Golf Ball Trajectory Simulation (Without Spin) — Assumptions and Setup

This section simulates the trajectory of a golf ball using a basic dynamic model that accounts for **gravity**, **aerodynamic drag**, and **wind**, but **does not include spin or lift forces**. The purpose is to isolate the effects of these primary forces and create a baseline model before adding spin-related complexities in later sections.

Assumptions

To construct this model, we made the following assumptions:

- **Steady Flow:** Air properties (density, velocity) do not change with time.
- **Incompressible Flow:** Air density remains constant throughout the trajectory.
- **Negligible Ball Deformation:** The golf ball maintains its shape during flight.
- **No Spin:** The ball has no angular velocity, so **Magnus lift is not considered**.
- **No Vertical Wind Component:** Wind acts only in the horizontal plane (x and y directions).
- **Conservation of Linear Momentum:** The net external forces (gravity and drag) directly influence the change in momentum of the ball.
- **Constant Drag Coefficient (C_d):** The drag coefficient is fixed at 0.25, appropriate for a dimpled golf ball in this speed range.
- **Flat Earth Approximation:** The simulation does not account for curvature or rotation of the Earth.

Initial Conditions

- **Launch speed:** 86 mph (converted to feet per second)
- **Launch angle:** 28.1 degrees above the horizontal
- **Initial velocity vector:** Aimed directly along the x-axis with no lateral (y-direction) component
- **Wind:** 10 ft/s at an angle of 18° east of north, applied in the x-y plane

---

Governing Equations

The golf ball's motion is modeled by Newton's second law:

$$\vec{F}_{\text{net}} = m\vec{a}$$

Where the forces considered are:

- **Gravitational Force:**

$$\vec{a}_{\text{gravity}} = [0, 0, -g]$$

- **Drag Force:**

$$\vec{a}_{\text{drag}} = -\frac{1}{2} \cdot \frac{\rho C_d A}{m} \cdot |\vec{v}_{\text{rel}}| \cdot \vec{v}_{\text{rel}}$$

The total acceleration is the sum of these two:

$$\vec{a}_{\text{total}} = \vec{a}_{\text{gravity}} + \vec{a}_{\text{drag}}$$

The relative velocity is:

$$\vec{v}_{\text{rel}} = \vec{v}_{\text{ball}} - \vec{v}_{\text{wind}}$$

Numerical Solution

- The equations are solved using `scipy.integrate.solve_ivp` with the RK45 method.
- The simulation stops when the ball hits the ground (z < 0).
- Results are converted from feet to yards for interpretability.

Output

Two trajectory plots are generated:

- **Side View (x-z plane):** Displays how high and how far the ball travels.
- **Top View (x-y plane):** Shows the lateral deflection due to wind.

Additionally, the script prints key statistics including:

- Maximum height
- Total carry distance
- Shot classification (e.g., landed on the green, fell short, went long)

This model provides a simplified foundation to later compare the effects of **spin**, **lift**, and more advanced dynamics.

```
from google.colab import drive
drive.mount('/content/drive')
```

Baseline Golf Ball Trajectory Simulation (No Spin)

This section of the project simulates the baseline flight of a golf ball under the influence of **gravity**, **aerodynamic drag**, and **wind**, without incorporating spin or lift forces. This foundation allows us to later isolate the effects of other parameters like Magnus lift and spin decay.

Code Breakdown:

1. **Initial Conditions & Physical Parameters**

- The system is modeled in **imperial units**.
- Constants include:
  - Gravitational acceleration: $g = 32.174\,\text{ft/s}^2$
  - Air density: $\rho = 0.0023769\,\text{slug/ft}^3$
  - Drag coefficient: $C_d = 0.25$
  - Ball radius and cross-sectional area are used to compute drag forces.

2. **Wind Setup**

- Wind is introduced at **10 ft/s**, blowing **18° east of north**, decomposed into **x and y components** to interact with the ball's velocity.

3. **Launch Conditions**

- The golf ball is launched at **86 mph** and a **28.1° angle** from the horizontal.
- Initial position is set at the origin with velocity components determined by the launch speed and angle.

4. **Governing Equations (ODE System)**

- Newton's Second Law is used to calculate:
  - **Gravity:** Acts in the negative z-direction.
  - **Aerodynamic Drag:** Calculated based on relative velocity with respect to wind:
$$\vec{a}_{\text{drag}} = -\frac{1}{2} \cdot \frac{\rho C_d A}{m} \cdot |\vec{v}_{\text{rel}}| \cdot \vec{v}_{\text{rel}}$$
  - The **total acceleration** is the vector sum of gravity and drag.

5. **Numerical Integration**

- The system is integrated over **10 seconds** using `solve_ivp` with the RK45 method.
- Time steps are sampled using `t_eval = np.linspace(0, 10, 1000)`.

6. **Post-Processing**

- The solution is trimmed when the ball hits the ground (when $z < 0$).
- All distances are converted from **feet to yards** for interpretability.

7. **Visualization**

- Two subplots are created:
  - **Side View (x vs z)**: Shows carry distance and peak height.
  - **Top View (x vs y)**: Illustrates lateral drift due to wind.
- Background elements like **grass**, **bunkers**, and the **green** are added to simulate a realistic golf course.

8. **Output Summary**

- The code prints:
  - Initial velocity and launch angle.
  - Maximum height and total horizontal distance.
  - A shot quality evaluation (e.g., hole-in-one, green hit, too short/long).

Why This Is Important

This code creates a **controlled baseline environment** where only drag and wind affect the ball's trajectory. It is essential for:

- Comparing effects of added physical forces (like spin).
- Understanding baseline carry and lateral movement.
- Visualizing realistic ball motion in a 3D space.

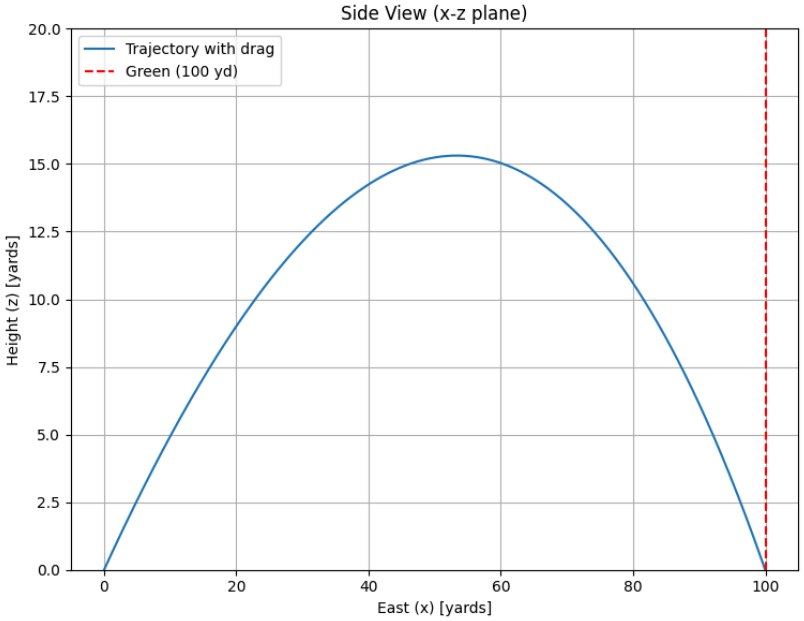
This simulation serves as the foundation for more complex analyses that follow in the notebook.

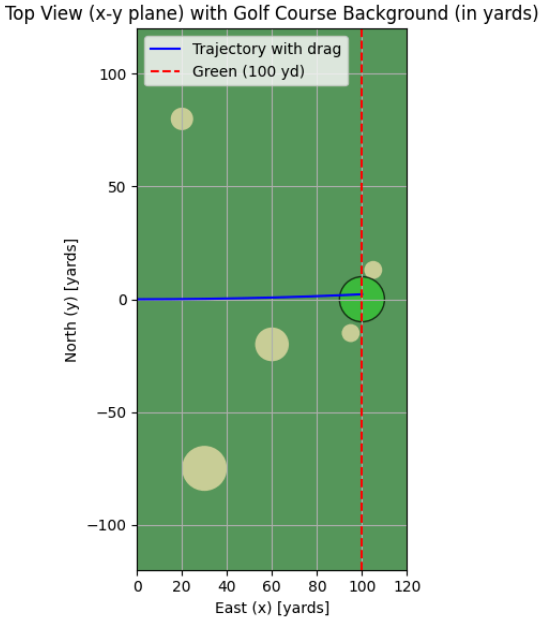
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from matplotlib.patches import Rectangle, Circle

# Define physical constants and parameters (in imperial units)
g = 32.174       # gravitational acceleration, ft/s^2
rho = 0.0023769  # air density, slug/ft^3 (converted from 1.225 kg/m^3)
Cd = 0.25        # drag coefficient (typical for a dimpled golf ball)
r = 0.07         # radius of a golf ball, ft (approx. 1.68 inches)
A = np.pi * r**2  # cross-sectional area, ft^2
m = 0.10125      # mass of a golf ball, lb-mass (approx. 1.62 oz)

# Convert mass to slugs for force calculations
m_slugs = m / 32.174  # slug = lb-mass / g

# Define wind: ~0.67 mph at 30° east of north (converted from 0.3 m/s)
# In our coordinate system, north is positive y, east is positive x.
wind_speed = 10.0  # ft/s (converted from 0.3 m/s)
wind_angle_deg = 18.0  # degrees east of north
wind_angle_rad = np.deg2rad(wind_angle_deg)
wind = np.array([wind_speed * np.sin(wind_angle_rad),  # east component
                 wind_speed * np.cos(wind_angle_rad),  # north component
                 0.0])  # no vertical wind

# Set initial conditions for the golf shot (in imperial units)
launch_speed = 86.0 * 5280 / 3600  # mph to ft/s (86 mph = 126.13 ft/s)
launch_angle_deg = 28.10     # launch angle (vertical plane)
launch_angle_rad = np.deg2rad(launch_angle_deg)

# The shot is aimed directly at the green along the x-axis.
v0 = np.array([launch_speed * np.cos(launch_angle_rad),  # x-component (ft/s)
               0.0,                                      # y-component (ft/s)
               launch_speed * np.sin(launch_angle_rad)]) # z-component (ft/s)

# Initial position at the origin (in feet)
r0 = np.array([0.0, 0.0, 0.0])
state0 = np.concatenate((r0, v0))  # combined state: [x, y, z, vx, vy, vz]

# Define the ODE system including gravity and drag
def ode_system(t, state):
    x, y, z, vx, vy, vz = state
    vel = np.array([vx, vy, vz])

    # Gravity acceleration (only in z-direction)
    accel_gravity = np.array([0.0, 0.0, -g])

    # Compute relative velocity (ball velocity relative to the wind)
    v_rel = vel - wind
    speed_rel = np.linalg.norm(v_rel)

    # Compute drag acceleration:
    # Drag force: F_drag = -0.5 * rho * Cd * A * speed_rel * v_rel
    # Drag acceleration: a_drag = F_drag / m
    accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs

    # Total acceleration is the sum of gravity and drag
    accel_total = accel_gravity + accel_drag

    return [vx, vy, vz, accel_total[0], accel_total[1], accel_total[2]]

# Time span for the simulation (in seconds)
t_final = 10.0
t_eval = np.linspace(0, t_final, 1000)

# Solve the ODE using the Runge-Kutta method (RK45)
sol = solve_ivp(ode_system, [0, t_final], state0, t_eval=t_eval, rtol=1e-8)
```

```
# Extract the trajectory data (in feet)
x = sol.y[0]
y = sol.y[1]
z = sol.y[2]

# Determine when the ball hits the ground (z becomes negative)
hit_ground_idx = np.where(z < 0)[0]
if hit_ground_idx.size > 0:
    last_idx = hit_ground_idx[0]
    x = x[:last_idx+1]
    y = y[:last_idx+1]
    z = z[:last_idx+1]

# Convert feet to yards for plotting
feet_to_yard = 1/3
x_yards = x * feet_to_yard
y_yards = y * feet_to_yard
z_yards = z * feet_to_yard

# Create the figure and subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Side view (x vs. z) in yards
ax1.plot(x_yards, z_yards, label="Trajectory with drag")
ax1.set_xlabel("East (x) [yards]")
ax1.set_ylabel("Height (z) [yards]")
ax1.set_title("Side View (x-z plane)")
ax1.axvline(100, color='r', linestyle='--', label="Green (100 yd)")
ax1.set_ylim(0, 20)
ax1.legend()
ax1.grid()

