude(signal, fp);
  fclose(fp);

  fftwl_execute(plan);

  printf("Saving FFT from signal...\n");
  fp = fopen("Output_FFT.dat", "w");
  print_magnitude(result, fp);
  fclose(fp);

  fftwl_destroy_plan(plan);
  return 0;
}
```

To compile this code you have to be very explicit on the locations of the libraries and headers because they are no included in the environmental variables that gcc uses to search for them. The compilation line will be:

```
gcc example_fftl.c -I$HOME/local/include -L$HOME/local/lib -
   lfftw3l -lm
```

When output is not declared like above, the executable is a file called `a.out`. We have the advantage that we produce a static library for the long-double precision version of FFTW.

The library is `libfftw3l.a`. The big advantage of static libraries is that the application can be certain that all its libraries are present and that they are the correct version. Being static for FFTW, our example simply runs with:

```
./a.out
```

You can check library dependencies executing

```
ldd ./a.out
```

Once executed you will have 2 files: `Input_FFT.dat` and `Output_FFT.dat`, those files contain the original signal and its Fourier-Transform function. We create a small python script to help you visualize both functions. The functions are in complex space, so you are drawing Real and Imaginary parts and the magnitude of the signal.

In the next section we will see how we can create modules that will facilitate compilation and execution of binaries that need these libraries.

1.5.1 Exercises:

- Compile the FFTW in their single precision (float) and double precision versions. You can use the same place `$HOME/local` as prefix for your installing the libraries. It is always good idea to clean the build directory before trying to compile a new version.
- Compile FFTW enabling the build of shared libraries. Try to compile the same code using them. Check dependencies with `ldd` and notice how the execution is not longer possible without explictly adding the environmental variable `LD_LIBRARY_PATH` pointing to the location of the libraries.

1.6 Creating Environmental Modules

Environment Modules (EM) provides a way for the dynamic modification of a user's environment via modulefiles. The Environment Modules package is a tool that simplify shell initialization and lets users easily modify their environment during the session. To achieve its goal, EM uses files called modules located on special locations, you can load and unload modules, changing the environment variables.

For this tutorial we will create a private repository for your own modules and we will create one module for the library we just create in our previous section.

The firs step is decide a place where we we locate the modules. For this tutorial we will use `$HOME/local/modules`. The location is arbitrary as soon as you can write in that directory. The first step is to create that directory.

The next step is to setup the variable `MODULEPATH` on your `.bashrc` pointing to the place where you will add your modules. Open you favorite text editor and edit your `$HOME/.bashrc`. You can add this at the very end of the file. Assuming you use bash:

```
export MODULEPATH=$HOME/local/modules:$MODULEPATH
```

In order to make effective the changes you need to "source" the file. incorporating the changes in your current session.

```
source $HOME/.bashrc
```

Now it is time to create your first module.

```
#%Module1.0#########################################
## Fast Fourier Transform Library
##

module-whatis "Name: fftw"
module-whatis "Version: 3.3.6"
module-whatis "Category: C subroutine library"
module-whatis "Description: Library for computing the discrete
   Fourier transform (DFT)"
module-whatis "URL: http://fftw.org/"

set  prefix     <ENTER_YOUR_HOME_DIR_HERE>/local

# This is used during compilation for searching for libraries
prepend-path  LIBRARY_PATH        ${prefix}/lib

# This is used during execution for searching for libraries
prepend-path  LD_LIBRARY_PATH     ${prefix}/lib

# This is used during compilation for searching headers (*.h
   and *.mod)
prepend-path  CPATH               ${prefix}/include

# These two are usual places where man pages and info pages are
    located
prepend-path  INFOPATH            ${prefix}/share/info:
prepend-path  MANPATH             ${prefix}/share/man

# This is a search path for searching for executables
prepend-path  PATH                ${prefix}/bin

# This is a helper tool used when compiling applications and
   libraries.
# It helps you insert the correct compiler options on the
   command line
prepend-path  PKG_CONFIG_PATH     ${prefix}/lib/pkgconfig
```

2 A glimpse on advanced topics

2.1 Advanced Bash/Python Scripting

Here we will cover a few more advanced topics not covered on our introductory scripting session. The examples here are oriented to usual operations that need to be done with the text output from several research codes.

2.1.1 Extracting text from output files (awk and grep)

Consider the data at `Day3_AdvancedTopics/1.Scripting`. There is a compressed file called `goldnano3cu.tgz`. Uncompress the tar file with

