Name: Helena Latorre Moreno

Student ID: 220561802

Question 1

(c) National Teams and Club teams:

I would create two additional header files: `national_team.hpp` and `club_team.hpp`, creating two new classes: NationalTeam and ClubTeam that are specialised versions of the class Team and they publicly inherit from it. I would make the ClubTeam class by writing `class ClubTeam : public Team`. So, a club team is a special kind of team, so it automatically gets everything in Team. I would add one extra private member: the country the club belongs to, plus get/set functions for it. The constructor of ClubTeam forwards the shared attributes (name, colour, list of Players) to the Team constructor in the initialiser list, then initialises country. Similarly, for NationalTeam, I would define `class NationalTeam : public Team` and add one private data member std::string continent. Its constructor forwards the name, colour and the list of players to the Team base constructor, then initialises continent. Because the inheritance is public, every method that is public in Team, such as addPlayer(), printPlayerList(), or the three average-ability functions, would remain public in these classes.

Question 2

(c) Maximum speed and plot: Using second-order centered differences to compute the speed components I obtained the following results. Figure 1 shows that the footballer's peak speed is $\mathbf{6.83\,m\,s^{-1}}$.

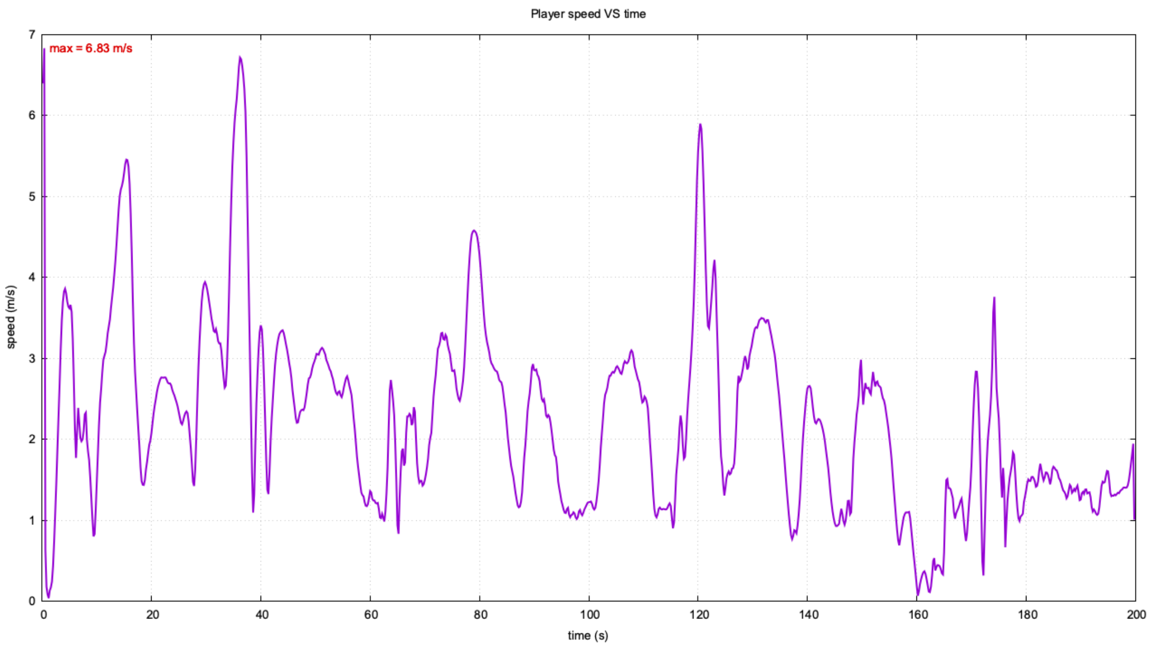

Figure 1: Player speed versus time (5 Hz, 0.2 s spacing).

(d) Lengths of the speed and acceleration valarray variables: Using the second derivative approximation using centered differences I computed the acceleration components. Both formulas require data at t_{n-1}, t_n, and t_{n+1}. At the first time point $t_0 = 0$, the value $x(t_{-1})$ is not available. Similarly, at the last time point $t_N = 200$, the value $x(t_{N+1})$ is not available.

As a result, we cannot compute centered finite differences at the endpoints t_0 and t_N. Because one grid point is lost at each end of the arrays, the **speed** and **acceleration** arrays contain $N - 2 = 1001 - 2 = 999$ elements, where $N = 1001$ is the length of the original position array.

