Numerical Methods in C++: Motion Tracking and Trajectory Simulation

Helena Latorre Moreno

December 26, 2025

Abstract

This report presents C++ implementations of core numerical methods used in data-driven kinematics and dynamical simulation. We estimate velocity and acceleration from discrete tracking observations using finite differences, reconstruct smooth motion via spline interpolation, approximate energy expenditure using quadrature rules, and simulate 3D trajectories using a Runge–Kutta method with optional aerodynamic forces.

Contents

1 Introduction

This report documents a small C++ workflow for motion modelling and simulation, combining object-oriented software design with core numerical methods. The overall aim is to support a simple sports-motion prototype: representing entities (players and teams), extracting kinematic information from discrete tracking data, estimating derived quantities via interpolation and quadrature, and simulating projectile motion by numerically integrating ordinary differential equations.

For the data-driven component, we analyse planar tracking measurements sampled at regular time intervals ($\Delta t = 0.2$s over 200s, with positional accuracy on the order of 0.2m). After transforming the recorded normalised coordinates into physical coordinates centred on the domain, we estimate velocity and acceleration using second-order centred finite differences and report summary quantities (e.g. peak speed) together with diagnostic plots. Since the measurement rate (5 Hz) is far lower than a typical display refresh rate (e.g. 120 Hz), we also discuss how to reconstruct smooth motion from sparse samples using spline interpolation and resampling on a finer time grid.

For energetic analysis, we interpolate a power–speed relationship efficiently (via divided differences / Newton form) and approximate the time integral of power using composite quadrature rules, enabling comparison between trapezoidal and Simpson-type schemes.

Finally, we simulate 3D projectile trajectories by formulating the dynamics as a first-order ODE system and integrating it with a fourth-order Runge–Kutta method (RK4), progressively adding physical effects such as drag and the Magnus force. Numerical outputs are visualised to support qualitative checks and interpretation.

2 Software Design

2.1 Team abstraction and specialisations

To represent different types of teams while reusing shared functionality, two derived classes are introduced: `ClubTeam` and `NationalTeam`, defined in `club_team.hpp` and `national_team.hpp`, respectively. Both classes *publicly* inherit from the base class `Team`, i.e. `class ClubTeam : public Team` and `class NationalTeam : public Team`. Public inheritance preserves the interface of `Team`, so existing public methods (e.g. `addPlayer()`, `printPlayerList()`, and the average-ability functions) remain accessible for both specialisations.

Each derived class extends `Team` with one additional attribute: `ClubTeam` stores the club's `country`, and `NationalTeam` stores the team's `continent`. Both classes provide simple getter/setter methods for the new field. Constructors forward the shared state (`name`, `colour`, and the `Players` list) to the `Team` constructor via the initializer list, and then initialise the additional member, ensuring code reuse and avoiding duplication of base-class setup.

3 Methods

3.1 Kinematics from discrete tracking data

3.1.1 Finite-difference estimators

Let $\{(x_n, y_n)\}_{n=0}^{N-1}$ denote the planar position samples recorded at uniform times $t_n = n\Delta t$, with sampling interval $\Delta t = 0.2\,\mathrm{s}$ (5 Hz). To estimate kinematic quantities from these discrete observations, we use second-order centred finite differences, which are symmetric and achieve $\mathcal{O}(\Delta t^2)$ truncation error for sufficiently smooth trajectories.

Velocity. For interior indices $n = 1, \ldots, N-2$, the velocity components are approximated by

$$v_x(t_n) \approx \frac{x_{n+1} - x_{n-1}}{2\Delta t}, \qquad v_y(t_n) \approx \frac{y_{n+1} - y_{n-1}}{2\Delta t}. \tag{1}$$

The scalar speed is then computed as

$$s(t_n) = \|v(t_n)\| \approx \sqrt{v_x(t_n)^2 + v_y(t_n)^2}. \tag{2}$$

Acceleration. Acceleration is obtained using the centred second-derivative formula (equivalently, differentiating the velocity again with a centred rule):

$$a_x(t_n) \approx \frac{x_{n+1} - 2x_n + x_{n-1}}{\Delta t^2}, \qquad a_y(t_n) \approx \frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta t^2}, \qquad n = 1, \ldots, N-2. \tag{3}$$

Endpoint handling. Centred differences require values at $n \pm 1$, so velocity and acceleration cannot be computed at the endpoints $n = 0$ and $n = N-1$. Consequently, the derived arrays (speed/acceleration) have length $N-2$ and correspond to times $t_1, \ldots, t_{N-2}$. In implementation, computations are performed in double precision to reduce round-off effects.

