Queen Mary
University of London

MTH6150: Numerical Computing with C and C++

Examiners: M. Agathos

This is the Assignment for Coursework 2, counting for 80% of the final grade. The submission deadline is **Monday 05/05/2025 @ 17:00 BST**. For every period of 24 hours, or part thereof, that the assignment is overdue there will be a deduction of five per cent of the total marks available (e.g. five marks for an assessment marked out of 100). After seven calendar days (168 hours or more late) the mark will be reduced to zero. Please give yourself plenty of time ahead of the deadline, to submit your files.
For this Coursework you will need to submit via QMplus a number of source code files with the indicated names and a .pdf document with your answers to the questions that require explanation. Keep your answers concise and relevant to the question; any irrelevant, superfluous or contradictory comments will contribute negative marks. The source code files should have a few short and concise comments describing the non-trivial parts of the code. The text in the .pdf should convey your understanding of the problem and its solution, and should include plots where necessary. You can use `gnuplot` or any plotting software of your choosing to produce the plots.

In Coursework 2 we will study a set of Numerical Computing problems towards simulating a football game. You may have heard of football before, as a sport that is popular in a large part of the world, including its birthplace, England. However, no prior football knowledge is required for you to solve the questions of this coursework. You only need to know that in a game of football there are two teams of players running around in a rectangular field (pitch), trying to kick a ball (the football) into a goal (a vertical rectangular opening) at the opposite team's end of the pitch. Each team has a special player, the goalkeeper, whose job is to block the ball from going into their goal.

We will use this problem set to test our understanding of and skills in

- programming in C++;
- object oriented programming;
- fundamental concepts of Numerical Analysis;
- numerical interpolation with C++;
- numerical differentiation with C++;
- numerical integration with C++;
- solving ordinary differential equations on a computer with C++.

A football game application in C++

Question 1 [25 marks]. First, define a couple of classes for organising the content of your game. Create the first couple of C++ source code files, `player.hpp` and `team.hpp`, for your football game application, with the following contents:

(a) A `Player` class, that can keep some basic data for a given football player that features in your game. Implement this class in a file named `player.hpp`. It should have the following `public` member variables:

- `string player_name`: the full name of the player;
- `int number`: the player's number;
- `int height`: the player's height (in cm);
- `int weight`: the player's weight (in kg);
- `bool left_footed`: whether the player's dominant foot is the left one;
- the following `int` variables, taking integer values between 0 and 100 that describe the player's skills: `speed`, `stamina`, `jumping`, `shooting`, `passing`, `dribbling`, `tackling`, `marking`, `strength`;
- `float x[2]`: 2D coordinates of player's position on the pitch (in m);
- `float v[2]`: 2D vector with player's speed components (in m/s);

It should also have the following member functions, which you should define:

- a constructor that creates a new object of the `Player` class, with a `string player_name` and `int number` as its only arguments that it uses to initialize the player's name and number variables; all other variables should be initialised to 0, meaning that they are waiting to be set (and the variable `left-footed` set by default to `false`).
- three `public` functions:
  - `int get_physical_ability()` returns the average over speed, stamina and jumping;
  - `int get_offence_ability()` returns the average over the player's shooting, passing and dribbling;
  - `int get_defence_ability()` returns the average over marking, tackling and strength;
- a `public` function `void printPlayerInfo()` that prints out a structured message summarising the player's information;
- two `public` functions: `void move()` that moves the player's position `x` based on her current speed vector `v` and `void update_speed(float v_new[2])` that updates the latter with the new values in `v_new`; assume that both of these functions are called at regular intervals of $dt = 0.01$s.

[10] [3]

(b) In a new file named `team.hpp`, define the `Team` class, whose purpose is to hold the necessary data for a team and perform related functions on that data. The `Team` class should have the following members

- `private` variables for: `name` (string), `colour` (string with the team's colour) and a `PlayerList` (list container of `Player` objects [1])
- a constructor that creates a new object of the `Team` class, whose arguments are two `string` objects to initialize the team's name and colour, and a `list` of `Player` objects to initialize the `PlayerList` member; if no player list is provided it should set it to the empty list by default.
- `public` functions `getX()` and `setX()` for X in {`name`, `colours`, `PlayerList` }. Getter functions should take no arguments and should return a value of the correct type, while setter functions should take an argument of the correct type and have `void` as return type;
- a `public` function `addPlayer()` that takes a `Player` object as argument and adds it to the front of the `playerList` container;
- a `public` function `printPlayerList()` that prints out the names of all players in the team, prefixed by their player number, e.g.

```
6  : Leah Williamson
10 : Lionel Messi
```

- three `public` functions `int get_offence_ability()`, `int get_defence_ability()` and `int get_physical_ability()` that return the average of the corresponding ability grades over all players in the `PlayerList`.