(e) Main sources of error:

- **Truncation Error:** The centered differences are only second-order accurate, i.e. accurate to $\mathcal{O}(dt^2)$, meaning the data quality and simulation duration increase significantly with a smaller time step. To mitigate this error, we could use higher-order centered differences or use a smaller time difference dt if we have available data.
- **Round-off Error (Float or Double):** In C++, a float has around 7 digits of precision. Since dt is very small, the difference between $f(x - dt)$ and $f(x + dt)$ is very small, so subtraction can lead to round-off errors that can accumulate over arithmetic operations. On the other hand, doubles have around 15 digits of precision, so they are better to work with to reduce this error.
- **Boundary Error:** At the first and last time points, centered differences cannot be applied, so it results in the loss of data in the boundaries. Alternative methods like forward or backwards differences could be used but have lower accuracy ($\mathcal{O}(dt)$), increasing error at the endpoints.

In my opinion, the **dominant error is the truncation error** since the boundary error only affects 2 data points, and the difference between $f(x - dt)$ and $f(x + dt)$ is not that small, so the round-off error is less significant.

(f) Reconstructing the player's motion: In order to make the motion look smooth I would use Catnum-Roll spline interpolation which gives you the smoothness of a single high-order polynomial without its spiky patterns. The method has the following properties:

- Treats the measured positions as control points $x_0, x_1, ..., x_N$ at $t_0, t_1, ..., t_N$, so the curve will pass through all of these points.
- The spline is C^1 continuous, so its tangent vector changes smoothly.

Evaluating it every $\frac{1}{120} = 0.833333$ seconds creates smooth 120 Hz motion. Because the spline is C^1 continuous, we can also obtain velocity by direct differentiation.

Question 3

(a) Evaluating it at v = 5.8m/s

$$P_A(5.8) = 1292.48W$$

(b) Polynomial degree: The Lagrange interpolation polynomial has a degree that is one less than the number of data points used for interpolation. Specifically, we have n+1 = 4 data points, the resulting **Lagrange polynomial is of degree 3**.

When evaluating the function on the speed data, calling interp_coeffs() once to store the coefficients and then using poly_eval() inside the 999 points is much faster and accumulates fewer rounding errors than calling Lagrange_Nk() for every speed sample and recomputing the full Lagrange sum each time. To produce the coefficients with interp_coeffs() we use Newton's divided differences method and store a vector of coefficients **coeffs** which is set up once and we can reuse it everywhere. Afterwards, determine the Newton polynomial with poly_eval(). So the number of operations in this method is significantly lower, making it safer of propagation errors and faster. The one time setup interp_coeffs() has $O(n^2)$ complexity and each evaluation takes $O(n)$ operations, instead of $O(n^2)$ for each evaluation with the Lagrange form.

(d) Composite trapezoid rule: Approximating the integral in Eq. (1) as a discrete sum using the composite version of the trapezoid rule I obtained:

$$E_A(t) \approx \sum_{i=0}^{N} c_i P_i = 111529 \text{ J}$$

(e) Simpson's 3/8 method We have 999 data points therefore we have 998 intervals so I used Simpson's 3/8 rule for the first 996 intervals because it's a multiple of 3. That left 2 remaining intervals, which were integrated using Simpson's 1/3 rule, since it works on 2 intervals. Therefore I obtained:

$$E_A(t) \approx \sum_{i=0}^{N} c_i P_i = 111532 \text{ J}$$

Since we used Simpson's 3/8 method + Simpson's 1/3 method for the last two intervals, that does not change the error term. The error in powers of 1/N is of **fourth order**, i.e. ($\mathcal{O}(N^{-4})$)

Question 4

(b) Plot simple_v25.dat

Using Gnuplot, the computed trajectory data were plotted on the (x, z) plane. The resulting plot is presented in Figure 2.

(c) Outcome of various velocities

Yes, the comparisons between the results are completely consistent with my mathematical and physical intuition. Figure 3 displays the plotted data.

$\lvert v_0 \rvert$ (m/s)	Outcome
20	GOOD SHOT!
22	The ball went over the horizontal bar
25	The ball went over the horizontal bar
26	The ball went over the horizontal bar
28	The ball went over the horizontal bar
30	The ball went over the horizontal bar