3.1.2 Implementation notes

The tracking data are read from file into arrays of length N containing uniformly sampled times $t_n = n\Delta t$ and planar positions (x_n, y_n), with $\Delta t = 0.2\,\mathrm{s}$ and $N = 1001$. All computations are performed in double precision.

Centred finite differences require neighbouring samples at indices $n \pm 1$, so derivative-based quantities are computed only on the interior grid points $n = 1, \ldots, N-2$. Consequently, the velocity, acceleration, and speed arrays have length $N - 2 = 999$ and correspond to the time vector $\{t_n\}_{n=1}^{N-2}$ (i.e. the endpoints t_0 and t_{N-1} are excluded). This choice preserves the second-order accuracy of the centred scheme; using one-sided differences at the endpoints would reduce accuracy to $\mathcal{O}(\Delta t)$ locally.

For plotting and downstream calculations, derived quantities are written to output files together with their associated time stamps (e.g. columns t_n and $s(t_n)$ for speed), ensuring consistent alignment between arrays.

3.1.3 Error sources

- **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.

3.2 Motion reconstruction via spline interpolation

The tracking measurements are available at 5 Hz, which is too coarse to produce visually smooth motion and can lead to jagged derivative estimates. To reconstruct a smooth continuous trajectory from the discrete samples, we use a **Catmull–Rom spline**, i.e. a piecewise cubic interpolant that passes through all measured positions while remaining smooth between samples.

Let $p_n = (x_n, y_n)$ denote the measured planar positions at times $t_n = n\Delta t$ with $\Delta t = 0.2\,\mathrm{s}$. Catmull–Rom splines can be written as cubic Hermite segments on each interval $[t_n, t_{n+1}]$, using tangent vectors estimated from neighbouring points. For interior indices, we take

$$m_n = \frac{p_{n+1} - p_{n-1}}{2},$$

which yields a curve that is continuous with continuous first derivative (C^1 continuity).

For $t \in [t_n, t_{n+1}]$ we define the local parameter

$$u = \frac{t - t_n}{t_{n+1} - t_n} \in [0, 1],$$

and evaluate the interpolated position using the Hermite form

$$p(t) = h_{00}(u)p_n + h_{10}(u)m_n + h_{01}(u)p_{n+1} + h_{11}(u)m_{n+1},$$

where

$$h_{00}(u) = 2u^3 - 3u^2 + 1, \quad h_{10}(u) = u^3 - 2u^2 + u, \quad h_{01}(u) = -2u^3 + 3u^2, \quad h_{11}(u) = u^3 - u^2.$$

(Endpoints can be handled by using one-sided tangents or duplicating the first/last point.)

Resampling at 120 Hz. To obtain smooth playback, the spline is evaluated on a finer grid with step $\Delta t_{\text{fine}} = \frac{1}{120}\,\mathrm{s} \approx 0.00833\,\mathrm{s}$, producing a 120 Hz trajectory.

Velocity from the spline. Since the spline is C^1, velocities can be obtained by differentiating $p(t)$. Writing $v(t) = p'(t)$ and using $u = (t - t_n)/\Delta t$ gives $du/dt = 1/\Delta t$, so

$$v(t) = \frac{dp}{dt} = \frac{1}{\Delta t}\frac{dp}{du},$$

which provides a smooth velocity estimate consistent with the interpolated motion.

3.3 Power interpolation and energy estimation

3.3.1 Power as a function of speed

A discrete power–speed relationship $P_A(v)$ is provided at four speed values. To evaluate power at arbitrary speeds (in particular, at the sampled speeds from the tracking data), we construct an interpolating polynomial through these data points. With $n+1=4$ points, the interpolant is a cubic polynomial (degree 3).

Although the Lagrange form is conceptually simple, repeatedly evaluating the full Lagrange sum at each speed sample is computationally wasteful. Instead, we compute the interpolating polynomial in *Newton form* using divided differences: coefficients are computed once via `interp_coeffs()` (one-time $\mathcal{O}(n^2)$ setup), and the polynomial is then evaluated efficiently at each sampled speed using `poly_eval()` (each evaluation $\mathcal{O}(n)$). This reduces both runtime and the accumulation of rounding error relative to recomputing the Lagrange basis for every sample.