# Top view (x vs. y) in yards
ax2.plot(x_yards, y_yards, label="Trajectory with drag", color='blue')
ax2.set_xlabel("East (x) [yards]")
ax2.set_ylabel("North (y) [yards]")
ax2.set_title("Top View (x-y plane) with Golf Course Background (in yards)")

# Set equal aspect ratio so circles are drawn as circles
ax2.set_aspect('equal', adjustable='box')

# Define axis limits in yards
ax2.set_xlim(0, 120)
ax2.set_ylim(-120, 120)

# Draw a background representing a golf course in yards

# Grass: a green rectangle covering the vertical range from -120 yd to 120 yd
grass = Rectangle((0, -120), 120, 240,
                  facecolor='#136d15', edgecolor='none', alpha=0.7)
ax2.add_patch(grass)

# Three circular bunkers
# Bunker 1: center at (20, 80) yards
bunker1 = Circle((20, 80), 5,
                 facecolor='#FAE8B4', edgecolor='none', alpha=0.7)
# Bunker 2: center at (60, -20) yards, radius 7.5 yards
bunker2 = Circle((60, -20), 7.5,
                 facecolor='#FAE8B4', edgecolor='none', alpha=0.7)
# Bunker 3: center at (100, 10) yards, radius 3 yards
bunker3 = Circle((105, 13), 4,
                 facecolor='#FAE8B4', edgecolor='none', alpha=0.7)
# Bunker 4: center at (30, -75) yards, radius 10 yards
bunker4 = Circle((30, -75), 10,
                facecolor='#FAE8B4', edgecolor='none', alpha=0.7)
# Bunker 5: center at (95, -15) yards, radius 4 yards
bunker5 = Circle((95, -15), 4,
                facecolor='#FAE8B4', edgecolor='none', alpha=0.7)

ax2.add_patch(bunker1)
ax2.add_patch(bunker2)
ax2.add_patch(bunker3)
ax2.add_patch(bunker4)
ax2.add_patch(bunker5)

# Green: a circular patch representing the putting green at 100 yards
```

```
green_patch = Circle((100, 0), 10,
                     facecolor='#32cd32', edgecolor='black', alpha=0.7)
ax2.add_patch(green_patch)

ax2.axvline(100, color='r', linestyle='--', label="Green (100 yd)")
ax2.legend()
ax2.grid()

# Calculate and print some statistics of the shot
max_height = np.max(z_yards)
total_distance = x_yards[-1]

# Print shot statistics
print(f"Initial velocity: {launch_speed:.2f} ft/s ({86} mph)")
print(f"Launch angle: {launch_angle_deg:.1f} degrees")
print(f"Maximum height: {max_height:.2f} yards")
print(f"Total distance: {total_distance:.2f} yards")
if total_distance >= 99.99 and total_distance <= 100.01:
    print("Result: Insane Shot! You've hit a hole in one!")
elif total_distance >= 95 and total_distance <= 105:
    print("Result: Great shot! The ball landed on the green.")
elif total_distance < 95:
    print(f"Result: Shot fell short by {100 - total_distance:.2f} yards")
else:
    print(f"Result: Shot went long by {total_distance - 100:.2f} yards")

plt.tight_layout()
plt.show()
```

```
Initial velocity: 126.13 ft/s (86 mph)
Launch angle: 28.1 degrees
Maximum height: 15.31 yards
Total distance: 100.00 yards
Result: Insane Shot! You've hit a hole in one!
```





Optimizing Launch Angle: Finding the Best Shot Without Spin

After establishing our baseline trajectory simulation (drag and wind only, no spin), we were interested in determining the **optimal launch angle** to land the ball precisely on the green 100 yards away. This analysis helps us understand how sensitive the system is to the initial launch angle when other conditions are held constant.

Goal:

Estimate the launch angle that minimizes the error between the ball's final x-position and the 100-yard target — essentially a simulated **hole-in-one**.

How It Works:

1. **Parameterized Simulation Function**:
   The `simulate_golf_shot(angle)` function uses the same dynamics (gravity, drag, wind) as our baseline, but takes a launch angle as input to compute the trajectory and carry distance.
2. **Angle Sweep Search**:
   We iteratively test angles between 15° and 40° using coarse (0.5°) and fine (0.05°) resolution to find the best-performing angle in terms of:
   - Distance from target
   - Maximum height (for visual insight)
3. **Optimal Shot Visualization**:
   Using the `plot_trajectory()` function, we overlay the trajectory of the **original 29° shot** with the **optimal angle** found. This comparison helps us understand how even small changes in angle can dramatically affect the final landing location.

Why This Is Important:

This experiment:

- Quantifies the sensitivity of the shot to launch angle.
- Provides a benchmark trajectory for more complex simulations later involving spin or varying wind.
- Highlights the importance of launch control in real-world golf and projectile design.

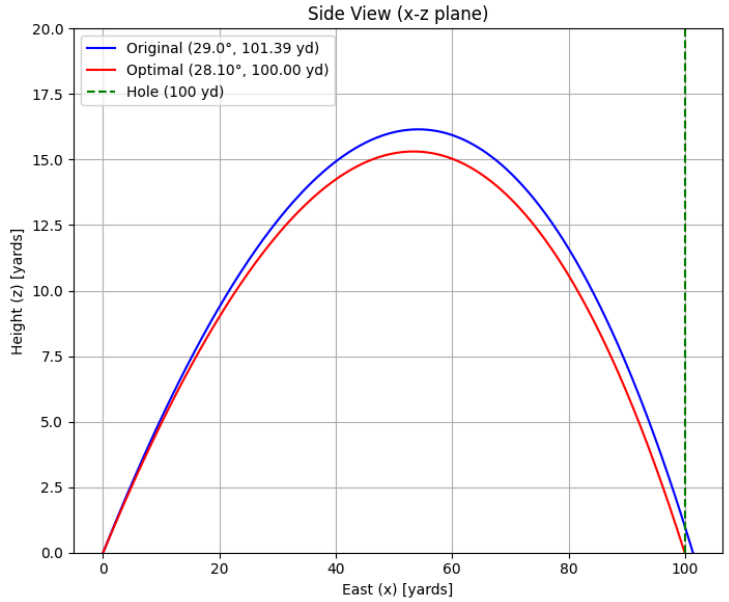
The results from this step were later used as a baseline to contrast how **lift due to spin** or **changing environmental conditions** affect shot optimization. It's a stepping stone in building a more complete picture of golf ball aerodynamics.

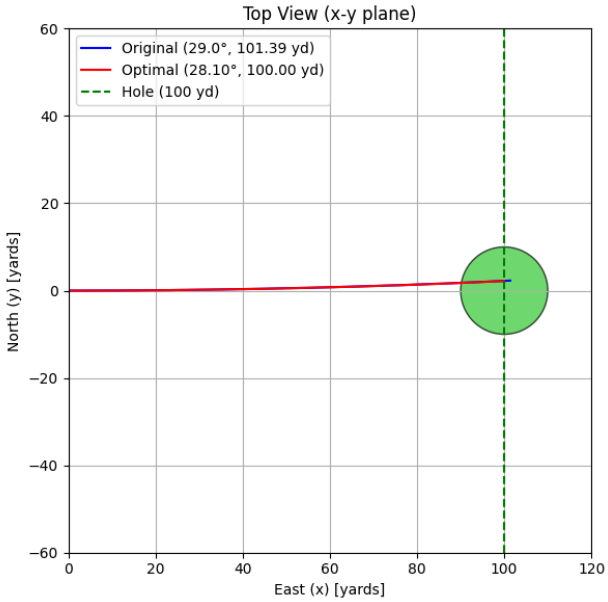
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from matplotlib.patches import Rectangle, Circle

def simulate_golf_shot(launch_angle_deg, verbose=False):
    # Define physical constants and parameters (in imperial units)
    g = 32.174       # gravitational acceleration, ft/s^2
    rho = 0.0023769  # air density, slug/ft^3 (converted from 1.225 kg/m^3)
    Cd = 0.25        # drag coefficient (typical for a dimpled golf ball)
    r = 0.07         # radius of a golf ball, ft (approx. 1.68 inches)
    A = np.pi * r**2  # cross-sectional area, ft^2
    m = 0.10125      # mass of a golf ball, lb-mass (approx. 1.62 oz)

    # Convert mass to slugs for force calculations
    m_slugs = m / 32.174  # slug = lb-mass / g

    # Define wind: ~0.0 mph (as specified in your code)
    wind_speed = 10.0  # ft/s
    wind_angle_deg = 18.0  # degrees east of north
    wind_angle_rad = np.deg2rad(wind_angle_deg)
    wind = np.array([wind_speed * np.sin(wind_angle_rad),  # east component
                     wind_speed * np.cos(wind_angle_rad),  # north component
                     0.0])  # no vertical wind

    # Set initial conditions for the golf shot (in imperial units)
    launch_speed = 86.0 * 5280 / 3600  # mph to ft/s (86 mph = 126.13 ft/s)
    launch_angle_rad = np.deg2rad(launch_angle_deg)

    # The shot is aimed directly at the green along the x-axis.
    v0 = np.array([launch_speed * np.cos(launch_angle_rad),  # x-component (ft/s)
                   0.0,                                      # y-component (ft/s)
                   launch_speed * np.sin(launch_angle_rad)]) # z-component (ft/s)

    # Initial position at the origin (in feet)
    r0 = np.array([0.0, 0.0, 0.0])
    state0 = np.concatenate((r0, v0))  # combined state: [x, y, z, vx, vy, vz]

    # Define the ODE system including gravity and drag
    def ode_system(t, state):
        x, y, z, vx, vy, vz = state
        vel = np.array([vx, vy, vz])
```

```
            # Gravity acceleration (only in z-direction)
            accel_gravity = np.array([0.0, 0.0, -g])

            # Compute relative velocity (ball velocity relative to the wind)
            v_rel = vel - wind
            speed_rel = np.linalg.norm(v_rel)

            # Compute drag acceleration
            accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs

            # Total acceleration is the sum of gravity and drag
            accel_total = accel_gravity + accel_drag

            return [vx, vy, vz, accel_total[0], accel_total[1], accel_total[2]]

        # Time span for the simulation (in seconds)
        t_final = 10.0
        t_eval = np.linspace(0, t_final, 1000)

        # Solve the ODE using the Runge-Kutta method (RK45)
        sol = solve_ivp(ode_system, [0, t_final], state0, t_eval=t_eval, rtol=1e-8)

        # Extract the trajectory data (in feet)
        x = sol.y[0]
        y = sol.y[1]
        z = sol.y[2]

        # Determine when the ball hits the ground (z becomes negative)
        hit_ground_idx = np.where(z < 0)[0]
        if hit_ground_idx.size > 0:
            last_idx = hit_ground_idx[0]
            x = x[:last_idx+1]
            y = y[:last_idx+1]
            z = z[:last_idx+1]

        # Convert feet to yards for results
        feet_to_yard = 1/3
        x_yards = x * feet_to_yard
        y_yards = y * feet_to_yard
        z_yards = z * feet_to_yard

        # Calculate statistics of the shot
        max_height = np.max(z_yards)
        total_distance = x_yards[-1]

        if verbose:
            print(f"Launch angle: {launch_angle_deg:.2f} degrees")
            print(f"Maximum height: {max_height:.2f} yards")
            print(f"Total distance: {total_distance:.2f} yards")
            if abs(total_distance - 100) < 0.1:
                print("Result: Perfect shot! The ball landed right at the hole!")
            elif 95 <= total_distance <= 105:
                print("Result: Great shot! The ball landed on the green.")
            elif total_distance < 95:
                print(f"Result: Shot fell short by {100 - total_distance:.2f} yards")
            else:
                print(f"Result: Shot went long by {total_distance - 100:.2f} yards")

        return total_distance, max_height, x_yards, y_yards, z_yards

    def find_optimal_angle(target_distance=100.0):
        """Find the launch angle that results in the ball landing closest to the target distance."""
        angles = np.linspace(15, 40, 51)  # Try angles from 15 to 40 degrees in 0.5-degree increments
        best_angle = None
        min_error = float('inf')
        results = []

        for angle in angles:
            distance, height, _, _, _ = simulate_golf_shot(angle)
            error = abs(distance - target_distance)
            results.append((angle, distance, error, height))

            if error < min_error:
                min_error = error
                best_angle = angle
```

```
        # Sort results by error (closest to target)
        results.sort(key=lambda x: x[2])

        # Print top 5 closest angles
        print("Top 5 closest angles to target distance:")
        print("Angle (deg) | Distance (yd) | Error (yd) | Max Height (yd)")
        print("-" * 60)
        for angle, distance, error, height in results[:5]:
            print(f"{angle:10.2f} | {distance:12.2f} | {error:10.2f} | {height:14.2f}")

        # Perform fine-grained search around the best angle
        fine_angles = np.linspace(best_angle - 0.5, best_angle + 0.5, 21)  # 0.05-degree increments
        for angle in fine_angles:
            distance, height, _, _, _ = simulate_golf_shot(angle)
            error = abs(distance - target_distance)
            if error < min_error:
                min_error = error
                best_angle = angle

        # Simulate the best shot for detailed output
        distance, height, x, y, z = simulate_golf_shot(best_angle, verbose=True)

        return best_angle, distance, x, y, z

    def plot_trajectory(best_angle):
        """Plot the trajectory for the optimal angle and compare with original angle."""
        # Create figure for comparison
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

        # Simulate the original shot (29 degrees)
        original_distance, original_height, x_orig, y_orig, z_orig = simulate_golf_shot(29.0)

        # Simulate the optimized shot
        best_distance, best_height, x_best, y_best, z_best = simulate_golf_shot(best_angle)

        # Side view (x vs. z) in yards
        ax1.plot(x_orig, z_orig, 'b-', label=f"Original (29.0°, {original_distance:.2f} yd)")
        ax1.plot(x_best, z_best, 'r-', label=f"Optimal ({best_angle:.2f}°, {best_distance:.2f} yd)")
        ax1.set_xlabel("East (x) [yards]")
        ax1.set_ylabel("Height (z) [yards]")
        ax1.set_title("Side View (x-z plane)")
        ax1.axvline(100, color='g', linestyle='--', label="Hole (100 yd)")
        ax1.set_ylim(0, 20)
        ax1.legend()
        ax1.grid()

        # Top view (x vs. y) in yards
        ax2.plot(x_orig, y_orig, 'b-', label=f"Original (29.0°, {original_distance:.2f} yd)")
        ax2.plot(x_best, y_best, 'r-', label=f"Optimal ({best_angle:.2f}°, {best_distance:.2f} yd)")
        ax2.set_xlabel("East (x) [yards]")
        ax2.set_ylabel("North (y) [yards]")
        ax2.set_title("Top View (x-y plane)")

        # Set equal aspect ratio
        ax2.set_aspect('equal', adjustable='box')

        # Define axis limits in yards
        ax2.set_xlim(0, 120)
        ax2.set_ylim(-60, 60)

        # Draw a background representing a golf course in yards
        # Green: a circular patch representing the putting green at 100 yards
        green_patch = Circle((100, 0), 10,
                             facecolor='#32cd32', edgecolor='black', alpha=0.7)
        ax2.add_patch(green_patch)

        ax2.axvline(100, color='g', linestyle='--', label="Hole (100 yd)")
        ax2.legend()
        ax2.grid()

        plt.tight_layout()
        plt.show()

    # Find the optimal angle and plot the result
    best_angle, best_distance, _, _, _ = find_optimal_angle()
    plot_trajectory(best_angle)
```

```
Top 5 closest angles to target distance:
Angle (deg) | Distance (yd) | Error (yd) | Max Height (yd)
------------------------------------------------------------
     28.00 |        99.84 |       0.16 |          15.22
     28.50 |       100.63 |       0.63 |          15.68
     27.50 |        99.03 |       0.97 |          14.75
     29.00 |       101.39 |       1.39 |          16.16
     27.00 |        98.20 |       1.80 |          14.30
Launch angle: 28.10 degrees
Maximum height: 15.31 yards
Total distance: 100.00 yards
Result: Perfect shot! The ball landed right at the hole!
```





