PHYS4070 Worksheet. Week 1: Introduction

This work sheet is broken into two parts.
Part A is a basic c++ tutorial designed to get you up and running. It has a few examples for you to compile/run and play around with. It's very important you can all get these running today; if not, we are here to help you. You will need these basic skills to complete the rest of the tasks and assignments.

Part B will introduce a basic physics problem: solving a very simple ODE. This will lead directly into your first assignment.

Contents

Part A: Basic introduction, examples

1. Getting started

Go through the **Quick Start Guide** that is posted on blackboard (Learning Resources / Examples, other resources).
This will explain how to get set up for several different systems.
Goal 1 is to compile the following code.
You may use Visual Studio Code to log on to smp-teaching and (alternatively, you may use your own PC/laptop).

```
#include <iostream>
int main(){
    std::cout<<"Welcome to PHYS4070!\n";
}
```

Place this code into a file (named, e.g., 'hello.cpp'), and:

- Compile it: `$ g++ -o hello hello.cpp`
- Run it: `$ ./hello`
- It should print "Welcome to PHYS4070!" to the screen. If this doesn't work, ask the lecturer/tutor for help

It may take you some time to get used to working with the linux terminal - that's pretty normal. In the QuickStart Guide, there is also a basic introduction to linux terminal. Goal 2 is to perform the following tasks to familiarise yourselves with the terminal. If you don't understand what they are doing, please ask one of us for help.

```
pwd
ls
mkdir temp
ls
cd temp
pwd
ls
touch hello.txt
ls
cd ..
pwd
cd ~
pwd
```

2. Basic tutorial/reference

c++ is a big language, and that can be daunting at first. However, everything you need to know will be covered in the course. We will start with the real basics, and work up from there. If you are familiar with c++, you may skip this section (though perhaps give it at least a quick look over)

There are many great websites online for learning c++ (as well as the books recommended on Blackboard). I recommend hackingcpp.com/ - which has a large array of nicely formatted tutorials, examples, and cheat-sheets. Start with

- **hackingcpp.com/cpp/beginners_guide**

You can also access an even more basic c++ "cheat sheet" here:

- github.com/benroberts999/cpp-cheatsheet
- This just has several exmaples of many of the most common c++ features
- Feel free to download this code and have a good play around, and use it as a reference if you need

Other c++ online resources
These are two main c++ resources/references online

- cppreference.com - The "standard" most complete/thorough resource, not very beginner friendly
- cplusplus.com - Less detailed than the above, but more beginner friendly, and has many nice examples
- Compiler Explorer (godbolt.org) is an online compiler that lets you easily test/share snippets of code. All of the following examples will be accompanied by a link to the interactive online compiler.

2.0 Compilation

Technically, c++ uses a two-step compilation system (just like C). First, each "compilation unit" (c++ file) in your project is compiled separately. Then, the resulting compiled files (known as objects) are *linked* together to form the executable program. For small projects, we often do these steps with a single command. For large projects, we typically use a build system (such as CMake or Makefile) to compile our code. For smaller projects, such as in this course, it's fine to compile directly from the command line. Later on, we will see an example of linking to an external set of functions (LAPACK).

Example 1 If we have a single c++ file, called, e.g., "main_program.cpp"

```
#include <iostream>
int main(){
    std::cout << "hello world!\n";
    return 0;
}
```

We would compile this like:

- `$ g++ -o main_program main_program.cpp`
- The '-o main_program' command tells g++ to name the executable 'main_program'. By default, it will otherwise be names `a.out`. By convention, the executable should have the same name as the main program, but either without a file extension (linux/mac) or '.exe'

Compiler warnings

It is not required, but is *highly* recommended that you turn on compiler warnings when compiling your code. These will warn you when you make likely mistakes. Most of all, they help you learn to write better code, and help prevent you from learning bad habbits. Then, you might compile, e.g., like this:

- `g++ -o main_program -Wall -Wextra -Wpedantic main_program.cpp`

Example 2 Say instead we have some functions we want to separate into a second file, which we call 'my_functions.cpp':