[**7**] [**2**]

(c) Teams can be national teams or club teams. National teams must specify their continent as an attribute, while club teams must specify their country of origin. For example, the England national football team is a national team in Europe; FC Barcelona is a club team in Spain. Describe in your own words how you would define a class for each case and what would be their relation to the `Team` class? [**3**]

[1] You will need to include `<list>` from the standard library.

Question 2 [25 marks]. Download the C++ source file `q2.cpp` the header file `q234.hpp` and the datafile `tracking_data.dat`. This data file has measurements of the position (x, y) of an actual football player on the pitch in regular time intervals of $dt = 0.2s$, for a total duration of 200s, with an accuracy of ~ 0.2m.

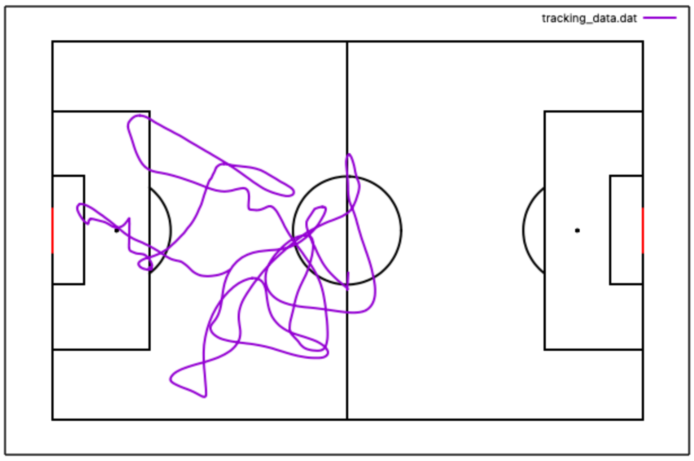

Figure 1: Tracking the path of a player on the football pitch for 200 seconds (you will find this data in the file `tracking_data.dat`).

The function `read_tracking_data()` reads the data from the file and stores all three columns in a `valarray`, while in the `main()` we copied the first column into `t_varr`, the second into `x_varr` and the third into `y_varr`.

(a) The original tracking data is provided in a normalised coordinate system, where (0,0) corresponds to the bottom left corner of the pitch as sketched in Fig. 1, and (1,1) corresponds to the top right corner. Write a function with signature

```
void coord_xfm(valarray<float> &x_out, valarray<float> &y_out,
               valarray<float> x_data, valarray<float> y_data)
```

that transforms the tracking data from these normalised coordinates (`x_data`, `y_data`) to physical coordinates (`x_out`, `y_out`) in meters, with the origin at the centre of the pitch. The physical length and width of the pitch are defined in `q234.hpp` as `PITCH_L`, `PITCH_W` respectively. For maximum marks do not use a loop. [6]

(b) In the `main()`, apply this function on the data read from file and store the results in two new `valarray` variables. [4]

(c) Use centered finite differences that are second-order accurate to compute the speed components v_x, v_y of the player. Then compute the speed magnitude $v = \sqrt{v_x^2 + v_y^2}$ and store it in a new variable `v_mag`. For maximum marks do not use loops. Output this data to a file with two columns: (t_i, v_i) and plot it. What is the maximum speed that the player reached? [6]

(d) Use centered finite differences that are second-order accurate to compute the acceleration components and acceleration magnitude; store the latter in a new variable `a_mag`. For maximum marks do not use loops. What are the lengths of the speed and acceleration `valarray` variables and why? [**1**] [**2**]

(e) Describe the different sources of error in the above computations. Which do you think is the dominant one, what type of error is it and what would you do to mitigate it? Is `float` a good choice for our real-valued variables or should we be using `double`? [**2**] [**1**]