```
tar -zxvf goldnano3cu.tgz
```

The folders and files are from actual simulations of gold nanoclusters. You can see a sample from the first 10 simulations. The actual simulation was done with actually several hundred files like those.

First, have a look to one of those files. For example `goldnano3cu/0/output.log`. Use `less` or your favorite text editor to get an idea about how the file actually looks. With less you can search for words typing / and the pattern. Search for etot on that file.

The first challenge is to know the value of the total energy per atom. You can see that such information is contained on lines such as

```
Time step =    232 SCF step =   3 etot/atom =    -277.072096
```

So you can use grep to extract those lines from the output

```
grep "etot/atom" goldnano3cu/0/output.log
```

grep is a tool to extract lines from text output using patterns to identify the lines that we need to be shown.

The output will look like this

```
...
   Time step =    251 SCF step =   3 etot/atom =    -277.070767
   Time step =    252 SCF step =   3 etot/atom =    -277.070811
   Time step =    253 SCF step =   3 etot/atom =    -277.070817
   Time step =    254 SCF step =   4 etot/atom =    -277.070855
   Time step =    255 SCF step =   3 etot/atom =    -277.070890
   Time step =    256 SCF step =   3 etot/atom =    -277.070922
   Time step =    257 SCF step =   3 etot/atom =    -277.070953
   Time step =    258 SCF step =   3 etot/atom =    -277.070978
```

You can see the file again and noticing that after that line there are more information that could be interesting to see. Execute for example:

```
grep -A 9 "etot/atom" goldnano3cu/0/output.log
```

Now, we are capturing the lines with `'etot/atom'` and the next 9 lines AFTER. You can also use `-B` to get lines before the pattern.

```
grep -B 1 -A 9 "etot/atom" goldnano3cu/0/output.log
```

The pattern so far is just a fixed string. grep was created to use far more complex patterns called regular expressions.

Consider for example that we are searching for lines with the word "Time"

```
grep "Time" goldnano3cu/0/output.log
```

But, now we realize that we do not want those lines that start with EBS, so we can indicate grep that the pattern matches only the word "Time" at the beggining of the line with some arbitrary number of spaces.

```
grep "^ *Time" goldnano3cu/0/output.log
```

The following list shows the basic elements for regular expressions.

```
. (dot) - a single character.
?       - the preceding character matches 0 or 1 times only.
*       - the preceding character matches 0 or more times.
+       - the preceding character matches 1 or more times.
{n}     - the preceding character matches exactly n times.
{n,m}   - the preceding character matches at least n times
          and not more than m times.
[agd]   - the character is one of those included
          within the square brackets.
[^agd]  - the character is not one of those included
          within the square brackets.
[c-f]   - the dash within the square brackets operates
          as a range. In this case it means either the
          letters c, d, e or f.
()      - allows us to group several characters to behave
          as one.
|       - the logical OR operation.
^       - matches the beginning of the line.
$       - matches the end of the line.
```

Regular expressions is whole subject by itself and widely used not only by grep, but also many other languages such as awk, perl and python. We will continue the exploration using AWK.

AWK is a programming language designed for text processing and typically used as a data extraction and reporting tool. In the 1990s, Perl became very popular, competing with AWK in the niche of Unix text-processing languages. Nowadays is Python the most popular language for many purposes including text processing, however for some basic operations is far easier to use a awk line or small script rather than a python one.

Lets go back to our grep command for extracting the `'etot/atom'`, lets use for example a line like this:

```
grep "^ *Time" goldnano3cu/0/output.log
```

Lets suppose that we would like to get just the number of the iteration and the energy, for example to plot the two columns later on. You can use grep and awk connected with a pipe to produce the result. For example

