SUPAC++ Labs 1 and 2

Gary Robertson

12^{th} November 2025

Important Information

Before the lab - I recommend trying the preliminary exercises before the first lab in order to verify everything (codebases, gitlab *etc.*) is working. If you have not yet done these you should start there before working on the main assignment. Note that the preliminary exercises do not count towards your final mark.

Try to follow good coding conventions when writing code:

- Comments - Add comments as you go. Helps keep track of what the code is doing, and makes it clearer for anyone looking at it down the line (me, but also future you). Consider also adding your name and the date at the top of the script as a comment.
- Repeated testing - Don't just write the code and then test it at the end. Test as you go to ensure each step is working as expected. Also carefully consider edge-cases which will likely break your code.
- Naming Conventions - Try to be consistent with names you use for variables, and make them clear.
- Indentation - C++ is a whitespace independent language, meaning that you can vary the spaces and indentation in commands without causing a crash (*e.g.* `int x = 5` will be correctly interpreted) but you should try to keep your code neat to ensure readability.

Submission

Submission of your code for the assignments will be done using git (see instructions at the bottom of the `README.md`). Please try to avoid uploading too many extra files beyond what is needed to complete the exercises. You can also add a `README.md` file explaining how to run your code. More information on what to include in the submission can be found at the end of this document.

Preliminary Exercises

1. Create a program that when compiled and executed prints "Hello World!" to the terminal.
2. Declare two variables `x` and `y` and assign them the values `7.5` and `3.4` respectively. Assuming these are components of a 2D vector, compute the magnitude of the vector and print the answer to the terminal.
3. Now create a generic function capable of calculating the magnitude of a 2D vector. It should take as input from the user (*i.e.* prompt the user on the command line) two values for the components and should print to the terminal the magnitude. Using this new function verify your calculation from part 2.

Assignment Exercises

The aim of these exercises is to make you more familiar with reading and manipulating different types of data using functions. We will go through the process of building up a complex piece of code in steps. The exercise is split into several steps through which we will slowly increase the complexity and functionality of the code. The final result should be a main steering script, `AnalyseData.cxx` which will allow the user to read data from a file and take a range of possible actions, all through the command line. The steering script will then call functions from other scripts based on the users choices in order to modify the data and write the results

to a file. You may find it useful to read through all of the steps to get a feel for the desired functionality of the final code, which can be helpful to keep in mind when working through the earlier steps. However, if you find this too overwhelming don't worry - this is why we start simple and slowly add complexity.

This assignment is due by the 28th of November

The goal of the assignment is to write code that allows the user to read in (`x`,`y`) coordinates from a plain text data file and then analyse it:

1. Print `N` lines (where `N` is user specified) of the data to the terminal.
2. Calculate the magnitude of each data point, assuming the point forms a vector with the origin (`0`, `0`).
3. Fit a straight line to the data using the least squares method, and assess the goodness of fit using a χ^2 test (we expect it to be bad).
   - Least squares method:
   $$y = px + q \quad \text{with} \quad p = \frac{N\sum x_i y_i - \sum x_i \sum y_i}{N\sum x_i^2 - \sum x_i \sum x_i}, \quad q = \frac{\sum x_i^2 \sum y_i - \sum x_i y_i \sum x_i}{N\sum x_i^2 - \sum x_i \sum x_i}$$
   - χ^2 test:
   $$\chi^2 = \sum \frac{(O_i - E_i)^2}{\sigma_i^2}$$
   where O_i are observed values, E_i are expected values, and σ_i is the expected error on the measurement.
4. Calculate x^y for each data point without using a for loop or any in-built power functions.
5. Adapt the code so that for each of the previous steps, the results are written to a file with differing names depending on the task chosen.

Instructions

It is always best practice to start with the simplest implementation that works, and then build complexity from there, and this is what we will do here.

1. Lets first make sure we can read the data file and that we understand what the data looks like. Create a script named `AnalyseData.cxx` which opens the data file `input2D_float.txt`, reads the contents line-by-line and prints them to the terminal. N.B. You can also see a visualisation of the data by opening `Outputs/png/plot.png`. This plot can be regenerated by running the command `gnuplot plotdata.gp` in the command line.
2. Now we have an idea what the structure and format of the data is, modify your code to write the (`x`, `y`) values into a suitable container (be sure to pick a suitable format) when reading the file. Then print the values to the terminal by printing these variables (as opposed to printing from the file in Step 1). N.B. We do not know exactly how many data points there are in the file (and ideally you want your code to be generic enough to handle different data sources) so make sure your chosen structure is dynamic.
3. Move your reading/printing functionality into two separate functions, modifying the printing function to only print `N` lines (remember `N` will be specified by the user!). Consider possible edge cases when implementing this. In particular, if `N` is larger than the total number of data points, the code should warn the user then print only the first 5 lines.
4. Now, add another function which calculates the magnitude of each data point (we can assume it is for the full file, not just `N` lines) assuming they are the (`x`, `y`) components of a vector. Print the magnitude values to the terminal also. N.B. Remember in the final code we would like to write the result of this function to a file, so make sure you are storing the magnitude in a sensible data structure and not just computing and printing the magnitude.
5. Now we want to streamline a bit. Move the extra functions you have defined out of the main file by adding a declaration in a file called `CustomFunctions.h` and implementing them in `CustomFunctions.cxx`. Now, if you haven't already, modify `AnalyseData.cxx` so that it prompts the user for which function they would like to use (print `N` lines of data or print the magnitude of every data point). N.B. Remember to modify your compilation command to include your new scripts. You may want to implement a simple `Makefile` (see Lecture 2) to make this easier.

6. Add a new function as an option in `CustomFunctions.cxx` (remembering to also declare it in the header file) which takes in the (`x`, `y`) data (*i.e.* takes a data file as an argument) and fits a straight line function using the least squares method. Save the final function as a string to a new file while also printing the result to the terminal.

- - - - - - Lecture 2 Concepts - - - - - -

7. Since we are printing many things in a very similar way, lets try to refactor the code to use the same print function in each case. Consider using overloads for the function to allow it to handle different objects as input.

8. We can assess how well the function we defined in Step 6 fits the data using a χ^2 test. A particularly useful measurement is the chi-squared per degree-of-freedom (commonly written as $\chi^2/NDOF$). Modify the least squares function to also calculate the $\chi^2/NDOF$ (perhaps by calling another function within), including it in the saving/printing. To do this you will also need the expected error for each data point, which you can find in the file `error2D_float.txt` (luckily you have a helpful function you wrote earlier for reading this file!).

9. Write a new function to calculate x^y using recursion for each data point with y rounded to the nearest whole number. Do this without using the `pow` function, a loop, or logarithms.

10. Add one last function (remember you may need overloads) which is capable of taking the output from any step that is ran and saves it to a file with a suitably descriptive name. If the main functions above (calculating the magnitude, fitting a straight line, and calculating x^y) are chosen by the user then there should be 3 total output files.

11. Finally, in case you haven't already, update `AnalyseData.cxx` to instruct the user through the terminal on the options available, and prompt them to select one. The code should then perform the selected action (try implementing a `switch`) and once finished prompt the user to either perform another action or exit.

Deliverables

For the submission you should include (at a minimum):

- `CustomFunctions.cxx` along with its corresponding header file.
- `AnalyseData.cxx` along with its corresponding header file.
- Optional: A `README` explaining how to compile/run your code.
- Optional: A `Makefile` if you have written one.

If there are any other scripts you have written that are required to reproduce the results, then you should also include them.