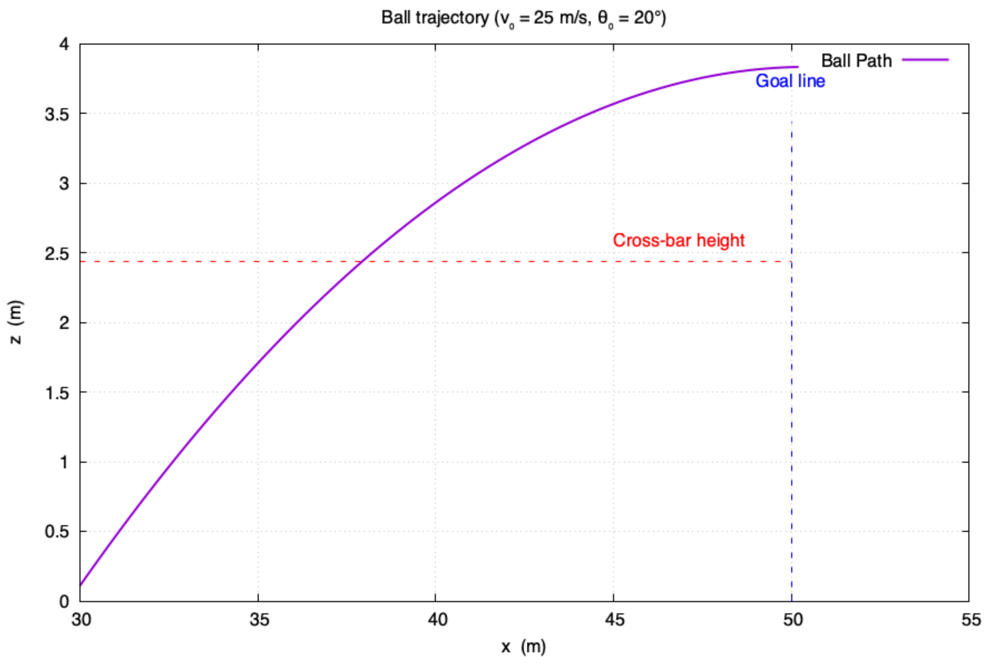

Figure 2: Ball trajectory (without F_d or F_m)

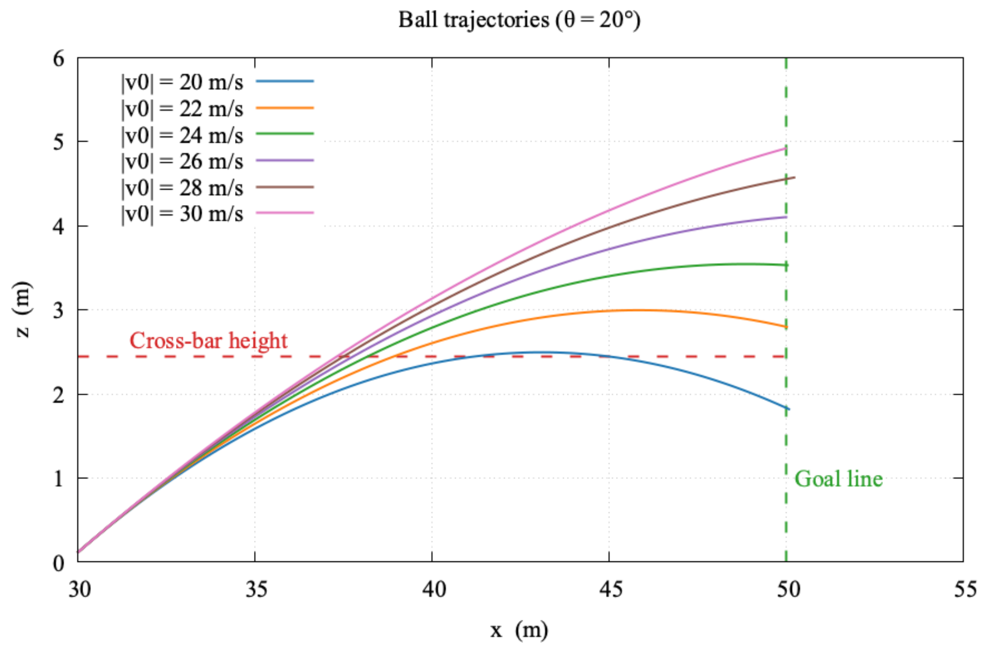

Figure 3: Ball trajectories (without F_d or F_m)

(d) Drag Force: I plotted the trajectory on the (x, z) plane together with its zero-drag counterpart, displayed in Figure 4. Taking into account the drag force **the goalkeeper has 1.129999 seconds before the ball reaches the goal line**.