(f) If we want to reconstruct this player's movement in our computer game using the tracking data, we face a problem: $dt = 0.2$s means a sampling rate of 5Hz (5 frames per second); however, a modern gaming device typically refreshes its output image at 120Hz, (i.e. showing 120 frames per second). Given that you only have data for a small fraction of the frames, describe what you would do and the method you would use to make the motion look smooth. [**2**] [**1**]

Question 3 [25 marks]. We want our game to keep track of each player's energy level. To approximate the energy E_A spent by player A, we map her instantaneous speed v to the power P_A that she spends when running at that speed. The integral of that power over time will give us an estimate of the total energy spent.

$$E_A(t) = \int_0^t P_A(v(t'))\mathrm{d}t' \tag{1}$$

Create a new file `q3.cpp` with a `main()` function, where you will perform this integral using the speed timeseries that you derived in the previous question. If you did not produce this data, use the provided file `speed_A.dat`.

(a) We have put real players through lab tests and monitored the power they consumed at different speeds. The data for player A is given in Table 1.

$v\,[\frac{\mathrm{m}}{\mathrm{s}}]$	$P\,[W]$
0	100
3	700
5	1100
8	2000

Table 1: Power that player A spends at different speeds.

Use the Lagrange interpolation code that we wrote for Exercise Set 6 to obtain the interpolating polynomial $P_A(v)$ that passes from the data points of Table 1. Evaluate it at $v = 5.8$. [**2**] [**6**]

(b) What degree is the polynomial? Between calling the `Lagrange_N()` function and deriving the polynomial coefficients using `interp_coeffs()`, which method would you prefer when evaluating the function on the provided speed data and why? [**2**]

(c) Compute the values $P_i = P_A(v_i) = P_A(v(t_i))$ using the timeseries of speeds and store the resulting values to a `valarray` variable called `p_i`. If in step (a) you were unable to define the interpolating polynomial, use the linear function $P(v) = 100 + 237.5\,v$ instead. [**4**]

(d) Employ the numerical integration code from the solutions of Exercise Set 8 to approximate the integral in Eq. (1) as a discrete sum $E_A(t) \simeq \sum_{i=0}^{N} c_i\,P_i$ with the appropriate coefficients, using the composite version of the trapezoid rule. [2] [**2**] [**4**]

(e) Modify `nintegrate1D()` and extend the code by implementing the next-order Newton-Cotes formula (a composite four-point formula) in a new function; use this to re-evaluate the integral. What order is the error (in powers of $1/N$) for the new formula? [**5**]

[2]If you want to assign units to your results, the power is measured in Watts [W], so the resulting energy will be in Joules [J].

Question 4 [25 marks]. Take the shot! For our game to be realistic, we need to simulate what happens when a player kicks the ball. It is the computer's job to calculate the trajectory of the ball, but only based on the physics that we implement in our game's code library. In this question, we will implement the differential equations that describe the motion of a football when airborn, starting with a very simplified model and gradually adding more physical effects, to make the behaviour more realistic. We will apply a familiar numerical ODE solver (`rk4`) to simulate a number of shots towards the goal.