- We separate the "my_functions" file into two parts - a .cpp file, which contains the function definition, and a header file (.h or .hpp), which contains the function declaration
- The reasons for this is to allow easy inclusion of separate files into our code, while allowing for faster compile times (since we can compile each file separately, we don't need to recompile everything whenever we change something.) You can learn more about header files here: https://www.learncpp.com/cpp-tutorial/header-files/

my_functions.cpp:

```
#include "my_functions.hpp"
void function(){
    std::cout << "Hello, from my function!\n";
}
```

my_functions.hpp:

```
#ifndef MY_FUNCTIONS_HPP
#define MY_FUNCTIONS_HPP
void function();
#endif
```

- The ifndef stuff here is a 'header guard'. You can read about them here : https://www.learncpp.com/cpp-tutorial/header-guards/
- In short, they prevent issues from having multiple definitions of the same function
- Most modern compilers have automatic header guards; instead we type '#pragma once' at the top of the file. You're welcome to use this

```
#pragma once
void function();
```

Then, our 'main_program.cpp' file would be::

```
#include <iostream>
#include "my_functions.hpp"
int main(){
    std::cout << "hello world!\n";
    function(); // this is defined in my_functions.hpp/cpp
    return 0;
}
```

- We must '#include' the header file in main_program.cpp, which makes the new functions available.

    - **Never** #include a .cpp file - only do this for header files (usually .h or .hpp)
- Then, we compile like:
- `$ g++ -o main_program main_program.cpp my_functions.cpp`

2.1 Variables, scope, if/for/while

- In c++, variables may only be accessed if there are 'in scope'. This is useful for understanding what the code is doing.
- It is generally good practise to define your variables in the most narrow scope possible (do not make global variables!), and to define them as close to where you use them as possible (unlike common practise in C and Fortran, where variables are often all defined at the front and used later). This is to make it easier to understand the code.
- It is possible to define a variable without assigning it a value - this is generally bad, as it can make the code confusing, and debugging very hard. Try to always define variables at the same time you declare them

Interactive version: godbolt.org/z/dco3MY37T

```cpp
#include <iostream>
#include <string>
int main() {
  // most common data types: double, int
  double x = 17.2;
  int i = 6;
  // 'auto' (available in c++-11) will deduce the type
  auto j = i;     // j is an int
  auto y = 14.6;  // y is a double
  // We must declare the variables type:
  // z = 12.0; this will not compile

  // be careful with integer division:
  double bad_division = 3 / 6;
  double good_division = 3.0 / 6.0;
  std::cout << bad_division << " " << good_division << '\n';

  // we can also makes 'strings' (text)
  std::string my_string = "Hello phys4070 class!";

  // Scope is defined by curly braces:
  {
    int i2 = 6;
    // we can access outer scope from in here:
    double y2 = 2 * y;
  }
  // This will not compile, i2 is out of scope.
  // std::cout << i2 << "\n";

  i = 3 * 6;  // same as i*=3
  std::cout << i << '\n';
  const int i3 = i;
  // but this will not compile: we cannot edit a const
  // i3 *= 2;

  double z;
  // z is not initialised (bad). This makes debugging code
  // difficult, and makes it easy to introduce errors
  std::cout << z << "\n";
}
```

- Bools, if statements, for loops:

Interactive version: godbolt.org/z/q3vazhW7E

```cpp
#include <iostream>
int main() {
  bool physics_is_fun = true; // bool may be true or false
  if (physics_is_fun) {
    std::cout << "yay!\n";
  } else {
    std::cout << "boo!\n";
  }

  if (16.5 > 2.0) {
    std::cout << "16.5 is greater than 2.0\n";
  } else {
    std::cout << "Oh no!\n";
  }

  for (int i = 0; i < 15; ++i) {
    std::cout << i << ", ";
  }
  std::cout << '\n';
}
```