```
grep "^ *Time" goldnano3cu/0/output.log | awk '{print $4, $11}'
```

What we are doing here is letting grep to extract the lines that we want and using awk to just print the 4th and 11th column from those lines. Notice the "," after $4 otherwise the two numbers will be stick together.

This is a very popular combination, using grep for line extraction and awk to print the output needed.

AWK is able to reproduce the grep execution by itself, for example:

```
awk '(/etot\/atom/) {print $0}' goldnano3cu/0/output.log
```

And the extraction of the iteration number and energy can be done with:

```
awk '(/etot\/atom/) {print $4, $11}' goldnano3cu/0/output.log
```

One of the advantages of AWK is that variables are converted on the fly based on use. That allow us to change the value of the energy if we need for example converted to different units. For example:

```
awk '(/etot\/atom/) {n=$11; print n*0.0367493, "Ha"}'
   goldnano3cu/0/output.log
```

This line also shows how you can create variables and use it them to manipulate the output.

At this point is good to see AWK as an old fashioned C-like scripting language. Take for example this script:

```
#!/bin/awk -f
BEGIN {
# Print the squares from 1 to 10 the first way
    i=1;
    printf("Squares:\n")
    while (i <= 10) {
  printf("The square of %3d is %3d\n", i, i*i);
  i = i+1;
    }
# do it again, using more concise code
    printf("Cubes:\n")
    for (i=1; i <= 10; i++) {
  printf("The cube of %3d is %4d\n", i, i**3);
    }
# now end
    exit;
}
```

Lets go back to the output of the simulation. Imagine now that you would like to know the minimum, maximum and average energy per atom. The next script in AWK will do that:

```
#/usr/bin/awk -f
BEGIN {
    emin=10000;
    emax=-10000;
    eavg=0.0
    nele=0
}
(/etot\/atom/){
    etot_atom = $11
    if ( etot_atom < emin ) {
  emin = etot_atom;
    }
    if ( etot_atom > emax ) {
  emax = etot_atom;
    }
    eavg+=etot_atom;
    nele+=1
}
END {
    printf("Minimum: %f Maximum: %f Average: %f\n", emin, emax,
    eavg/nele);
}
```

You can use it like this:

```
awk -f awk_min_max_avg.awk goldnano3cu/0/output.log
```

In fact you can go further and process all the outputs from the 10 simulations like this:

```
awk -f awk_min_max_avg.awk goldnano3cu/*/output.log
```

AWK is actually reading more than 7 million lines of output, extracting the relevant lines and computing the maximum, minimum and average from them.

2.1.2 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:

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

import re
import numpy as np
import json

rf = open('OUTCAR')
data = rf.read()

# Parsing of the data
kp=re.findall('k-point([\d\s]*):([\d\s.]*)band[\s\w.]*
   occupation([\s\d:.\-]*)\n\n', data)

# Giving structure to the data
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)

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

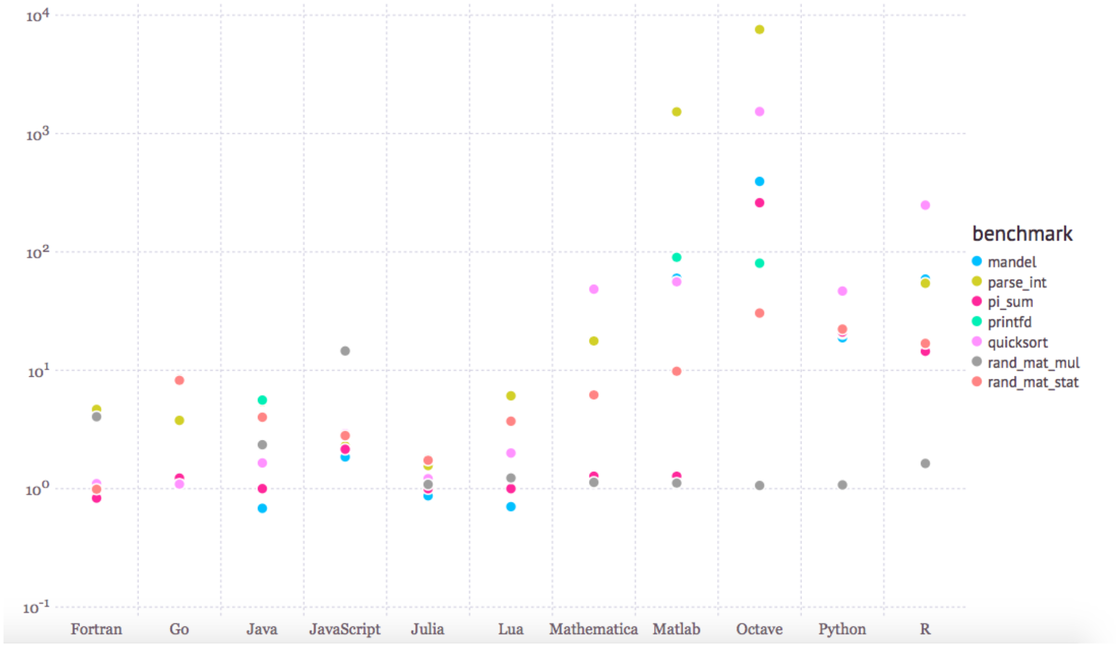

Figure 2.1: Comparison of performance for several computing languages. Benchmark times relative to C (smaller is better, C performance = 1.0). Source: https://julialang.org/benchmarks/.

2.2 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 2.2 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.

2.2.1 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
```