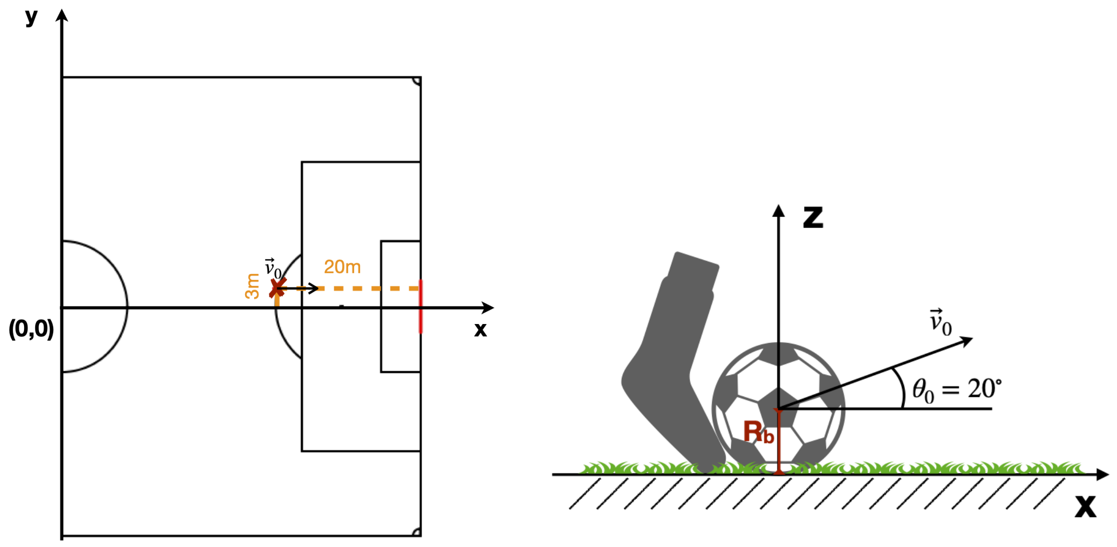

Figure 2: Left: initial positon and velocity of the ball with respect to the football pitch as viewed from above on the $x-y$ plane. Right: the initial position and velocity of the ball at the time of the kick as viewed from the side, on the $x-z$ plane.

The simplest model we can think of only includes the effect of a uniform gravitational force $\vec{F}_g$ that results in a downwards acceleration equal to $g \simeq 9.81\,m/s^2$. Newton's second law

$$m\,\vec{a} = \vec{F}_g = -mg\hat{z}\,, \tag{2}$$

(where $\hat{z}$ is the unit vector pointing upwards) gives us the equation of motion in the vertical direction:

$$z''(t) = a_z = -g \quad , \tag{3}$$

whilst $x'' = 0$ and $y'' = 0$ because no forces act on those directions. Here again we place the origin $\vec{O} = (0,0,0)$ of the coordinate system $\vec{x} = (x,y,z)$ right at the centre of the football pitch, at ground level. This is a second order system of ODEs, which we can reformulate as a first order one

$$\vec{x}\,'(t) = \vec{v}(t) \tag{4}$$

$$\vec{v}\,'(t) = -g\hat{z} \quad . \tag{5}$$

The system of equations governing the motion of the football is thus given by this system of six first order ODEs

$$\vec{Y}'(t) := \begin{bmatrix} x'(t) \\ y'(t) \\ z'(t) \\ v_x'(t) \\ v_y'(t) \\ v_z'(t) \end{bmatrix} = \begin{bmatrix} v_x(t) \\ v_y(t) \\ v_z(t) \\ 0 \\ 0 \\ -g \end{bmatrix} =: \vec{f}(t, \vec{Y}) \tag{6}$$

where we denote $\vec{Y} = (x, y, z, v_x, v_y, v_z)$. As there are no forces along the x or y axes in this simplified model, the ball will move with constant speed along the horizontal direction determined by its initial velocity. On the vertical dimension (z-axis), the only force acting on the ball is that of gravity, causing a constant downwards acceleration.

(a) Download the source files

  (i) `rk4.cpp` that implements the Runge-Kutta method RK4;

  (ii) `q234.hpp` that contains some constants (dimensions of the pitch, goal and ball, physical constants, etc.), and prototypes of the functions that will be used in this exercise;

  into your working directory, to solve this system of ODEs and go through them carefully. Create a new file `q4.cpp` with a `main()` function. First implement the vector of functions on the right hand side (RHS) of Eq. (6) into the `rhs` function. The initial value problem (IVP) is complete once we provide the inital data for the ball's position $\vec{x}(t = 0) = \vec{x}_0 = (x_0, y_0, z_0)$ and velocity $\vec{v}(t = 0) = \vec{v}_0 = (v_{x,0}, v_{y,0}, v_{z,0})$. Use the following information to set their values (in units of m and m/s respectively). The moment the player kicks the ball, at $t = 0$:

  - the ball is touching the ground;
  - the ball is located 20m from the goal line and 3m left to the centre of the goal (see Fig.2);
  - the ball has an initial velocity $\vec{v_0}$, with $|\vec{v_0}| = 25$ m/s, with a direction straight towards the goal ($v_{0,y} = 0$) and an upwards angle of $\theta_0 = 20°$ (see Fig.2).