2.2 Basic in/out, headers

- Input and output are dealt with using 'streams' in c++. 'cout' is the default output stream, which prints to the terminal screen. 'cin' reads input from the terminal (this example won't work so well on CompilerExplorer)
- fstream reads/writes to text files in much the same way (ofstream for output, ifstream for input, fstream for input and output)
- We need to include headers (`#include <...>`) to make certain functions/classes available
- We are not limited to the inbuilt c++ headers - we can write our own (we'll see this later). Generally, `<>` implies it is an internal c++ header, while `""` indicate a user-defined header

```cpp
#include <fstream>   // for ifstream, ofstream
#include <iostream>  // for cin, cout
int main() {
  std::cout << "Hello. Please enter your name:\n";
  std::string name{""};
  std::cin >> name;
  std::cout << "hello " << name << '\n';
  // Notice the direction of the '<<' or '>>' operator

  std::ofstream ofile("out.txt");
  ofile << 0.1 << " " << 12.5 << '\n';
  ofile << 45.7 << " " << 16.3 << '\n';
}
```

```cpp
#include <fstream>   // for ifstream, ofstream
#include <iostream>  // for cin, cout
#include <sstream>   // for string stream
#include <string>    // for string

int main() {
  std::ifstream in_file("out.txt");
  // first get each line as a string
  std::string line{""};
  while (std::getline(in_file, line)) {
    // then, read input from each line, using stringstream:
    double x{0.0};
    double y{0.0};
    std::cout << "The whole line: " << line << '\n';
    auto ss = std::stringstream(line);
    ss >> x >> y;
    std::cout << "x=" << x << ", y=" << y << "\n";
  }
}
```

2.3 Functions

- In c++ program, the main() function always runs first. Only other functions that are called from main() are executed
- Functions take input, and return output
- In c++, we can have functions with the same name, that take different intputs (function overloading)
- Often, it is useful to use templates for situations where the same function would work for many different types - this saves a lot of typing, and reduces the likelihood for transcription errors

Interactive version: godbolt.org/z/MT9dxWx9Y

```
#include <iostream>

double my_function(double x, double y) {
  double value = x * y;
  return value;
}

int my_function(int x, int y) {
  int value = x * y;
  return value;
}

template <typename T>
T my_function2(T x, T y) {
  T value = x * y;
  return value;
}

int main() {
  auto z1 = my_function(17.2, 16.5);
  auto z2 = my_function(17, 16);
  auto z3 = my_function2(17.2, 16.5);
  auto z4 = my_function2(17, 16);

  std::cout << z1 << " " << z2 << " " << z3 << " " << z4 << "\n";
}
```

main() can take arguments too, from the command line. They must be in the following form. They are stored in an array of character arrays (this is very old school, but is like this to ensure back compatibility with C). Run this code with with command line argument, e.g.,

- `$ g++ example.cpp -o example`
- `$ ./example hello phys4070 class`

```
#include <iostream>
int main(int argc, char** argv) {
  std::cout << "You entered " << argc << " arguments:\n";
  for (int i = 0; i < argc; ++i) {
    std::cout << argv[i] << "\n";
  }
}
```

2.4 Pass by value, reference, pointer

There are three main ways to pass variables into functions:

- by value (sometimes called *by copy*): the function will get a new copy of the variable, not changes to this variable inside the function will be visible. This should usually be the standard -

only use another method if there's a good reason (e.g., when a big data object is expensive to copy)

- by reference: the function will get *a reference* to the same variable - changes will be made to the original variable. This is often considered bad, because it makes it hard to reason about who has access to which variables
- pointer: a pointer is a variable that stores a memory address. Similar to by reference. We very rarely use raw pointers in c++ - however, we sometimes have to, e.g., when interfacing with C or Fortran libraries (like LAPACK), as we'll see later on

Interactive version: godbolt.org/z/5xbczehTf

```
#include <iostream>
int pass_by_value(int a, int b) {
  a = a + b;
  return a;
}
int pass_by_reference(int &a, int b) {
  a = a + b;
  return a;
}

void pass_by_pointer(int *a, int b) { *a = *a + b; }

int main() {
  int a = 1;
  int b = 1;
  std::cout << a << " " << b << "\n";
  int c = pass_by_value(a, b);
  std::cout << a << " " << b << " " << c << "\n";
  c = pass_by_reference(a, b);
  std::cout << a << " " << b << " " << c << "\n";
  std::cout<< "Value of a: " << a << "; memory address of a: " << &a << "\n";

  int* pointer_to_a = &a;
  pass_by_pointer(&a, b);
  std::cout << a << " " << b << "\n";
  pass_by_pointer(pointer_to_a, b);
  std::cout << a << " " << b << "\n";
}
```

- In cases when we want to avoid a copy (e.g., passing a large array/matrix to a function), it's often wise to pass by reference. In these cases, it is usually best to pass by *const reference*, to avoid accidentally modifying data we shouldn't

```
BigMatrix pass_by_const_reference(const BigMatrix &a, const BigMatrix &b) {
  //a = a + b; cannot modify a const!
  auto new_a = a + b;
  return new_a;
}
```

2.5 std::vector

std::vector is the c++ workhorse. It is a dynamically allocated, resizable array. It should generally be the default for storing array-like data in c++.
Data is stored in contiguous block (good for speed + needed for libraries), and is completely compatible with regular c-style arrays.