As a numerical check, evaluating the interpolant at $v = 5.8\,\mathrm{m\,s^{-1}}$ gives

$$P_A(5.8) = 1292.48 \text{ W}.$$

Energy via composite quadrature. Let $P_i := P_A(s(t_i))$ denote the power evaluated at the sampled speeds, with uniform time step $\Delta t = 0.2\,\mathrm{s}$. The total energy is approximated by

$$E_A \;=\; \int_0^T P_A(s(t))\,dt \;\approx\; \sum_i w_i P_i,$$

where the weights $\{w_i\}$ depend on the quadrature rule.

Composite trapezoidal rule. Using the composite trapezoidal rule,

$$E_A \approx \Delta t \left(\tfrac{1}{2}P_0 + \sum_{i=1}^{N-1} P_i + \tfrac{1}{2}P_N \right),$$

which yields

$$E_A \approx 111529 \text{ J}.$$

Simpson-type rule (3/8 + 1/3). The speed series contains 999 samples, hence 998 subintervals. Since 998 is not a multiple of 3, we apply Simpson's 3/8 rule over the first 996 subintervals (a multiple of 3), and use Simpson's 1/3 rule over the remaining two subintervals. This gives

$$E_A \approx 111532 \text{ J}.$$

Both Simpson rules are fourth-order accurate for smooth integrands, so the overall scheme retains $\mathcal{O}(\Delta t^4)$ accuracy (equivalently $\mathcal{O}(N^{-4})$ for fixed T).

3.4 Trajectory simulation

We model the projectile as a point mass moving in 3D under gravity, with optional aerodynamic forces. Let the state vector be

$$Y(t) = \big(x(t), y(t), z(t), v_x(t), v_y(t), v_z(t)\big)^{\mathsf{T}}, \qquad v(t) = (v_x, v_y, v_z)^{\mathsf{T}}.$$

The dynamics are written as a first-order ODE system

$$\frac{dY}{dt} = f(Y),$$

with the kinematic relations

$$x' = v_x, \qquad y' = v_y, \qquad z' = v_z.$$

Gravity contributes $-g$ in the vertical direction:

$$v_x' = 0, \qquad v_y' = 0, \qquad v_z' = -g \quad \text{(in vacuum)}.$$

Drag model

To account for air resistance, we use a quadratic drag force opposing the velocity:

$$a_{\mathrm{d}}(v) = -D\,\|v\|\,v, \qquad D = \frac{C_d \rho A}{2m},$$

where C_d is the drag coefficient, ρ the air density, A the cross-sectional area, and m the mass. With drag, the acceleration becomes

$$v' = -g\,e_z + a_{\mathrm{d}}(v), \quad e_z = (0,0,1)^{\mathsf{T}}.$$

Magnus effect

Spin-induced lift is included via a Magnus term proportional to v and the spin vector ω:

$$a_{\mathrm{m}}(v) = \frac{S}{m}\,(\omega \times v),$$

where S is a lift coefficient. For spin around the vertical axis $\omega = (0,0,\omega_z)$ this reduces to a horizontal acceleration orthogonal to the velocity. Writing

$$M = \frac{S\omega_z}{m},$$

one may use the equivalent component form

$$a_{\mathrm{m}}(v) = M\,(-v_y,\ v_x,\ 0)^{\mathsf{T}},$$

where the sign depends on the chosen spin direction convention.

Combining effects, the full model takes the form

$$v' = -g\,e_z - D\|v\|v + a_{\mathrm{m}}(v).$$

Time integration (RK4)

The ODE is integrated using the classical fourth-order Runge–Kutta method (RK4) with fixed time step Δt. Given $Y_n \approx Y(t_n)$, RK4 computes

$$\begin{aligned} k_1 &= f(Y_n), \\ k_2 &= f\!\left(Y_n + \tfrac{\Delta t}{2} k_1\right), \\ k_3 &= f\!\left(Y_n + \tfrac{\Delta t}{2} k_2\right), \\ k_4 &= f\!\left(Y_n + \Delta t\, k_3\right), \end{aligned} \qquad Y_{n+1} = Y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right).$$

Trajectory samples (x_n, y_n, z_n) are written to file for plotting and comparison across force models.

Stopping criteria

The integration is terminated when the projectile reaches a prescribed target plane (e.g. $x \geq x_\star$), or when it hits the ground ($z \leq 0$). Additional geometric checks (e.g. height at $x = x_\star$ or lateral position bounds) can be evaluated at the stopping time to classify the outcome.