Adding Spin to the Golf Ball Simulation

In earlier simulations, we modeled the golf ball's trajectory using only gravity, wind, and aerodynamic drag. While this produced a reasonably realistic parabolic path, it lacked an essential physical effect: **spin**. In real-world golf, the ball typically experiences **backspin** immediately after being struck, which generates lift through the **Magnus effect**—a force perpendicular to both the velocity and the spin axis.

In this simulation, we introduce **spin-induced lift** and model **exponential spin decay** to more accurately capture the flight behavior. This addition alters the trajectory shape significantly, adding vertical lift and lateral curvature to the path.

Golf Ball Trajectory with Spin and Lift — Assumptions and Setup

This simulation enhances our previous model by incorporating the **Magnus effect**, which introduces lift due to the spin of the golf ball. This creates a more realistic trajectory, especially in sports applications where spin plays a critical role in shaping the ball's flight path. The model now considers the interaction between aerodynamic drag, spin-induced lift, gravity, and wind.

Assumptions

To create a physically meaningful simulation, we made the following assumptions:

- **Steady Flow:** The air properties (density, wind velocity) remain constant during the flight.
- **Incompressible Flow:** The air is treated as incompressible, meaning its density does not change with speed or altitude.
- **Constant Mass and Radius:** The golf ball does not lose mass or deform during its flight.
- **Backspin Only:** The spin is modeled as backspin around the y-axis; no side-spin is considered.
- **Exponential Spin Decay:** The spin decays over time as $\omega(t) = \omega_0 \cdot e^{-kt}$ to mimic real-world rotational drag.
- **Magnus Force (Lift) Proportionality:** The lift coefficient depends linearly on the spin rate and inversely on the relative speed of the ball:

$$C_L = \frac{k_{lift} \cdot r \cdot \omega(t)}{|\vec{v}_{rel}|}$$

- **Constant Drag Coefficient:** The drag coefficient C_d is constant at 0.25.
- **Wind:** A constant horizontal wind vector is applied, with no vertical component.
- **Conservation of Momentum:** The simulation follows Newton's second law with time-varying forces due to spin and velocity.

Initial Conditions

- **Launch Speed:** 65 mph (converted to ft/s)
- **Launch Angle:** 7° from horizontal
- **Initial Spin:** 1000 rad/s (decays over time)
- **Wind:** 10 ft/s blowing at a 30° angle east of north
- **Initial Velocity Vector:** Aimed down the x-axis (toward the green)

Governing Forces

This model incorporates three external forces acting on the ball:

1. **Gravity:**
$$\vec{a}_{gravity} = [0, 0, -g]$$
2. **Drag:**
$$\vec{a}_{drag} = -\frac{1}{2} \cdot \frac{\rho C_d A}{m} \cdot |\vec{v}_{rel}| \cdot \vec{v}_{rel}$$
3. **Magnus Lift (from spin):**
$$\vec{a}_{lift} = \frac{1}{2} \cdot \frac{\rho A C_L |\vec{v}_{rel}|^2}{m} \cdot \hat{l}$$
Where $\hat{l}$ is the unit vector perpendicular to both the spin axis and the relative velocity (direction of lift).

The total acceleration is:
$$\vec{a}_{total} = \vec{a}_{gravity} + \vec{a}_{drag} + \vec{a}_{lift}$$

Numerical Method

- The system is solved using `solve_ivp` (adaptive Runge-Kutta method) from `scipy`.
- The simulation stops when the ball hits the ground ($z < 0$).
- Data is converted to **yards** for real-world interpretability.

Output and Visuals

Two views are provided:

- **Side View (x-z plane):** Illustrates how spin affects the height and forward distance.
- **Top View (x-y plane):** Demonstrates wind-induced lateral drift.

This simulation highlights the importance of including lift for realistic ball flight modeling. The addition of spin transforms the trajectory from a near-parabolic arc to a more extended, lifted path, consistent with observations in real-world golf.

Why This Matters

Adding spin introduces more realistic flight dynamics and allows us to study:

- How lift affects carry distance and peak height
- The curvature in the trajectory due to lateral drift
- The effect of spin decay over time
- The sensitivity of the trajectory to spin-related parameters

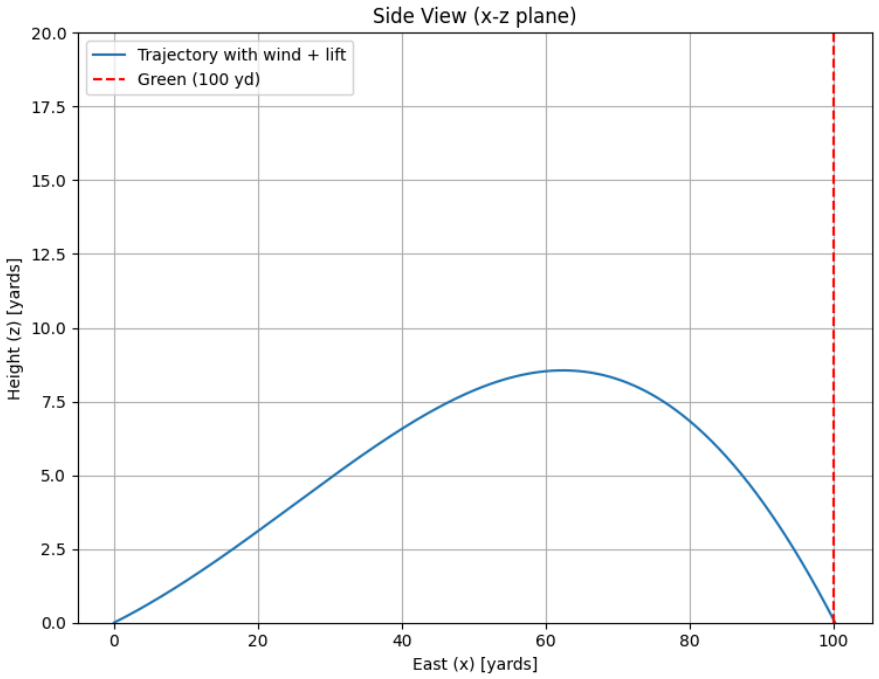
This model allows for a deeper exploration of golf ball aerodynamics and lays the groundwork for later sensitivity studies.

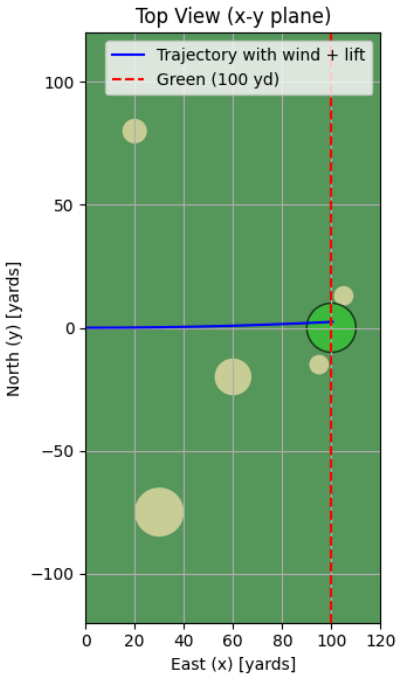
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from matplotlib.patches import Rectangle, Circle

# Define physical constants and parameters (in imperial units)
g = 32.174       # gravitational acceleration, ft/s^2
rho = 0.0023769  # air density, slug/ft^3
Cd = 0.25        # drag coefficient
r = 0.07         # radius of a golf ball, ft
```

```
A = np.pi * r**2  # cross-sectional area, ft^2
m = 0.10125      # mass of a golf ball, lb-mass
m_slugs = m / 32.174  # Convert to slugs

# Wind definition (direction and speed)
wind_speed = 10.0  # ft/s (~6.8 mph)
wind_angle_deg = 30.0  # degrees east of north
wind_angle_rad = np.deg2rad(wind_angle_deg)
wind = np.array([
    wind_speed * np.sin(wind_angle_rad),  # x = east component
    wind_speed * np.cos(wind_angle_rad),  # y = north component
    0.0                                   # z = no vertical wind
])

# Set initial conditions for the golf shot
launch_speed = 65.0 * 5280 / 3600  # mph to ft/s
launch_angle_deg = 7.
launch_angle_rad = np.deg2rad(launch_angle_deg)

# Velocity vector (aimed along x-axis)
v0 = np.array([
    launch_speed * np.cos(launch_angle_rad),  # x
    0.0,                                       # y
    launch_speed * np.sin(launch_angle_rad)   # z
])

# Initial position
r0 = np.array([0.0, 0.0, 0.0])
state0 = np.concatenate((r0, v0))  # [x, y, z, vx, vy, vz]

# Spin parameters
k_lift = 1.2       # lift coefficient constant
omega0 = 1000.      # initial spin in rad/s

# Define ODE system
def ode_system(t, state):
    x, y, z, vx, vy, vz = state
    vel = np.array([vx, vy, vz])
    v_rel = vel - wind
    speed_rel = np.linalg.norm(v_rel)