The following is a very basic example. We'll get more into this in the coming weeks.

```
#include <iostream>
#include <vector>

int main() {
  // Create a vector of ints (can be of any type)
  std::vector<int> a{1, 2, 3, 4, 5};

  // add elements to the vector:
  a.push_back(6);
  a.push_back(7);
  a.push_back(8);

  std::cout << "Ranged for loop:\n";
  for (int element : a) {
    std::cout << element << "\n";
  }
}
```

A slightly bigger example can be found here: **godbolt.org/z/8a796Ev1o**

Full reference:
cplusplus.com: https://www.cplusplus.com/reference/vector/vector/
cppreference: https://en.cppreference.com/w/cpp/container/vector

2.6 classes

A class in c++ is a way to group functions and data that belong together. There are many classes provided by the c++ standard (e.g., vector we looked at above is a class), and we may define our own. You can think of this as defining our own data types. A very basic example is given here; we'll look at classes more down the road

Here is just a very simple example of a class, which emulates a particle in 1 dimension:
Interactive version: godbolt.org/z/az7McfYc4

```
#include <iostream>

class Particle1D {
  // these variables are "private" - only accessible from inside the class
 private:
  double m_mass;
  double m_vecocity;

 public:
  // constructor: special function that runs when we construct the object, and sets the initial
  Particle1D(double mass, double initial_velocity)
      : m_mass(mass), m_vecocity(initial_velocity) {}

  // these functions just report on the current mass/velocity
  double mass() const { return m_mass; }
  double velocity() const { return m_vecocity; }

  // this function applies a force to the particle
  void accelerate(double applied_force, double delta_t) {
    m_vecocity += (applied_force / m_mass) * delta_t;
  }
}; //notice the Class ends with an ';'. For reasons.

int main() {
  // create particle with mass 10, and 0 velocity
  Particle1D particle{10.0, 0.0};
  std::cout << particle.mass() << " " << particle.velocity() << "\n";
  // Impart a force of 6.0 units, for time of 0.5 units:
  particle.accelerate(6.0, 0.5);
  // new mass/velocity:
  std::cout << particle.mass() << " " << particle.velocity() << "\n";
}
```

2.7 cmath library

Being a physics course, we will need to use mathematics functions - many are available in cmath header:

```
#include <cmath>
#include <iostream>

int main() {
  std::cout << M_PI << "\n";
  std::cout << std::sin(6.2) << " " << std::abs(-12.6) << " " << std::exp(-0.1)
            << " " << std::log(1.0) << " " << std::sqrt(6.1) << "\n";
}
```

3. Plotting

You will need to plot data for several parts of the course. You can use any program you like for plotting, e.g., gnuplot or matplotlib from python. The following are two basic examples that plot the data contained in "data.txt" file, included on blackboard

3.1 GNUplot

The following will plot to a GNUPLOT window: you can then click the button (top left) to export to png/pdf etc:

```
$ gnuplot
gnuplot> set title 'Title of my plot'
gnuplot> set xlabel 'time (s)'
gnuplot> set ylabel 'y(t)'
gnuplot> plot 'data.txt' u 1:2 with lines title 'data 1',\
     '' u 1:3 w points t 'data 2',\
     '' u 1:4 w l linetype 0 t 'data 3',\
     '' u 1:5 w p dt 2 t 'data 4'
gnuplot> quit
```

- 'data.txt' is on blackboard (alternatively, try any other data)
- 'using 1:2' means using column 1 vs column 2 (x and y). You can use 'u' as short-hand for 'using'
- 'with lines' means 'with lines', instead of just plotting the points. You can use 'w l' as short-hand (or 'w p' to plot the points only, or 'w lp' to plot both)
- "title 'line title'" means set the title of that line to 'line title' (often not needed)