3.4.1 Model parameters

Table 1 summarises the parameters used in the trajectory simulations.

Symbol / name	Description	Value
Δt	RK4 time step	0.2 s
g	Gravitational acceleration	9.812 m/s^2
R	Projectile radius	0.111 m
m	Projectile mass	0.436 kg
ρ	Air density	1.22 kg/m^3
C_d	Drag coefficient	0.473
S	Magnus coefficient	0.002
ω_z	Spin rate (about z axis)	10 rad/s
L	Domain length	100.0 m
W	Domain width	64.0 m
W_g	Target width	7.32 m
H_g	Target height	2.44 m

Table 1: Simulation constants used in the dynamical model (defined in `q234.hpp`).

In all experiments, the drag and Magnus terms can be toggled via the boolean flags `drag_on` and `magnus_on`.

4 Results

4.1 Speed profile and peak speed

Figure 1 shows the speed time series computed from the tracking data using second-order centred finite differences with sampling interval $\Delta t = 0.2\,\text{s}$ (5 Hz). The speed varies over the observation window and attains a peak value of

$$s_{\max} = 6.83\,\text{m/s}.$$

This peak occurs at a time corresponding to the maximum of the computed speed array.

4.2 Energy estimates

Applying the quadrature schemes described in Section 3.3 to the sampled power series yields

$$E_A^{\text{trap}} = 111529\ \text{J}, \qquad E_A^{\text{Simpson}} = 111532\ \text{J}.$$

The difference is 3 J, corresponding to a relative discrepancy of

$$\frac{|E_A^{\text{Simpson}} - E_A^{\text{trap}}|}{E_A^{\text{Simpson}}} \approx 2.7 \times 10^{-5},$$

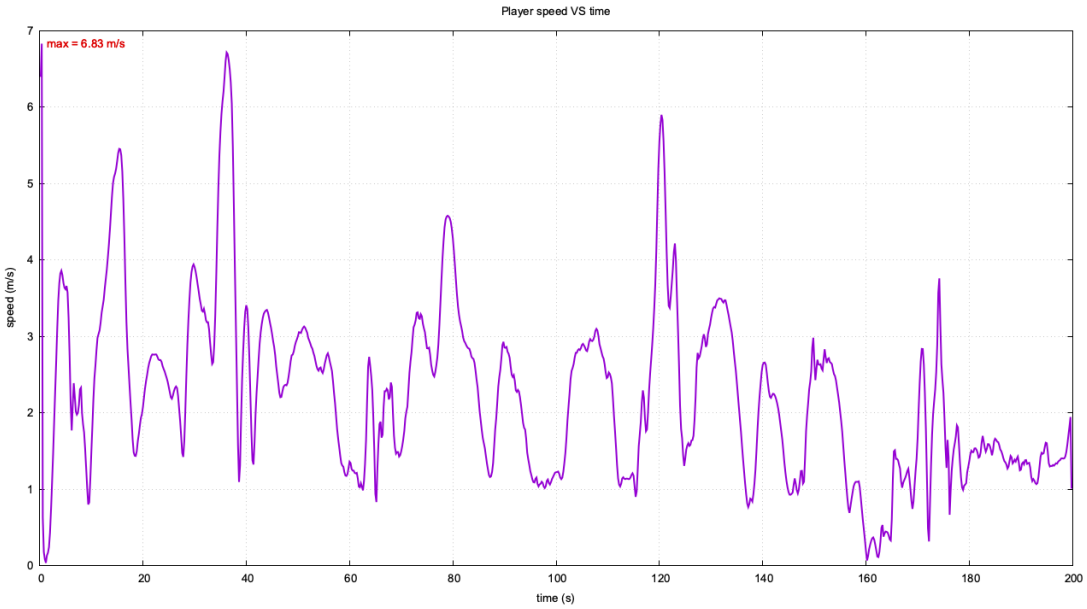

Figure 1: Estimated speed versus time from discrete tracking data (5 Hz).

so both rules give essentially the same energy estimate on this dataset. This agreement suggests that the quadrature discretisation error is small compared with other sources of uncertainty (e.g. measurement noise and the accuracy of the interpolated power–speed relationship).

Method	E_A (J)
Composite trapezoidal rule	111529
Simpson-type (3/8 + 1/3)	111532

Table 2: Energy estimates from two composite quadrature rules.