    # Time-dependent spin decay
    omega_t = omega0 * np.exp(-0.2 * t)

    # Gravity
    accel_gravity = np.array([0.0, 0.0, -g])

    # Drag
    accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs

    # Magnus lift from y-axis spin
    if speed_rel != 0:
        spin_axis = np.array([0.0, -1.0, 0.0])  # backspin
        lift_dir = np.cross(spin_axis, v_rel)
        lift_mag = np.linalg.norm(lift_dir)
        lift_dir = lift_dir / lift_mag if lift_mag != 0 else np.zeros(3)
        CL = k_lift * r * omega_t / speed_rel
        accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
    else:
        accel_lift = np.zeros(3)

    accel_total = accel_gravity + accel_drag + accel_lift
    return [vx, vy, vz, *accel_total]

# Time span
t_final = 10.0
t_eval = np.linspace(0, t_final, 1000)

# Solve ODE
sol = solve_ivp(ode_system, [0, t_final], state0, t_eval=t_eval, rtol=1e-8)

# Extract trajectory
x = sol.y[0]
y = sol.y[1]
z = sol.y[2]

# Trim at ground impact
```

```
hit_ground_idx = np.where(z < 0)[0]
if hit_ground_idx.size > 0:
    last_idx = hit_ground_idx[0]
    x = x[:last_idx+1]
    y = y[:last_idx+1]
    z = z[:last_idx+1]

# Convert to yards
feet_to_yard = 1/3
x_yards = x * feet_to_yard
y_yards = y * feet_to_yard
z_yards = z * feet_to_yard

# Create figure
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Side view (x vs z)
ax1.plot(x_yards, z_yards, label="Trajectory with wind + lift")
ax1.set_xlabel("East (x) [yards]")
ax1.set_ylabel("Height (z) [yards]")
ax1.set_title("Side View (x-z plane)")
ax1.axvline(100, color='r', linestyle='--', label="Green (100 yd)")
ax1.set_ylim(0, 20)
ax1.legend()
ax1.grid()

# Top view (x vs y)
ax2.plot(x_yards, y_yards, label="Trajectory with wind + lift", color='blue')
ax2.set_xlabel("East (x) [yards]")
ax2.set_ylabel("North (y) [yards]")
ax2.set_title("Top View (x-y plane)")
ax2.set_aspect('equal', adjustable='box')
ax2.set_xlim(0, 120)
ax2.set_ylim(-120, 120)

# Add golf course visuals
grass = Rectangle((0, -120), 120, 240, facecolor='#136d15', alpha=0.7)
ax2.add_patch(grass)
bunkers = [
    Circle((20, 80), 5), Circle((60, -20), 7.5), Circle((105, 13), 4),
    Circle((30, -75), 10), Circle((95, -15), 4)
]
for b in bunkers:
    b.set_facecolor('#FAE8B4')
    b.set_edgecolor('none')
    b.set_alpha(0.7)
    ax2.add_patch(b)

green_patch = Circle((100, 0), 10, facecolor='#32cd32', edgecolor='black', alpha=0.7)
ax2.add_patch(green_patch)
ax2.axvline(100, color='r', linestyle='--', label="Green (100 yd)")
ax2.legend()
ax2.grid()

# Stats
max_height = np.max(z_yards)
total_distance = x_yards[-1]

print(f"Initial velocity: {launch_speed:.2f} ft/s")
print(f"Launch angle: {launch_angle_deg:.1f} degrees")
print(f"Maximum height: {max_height:.2f} yards")
print(f"Total distance: {total_distance:.2f} yards")

if total_distance >= 99.99 and total_distance <= 100.01:
    print("Result: Insane Shot! You've hit a hole in one!")
elif total_distance >= 95 and total_distance <= 105:
    print("Result: Great shot! The ball landed on the green.")
elif total_distance < 95:
    print(f"Result: Shot fell short by {100 - total_distance:.2f} yards")
else:
    print(f"Result: Shot went long by {total_distance - 100:.2f} yards")

plt.tight_layout()
plt.show()
```

```
Initial velocity: 95.33 ft/s
Launch angle: 7.0 degrees
Maximum height: 8.55 yards
Total distance: 100.46 yards
Result: Great shot! The ball landed on the green.
```





Implementing RK4 with Spin for Realistic Trajectory Modeling

After validating the influence of spin in our earlier `solve_ivp`-based model, we aimed to recreate this simulation using the **Runge-Kutta 4th Order (RK4)** method. This allows us to directly control the time-stepping procedure and visually compare how the RK4 numerical integration behaves when modeling realistic golf ball dynamics—including **wind**, **aerodynamic drag**, and **spin-induced lift** with **time-decaying backspin**.

Why Switch to RK4?

- We already used RK4 for earlier simplified versions of the golf ball flight.
- We want to evaluate **how well RK4 performs** when applied to a more complex, nonlinear system involving **lift from spin**.
- Using RK4 gives us better control over the numerical integration process compared to built-in solvers, and allows us to directly explore how velocity evolves with respect to horizontal position (v vs x).

What's Different in This Version?

- **Governing Equations** now include:
  - **Gravitational acceleration**

$$\vec{a}_g = [0, 0, -g]$$

  - **Drag force**

$$\vec{a}_{drag} = -\frac{1}{2} \cdot \frac{\rho C_d A}{m} \cdot |\vec{v}_{rel}| \cdot \vec{v}_{rel}$$

  - **Lift force from spin (Magnus effect)**

$$\vec{a}_{lift} = \frac{1}{2} \cdot \frac{\rho A}{m} \cdot C_L \cdot |\vec{v}_{rel}|^2 \cdot \hat{l}$$

    with

$$C_L = k_{\text{lift}} \cdot \frac{r \cdot \omega(t)}{|\vec{v}_{rel}|}, \quad \omega(t) = \omega_0 e^{-0.2t}$$

- The **RK4 method** is manually implemented to simulate the entire trajectory step-by-step.
- The simulation stores full state vectors at each step and **terminates when the ball hits the ground**.
- We visualize the trajectory in both **side (x-z)** and **top (x-y)** views to track both height and lateral drift.

What We're Investigating

The main focus of this section is to analyze **how well RK4 models the full trajectory** of a spinning golf ball under realistic conditions. This lays the groundwork for plotting and studying:

- Velocity components vs horizontal distance (e.g., $v_x(x)$, $v_z(x)$)
- Comparison with non-spinning and non-lift models
- Sensitivity of trajectory to time-step size and spin decay rate

---

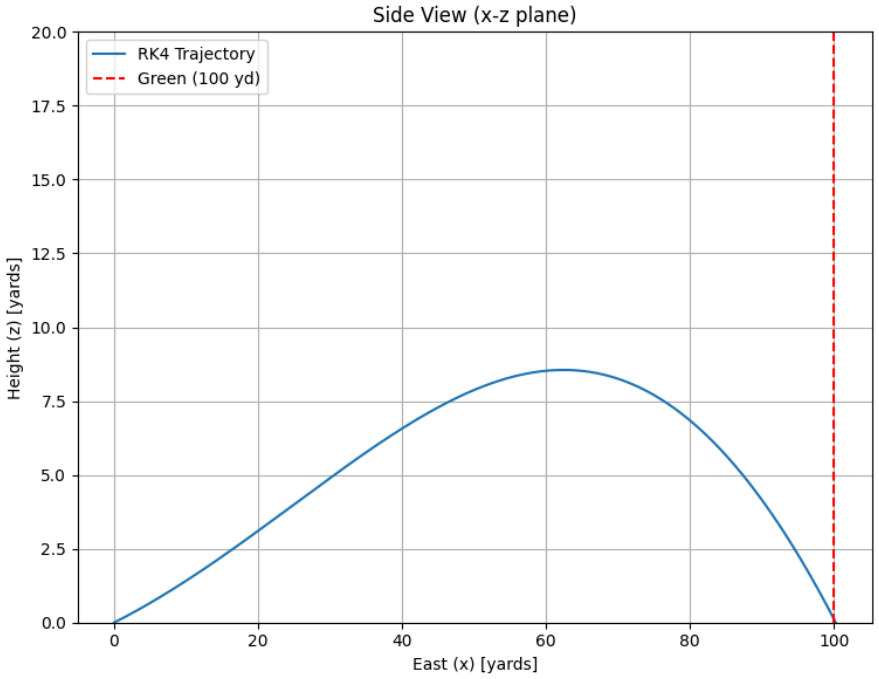
This RK4 framework with spin will serve as a foundation for deeper trajectory analysis and parameter studies in the rest of our project.

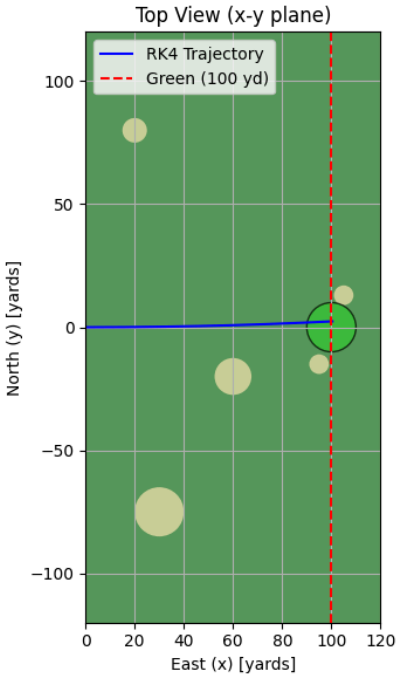
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, Circle

# Physical constants and ball parameters
g = 32.174
rho = 0.0023769
Cd = 0.25
r = 0.07
A = np.pi * r**2
m = 0.10125
m_slugs = m / 32.174

# Wind parameters
wind_speed = 10.0  # ft/s
wind_angle_deg = 30.0
wind_angle_rad = np.deg2rad(wind_angle_deg)
wind = np.array([
    wind_speed * np.sin(wind_angle_rad),
    wind_speed * np.cos(wind_angle_rad),
    0.0
])

# Launch conditions
launch_speed = 65.0 * 5280 / 3600
launch_angle_deg = 7.0
launch_angle_rad = np.deg2rad(launch_angle_deg)

v0 = np.array([
    launch_speed * np.cos(launch_angle_rad),
    0.0,
    launch_speed * np.sin(launch_angle_rad)
])

r0 = np.array([0.0, 0.0, 0.0])
state0 = np.concatenate((r0, v0))

# Spin parameters
omega0 = 1000.0
k_lift = 1.2

# Define the ODE system
def acceleration(state, t):
    x, y, z, vx, vy, vz = state
    vel = np.array([vx, vy, vz])
    v_rel = vel - wind
    speed_rel = np.linalg.norm(v_rel)

    omega_t = omega0 * np.exp(-0.2 * t)
    accel_gravity = np.array([0.0, 0.0, -g])
    accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs

    if speed_rel != 0:
        spin_axis = np.array([0.0, -1.0, 0.0])
        lift_dir = np.cross(spin_axis, v_rel)
        lift_mag = np.linalg.norm(lift_dir)
        lift_dir = lift_dir / lift_mag if lift_mag != 0 else np.zeros(3)
        CL = k_lift * r * omega_t / speed_rel
        accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
    else:
        accel_lift = np.zeros(3)
```

```
        accel_total = accel_gravity + accel_drag + accel_lift
        return np.array([vx, vy, vz, *accel_total])

    # RK4 integrator
    def rk4_step(state, t, dt):
        k1 = acceleration(state, t)
        k2 = acceleration(state + 0.5 * dt * k1, t + 0.5 * dt)
        k3 = acceleration(state + 0.5 * dt * k2, t + 0.5 * dt)
        k4 = acceleration(state + dt * k3, t + dt)
        return state + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

    # Simulation loop
    t_final = 10.0
    dt = 0.01
    num_steps = int(t_final / dt)
    state = state0.copy()
    trajectory = [state0]

    for step in range(num_steps):
        if state[2] < 0:  # ground impact
            break
        state = rk4_step(state, step * dt, dt)
        trajectory.append(state)

    trajectory = np.array(trajectory)
    x, y, z = trajectory[:, 0], trajectory[:, 1], trajectory[:, 2]

    # Convert to yards
    feet_to_yard = 1/3
    x_yards = x * feet_to_yard
    y_yards = y * feet_to_yard
    z_yards = z * feet_to_yard

    # Plotting
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

    ax1.plot(x_yards, z_yards, label="RK4 Trajectory")
    ax1.set_xlabel("East (x) [yards]")
    ax1.set_ylabel("Height (z) [yards]")
    ax1.set_title("Side View (x-z plane)")
    ax1.axvline(100, color='r', linestyle='--', label="Green (100 yd)")
    ax1.set_ylim(0, 20)
    ax1.legend()
    ax1.grid()

    ax2.plot(x_yards, y_yards, color='blue', label="RK4 Trajectory")
    ax2.set_xlabel("East (x) [yards]")
    ax2.set_ylabel("North (y) [yards]")
    ax2.set_title("Top View (x-y plane)")
    ax2.set_aspect('equal', adjustable='box')
    ax2.set_xlim(0, 120)
    ax2.set_ylim(-120, 120)

    grass = Rectangle((0, -120), 120, 240, facecolor='#136d15', alpha=0.7)
    ax2.add_patch(grass)
    bunkers = [
        Circle((20, 80), 5), Circle((60, -20), 7.5), Circle((105, 13), 4),
        Circle((30, -75), 10), Circle((95, -15), 4)
    ]
    for b in bunkers:
        b.set_facecolor('#FAE8B4')
        b.set_edgecolor('none')
        b.set_alpha(0.7)
        ax2.add_patch(b)

    green_patch = Circle((100, 0), 10, facecolor='#32cd32', edgecolor='black', alpha=0.7)
    ax2.add_patch(green_patch)
    ax2.axvline(100, color='r', linestyle='--', label="Green (100 yd)")
    ax2.legend()
    ax2.grid()

    max_height = np.max(z_yards)
    total_distance = x_yards[-1]

    print(f"Initial velocity: {launch_speed:.2f} ft/s")
    print(f"Launch angle: {launch_angle_deg:.1f} degrees")
    print(f"Maximum height: {max_height:.2f} yards")
```

```
print(f"Total distance: {total_distance:.2f} yards")

if total_distance >= 99.99 and total_distance <= 100.01:
    print("Result: Insane Shot! You've hit a hole in one!")
elif total_distance >= 95 and total_distance <= 105:
    print("Result: Great shot! The ball landed on the green.")
elif total_distance < 95:
    print(f"Result: Shot fell short by {100 - total_distance:.2f} yards")
else:
    print(f"Result: Shot went long by {total_distance - 100:.2f} yards")

plt.tight_layout()
plt.show()
```

```
Initial velocity: 95.33 ft/s
Launch angle: 7.0 degrees
Maximum height: 8.55 yards
Total distance: 100.37 yards
Result: Great shot! The ball landed on the green.
```