Or include the following, which will directly output the plot to a png:

```
gnuplot> set terminal png
gnuplot> set output "myplot.png"
```

Instead of running from the command line, you can put all the gnuplot functions into a plain textfile (called, e.g., plot.gnu), then run gnuplot on that file. The example plot.gnu is on blackboard

```
 gnuplot plot.gnu
```

3.2 matplotlib (from python)

matplotlib is a very popular plotting program, available through python, that is widely used in scientific publishing. The smp-teaching server has python2 installed, not python3, but it will still work fine.

```
python matplotlib_tutorial.py
```

Or, you an add the following `#! /usr/bin/env python` to the start of the python file, change the permissions to allow direct execution of the file using `chmod +x matplotlib_tutorial.py`, and then simply run as:

```
./matplotlib_tutorial.py
```

Part B: Worksheet Tasks

A spherical projectile moves vertically in a viscous medium near the surface of the earth.
It experiences two forces, the gravitational attraction of the $(-mg)$ earth and the viscous force F_v from the medium in which it moves. The former is directed downward (negative z) direction and the latter is directed opposite to the velocity v. Usually, the magnitude of the viscous force is a function of the speed with which the projectile moves, i.e.
$F_v = f(v)$
for some f. Assuming that $f = bv^2$ (where $b > 0$ denotes some positive constant), the projectile's equation of motion is then given by:

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = -m|g| - bv|v| \tag{1}$$

- $m = 50\,\mathrm{g}$ is particle mass
- $|g| \approx 9.8\,\mathrm{m/s^2}$ is acceleration due to gravity
- $b = \rho C_d A/2$,
  - $\rho = 1500\,\mathrm{kg/m^3}$ is the medium density
  - $C_d = 0.45$ is the drag coefficient
  - $A = \pi r^2$ is the cross-sectional area, $r = 5\,\mathrm{mm}$

To determine the velocity, we must solve the above first-order ODE. You'll learn much more about solving ODEs in the lectures. Here, we'll use a simple numerical method: Euler's method

Euler's Method

The general problem is of the form:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = g(x, f(x))$$

with some initial value, $f(x_0) = a$; we wish to solve for f numerically. Expanding around some point x, we have:

$$f(x + \delta x) = f(x) + \frac{\mathrm{d}f}{\mathrm{d}x}\delta x + \mathcal{O}(\delta x^2).$$

If δx is sufficiently small, we can neglect the higher-order terms. If we know $f(x_0)$, we can estimate $f(x_1)$, and so on:

$$f(x_0 + \delta x) \approx f(x_0) + g(x, f(x_0))\delta x. \tag{2}$$

We wish to solve the problem over some domain $x \in [x_0, x_N]$, which we break into $N + 1$ discrete steps, with $x_N = x_0 + N\delta x$.

Q1. Write a program that solves the ODE

Write a c++ program that solves Eq. (1), using Euler's method, Eq. (2), for $t \in (0,\, \delta t,\, \ldots,\, t_N)$. It should use variables for N, t_N, and the initial value $v_0 = v(0)$, so you can easily change these parameters. You may take the parameters as command-line input, read them from a text file, or just code them in. You can print the output of your solver to screen, or better yet, write the data to a text file, in the form "t v(t)" on each line, so you can plot it.

Q2. Run the program

- (a) First, run your program with $t_N = 10\,\mathrm{s}$, $N = 50$, and $v_0 = 0$. Does the result make sense physically?
- (b) Then, run the program again, but with $N = 10$. What happens now? Why?
- (c) Run your program again, with $N = 100,\, 200,\, 400$.... What happens as we increase N? Why?
- (d) Run your program with $t_N = 10\,\mathrm{s}$, $N = 25$, $v_0 = 0\,\mathrm{m/s}$, and plot the result. What is happening? Is this physical?
- (e) Run your program with $t_N = 2\,\mathrm{s}$, $N = 100$, for $v_0 = \{-10,\, 0,\, +10\}\,\mathrm{m/s}$, and plot the results. What is happening? Is it what you expect?
- Try plotting some of your solutions together on the same plot