# Computational Physics – Lecture 1

Andrei Alexandru

August 28, 2017

## 1 Introduction

The purpose of this course is to discuss a number of numerical methods that are useful in solving Physics, Engineering and Applied Mathematics problems. We plan to cover a good fraction of methods that are frequently encountered in these fields. These methods are well established and have a solid theoretical foundation. Our focus will be less on the theoretical underpinnings and more on learning how to use these methods to solve concrete problems.

The course is project driven and we will discuss a new topic every week. To help you understand the material and to provide an assessment of your progress, you will have to solve a homework each week. The homework will be motivated by a Physics problem and will rely on the numerical methods covered in class. Most of the problems will be connected to mechanical systems but a few will be motivated by Statistical Mechanics and Electricity and Magnetism problems. We will review the required physics background for each homework.

The tools we plan to use in this course are C and Mathematica. We will use C to handle the low-level, computationally demanding tasks and Mathematica to assist with debugging our codes, analyze the results, create plots and animations, etc. We plan to use C since it is a high-level programming language that offers a rather accurate computational model, that is, the abstractions used to define computation in C are very similar to the way the computer operates. This makes it possible to write very efficient codes, which is required when dealing with numerically demanding tasks.

Mathematica will serve a complementary role. This is a high level language that makes it easy to plot, fit, and analyze data and develop and test models of moderate complexity. Mathematica provides easy access to high quality implementations for most standard numerical tasks, arbitrary precision arithmetic, symbolic computation, many interesting databases, etc. We will introduce these features in class and you are expected to use them to prepare your reports.

Another goal in this course is to get you familiar with the workflow required to produce scientific reports. For each homework set, you will be asked to write a report in LaTeX, a tool that will allow you to typeset equations, include graphs to support your presentation, and create a bibliography. An effective workflow depends on your ability to easily connect these tools. We will use Linux in this class and, for most tasks, we will use a simple text terminal. We encourage students to bring their own laptops and configure them appropriately. To help with this task we provide a virtual machine image properly configured.

## 2 Getting your machine ready

The tasks for the first week is to get a simple C code compiled, write a simple code looping over a data set, print out a conversion table and plot this using Mathematica. At the end of this week you should be able to:

- navigate through the file system from the terminal,
- create and edit a text file using `nano`,
- compile a simple C code using `gcc`,
- read in data in a C code and print it out as text (`scanf` and `printf`),

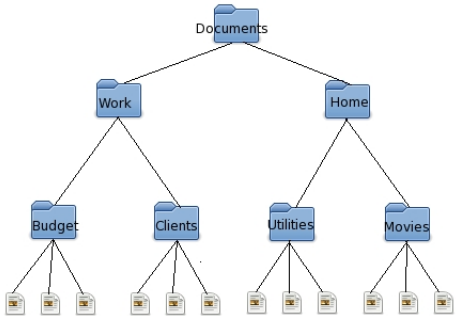


Figure 1: Example file tree structure

– use `for` to loop over a code section,

– read in data in Mathematica and plot it,

– put together a report in LaTeX.

The instructions below have been tested in the virtual machine provided. If you have your own Linux installation, similar steps should be followed. Ask for assistance if you run into any problems.

To set up we will boot up the virtual machine, log in and open a terminal. When you open the terminal the current directory is set to your home directory. You are probably familiar with the idea that the files stored on your machine live in folders and that the folders can store not only files but also other folders. In most cases this structure can be viewed as a tree, where a folder is represented by a node and its contents, files and folders, are viewed as branches. The files sit at the end of these braches (we can call them leaves). An example of this structure is shown in Fig. 1. To access a particular file, we need to navigate up and down in this tree until our current directory is the directory where this file is stored or we need to use the full path name for the file.

In Linux the *root* of the directory tree, that is, the uppermost folder, is called `/`. If we have a folder `home` under the root directory with a file `exam.tex` in it, the path name for the file is `/home/exam.tex`. To edit this file we can type `nano /home/exam.tex`[1] irrespective of where we are in the directory tree. This is because this path is an *absolute* path that defines the file uniquely. Absolute paths start with `/`. A path that does not start with `/` is a *relative* path. This means that to determine the location refered to by this path we need to append it to the current directory. For example, if our current directory is `/home` we can edit `/home/exam.tex` by typing `nano exam.tex`.

To determine the current directory, type `pwd` at the command prompt. To list the contents of the current directory use `ls`. To navigate to another folder we use `cd path-to-folder`. Two relative paths have special names: `.` indicates the current directory and `..` indicates the parent directory. Thus `cd .` does nothing and `cd ..` navigates to the parent directory. While `..` is obviously useful when navigating the directory tree, you may wonder why we need `.`: many linux tools allow you to refer to files that are not in the current directory without specifying the full path. In this case it is sometimes required to use `.` to make it clear that you refer to files in the current directory.