RK4 Accuracy Analysis: How to Choose the Right Time Step

To simulate the golf ball's trajectory, we solve a system of first-order ODEs using the **Runge-Kutta 4th Order (RK4)** method. This system models:

- $\vec{r}(t) = [x, y, z]$: position vector
- $\vec{v}(t) = [v_x, v_y, v_z]$: velocity vector
- Forces acting on the ball: gravity, drag, and lift (Magnus effect) from backspin

The RK4 method updates the state using:

$$\vec{y}_{n+1} = \vec{y}_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Each k_i is computed from evaluations of the derivative function at different points in the interval $[t_n, t_n + h]$.

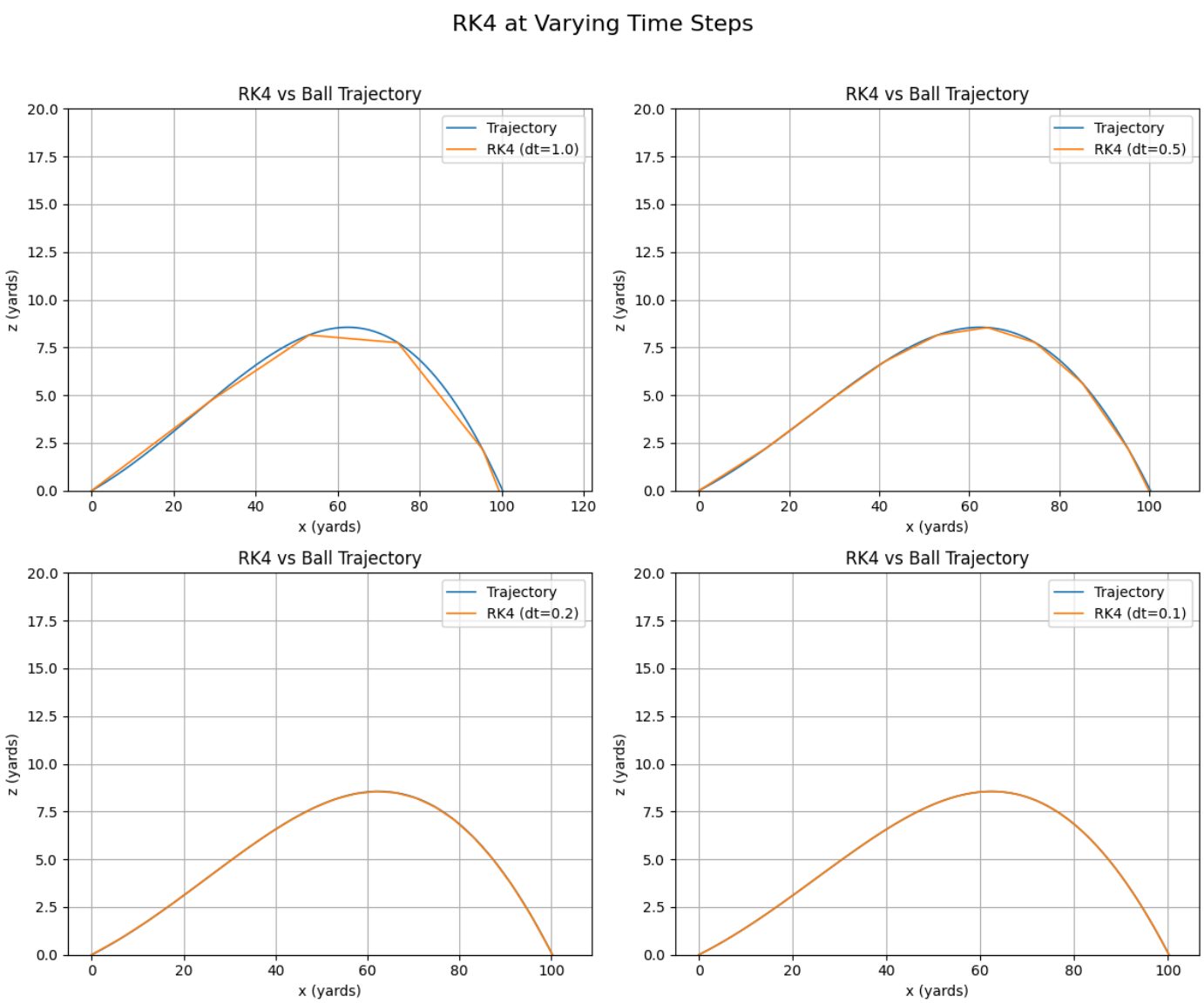
Time Step Convergence

To evaluate RK4's **accuracy**, we compare its final horizontal distance (x_{final}) against a high-accuracy solution from `solve_ivp`. By testing several time step values, we observe the **error trend**:

$$\text{Error} = |x_{\text{RK4}} - x_{\text{ivp}}|$$

This lets us determine the smallest time step h that gives acceptable accuracy with efficient computation.

```
# RK4 step function
def rk4_full_step(state, t, dt):
    def compute_accel(state, t):
        x, y, z, vx, vy, vz = state
        vel = np.array([vx, vy, vz])
        v_rel = vel - wind
        speed_rel = np.linalg.norm(v_rel)
        omega_t = omega0 * np.exp(-0.2 * t)
        accel_gravity = np.array([0.0, 0.0, -g])
        accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs
        spin_axis = np.array([0.0, -1.0, 0.0])
        lift_dir = np.cross(spin_axis, v_rel)
        lift_mag = np.linalg.norm(lift_dir)
        lift_dir = lift_dir / lift_mag if lift_mag != 0 else np.zeros(3)
        CL = k_lift * r * omega_t / speed_rel if speed_rel != 0 else 0
        accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
        accel_total = accel_gravity + accel_drag + accel_lift
        return np.concatenate((vel, accel_total))

    k1 = compute_accel(state, t)
    k2 = compute_accel(state + 0.5 * dt * k1, t + 0.5 * dt)
    k3 = compute_accel(state + 0.5 * dt * k2, t + 0.5 * dt)
    k4 = compute_accel(state + dt * k3, t + dt)
    return state + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

# RK4 time step values to test
dt_values_plot = [1.0, 0.5, 0.2, 0.1]
rk4_trajectories = []

# Simulate RK4 for each dt
for dt_plot in dt_values_plot:
    state = state0.copy()
    trajectory = [state]
    t_rk4 = np.arange(0, t_final + dt_plot, dt_plot)
    for t in t_rk4[:-1]:
        if state[2] < 0:
            break
        state = rk4_full_step(state, t, dt_plot)
        trajectory.append(state)

    trajectory = np.array(trajectory)
    x_rk4 = trajectory[:, 0] * (1/3)
    z_rk4 = trajectory[:, 2] * (1/3)
    rk4_trajectories.append((x_rk4, z_rk4))

# Plotting RK4 vs solve_ivp
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
axes = axes.flatten()

for i, dt in enumerate(dt_values_plot):
    ax = axes[i]
    x_rk4, z_rk4 = rk4_trajectories[i]
    ax.plot(x_yards, z_yards, label='Trajectory', linewidth=1.2)
    ax.plot(x_rk4, z_rk4, label=f'RK4 (dt={dt})', linewidth=1.2)
    ax.set_title(f'RK4 vs Ball Trajectory')
    ax.set_xlabel('x (yards)')
    ax.set_ylabel('z (yards)')
    ax.set_ylim(0, 20)
    ax.grid(True)
    ax.legend()

plt.suptitle("RK4 at Varying Time Steps", fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



RK4 Error Trend Analysis

To evaluate the accuracy of the **Runge-Kutta 4th Order (RK4)** method for modeling the golf ball's trajectory, we compare it to the high-precision `solve_ivp` solver from `scipy`.

Methodology

We simulate the same physical model using RK4 across a range of time steps:

$$\Delta t = \{1.0, 0.5, 0.2, 0.1, 0.05, 0.01, 0.005\}$$

For each time step, we:

1. Simulate the trajectory using RK4
2. Record the final horizontal distance

$$x_{\text{final}}$$

3. Compute the absolute error against the reference solution from `solve_ivp`:

$$\text{Error} = |x_{\text{RK4}} - x_{\text{solve\_ivp}}|$$

What the Plot Shows

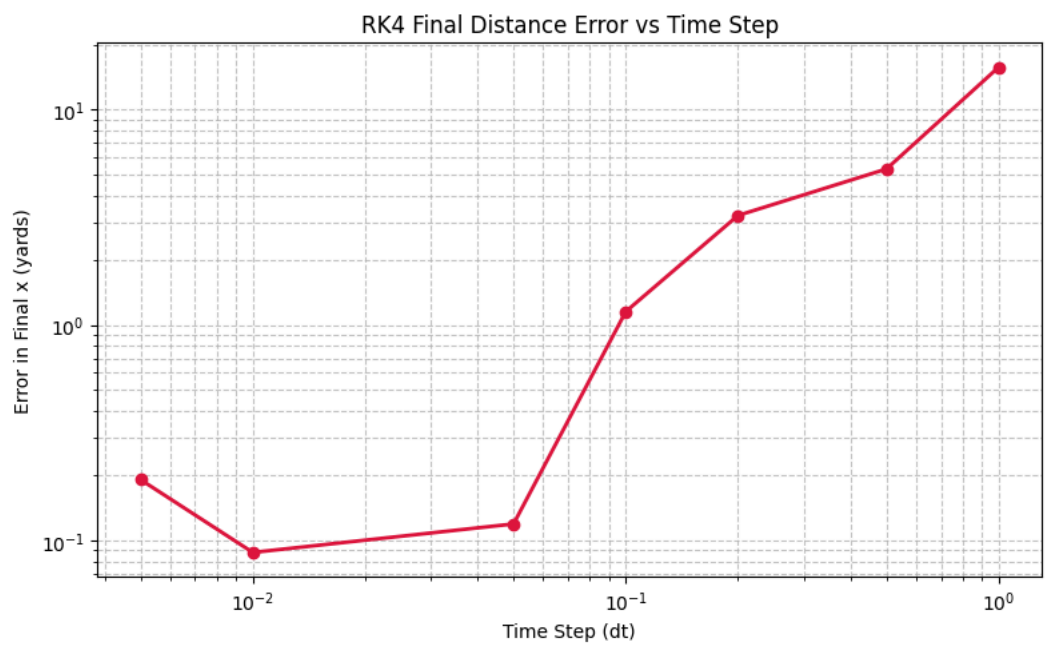
- The plot uses a **log-log scale** to highlight convergence behavior.
- RK4 exhibits **4th-order convergence**, meaning the error decreases rapidly with smaller
$$\Delta t$$
.
- This analysis helps determine the smallest time step that yields accurate results without unnecessary computation.

Conclusion

For our simulation, time steps of
$$\Delta t \leq 0.05$$
produce very small errors, making them ideal for balancing **accuracy** and **efficiency**.