2.2.2 Matrix inversion

The purpose here is not to show the algorithm behind matrix inversion but to show how that could be achieve in several programming languages using external libraries, in particular we will show you the problem solved using LAPACK for Fortran, GSL for C and Numpy for python

Lets start with the Fortran version. BLAS and LAPACK are a set of well known libraries to perform Linear Algebra calculations. This is a simple example of inverting a real matrix.

```
program inverse_matrix
  implicit none
  double precision, allocatable, dimension(:,:) :: a, ainv
  double precision, allocatable, dimension(:) :: work
  integer :: i,j, lwork

  integer :: info, lda, m, n
  integer, allocatable, dimension(:) :: ipiv

  integer deallocatestatus
  character(len=15) :: mformat='(100(E14.6,1x))'

  external dgetrf
  external dgetri

  n = 4
  lda = n
  lwork = n*n
```

```
allocate (a(lda,n))
allocate (ainv(lda,n))
allocate (work(lwork))
allocate (ipiv(n))

call random_seed()
call random_number(a)

print '(" ")'
print*,"LU matrix:"
do i = 1, n
   write(*,mformat) (a(i,j), j = 1, n)
end do

print '(" ")'
! dgetrf computes an lu factorization of a general m-by-n
 matrix a
! using partial pivoting with row interchanges.

m=n
lda=n

! store a in ainv to prevent it from being overwritten by
 lapack
ainv = a

call dgetrf( m, n, ainv, lda, ipiv, info )

if(info.eq.0)then
   print '(" LU decomposition successful ")'
endif
if(info.lt.0)then
   print '(" LU decomposition:  illegal value ")'
   stop
endif
if(info.gt.0)then
   write(*,'(a,i4)') 'LU decomposition return',info
endif

print '(" ")'
print*,"LU matrix:"
do i = 1, n
   write(*,mformat) (ainv(i,j), j = 1, n)
end do

!  dgetri computes the inverse of a matrix using the lu
```

```
   factorization
  !  computed by dgetrf.
  call dgetri(n, ainv, n, ipiv, work, lwork, info)

  print '(" ")'
  if (info.ne.0) then
     stop 'Matrix inversion failed!'
  else
     print '(" Inverse successful ")'
  endif

  print '(" ")'
  print*,"Inverse matrix:"
  do i = 1, n
     write(*,mformat)(ainv(i,j), j = 1, n)
  end do

  print '(" ")'

  deallocate (a, stat = deallocatestatus)
  deallocate (ainv, stat = deallocatestatus)
  deallocate (ipiv, stat = deallocatestatus)
  deallocate (work, stat = deallocatestatus)

  print '(" done ")'
  print '(" ")'

  stop
end program inverse_matrix
```

2.3 Introduction to Parallel Programming

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.

2.4 Parallel Programming with 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
```

2.5 Parallel Programming with MPI

2.6 Test Driven Development

2.7 Version control with Git

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

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