4.3 Trajectory outcomes

We simulated 3D trajectories using RK4 under three progressively richer force models: (i) gravity only, (ii) gravity + drag, and (iii) gravity + drag + Magnus effect. Trajectories were exported and visualised in the (x, z) and (x, y) planes to compare range, peak height, and lateral deviation.

Baseline trajectories (no drag, no Magnus). Figure 2 shows a representative trajectory in the (x, z) plane for a fixed initial speed, while Figure 3 compares trajectories for multiple launch speeds. As expected, increasing v_0 increases both range and maximum height. The qualitative behaviour across $|v_0| \in \{20, 22, 25, 26, 28, 30\}$ m/s is consistent with ballistic intuition.

$\lvert v_0 \rvert$ (m/s)	Outcome
20	Successful
22	Above target height
25	Above target height
26	Above target height
28	Above target height
30	Above target height

Table 3: Outcome classification versus initial speed for the baseline model (gravity only; drag and Magnus disabled).

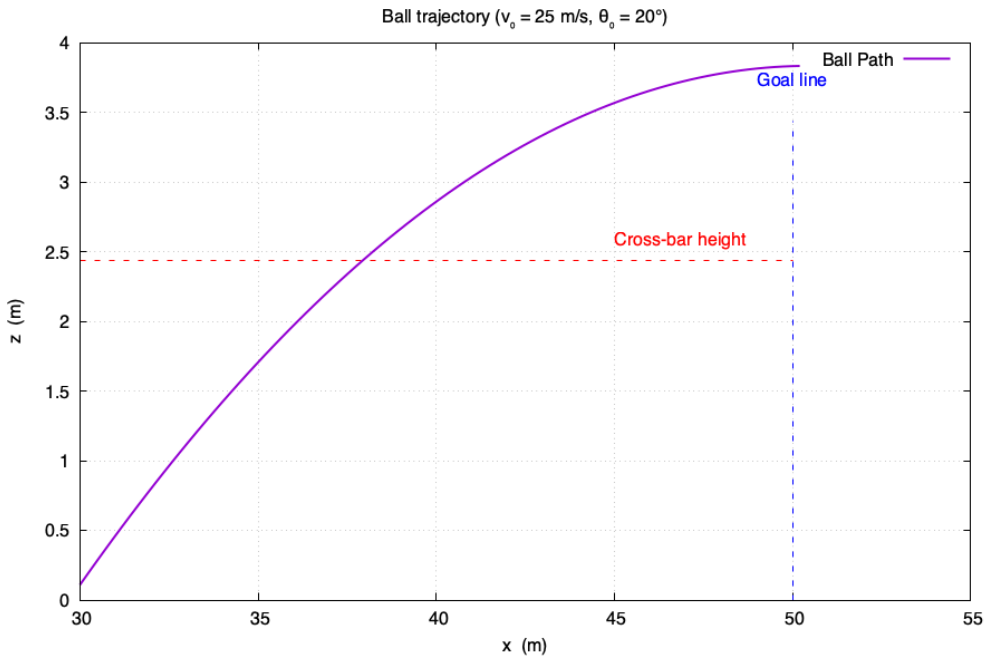

Figure 2: Trajectory in the (x, z) plane (gravity only; drag and Magnus disabled).