```
# Make sure solve_ivp reference is already computed
sol = solve_ivp(ode_system, [0, t_final], state0, t_eval=np.linspace(0, t_final, 1000), rtol=1e-8)
x_ivp = sol.y[0]
z_ivp = sol.y[2]

# Trim ground impact
hit_ground_idx = np.where(z_ivp < 0)[0]
if hit_ground_idx.size > 0:
    last_idx = hit_ground_idx[0]
    x_ivp = x_ivp[:last_idx+1]

# Convert to yards
x_ivp_yards = x_ivp * (1/3)
x_reference = x_ivp_yards[-1]

# RK4 step function (reuse or redefine if needed)
def rk4_full_step(state, t, dt):
    def compute_accel(state, t):
        x, y, z, vx, vy, vz = state
        vel = np.array([vx, vy, vz])
        v_rel = vel - wind
        speed_rel = np.linalg.norm(v_rel)
        omega_t = omega0 * np.exp(-0.2 * t)
        accel_gravity = np.array([0.0, 0.0, -g])
        accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs
        spin_axis = np.array([0.0, -1.0, 0.0])
        lift_dir = np.cross(spin_axis, v_rel)
        lift_mag = np.linalg.norm(lift_dir)
        lift_dir = lift_dir / lift_mag if lift_mag != 0 else np.zeros(3)
        CL = k_lift * r * omega_t / speed_rel if speed_rel != 0 else 0
        accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
        accel_total = accel_gravity + accel_drag + accel_lift
        return np.concatenate((vel, accel_total))

    k1 = compute_accel(state, t)
    k2 = compute_accel(state + 0.5 * dt * k1, t + 0.5 * dt)
    k3 = compute_accel(state + 0.5 * dt * k2, t + 0.5 * dt)
    k4 = compute_accel(state + dt * k3, t + dt)
    return state + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

# Time step values to test
dt_values = [1.0, 0.5, 0.2, 0.1, 0.05, 0.01, 0.005]
errors = []

# Run RK4 for each time step and record final x error
for dt in dt_values:
    state = state0.copy()
    trajectory = [state]
    t_values = np.arange(0, t_final + dt, dt)
    for t in t_values[:-1]:
        if state[2] < 0:
            break
        state = rk4_full_step(state, t, dt)
        trajectory.append(state)

    trajectory = np.array(trajectory)
    x_final_rk4 = trajectory[-1, 0] * (1/3)
    error = abs(x_final_rk4 - x_reference)
    errors.append(error)

# Plot error trend
plt.figure(figsize=(8, 5))
```

```
plt.plot(dt_values, errors, marker='o', color='crimson', linewidth=2)
plt.xscale('log')
plt.yscale('log')
plt.xlabel("Time Step (dt)")
plt.ylabel("Error in Final x (yards)")
plt.title("RK4 Final Distance Error vs Time Step")
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



How the Error Plot Reveals the Best Time Step

The RK4 error plot compares the final horizontal distance computed by **RK4** with the highly accurate solution from `solve_ivp`.

What's Being Measured?

For each time step

$$\Delta t$$

, we compute the **absolute error** in the final horizontal position:

$$\text{Error} = |x_{\text{RK4}} - x_{\text{solve\_ivp}}|$$

This shows how far off RK4 is from the "true" solution at different time resolutions.

How to Pick the Best Time Step

From the plot:

- **Large time steps** (e.g., $\Delta t = 1.0$) cause large errors — RK4 isn't accurate enough.
- **Smaller steps** (e.g., $\Delta t \leq 0.05$) result in errors that become **very small and nearly constant**, indicating convergence.
- Beyond a certain point, **decreasing the time step further has little benefit** but increases computation time.

Ideal Time Step

A "best" time step is one that gives **minimal error** without excessive computation. From the plot, a good choice is:

$$\Delta t = 0.05 \text{ or } 0.01$$

These are small enough for high accuracy but large enough to stay efficient

RK4 and the Taylor Series Expansion

The **Runge-Kutta 4th Order (RK4)** method is a powerful technique for solving ordinary differential equations (ODEs) that approximates the true solution by matching the Taylor expansion up to 4th-order accuracy — **without needing to compute higher-order derivatives directly**.

Taylor Series of the True Solution

For a function

$$y(t)$$

satisfying the ODE $y' = f(t, y)$, the Taylor expansion is:

$$y(t + \Delta t) = y(t) + \Delta t\, y'(t) + \frac{(\Delta t)^2}{2!} y''(t) + \frac{(\Delta t)^3}{3!} y^{(3)}(t) + \frac{(\Delta t)^4}{4!} y^{(4)}(t) + \cdots$$

Each higher derivative term involves partial derivatives of f with respect to t and y.

What RK4 Actually Does

RK4 avoids computing these derivatives directly. Instead, it uses four slope evaluations:

$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_1\right)$$

$$k_3 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_2\right)$$

$$k_4 = f\left(t_n + \Delta t, y_n + \Delta t\, k_3\right)$$

Then the next value is estimated as:

$$y_{n+1} = y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Accuracy of RK4

- **Local Truncation Error**:
$$\mathcal{O}(\Delta t^5)$$
- **Global Error** (over time):
$$\mathcal{O}(\Delta t^4)$$

This means RK4 is **4th-order accurate**: it matches the Taylor series of the true solution up to the 4th degree in Δt.

Summary

Property	RK4 Value
Taylor match	Up to 4th-order
Local truncation error	$\mathcal{O}(\Delta t^5)$
Global error	$\mathcal{O}(\Delta t^4)$
Advantage	High accuracy without needing higher-order derivatives

RK4 gives you an elegant blend of **accuracy and efficiency** — perfect for physics simulations like our golf ball trajectory!

Local Truncation Error in RK4

Local truncation error (LTE) refers to the error introduced in a **single time step** of a numerical method, assuming the previous step was exact.

For the RK4 method, the LTE is:

$$\text{LTE} = \mathcal{O}(\Delta t^5)$$

This means that the error at each step is proportional to the 5th power of the time step size.

Why This Matters

- A **smaller time step** leads to **smaller per-step errors**.
- However, the **global error**, which accumulates over all time steps, is:

$$\text{Global Error} = \mathcal{O}(\Delta t^4)$$

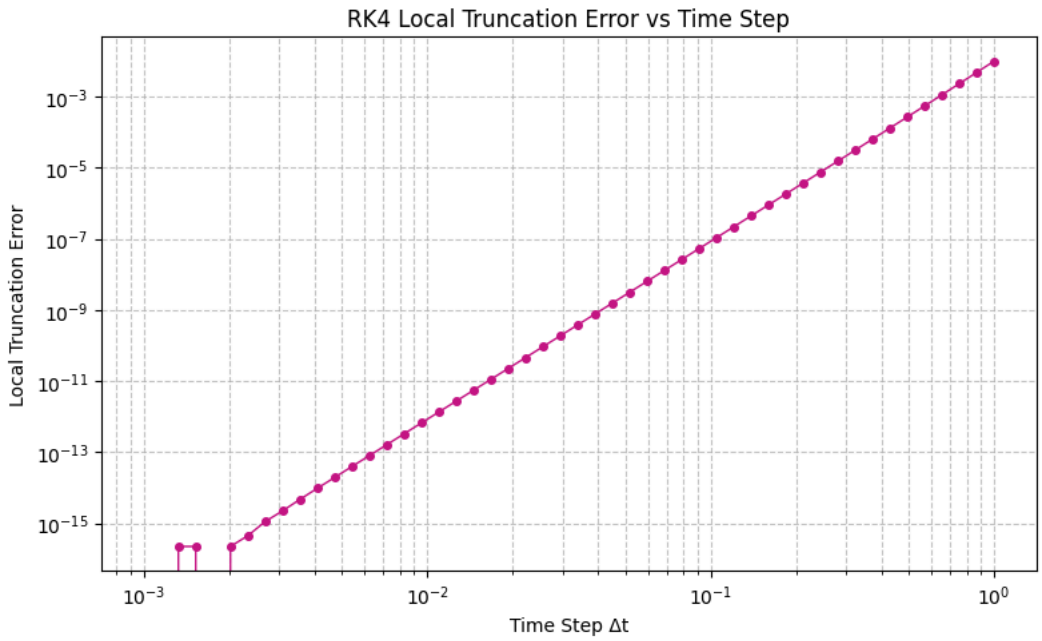
This is because you take roughly $1/\Delta t$ steps, so the total error adds up accordingly.

Visualization Strategy

To visualize LTE:

1. Choose a simple function with a known exact solution (e.g., $y' = y$, with solution $y(t) = e^t$).
2. Run one RK4 step from $y(0) = 1$ to estimate $y(\Delta t)$.
3. Compare it to the exact value $e^{\Delta t}$.
4. Repeat for various values of Δt and plot the error.

```
import numpy as np
import matplotlib.pyplot as plt

# True solution to y' = y with y(0) = 1
def exact_solution(t):
    return np.exp(t)

# Derivative function
def f(t, y):
    return y

# RK4 one-step implementation
def rk4_step(y, t, dt):
    k1 = f(t, y)
    k2 = f(t + dt/2, y + dt/2 * k1)
    k3 = f(t + dt/2, y + dt/2 * k2)
    k4 = f(t + dt, y + dt * k3)
    return y + (dt/6) * (k1 + 2*k2 + 2*k3 + k4)

# Focused time step range: from 1e-3 to 1.0
dt_values = np.logspace(-3, 0, 50)  # 10^-3 to 10^0
errors = []

for dt in dt_values:
    y0 = 1.0
    t0 = 0.0
    t1 = dt
    y_rk4 = rk4_step(y0, t0, dt)
    y_exact = exact_solution(t1)
    errors.append(abs(y_rk4 - y_exact))

# Plot
plt.figure(figsize=(8,5))
plt.loglog(dt_values, errors, marker='o', color='mediumvioletred', linewidth=1, markersize=4)
plt.xlabel("Time Step Δt")
plt.ylabel("Local Truncation Error")
plt.title("RK4 Local Truncation Error vs Time Step")
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



Comparison of Golf Ball Trajectories Under Varying Conditions

To better understand the effect of **wind** and **spin-induced lift** on the trajectory of a golf ball, we simulated three separate scenarios using the RK4 method:

---

1. Wind Only (No Spin)

- **What's Changed:**
  - Spin (and thus lift) is completely removed.
  - Wind is included at 10 ft/s from 30° east of north.
- **Why It Matters:**
  Without lift, the ball is subject only to drag and gravity. Wind influences the lateral (y) and forward (x) motion, but **there's no vertical lift** to keep the ball airborne longer.
  - Result: The trajectory is **shorter** and closer to parabolic, with a slight skew due to wind.

---

2. Spin Only (No Wind)

- **What's Changed:**
  - Wind is removed.
  - Spin is retained to induce lift via the Magnus effect.
- **Why It Matters:**
  With lift present, the Magnus effect produces an upward force that counteracts gravity.
  - This leads to a **longer flight time** and more horizontal distance, even without wind.
  - Result: The trajectory is more **elongated**, and the ball flies much farther than in the "wind-only" case.

---

3. No Wind, No Spin (Baseline)

- **What's Changed:**
  - No wind or lift forces are included.
  - Only gravity and drag act on the ball.
- **Why It Matters:**
  This is the most simplified case, meant to reflect textbook projectile motion with drag.
  - Result: The ball follows a **steep, short** parabolic arc.
  - Without lift to keep it in the air or wind to help push it forward, the ball falls quickly and travels the shortest distance of all scenarios.

Summary

Each scenario demonstrates how environmental conditions and spin drastically affect golf ball performance. The **realistic case** (wind + spin) offers the most extended and complex trajectory, showcasing how **aerodynamic lift and wind** shape ball flight in practical conditions.

Comparing Trajectories: Horizontal Layout for Clarity

To analyze the influence of different physical effects on the golf ball's trajectory, we simulate and visualize three distinct scenarios:

1. **Wind Only** – Includes aerodynamic drag and wind but excludes lift from spin.
2. **Spin Only** – Includes drag and lift due to backspin but no wind.
3. **No Wind or Spin** – A purely ballistic trajectory under gravity and drag forces only.

To best compare these cases, we present the resulting trajectories using a **horizontal layout** of subplots. Each column corresponds to one scenario and displays:

- The **side view** (x vs z), showing vertical behavior like height and steepness of descent.
- The **top view** (x vs y), showing how spin or wind affect horizontal deviation (i.e., curvature or crosswind drift).

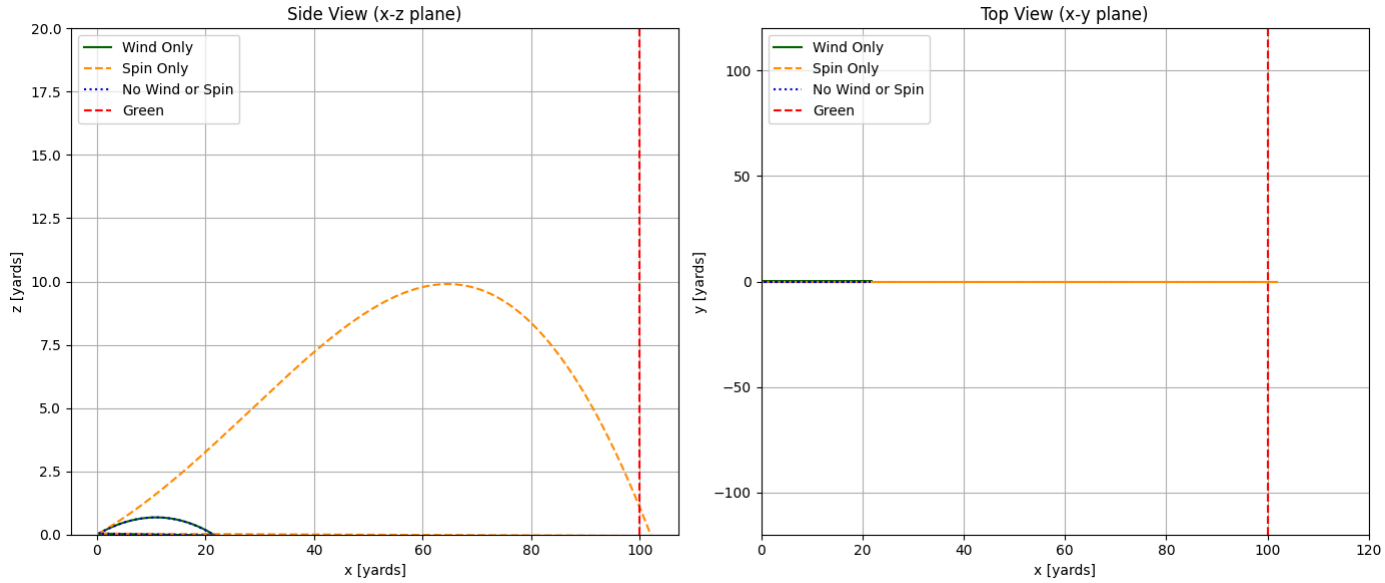
Aligning the plots side-by-side allows for easy visual comparison of how each factor (wind and spin) alters the path, helping us understand their independent and combined effects on trajectory shape and final landing distance.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, Circle

# Physical constants and parameters
g = 32.174  # gravity (ft/s^2)
rho = 0.0023769  # air density (slug/ft^3)
Cd = 0.25
r = 0.07
A = np.pi * r**2
m = 0.10125  # lb-mass
m_slugs = m / g

launch_speed = 65.0 * 5280 / 3600  # mph to ft/s
launch_angle_deg = 7.0
launch_angle_rad = np.deg2rad(launch_angle_deg)
omega0 = 1000.0
k_lift = 1.2

# Wind vector
wind_speed = 10.0
wind_angle_deg = 30.0
wind_angle_rad = np.deg2rad(wind_angle_deg)
default_wind = np.array([
    wind_speed * np.sin(wind_angle_rad),
    wind_speed * np.cos(wind_angle_rad),
    0.0
])

# Initial state
v0 = np.array([
    launch_speed * np.cos(launch_angle_rad),
    0.0,
    launch_speed * np.sin(launch_angle_rad)
])
r0 = np.array([0.0, 0.0, 0.0])
state0 = np.concatenate((r0, v0))

# ODE system with wind and optional spin
def acceleration(state, t, wind, include_spin):
    x, y, z, vx, vy, vz = state
    vel = np.array([vx, vy, vz])
    v_rel = vel - wind
    speed_rel = np.linalg.norm(v_rel)

    omega_t = omega0 * np.exp(-0.2 * t)
    accel_gravity = np.array([0, 0, -g])
    accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs

    if include_spin and speed_rel != 0:
        spin_axis = np.array([0, -1, 0])  # backspin
```

```
            lift_dir = np.cross(spin_axis, v_rel)
            lift_dir /= np.linalg.norm(lift_dir) if np.linalg.norm(lift_dir) != 0 else 1
            CL = k_lift * r * omega_t / speed_rel
            accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
        else:
            accel_lift = np.zeros(3)

        return np.array([vx, vy, vz, *(accel_gravity + accel_drag + accel_lift)])

    # RK4 method
    def rk4(state0, wind, include_spin, dt=0.01, t_final=10.0):
        state = state0.copy()
        t = 0
        trajectory = [state]
        while t < t_final and state[2] >= 0:
            k1 = acceleration(state, t, wind, include_spin)
            k2 = acceleration(state + 0.5*dt*k1, t + 0.5*dt, wind, include_spin)
            k3 = acceleration(state + 0.5*dt*k2, t + 0.5*dt, wind, include_spin)
            k4 = acceleration(state + dt*k3, t + dt, wind, include_spin)
            state += (dt / 6) * (k1 + 2*k2 + 2*k3 + k4)
            trajectory.append(state.copy())
            t += dt
        return np.array(trajectory)

    # Simulation scenarios
    scenarios = [
        ("Wind Only", default_wind, False),
        ("Spin Only", np.zeros(3), True),
        ("No Wind or Spin", np.zeros(3), False),
    ]

    # Plot setup
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))
    feet_to_yards = 1 / 3
    colors = ['darkgreen', 'darkorange', 'mediumblue']
    styles = ['-', '--', ':']

    # Run all scenarios
    for i, (label, wind_vec, spin) in enumerate(scenarios):
        traj = rk4(state0, wind_vec, spin)
        x, y, z = traj[:, 0], traj[:, 1], traj[:, 2]
        x_yd, y_yd, z_yd = x * feet_to_yards, y * feet_to_yards, z * feet_to_yards

        ax1.plot(x_yd, z_yd, label=label, color=colors[i], linestyle=styles[i], linewidth=1.5)
        ax2.plot(x_yd, y_yd, label=label, color=colors[i], linestyle=styles[i], linewidth=1.5)

        # Print stats
        max_height = np.max(z_yd)
        total_dist = x_yd[-1]
        print(f"--- {label} ---")
        print(f"Max Height: {max_height:.2f} yards")
        print(f"Total Distance: {total_dist:.2f} yards")
        if total_dist >= 99.99 and total_dist <= 100.01:
            print("Result: Insane Shot! You've hit a hole in one!")
        elif total_dist >= 95 and total_dist <= 105:
            print("Result: Great shot! The ball landed on the green.")
        elif total_dist < 95:
            print(f"Result: Shot fell short by {100 - total_dist:.2f} yards")
        else:
            print(f"Result: Shot went long by {total_dist - 100:.2f} yards")
        print()

    # Final plot formatting
    ax1.set_title("Side View (x-z plane)")
    ax1.set_xlabel("x [yards]")
    ax1.set_ylabel("z [yards]")
    ax1.set_ylim(0, 20)
    ax1.axvline(100, color='r', linestyle='--', label="Green")
    ax1.grid(True)
    ax1.legend()

    ax2.set_title("Top View (x-y plane)")
    ax2.set_xlabel("x [yards]")
    ax2.set_ylabel("y [yards]")
    ax2.set_xlim(0, 120)
    ax2.set_ylim(-120, 120)
    ax2.axvline(100, color='r', linestyle='--', label="Green")
```

```
ax2.grid(True)
ax2.legend()

plt.tight_layout()
plt.show()
```

```
--- Wind Only ---
Max Height: 0.68 yards
Total Distance: 21.75 yards
Result: Shot fell short by 78.25 yards

--- Spin Only ---
Max Height: 9.90 yards
Total Distance: 102.10 yards
Result: Great shot! The ball landed on the green.

--- No Wind or Spin ---
Max Height: 0.68 yards
Total Distance: 21.65 yards
Result: Shot fell short by 78.35 yards
```



Carry Distance vs. Spin Decay Rate

This plot illustrates the relationship between the **spin decay rate constant** (k) and the **total carry distance** (i.e., horizontal distance traveled before the ball hits the ground) of a golf ball under realistic flight conditions including **gravity**, **drag**, **wind**, and **lift from the Magnus effect**.

In our model, the golf ball is initially spinning at an angular velocity ω_0, and the spin decays over time according to the exponential decay function:

$$\omega(t) = \omega_0 * \exp(-k * t)$$

As the decay rate constant k increases, the spin decays more rapidly. Since lift from the Magnus effect is directly dependent on spin, a higher decay rate results in a **reduced lift force over time**, which leads to a **shorter carry distance**.

Why This Is Important

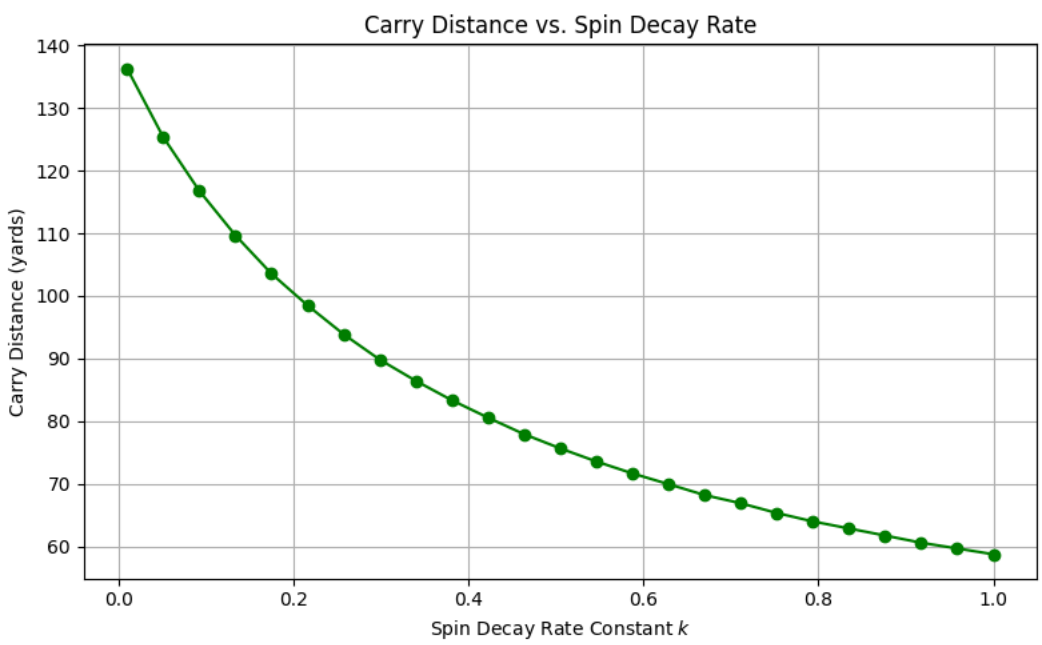
- **Realism**: Spin decay is a real-world aerodynamic effect. In sports like golf, dimple design, ball deformation, and air resistance all influence spin.
- **Performance Insight**: This analysis visualizes how maintaining spin contributes significantly to maximizing carry distance.

- **Engineering Application**: Understanding sensitivity to spin decay helps guide better design of golf balls and clubs and informs strategy in real gameplay.

