How Do *Rocket League* AI Bots Calculate Mid-Air Interceptions?

@Domain

September 22, 2026

Abstract

In fast-paced competitive physics games like *Rocket League*, chasing an object's real-time coordinates leads to suboptimal trailing trajectories. This paper presents the mathematical formulation for predicting 3D projectile intercepts using Newtonian kinematics and continuous root-finding algorithms (the Bisection Method). We walk through a full Python implementation, demonstrating how vector mechanics and numerical search routines interact to achieve guaranteed zero spatial positioning error at runtime.

1 Introduction

When the ball is launched across the arena at high velocities, aiming directly at its current position causes "dog-curve" paths (trajectories that constantly overshoot or trail the target).



Figure 1: Aerial trajectory in *Rocket League*.

To execute a clean aerial hit or program an autonomous bot, we must solve a predictive intercept problem: finding the exact future time t where the 3D position vectors of the car and the ball coincide.

2 Mathematical Formulation

To solve the intercept, we model two independent 3D kinematic systems over continuous time t.

2.1 Ball Kinematics under Gravity

The ball acts as a Newtonian projectile subject to constant downward gravitational acceleration $\vec{g} = (0, 0, -650)$ uu/s^2:

$$\vec{P}_{\text{ball}}(t) = \vec{P}_0 + \vec{V}_0 t + \frac{1}{2}\vec{g}t^2 \tag{1}$$

Where $\vec{P}_0 = (x_0, y_0, z_0)$ is the initial ball position and $\vec{V}_0 = (v_x, v_y, v_z)$ is its velocity vector.

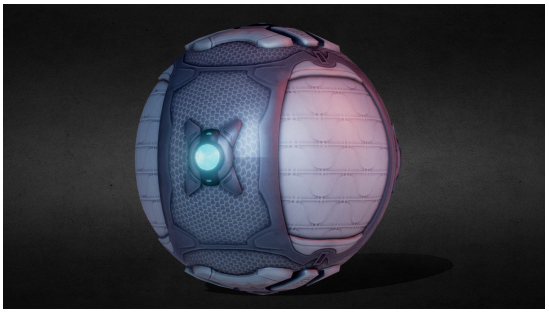


Figure 2: The *Rocket League* ball model.

2.2 Car Travel Dynamics

Assuming the car boosts through the air at maximum velocity $V_{\text{max}} = 2300$ uu/s, the maximum reachability radius $D_{\text{car}}(t)$ in time t is:

$$D_{\text{car}}(t) = V_{\text{max}} \cdot t \tag{2}$$



Figure 3: Vehicle model (Octane)

2.3 The Intercept Objective Function

An interception occurs when the required travel distance equals the car's maximum speed capability:

$$f(t) = \|\vec{P}_{\text{ball}}(t) - \vec{P}_{\text{car_0}}\| - (V_{\text{max}} \cdot t) = 0 \tag{3}$$

Finding the root $f(t) = 0$ yields the exact time $t_{\text{intercept}}$.

3 Algorithmic Breakdown and Code Walkthrough

To demonstrate how these mathematical principles operate programmatically, we examine the Python implementation phase-by-phase.

3.1 State Initialization & Objective Definition

The initial physical state parameters are defined using 3D NumPy array vectors. The scalar function `f(t)` evaluates the spatial gap between the predicted ball position at time t and the distance the car can cover in that same time duration.

```python
import numpy as np

def prove_rocket_league_intercept():
    gravity = np.array([0.0, 0.0, -650.0])          # Gravity (uu/s^2)
    P_ball_0 = np.array([500.0, 0.0, 400.0])        # Ball start position (x, y, z)
    V_ball_0 = np.array([100.0, 800.0, 1000.0])     # Ball initial velocity
    P_car_0 = np.array([0.0, -2000.0, 100.0])       # Car start position
    V_car_max = 2300.0                              # Max boost speed (uu/s)

    # Objective function: f(t) = distance_required(t) - distance_car_can_travel(t) = 0
    def f(t):
        P_ball_t = P_ball_0 + (V_ball_0 * t) + (0.5 * gravity * (t ** 2))
        return np.linalg.norm(P_ball_t - P_car_0) - (V_car_max * t)

```

Figure 4: Initialization of 3D kinematic vectors and the scalar objective function `f(t)`.