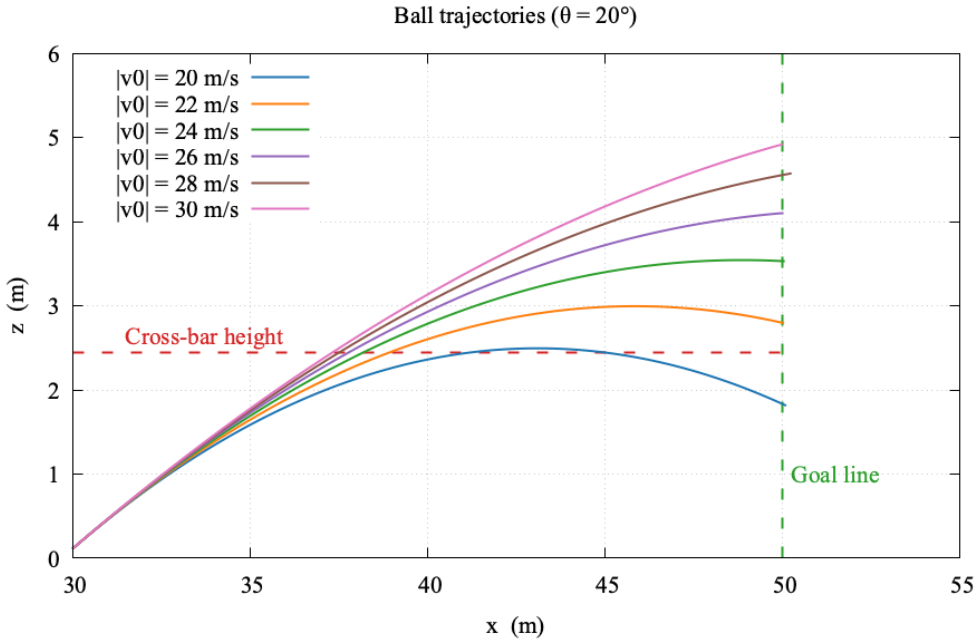

Figure 3: Trajectories in the (x, z) plane for multiple initial speeds (gravity only).

Effect of drag. Including drag reduces the range and peak height relative to the no-drag trajectory, as shown in Figure 4. Under this model the time to reach the target plane is 1.129 999 s, which can be interpreted as the available reaction time before crossing.

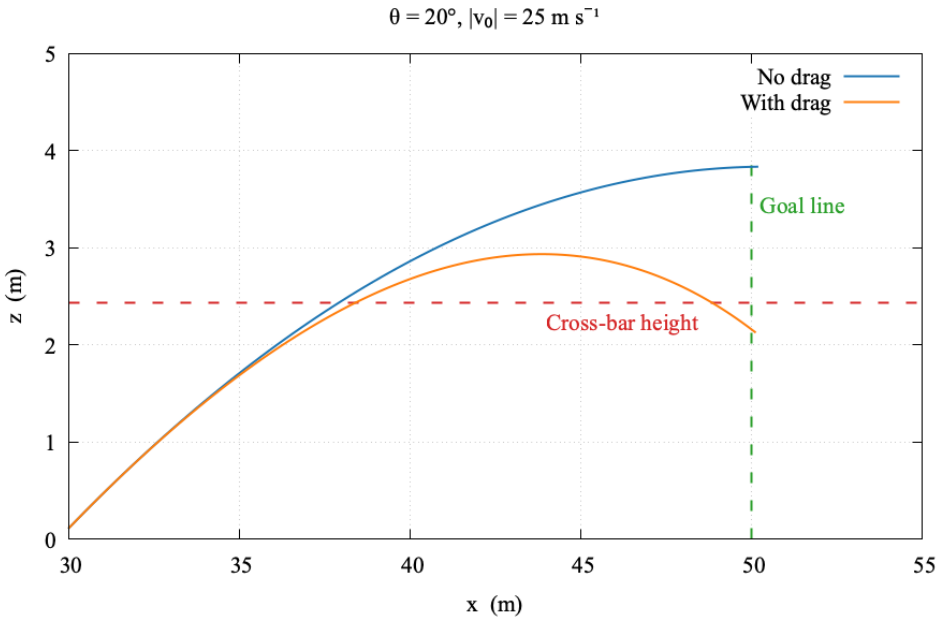

Figure 4: Comparison of trajectories with and without quadratic drag.

Effect of Magnus (spin-induced curvature). With both drag and Magnus enabled, the trajectory curves laterally in the (x, y) plane. In the tested configuration (Figure 5), the lateral position at the target plane is $y = 3.476309$, while the relevant boundary is at $y = 3.66$, leaving a clearance of $3.66 - 3.476309 = 0.183691$ m.