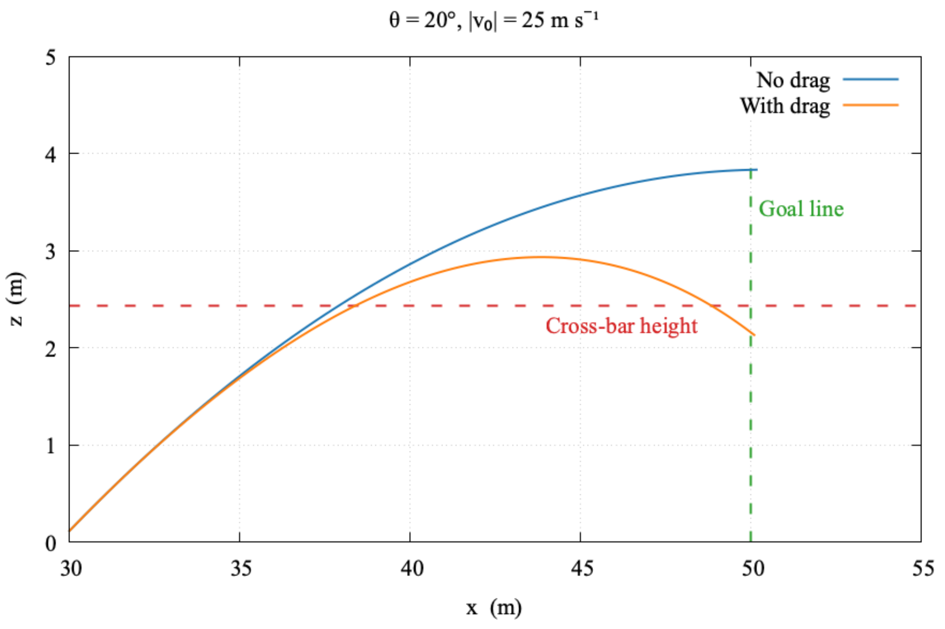

Figure 4: Comparison of Ball Trajectories: With vs. Without Drag Force F_d

(e) Magnus Effect: The new version of Eq.(6) if we add both the air resistance and the Magnus effect looks like:

$$\vec{Y}'(t) := \begin{bmatrix} x'(t) \\ y'(t) \\ z'(t) \\ v_x'(t) \\ v_y'(t) \\ v_z'(t) \end{bmatrix} = \begin{bmatrix} v_x(t) \\ v_y(t) \\ v_z(t) \\ -D|v|v_x - Mv_y \\ -D|v|v_x + Mv_y \\ -g - D|v|v_z \end{bmatrix} =: \vec{f}(t, \vec{Y})$$

where

$$D = \frac{C_d \rho A}{2m} \qquad\qquad M = \frac{S\omega_z}{m}$$

(f) Yes, the shot went in, the trajectory data was plotted and is shown in Figure 5. The ball is at $y = 3.476309$ when it crosses the goal line, and the left vertical goalpost is at $y = 3.66$. **So the ball is 0.183691 m away from the goalpost**.

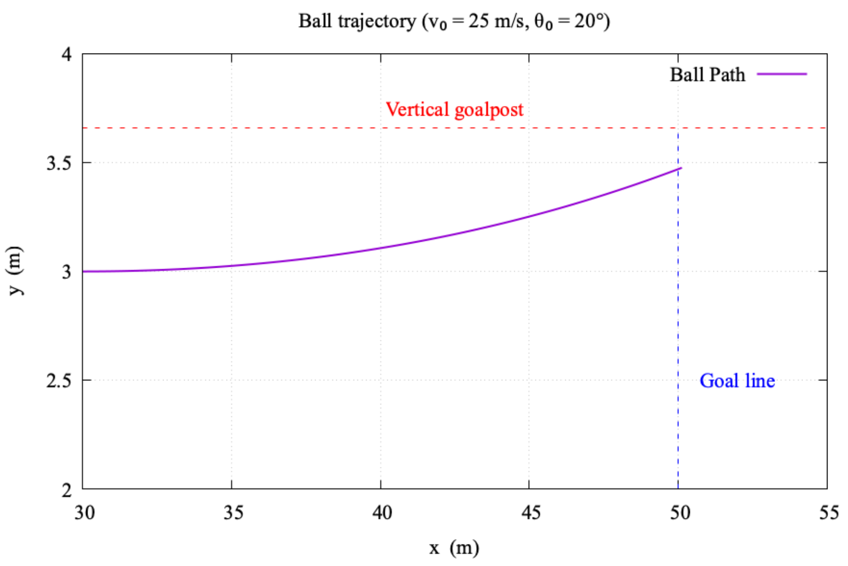

Figure 5: Ball Trajectory Considering Drag F_d and Magnus Effect F_m