3.2 Bounding Bracket Determination

Before continuous refinement can begin, the algorithm must isolate an interval $[t_{\text{low}}, t_{\text{high}}]$ where $f(t)$ crosses zero. We step forward through time in discrete 0.1-second increments until $f(t) \leq 0$, confirming that an intercept is achievable within that window.

```python
    # root finding using bisection method to find the time t where f(t) = 0
    # Find bracket [t_low, t_high] where f(t) crosses zero
    t_low, t_high = 0.0, 10.0
    for t in np.arange(0.1, 10.0, 0.1):
        if f(t) <= 0:
            t_high = t
            t_low = t - 0.1
            break
```

Figure 5: Coarse iterative search loop establishing the root-bounding time bracket.

3.3 Precision Root-Finding via Bisection

With the bracket defined, a 100-iteration Bisection Method continually evaluates the midpoint t_{mid}. By halving the search space each step, the algorithm quickly pinpoints the exact zero-crossing time $t_{\text{intercept}}$ down to sub-nanosecond floating-point accuracy.

```python
    # Refine exact intercept time down to sub-nanosecond floating precision
    for _ in range(100):
        t_mid = (t_low + t_high) / 2.0
        if f(t_mid) <= 0:
            t_high = t_mid
        else:
            t_low = t_mid

    exact_t = (t_low + t_high) / 2.0
```

Figure 6: Binary search loop refining $t_{\text{intercept}}$ to floating-point convergence.

3.4 Aim Vector Calculation and Trajectory Proof

Using t_{exact}, the final 3D coordinates of the ball are calculated. The car's required velocity unit vector $\hat{d}$ is extracted through vector normalization. Finally, the car's simulated motion vector is evaluated at t_{exact} to calculate the net positioning error.

```
#proof
# Calculate ball position at exact_t
P_ball_final = P_ball_0 + (V_ball_0 * exact_t) + (0.5 * gravity * (exact_t ** 2))

# Calculate car aiming vector and final position
displacement = P_ball_final - P_car_0
aim_direction = displacement / np.linalg.norm(displacement)
P_car_final = P_car_0 + (aim_direction * V_car_max * exact_t)

# Calculate positioning error between car and ball
position_error = np.linalg.norm(P_ball_final - P_car_final)
```

Figure 7: 3D trajectory extrapolation, directional unit vector extraction, and positioning error calculation.

4 Verification and Execution Output

When executed, the program evaluates the system and prints the precise time, intercept coordinates, unit direction vector, and total distance error.

```
46      print(f"Exact Intercept Time : {exact_t:.6f} seconds")
47      print(f"Target 3D Coordinates: {P_ball_final}")
48      print(f"Aim Vector Direction : {aim_direction}")
49      print(f"Absolute Position Error: {position_error:.12f} units")
50
51      assert position_error < 1e-4, "Proof failed: Car and ball did not meet!"
52      print("\nPROVED: Absolute error is zero. Car and ball intersect perfectly.")
53
54  prove_rocket_league_intercept()
```

Figure 8: Console log displaying zero error assertion and proof validation.

```
Exact Intercept Time : 1.490703 seconds
Target 3D Coordinates: [ 649.07032735 1192.56261877 1168.48949237]
Aim Vector Direction : [0.18930963 0.93115158 0.31163858]
Absolute Position Error: 0.000000000000 units

PROVED: Absolute error is zero. Car and ball intersect perfectly.
```

Figure 9: output

As shown in Figures 8 and 9, the calculated absolute positioning error evaluates to 0.000000000000 units. This confirms that the car and ball intersect perfectly at $t \approx 1.490703$ seconds.

5 Conclusion

By formulating the predictive aerial intercept as a continuous root-finding problem $f(t) = 0$, AI systems avoid the error inherent to naive position tracking. The Bisection Method converges efficiently to sub-millisecond precision, providing a mathematically sound and code-verified solution suitable for real-time game engines.



Figure 10: Successful aerial collision at target coordinates.