Now, to begin our task: we first create a directory where to store our first program. I will assume that you collect all the files for this class in a folder called `compphys` under your home folder. To create this folder we type `mkdir compphys`. We navigate to this folder using `cd compphys` and then create the project folder: `mkdir week1proj1`. Finally we change to this directory, `cd week1proj1`, and then start editing our first program, `nano tempconv.c`.

Your first program will be very simple: it will just print a string on the console and then exit. All C programs will have a function called `main`. The *signature* of the function is `int main()` which means

[1] `nano` is a simple text editor that we will use in this class.

that it will take no input parameters, thus the empty parentheses, and returns an `int`. The program's execution starts on line 5 where the `printf` function is called and once this is done the program completes by returning 0. This `int` is returned to the operating system and it is used to indicate whether the program completed successfully, by convention this is indicated by 0, or whether an error occurred. In case of error the program usually returns a code to help identify the type of error that occurred.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    printf("Hello, class!\n");
6    return 0;
7  }
```

Note that `printf` is a function defined in the standard C library so we have to include its declaration, `#include <stdio.h>`, for the code to compile successfully. After typing this code in `nano` we save it, `ctrl+O`, and then exit, `ctrl+X`. To *compile* the code we type at command prompt `gcc -std=c99 -o tempconv tempconv.c`. This command uses the GNU C compiler to compile and link the program. The options passed to the compiler specify that we want to use the C99 standard and that the program name should be `tempconv`. To run this program we type `./tempconv` at the command prompt. Note that we need to specify the relative path of the program, because the environment setup in your virtual machine does not instruct the shell to look in the current directory for an executable. When we run this code it prints `Hello, class!` on the screen and then returns. Note that the last part of the string `\n` does not seem to appear on the terminal. This is because it represents a *newline* which advances the cursor to the next line. To understand the effect of this *special* character, edit the program to remove the newline, compile the code and run it again.

## 3 Temperature conversion

We are now ready to write a program that converts temperatures from Fahrenheit scale to Celsius. To convert the temperature we use the relations

$$T_F = T_C \times \frac{9}{5} + 32 \qquad \text{and} \qquad T_C = (T_F - 32) \times \frac{5}{9}\,. \tag{1}$$

A first attempt for coding this is given in the listing below. Note that we introduce some variables to hold the values of the temperature, `Tf` and `Tc`. We read `Tf` from the terminal using the standard library function `scanf`, that takes a formatting string and the *addresses* of the variables designated to hold the input values.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int Tf;
6    printf("Enter temperature in Fahrenheit: ");
7    scanf("%d", &Tf);
8
9    int Tc = (Tf-32)*(5/9);
10   printf("Temperature in Fahrenheit %d corresponds to %d in Celsius\n",
11     Tf, Tc);
12   return 0;
13 }
```

We pause here to briefly discuss `printf` and `scanf`. These routines provide a way to interact with the user via formatted strings. They both take as a first parameter a string which describes the expected input/output format and then a variable number of parameters that are either used to produce the final output for `printf` or they are initialized with the values read in for `scanf`. In the format string the tokens

starting with `%` indicate that a value is expected, with the character following indicating the type. In our code we use `%d` on line 6 to indicate that we expect to read an `int` value and on line 10 to indicate that we will take an `int` value and represent it as a string. Note that on line 10 there are two `%d` which are matched with `Tf` and `Tc`, the two parameters following the formatting string. For more details read the detailed help provided by the online help system. This can be access by typing at the command prompt `man prinft` and `man scanf`.

We compile this code using the same command as before `gcc -std=c99 -o tempconv tempconv.c`. When we run it, we are asked to enter the temperature. However, there seems to be a problem since the program returns 0 for any temperature we type. The problem is actually on line 9, where we use a conversion factor of `(5/9)`. Since both 5 and 9 are recognized as integral types by the compiler, the division used is the integer division, which returns the quotient of the division. The result here is 0 and `Tc` ends up being 0 irrespective of the value of `Tf`. One easy fix is to remove the parentheses around `(5/9)`. Then the multiplication `(Tf-32)*5` is performed first and then the integer division, which leads to a better answer. Edit the code, recompile and run it.

The final task today is to produce a table to help us with the temperature conversions. We will generate a table with Fahrenheit temperatures 0, 5, 10, ..., 100 and their equivalent on the Celsius scale. To avoid dealing with the integer division issue, we will use floating point numbers to store the values. In C we have two types that are used to store floating point numbers: `float` and `double`. They have similar properties except that `double` is represented using 64 bits (versus 32 for `float`) and it can be used to store a greater range of numbers and perform computations more precisely. `float` requires less memory and in certain situations it can increase the speed of the code at the expense of numerical precision. In our class we will use `double` in most cases.

