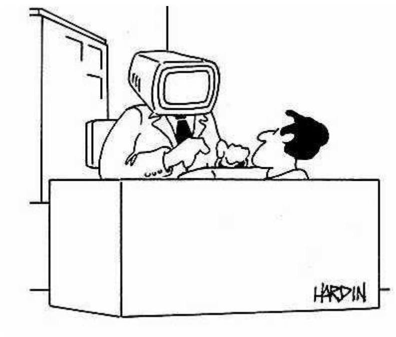
Y2 Computational Physics Laboratory

UNIVERSITY OF BIRMINGHAM



Exercise 1

A ballistics program using C++ functions

Aims:

- To write a short C++ program to model the ballistic trajectory of a projectile
- To introduce you to programming with Microsoft Visual C++
- To introduce you to using functions in C++
- **1 mark will be given for overall presentation**

Duration: 2 weeks

Hand-in deadline: End of session #2

Introduction

This first exercise will build upon what you learnt last year in the Y1 C++ course. It will also introduce you to programming using Microsoft Visual C++. You will be writing a short C++ program using functions to calculate the trajectory of a ballistic projectile. In addition, you'll be asked to derive some simple equations, to plot graphs in Excel and to comment on your results.

Before you start programming, please look at the examples on Canvas to remind yourself of the structure and basic syntax of a simple C++ program. (There are also many useful websites on C++ programming, such as www.cplusplus.com .)

Example #1 is the standard "Hello, World!" program, whilst Example #2 uses a 'for' loop to tabulate position *y* at time *t* for an object falling from rest under gravity. Example #3 shows how to save the results to a text file (output.txt) that can then be opened in Excel. Example #4 uses a function to calculate the position. A sample spreadsheet (example4.xls) is also provided. This is exactly what you'll be aiming to produce with your ballistics program. (Note that you could actually do the ballistics calculations using formulae in Excel. However, the aim here is to learn how to use C++ functions. You'll use them later on to tackle problems that you can't solve in Excel.)

Visual C++

It's recommended that you use Microsoft Visual C++ 2010 to write, build and debug your program. This is the industry-standard tool for C++ programming on Windows PCs. For use on your own PC, you can either download the Express version from Microsoft free of charge at www.microsoft.com/visualstudio/en-us/products/2010-editions/express or you can download a free, full-featured version of Visual Studio 2010 through the University's Microsoft DreamSpark Premium agreement at https://www.epsit.bham.ac.uk/software/dreamspark/?action=signin .

All of the C++ exercises in this lab can be written inside a single .cpp source file. However, Visual Studio requires that file to be part of a 'project', which specifies the type of program (a 'Win32 Console Application') and all the compiler options, *etc*. Moreover, several related projects can be kept together in Visual Studio in what Microsoft calls a 'solution'. This is how more complex programs are constructed, and it's best that you learn to work in this way. The examples and sample projects on Canvas are provided as ready-made solutions for Visual C++. There are also solution files for tackling each exercise. Hence, you may not need to create a new project from scratch.

Visual C++ produces many other files than just the final executable .exe file for your program. Some of these ancillary files can be quite large (>10 MB). You can get rid of most of them at the end of the session by selecting Build|Clean Solution in Visual C++. That will leave you with a more compact solution folder that you can save to a USB flash drive.

To use the examples, first copy the Y2C++Examples.zip file from Canvas to your PC and unzip it. You can then double-click the solution file Y2C++Examples.sln. Alternatively, start Visual C++ and navigate to this file from "Open Project…" on the Start Page or from File|Open|Project/Solution… on the menu bar. (Visual Studio may ask you to "Choose Default Environment Settings", in which case you should select "Visual C++ Development Settings" and wait a minute or so for them to take effect. On your own PC, you only have to do this once, as Visual Studio will then remember your settings.)

To build and run one of the projects, select the project in the Solution Explorer window and then press the small green arrow below the menu bar (or just press F5). The projects are configured for debugging. However, if you wish, you can produce a final 'release' version of your .exe file when you've finished debugging the program.

Structured programming

The key skill that you need to develop for writing any computer program is called ***structured programming***. This requires you to break down the program into a series of logical steps, each of which is controlled by a simple statement (`if`, `for`, `while`, *etc*.). Before you write any C++ code, it's useful to sketch the program's logic in a flow diagram. In structured programming, there should be a direct logical flow from start to finish, with no jumping around. When you translate this into code, each logical step becomes a statement block `{...}` (or perhaps just a single statement) whose execution is controlled by a simple logical test. In a structured C++ program, there is therefore a straightforward logical flow from top to bottom of the page. This makes the program efficient, easy to understand, easy to debug and easy to modify in the future. Simple problems usually have simple solutions. So, keep your program as simple as you can!