  In the `main()` function, set up the initial conditions for all six components into a `valarray` variable. Print out a line to standard output, with the values of the start time, initial positions and initial velocities seperated by spaces. **[3]** **[2]**

(b) Write a loop that in each iteration evolves the state vector by a single step, using the `rhs` function that you wrote with $dt = 0.01$s. Set its condition so that it evolves the system of ODEs until the ball crosses the goal line at $x =$`PITCH_L`$/2$ or until it reaches the ground again. In each step, print out the time and the components of the state vector $\vec{Y}$ to standard output in exactly the same format as in the previous step. Compile your code and run your program, redirecting the output to a file `simple_v25.dat`. Plot the resulting coordinate data on the (x, z) plane using `gnuplot` or an alternative plotting software of your choice. If the ball crosses the goal line (the $x =$`PITCH_L`$/2$ plane) clear under the horizontal bar at $z =$`GOAL_H` without touching it, the shot is **good**; if it goes over, the shot is **over**;

if the ball touches the ground before it gets there, then the shot is **weak**. At the end of your program, evaluate the outcome of the shot and add a print statement to `cerr` to inform the user about it. [**2**] [4]

(c) Repeat the process for shots with different magnitudes of the initial velocity $|\vec{v}_0| = \{20, 22, 24, 26, 28, 30\}$ m/s, and redirect the output to different, appropriately named files. Don't hardcode the value each time, but instead introduce a command-line argument using `argc` and `argv[]`, like we did in the exercises, that can override the default value (25 m/s). Again, plot the trajectories of the different shots in the same figure and compare. Which ones are good, over, weak? Are the comparisons between results consistent with your mathematical and physical intuition? [**3**]

(d) In the above oversimplified version of the problem, we have neglected a crucial physical effect, the air resistance. The presence of air introduces a drag force, directed opposite to the ball's velocity, slowing the ball down as it moves through the air. This drag force can be modelled as

$$F_d = \frac{1}{2} C_d \, \rho \, A \, |v| \vec{v} \,,$$

where C_d is known as the drag coefficient (find its value in the header file) and $A = \pi R_b^2$ is the cross-sectional area of the football. What will Eq.(6) look like, if we again start from Eq. (2) (Newton's 2nd law) and introduce this new force, in addition to gravity? Add the effect of the drag force to the RHS. Do it in such a way that you can switch the effect on or off by setting the value of the global variable `drag_on` (declared in `q234.hpp`) to `true` or `false` respectively. Compile your code and run the new version of your program with $v_0 = 25$ m/s. Is the shot good, over, or weak? Plot the trajectory on the (x, z) plane together with its zero-drag counterpart. How much time does the goalkeeper have before the ball crosses the goal line? [**4**]

(e) **"Bend it like Beckham!"** Your player is about to take the straight shot at 25 m/s at a 20° tilt angle, but the goalkeeper is well positioned to block the shot. Due to pressure from defenders you cannot aim further to the left, or your shot will get deflected. There is however one thing you can do to send the ball closer to the left corner of the goal: add some spin to it. The trajectory of a spinning ball will curve towards a direction that is perpendicular to its velocity and its axis of rotation. This is known as the Magnus effect and can be incorporated into the equations of motion as an additional force

$$\vec{F}_m = S(\vec{\omega} \times \vec{v}) \,, \tag{7}$$

where S is the football's lift coefficient and $\vec{\omega}$ is its angular velocity and is assumed to be along the z-axis $\vec{\omega} = (0, 0, \omega_z)$ (see header file for the default values). Hence the RHS will have an additional contribution $\vec{F}_m/m$ that looks like

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ -\frac{S}{m}\omega_z v_y \\ +\frac{S}{m}\omega_z v_x \\ 0 \end{bmatrix} \tag{8}$$

What does the new version of Eq.(6) look like, if we add both the air resistance and the Magnus effect? Add the Magnus effect to the `rhs` function. Again, do it in such a way that it can be switched off by setting the global variable `magnus_on` to `false`. [**4**]

(f) Run your program including the Magnus effect, using the default value $\omega_z = 10$. Redirect the output to a new data file and plot the resulting trajectory on the (x, y) plane. Confirm that the shot went in. How close is the ball to the vertical goalpost at $y =$`GOAL_W`$/2$ the moment it crosses the goal line? [**3**]