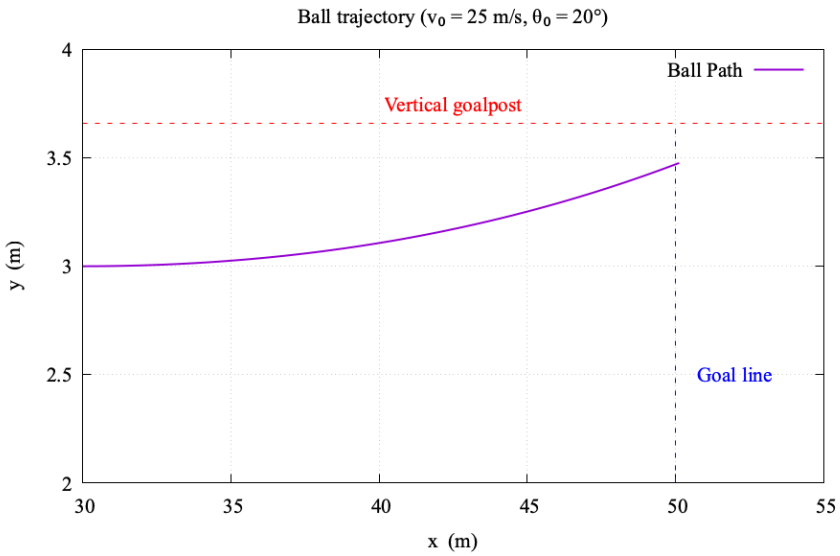

Figure 5: Trajectory in the (x, y) plane with drag and Magnus enabled (example configuration).

5 Discussion

Overall, the numerical results are consistent with physical intuition. In the ballistic (gravity-only) model, increasing the initial speed increases both range and maximum height, which is reflected in the family of trajectories. Introducing quadratic drag reduces range and peak height and changes the time to reach the target plane, as expected for a dissipative force opposing motion. When the Magnus term is enabled, the trajectory exhibits lateral curvature in the horizontal plane, with the direction and magnitude controlled by the spin parameter.

From a numerical perspective, RK4 provides a good balance of accuracy and simplicity for this smooth ODE system, but the results still depend on the choice of time step Δt and on parameter values (C_d, ρ, S, ω_z, etc.). A systematic sensitivity analysis (varying Δt and the physical constants) would quantify numerical error versus modelling uncertainty. In addition, event detection is handled by discrete stepping (checking when a plane or boundary is crossed), so the reported crossing time and position are limited by time-step resolution unless interpolation between steps is used.

Finally, several modelling simplifications remain: the projectile is treated as a point mass, wind and spin decay are neglected, and coefficients are assumed constant. Extending the model to include these effects, together with higher-fidelity calibration against data, would improve realism if required. .

6 Conclusion

This report presented C++ implementations of numerical methods for both data-driven kinematics and dynamical simulation. From discrete tracking data, velocity, acceleration, and speed were estimated using second-order centred finite differences, and smooth motion reconstruction was achieved via spline interpolation and resampling. Power was interpolated efficiently using divided differences, and total energy expenditure was estimated using composite quadrature rules, with trapezoidal and Simpson-type schemes giving closely agreeing results.

For trajectory modelling, a 3D projectile ODE was integrated using RK4 under gravity-only,

drag, and drag+Magnus force models. Visualisations demonstrated the qualitative influence of each force term on range, height, timing, and lateral curvature. Together, these components form a coherent numerical workflow linking discrete observations to derived kinematics and forward simulation.

A Additional implementation details

A.1 Data format and preprocessing

Input format. The tracking dataset is read from `tracking_data.dat`. Each row contains a single time sample with space-separated columns

$$t \;\; x \;\; y,$$

where t is time (s) and (x, y) are recorded planar coordinates.

Coordinate transform. The raw coordinates are mapped to physical coordinates on a rectangular domain of size $L = 100\,\text{m}$ by $W = 64\,\text{m}$ and recentred so that $(0, 0)$ corresponds to the pitch centre. (Implementation: `coord_xfm` in `q2.cpp`.)

A.2 Array lengths and indexing

The original position arrays have length $N = 1001$ with $\Delta t = 0.2\,\text{s}$. Centred finite differences are computed on indices $n = 1, \ldots, N - 2$, so velocity/speed/acceleration arrays have length $N - 2 = 999$ and correspond to times $t_1, \ldots, t_{N-2}$.

A.3 Output files

The following outputs are written for plotting and diagnostics:

- `speed_data.dat`: time and estimated speed.
- `simple_vXX.dat` (or similar): simulated trajectory samples for a given initial speed.

All outputs are plain text files suitable for plotting (e.g. with Gnuplot).

A.4 RK4 step size and stopping criteria

The RK4 integrator is run with fixed step size Δt (as set in `q4.cpp`). Integration terminates when the projectile hits the ground ($z \leq 0$) or crosses the target plane ($x \geq x_\star$). Outcome classification is computed from the state at the crossing time (height and lateral position).

A.5 Physical constants

Constants used in the simulation are defined in `q234.hpp` (gravity, mass, radius, air density, drag coefficient, Magnus coefficient, and geometry parameters). Table 1 lists their values.