The following *syntax-highlighted* code from Example #3 illustrates a simple structured program:

```
#include <iostream>   //Declares cout
#include <fstream>    //Use for file output
#include <conio.h>    //Declares the _getch() function
using namespace std;  //Allows use of cout instead of std::cout

//Main program:
int main(void)
{
   //Declare parameters for the problem as constants:
   const double g=9.81;     //Standard value of g (in m/s^2) cannot be changed.
   const double y0=1000.;      //y0 = Initial position at t=0 (in m)
   const double tmax=10.;      //tmax = Maximum time (in s)
   const int npoints=100;      //npoints = Number of points
   const double dt=tmax/npoints;    //dt = Time increment (in s)
   //Declare variables used in the calculation:
   double y, t;       //y = Vertical position (in m), t = time taken (in s)

   ofstream outs;           //outs is the name of the output stream
   outs.open("output.txt"); //Opens the file output.txt

   //Print a column header for the t & y values on the screen:
   cout << "t" << "\t" << "y" << endl; //Note that "\t" inserts a tab separator
   outs << "t" << "\t" << "y" << endl; //Writes to the output file

   //Loop from i=0 to 100:
   for (int i=0; i<=npoints; i++) //Starts the 'for' loop (Can declare counter i here)
      {
         t = i*dt;          //Calculates the time taken
         y = y0-0.5*g*t*t;  //Calculates the position (taking y=y0 at t=0)

         //Print the t & y values to the screen:
         cout << t << "\t" << y << endl;
         outs << t << "\t" << y << endl; //Saves results to output.txt
      } //End of the ‘for’ loop

   outs.close(); //Closes the output file

   cout<<"Press a key to exit...";
   _getch();  //Waits for a keystroke before closing the console window
   return 0;
} //End of main()
```

Points to note:

1. In C++, we must declare the types of functions and variables before we use them. The `#include` statements at the top of the program instruct the pre-processor to add header files to our source code before compilation. The header files declare `cout`, `ofstream`, `_getch()`, *etc*.
2. Throughout the program, we add many, many comments to explain the code. We also use indentation to emphasize the program’s block structure.
3. All of the executable statements in this simple program are contained inside the `main()` function.
4. We generally declare constants and variables at the start of `main()`. Declaring the parameters for the problem as `const` makes the program easy to understand and easy to modify.
5. In scientific programming, it’s a good idea to follow the FORTRAN naming convention: integer variables begin with `i - n`, and other letters are used for floating-point variables.
6. Wherever possible, match the names in the program with the symbols in the algebraic theory; for example, an angle θ could be called `theta` in C++. Using meaningful names makes the code easy to understand and leads to fewer errors.
7. In scientific programming, we usually use type `double` (64-bit) instead of `float` (32-bit) for floating-point numbers. Type `double` is more accurate and has a much wider range.
8. The `for` loop allows us to repeat a calculation a specified number of times. Note that it’s legal (and slightly more efficient) to declare the counter variable `i` inside the `for` statement.
9. It’s best to use an integer counter in the `for` loop to avoid repeatedly adding floating-point numbers, which can produce significant rounding errors. That’s why we use the single multiplication `t = i*dt` inside the loop rather than repeatedly adding `t = t+dt` (or `t += dt`).
10. The `for` loop runs from `0` to `npoints`, where `npoints` is a parameter that we set by writing `const int npoints=100` at the top of the program. Using a `const` parameter avoids burying the integer constant `100` inside the code, where it can be hard to spot. A more complex program may contain several `for` loops, so it’s useful to define `npoints` just once.
11. Division is *much* slower than multiplication, so we write `0.5*g*t*t` rather than `g*t*t/2` – it won’t make any real difference here, but it’s best to develop good programming habits.
12. To write to a file (`output.txt`), we open an output stream called `outs` of type `ofstream`. We can then use the same format for writing to `outs` as for writing to the console using `cout`.
13. For simplicity, the output uses the default format for `cout`. We just separate the output fields by a tab character with `"\t"`. (Formatting output with `cout` is messy. Programmers often use the `printf()` function instead, but the default format for `cout` works fine here.)
14. Before the program finishes, we must close the output file. (This ensures that any data in the output-stream buffer in memory is written to the disk file before the program finishes.)
15. To prevent the console window from closing before we can read the screen, we prompt the user to press a key. The function `_getch()` reads a single character from the keyboard.
16. The `main()` function must return a result of type `int`, so we finish the program with `return 0`. (In this case, the actual value returned by `main()` doesn’t matter.)