To repeat the calculation for all the values in our table we will use a `for` loop. The syntax of the `for` loop is

```
for ( initialization ; iterate test ; increment ) { body }
```

The `initialization` sets loop variables at the beginning of the first iteration, then the `iterate test` is performed and, if true, the `body` is evaluated. At the end of the iteration `increment` is evaluated and the whole loop begins again if `iterate test` is true. This repeats until the test is false.

The final version of our code is listed below. Note the new format tokens in the `printf` routine calls on line 5 and 9: `%12s` indicates a string of length 12 (the string is padded with blank spaces on the left if it is shorter) and `% 12.2f` which indicates that the floating point variable should be represented using fixed point notation with 2 digits after the decimal dot and 12 characters overall. The space following `%` indicates that if the string representation of the number is shorter than 12, it will be padded with blanks on left. This helps up produce a nicely aligned table. Please compile the program and run it.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    printf("%12s   %12s\n", "Fahrenheit", "Celsius");
6    for(double Tf = 0; Tf < 101; Tf += 5)
7    {
8      double Tc = (Tf-32)*5/9;
9      printf("% 12.2f   % 12.2f\n", Tf, Tc);
10   }
11   return 0;
12 }
```

# 4 Graphs

For your reports you will be asked often to produce graphs to illustrated your arguments. We will use Mathematica in this class to analyze the data and produce graphs. We will now use the output of the program we wrote in the previous section, import it in Mathematica and plot it.

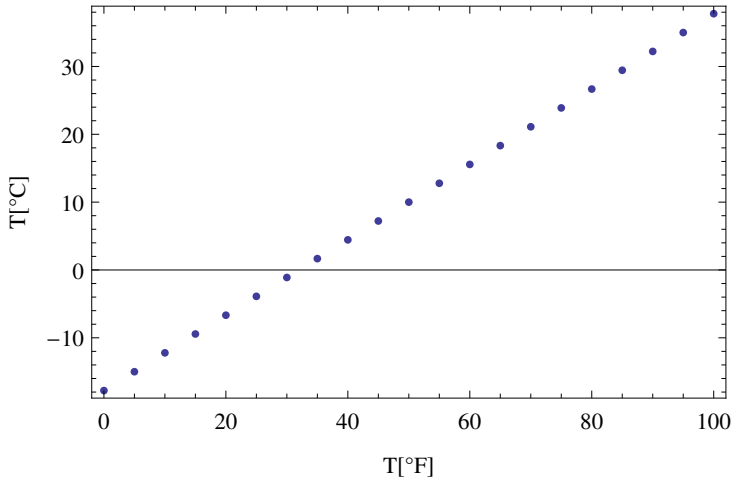


Figure 2: Plot produced by Mathematica using the steps indicated in the text.

The first step is to save the output in a file. To accomplish this we can modify the program to write directly to a file. A simpler option is to redirect the output of the program from the screen to a file. This is done by typing `./tempconv > ouput.log`. If you now list the contents of your directory –remember `ls`– you will see that you have a new file: `output.log`. Let's edit the contents of the file to make it easy to read it in Mathematica. Type `nano output.log` and remove the first line in the file, the heading of our temperature table. (Hint: use `ctrl+K` to erase the entire line at once.) Save the file and exit the editor. Now `output.log` has a number of lines with numeric entries.

Open a Mathematica notebook and read in this file. The first task is to identify the directory where `output.log` is stored. On my machine `pwd` reports that this is `/home/student/compphys/week1proj1`. In Mathematica I will set this directory using

```
SetDirectory["/home/student/compphys/week1proj1"]
```

and evaluate this line by pressing `shift+Enter`. To read in the file I use Mathematica command

```
data=ReadLine["output.log", {Number, Number}]
```

and evaluate this line by pressing `shift+Enter`. Note that I told `ReadLine` that the expected input is a list of pairs of `Number`s. We are now ready to plot this data. We do this using `ListPlot[data]` and evaluate this line by pressing `shift+Enter`. This produces a graph of the data, but it is rather barebones. A better version of this plot can be produced by passing custom options to `ListPlot`. A complete description of the options can be found in Mathematica's Help system. To get help for `ListPlot` place your cursor inside this command and press `F1`. Note that this help is interactive. A better looking plot can be produced using

```
g=ListPlot[data, Frame->True, FrameLabel->{"T[°F]", "T[°C]", "", ""}]
```

where ° is produced by pressing `Esc`, type `deg` and then press `Esc` again. If you want to include this plot in a report you need to give it a name –we called it `g`– and then export it to a file. The preferred file format is PDF but you can use EPS or other file formats if you need to. Read the Help on `Export` to see what other options Mathematica offers. To export to PDF we use

```
Export["data.pdf", g, ImageSize->300]
```

and evaluate this line by pressing `shift+Enter`. You should check that you have the file `data.pdf` in your original directory. Go to the terminal and check this. To see the contents of the file you can use `evince data.pdf`. `evince` is the viewer installed in your virtual machine. The image produced is shown in Fig. 2.