Modular programming

For more complex programs, the next stage beyond structured programming is **modular programming**. This breaks up the program into a collection of *functions* (also called *procedures*, *routines* or *modules*), each of which performs a specific task.

As a simple example, we might want to use a function to do the job of calculating the position of the particle in the above example. To do this, we could replace the line

```
      y = y0-0.5*g*t*t;  //Calculates the position (taking y=0 at t=0)
```

by

```
      y = position(y0,t);//Calculates the position (taking y=0 at t=0)
```

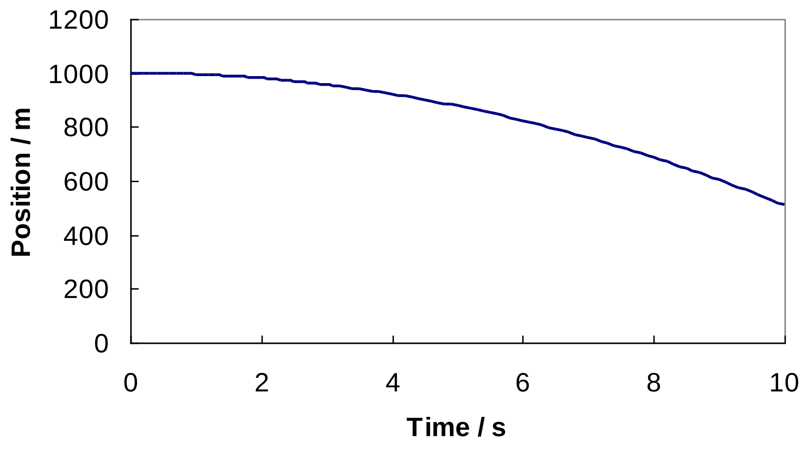
The `position()` function here takes two arguments (or parameters), both of type `double`, and it also returns a result of type `double`. We must give this information to the compiler in a *function declaration* before we use the `position()` function.

The following code from Example #4 illustrates the resulting program, which you can use as the starting point for Exercise #1 below:

```
#include <iostream>      //Declares cout
#include <fstream>       //Use for file output
#include <conio.h>       //Declares the _getch() function
using namespace std;     //Allows use of cout instead of std::cout

//Global declarations:
inline double position(double y0, double t);  //Declares the position() function

//Main program:
int main(void)
{
   //Declare constants and variables:
   const double y0=1000.;   //Initial position at t=0 (in m)
   const double tmax=10.;   //Maximum time (in s)
   const int npoints=100;      //npoints = Number of points
   const double dt=tmax/npoints;    //dt = Time increment (in s)
   double y, t;       //y = Vertical position (in m), t = time taken (in s)

   ofstream outs;           //outs is the name of the output stream
   outs.open("output.txt"); //Opens the file output.txt

   //Print a column header for the t & y values on the screen:
   cout << "t\ty\n"; //Note that "\t" = tab, “\n” = new line
   outs << "t\ty\n"; //Writes to the output file

   //Loop from t=0 to t=tmax in steps of size dt:
   for (int i=0; i<=nsteps; i++) //Starts the ‘for’ loop (Can declare counter i here)
   {
      t = i*dt;              //Calculates the time taken
      y = position(y0,t);    //Calculates the position (taking y=y0 at t=0)
      cout << t << "\t" << y << endl; //Prints the t & y values to the screen
      outs << t << "\t" << y << endl; //Saves results to output.txt
   } //End of the ‘for’ loop

   outs.close(); //Closes the output file

   cout<<"Press a key to exit...";
   _getch();  //Waits for a keystroke before closing the console window
   return 0;
} //End of main()

//Definition of the position() function:
inline double position(double y0, double t)
{
   const double g=9.81;  //Standard value of g (in m/s^2) cannot be changed.
   return y0-0.5*g*t*t;
} //End of the position() function
```

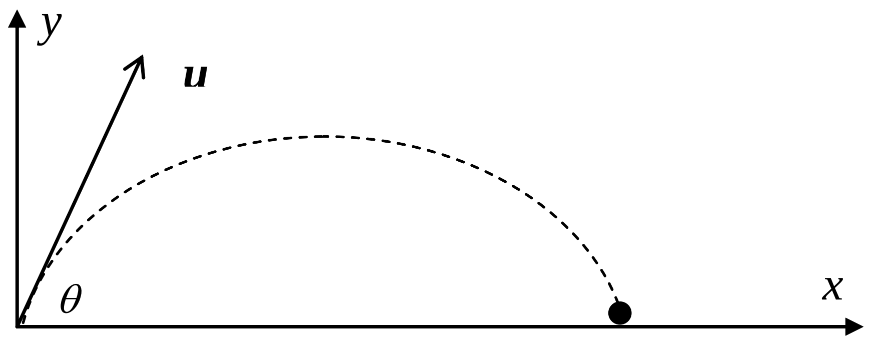
Points to note:

1. The function *declaration* (or function *prototype*) tells the compiler the number and types of parameters and the return type of the function. Parameter names are optional in the declaration, so it would be valid to declare the `position()` function more succinctly as `double position(double, double)`. However, the names of the parameters (`y0` and `t`) indicate what they actually are and hence make the code more readable.
2. The body of the function *definition* contains the actual code for the function. The argument list declares the parameter names that are used inside the function. The number and types of parameters must exactly match the function declaration.
3. Additional variables can be declared inside the function body as required. In this case, we declare `g` as a constant inside the `position()` function rather than inside `main()`.
4. Variables declared inside a function (or, indeed, inside any statement block) are not visible outside the function (or outside the statement block in which they are declared).
5. We can share variables between functions by declaring them in the 'global declarations' section outside all the functions in the program. Such 'global variables' are often useful in scientific programming.
6. In professional programming, the use of global variables is generally frowned upon. Their value can be changed from anywhere in the program, which can produce run-time errors that are hard to track down. However, variables that are declared to be `const`, as with `g` above, may be safely declared in the global section, as their value cannot be changed.
7. To make the program more efficient, the short `position()` function in Example #4 is declared to be `inline`. This is to eliminate the overhead of processing the call to the `position()` function and the return to `main()`. The `inline` keyword asks the compiler to replace the function call by the code inside the function body (which here just amounts to the expression `y0-0.5*g*t*t`, as in Example #3). The compiler itself decides whether or not to inline the function. Long functions will not be inlined, as the speed advantage is offset by the increase in code size from expanding the function.
8. The fixed parameters `y0`, `tmax` and `dt` are all declared to be `const` at the top of `main()`, as also is the number of steps `nsteps=tmax/dt`, which is calculated by the compiler. This makes the code efficient and easy to maintain.
9. Although the `position()` function is only called once in Example #4, functions are most useful when we need to perform the same calculation at different points in a program. Instead of replicating the code, we place it inside a function. We can then call that function whenever we need it. Hence, modular programming avoids unnecessary repetition of code.
10. You can use Excel to draw a graph of the results by first opening the file output.txt. Excel puts the tab-delimited data into columns, which you can then copy and paste into a spreadsheet. Select the columns and then use the chart wizard to draw a graph (an 'XY scatter chart'), as in example4.xls. This is a useful skill to master for project and lab reports and seminars, as you can copy and paste an Excel chart directly into Word or PowerPoint:



The problem: ballistic motion of a projectile

The diagram below illustrates the parabolic ballistic trajectory of a projectile with an initial speed u at an angle θ to the horizontal. The horizontal range X, the maximum height Y reached, and the time T taken to reach the maximum height are all functions of u and θ. Your first task is therefore to derive expressions for X, Y and T in terms of u, θ and g, the acceleration due to gravity. [For simplicity, neglect air resistance and the curvature and rotation of the earth.]



Ballistic trajectory of a projectile

The formulae for X, Y and T involve sines and cosines, so you will need to `#include <cmath>` at the top of your program in order to use the `sin()` and `cos()` functions in C++. The arguments of these functions are in radians, rather than degrees, so you will also need to know the value of π, which is defined as a constant in `cmath`. However, a more elegant solution uses the relation $\tan(\pi/4) = 1$, from which the compiler can calculate a `const double` value for π using the `atan()` function.

The problem—what's required

1. Write a C++ program using functions to output a four-column, tab-delimited text file containing the values of θ, $X(u, \theta)$, $Y(u, \theta)$ and $T(u, \theta)$ for θ between 0 and 90° in steps of 0.1 ° with a fixed initial speed u = 10 m s^{-1}.
2. Use Excel to produce labelled plots of each parameter versus θ.

Your solution—what's needed

1. Include in your solution clear derivations of the expressions for X, Y and T. **[3 marks]**
2. Include a clearly commented printout of your code. The marker must easily be able to follow the code and understand its logic and function. **[10 marks]**
3. Include labelled plots of X, Y and T against θ. **[3 marks]**
4. From the plot of X against θ, deduce the angle of launch that gives the largest range for the projectile. Then derive this result analytically. **[3 marks]**

Further reading

From the many books on C++, I recommend *C++ for Programmers* by Leen Ammeraal (Wiley 2000). In addition, you should consult the megabytes of comprehensive on-line help files in Visual Studio and dedicated programming websites such as www.cplusplus.com .