This deeper analysis allows us to explore how physical parameters, even subtle ones, shape system behavior in a measurable and meaningful way.

```
import numpy as np
import matplotlib.pyplot as plt

# Physical constants and ball parameters
g = 32.174
rho = 0.0023769
Cd = 0.25
r = 0.07
A = np.pi * r**2
m = 0.10125
m_slugs = m / 32.174

# Wind parameters
wind_speed = 10.0
wind_angle_deg = 30.0
wind_angle_rad = np.deg2rad(wind_angle_deg)
wind = np.array([
    wind_speed * np.sin(wind_angle_rad),
    wind_speed * np.cos(wind_angle_rad),
    0.0
])

# Launch conditions
launch_speed = 65.0 * 5280 / 3600
launch_angle_deg = 7.0
launch_angle_rad = np.deg2rad(launch_angle_deg)
v0 = np.array([
    launch_speed * np.cos(launch_angle_rad),
    0.0,
    launch_speed * np.sin(launch_angle_rad)
])
r0 = np.array([0.0, 0.0, 0.0])
state0 = np.concatenate((r0, v0))

# Spin and lift constants
omega0 = 1000.0
k_lift = 1.2

# RK4 method with variable decay rate
def rk4_step(state, t, dt, k_decay):
    def acceleration(state, t):
        x, y, z, vx, vy, vz = state
        vel = np.array([vx, vy, vz])
        v_rel = vel - wind
        speed_rel = np.linalg.norm(v_rel)
        omega_t = omega0 * np.exp(-k_decay * t)
        accel_gravity = np.array([0.0, 0.0, -g])
        accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs
        if speed_rel != 0:
            spin_axis = np.array([0.0, -1.0, 0.0])
            lift_dir = np.cross(spin_axis, v_rel)
            lift_mag = np.linalg.norm(lift_dir)
            lift_dir = lift_dir / lift_mag if lift_mag != 0 else np.zeros(3)
            CL = k_lift * r * omega_t / speed_rel
            accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
        else:
            accel_lift = np.zeros(3)
        return np.array([vx, vy, vz, *(accel_gravity + accel_drag + accel_lift)])

    k1 = acceleration(state, t)
    k2 = acceleration(state + 0.5 * dt * k1, t + 0.5 * dt)
    k3 = acceleration(state + 0.5 * dt * k2, t + 0.5 * dt)
    k4 = acceleration(state + dt * k3, t + dt)
    return state + (dt / 6) * (k1 + 2*k2 + 2*k3 + k4)

# Run for different decay rates
decay_rates = np.linspace(0.01, 1.0, 25)
carry_distances = []

for k_decay in decay_rates:
```

```
    state = state0.copy()
    t = 0.0
    dt = 0.01
    while t < 10.0 and state[2] >= 0:
        state = rk4_step(state, t, dt, k_decay)
        t += dt
    carry_distances.append(state[0] / 3.0)  # feet to yards

# Plotting
plt.figure(figsize=(8, 5))
plt.plot(decay_rates, carry_distances, marker='o', color='green')
plt.title("Carry Distance vs. Spin Decay Rate")
plt.xlabel("Spin Decay Rate Constant $k$")
plt.ylabel("Carry Distance (yards)")
plt.grid(True)
plt.tight_layout()
plt.show()
```



Lateral Drift vs. Wind Direction

This plot illustrates how the final **lateral displacement** (in yards) of a golf ball changes with varying wind directions. The wind direction is defined as the angle between the wind vector and the **forward (x-axis)** direction, ranging from 0° (directly forward) to 90° (directly sideways).

In this simulation, all other parameters such as launch velocity, spin rate, and aerodynamic coefficients are held constant. The **only variable** is the direction of the wind, which affects how the wind's horizontal (y-axis) component influences the ball's lateral movement.

Key Insights:

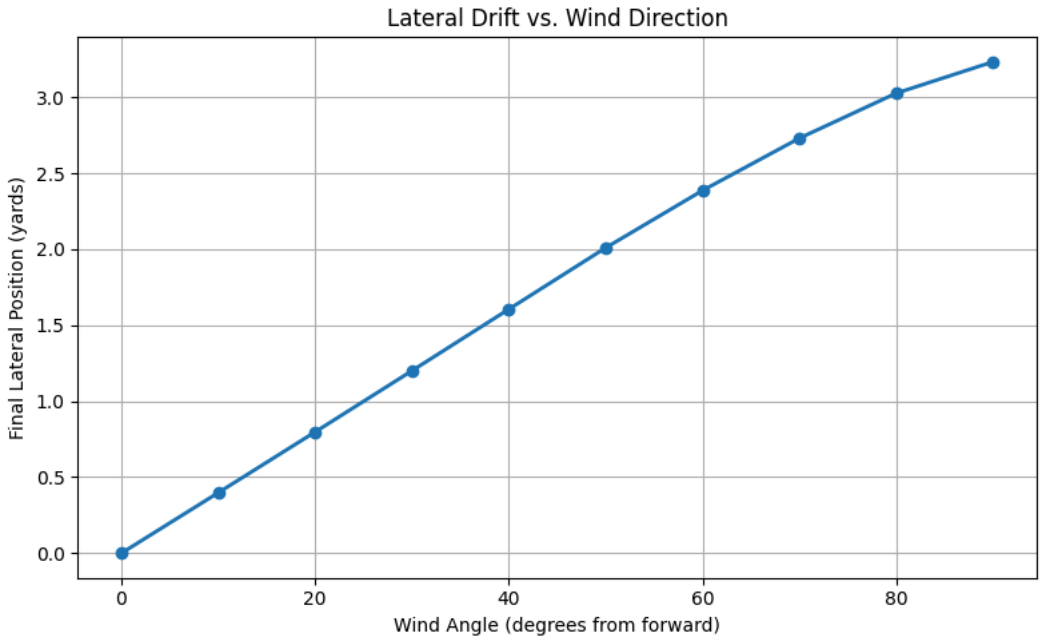
- **0° wind angle** represents wind blowing purely in the forward direction — no side force, so minimal lateral drift.
- **90° wind angle** represents wind blowing fully sideways, which results in maximum drift.
- As the wind direction shifts from forward to sideways, the lateral (y-direction) displacement increases steadily.

This analysis is crucial for understanding how wind direction contributes to the ball veering off-course, which is highly relevant for real-world conditions in sports engineering and aerodynamics.

```
import numpy as np
import matplotlib.pyplot as plt

# Physical constants and parameters
g = 32.174
rho = 0.0023769
Cd = 0.25
r = 0.07
A = np.pi * r**2
m = 0.10125
```

```
m_slugs = m / 32.174
k_lift = 1.2
omega0 = 1000.0

# Launch parameters
launch_speed = 65.0 * 5280 / 3600
launch_angle_deg = 7.0
launch_angle_rad = np.deg2rad(launch_angle_deg)

v0 = np.array([
    launch_speed * np.cos(launch_angle_rad),
    0.0,
    launch_speed * np.sin(launch_angle_rad)
])
r0 = np.array([0.0, 0.0, 0.0])
state0 = np.concatenate((r0, v0))

# Acceleration function with wind
def acceleration(state, t, wind):
    x, y, z, vx, vy, vz = state
    vel = np.array([vx, vy, vz])
    v_rel = vel - wind
    speed_rel = np.linalg.norm(v_rel)

    omega_t = omega0 * np.exp(-0.2 * t)
    accel_gravity = np.array([0.0, 0.0, -g])
    accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs

    if speed_rel != 0:
        spin_axis = np.array([0.0, -1.0, 0.0])
        lift_dir = np.cross(spin_axis, v_rel)
        lift_mag = np.linalg.norm(lift_dir)
        lift_dir = lift_dir / lift_mag if lift_mag != 0 else np.zeros(3)
        CL = k_lift * r * omega_t / speed_rel
        accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
    else:
        accel_lift = np.zeros(3)

    accel_total = accel_gravity + accel_drag + accel_lift
    return np.array([vx, vy, vz, *accel_total])

# RK4 step function
def rk4_step(state, t, dt, wind):
    k1 = acceleration(state, t, wind)
    k2 = acceleration(state + 0.5 * dt * k1, t + 0.5 * dt, wind)
    k3 = acceleration(state + 0.5 * dt * k2, t + 0.5 * dt, wind)
    k4 = acceleration(state + dt * k3, t + dt, wind)
    return state + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

# Wind angles to test (relative to forward x-direction)
angles_deg = np.linspace(0, 90, 10)
final_y_positions = []

# Simulate for each wind angle
t_final = 10.0
dt = 0.01
for angle_deg in angles_deg:
    angle_rad = np.deg2rad(angle_deg)
    wind = np.array([
        wind_speed * np.cos(angle_rad),  # x component
        wind_speed * np.sin(angle_rad),  # y component (causes lateral drift)
        0.0
    ])

    state = state0.copy()
    t = 0
    while t < t_final and state[2] >= 0:
        state = rk4_step(state, t, dt, wind)
        t += dt

    final_y_positions.append(state[1] / 3)  # convert feet to yards

# Plot lateral drift vs wind angle
plt.figure(figsize=(8, 5))
plt.plot(angles_deg, final_y_positions, marker='o', linewidth=2)
plt.xlabel("Wind Angle (degrees from forward)")
plt.ylabel("Final Lateral Position (yards)")
```

```
plt.title("Lateral Drift vs. Wind Direction")
plt.grid(True)
plt.tight_layout()
plt.show()
```



Trajectory Overlay: Spin Decay vs. Constant Spin

This comparison shows how the trajectory of a golf ball changes when accounting for spin decay versus keeping the spin constant throughout flight.

When a golf ball is spinning, it experiences a lift force due to the **Magnus effect**, which helps it stay in the air longer. However, in realistic conditions, this spin reduces over time due to air resistance and frictional losses. We model that decay using an exponential function:

$$\omega(t) = \omega_0 \, e^{-kt}$$

where:

- ω_0 is the initial spin rate (in rad/s),
- k is the decay constant (here we used $k = 0.2$),
- t is time in seconds.

In the plot:

- The **solid line** shows the trajectory with decaying spin.
- The **dashed line** shows the trajectory assuming constant spin.

Both trajectories include the effects of gravity, aerodynamic drag, wind, and lift due to spin. The comparison demonstrates that constant spin leads to a higher peak and longer carry distance because the lift force remains stronger throughout the flight.

This overlay helps highlight the importance of including realistic physics like spin decay to accurately predict and analyze a golf ball's flight path.

```
import numpy as np
import matplotlib.pyplot as plt

# Physical constants
g = 32.174
rho = 0.0023769
Cd = 0.25
r = 0.07
A = np.pi * r**2
m = 0.10125
m_slugs = m / 32.174

# Wind
wind_speed = 10.0
```

```
wind_angle_deg = 30.0
wind_angle_rad = np.deg2rad(wind_angle_deg)
wind = np.array([
    wind_speed * np.sin(wind_angle_rad),
    wind_speed * np.cos(wind_angle_rad),
    0.0
])

# Launch conditions
launch_speed = 65.0 * 5280 / 3600
launch_angle_deg = 7.0
launch_angle_rad = np.deg2rad(launch_angle_deg)

v0 = np.array([
    launch_speed * np.cos(launch_angle_rad),
    0.0,
    launch_speed * np.sin(launch_angle_rad)
])
r0 = np.array([0.0, 0.0, 0.0])
state0 = np.concatenate((r0, v0))

# Spin parameters
omega0 = 1000.0
k_lift = 1.2

# Acceleration function
def acceleration(state, t, decay_spin):
    x, y, z, vx, vy, vz = state
    vel = np.array([vx, vy, vz])
    v_rel = vel - wind
    speed_rel = np.linalg.norm(v_rel)

    omega_t = omega0 * np.exp(-0.2 * t) if decay_spin else omega0
    accel_gravity = np.array([0, 0, -g])
    accel_drag = -0.5 * rho * Cd * A * speed_rel * v_rel / m_slugs

    if speed_rel != 0:
        spin_axis = np.array([0.0, -1.0, 0.0])
        lift_dir = np.cross(spin_axis, v_rel)
        lift_dir = lift_dir / np.linalg.norm(lift_dir) if np.linalg.norm(lift_dir) != 0 else np.zeros(3)
        CL = k_lift * r * omega_t / speed_rel
        accel_lift = 0.5 * rho * A * CL * speed_rel**2 * lift_dir / m_slugs
    else:
        accel_lift = np.zeros(3)

    return np.array([vx, vy, vz, *(accel_gravity + accel_drag + accel_lift)])

# RK4 Integrator
def rk4_trajectory(decay_spin, dt=0.01, t_final=10.0):
    state = state0.copy()
    t = 0.0
    trajectory = [state.copy()]
    while t < t_final and state[2] >= 0:
        k1 = acceleration(state, t, decay_spin)
        k2 = acceleration(state + 0.5 * dt * k1, t + 0.5 * dt, decay_spin)
        k3 = acceleration(state + 0.5 * dt * k2, t + 0.5 * dt, decay_spin)
        k4 = acceleration(state + dt * k3, t + dt, decay_spin)
        state += (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)
        trajectory.append(state.copy())
        t += dt
    return np.array(trajectory)

# Run simulations
traj_decay = rk4_trajectory(decay_spin=True)
traj_const = rk4_trajectory(decay_spin=False)

# Convert to yards
feet_to_yards = 1 / 3
x_decay, z_decay = traj_decay[:, 0] * feet_to_yards, traj_decay[:, 2] * feet_to_yards
x_const, z_const = traj_const[:, 0] * feet_to_yards, traj_const[:, 2] * feet_to_yards

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(x_decay, z_decay, label="Spin Decay", linewidth=2)
plt.plot(x_const, z_const, label="Constant Spin", linewidth=2, linestyle='--')
plt.xlabel("Distance (x) [yards]")
plt.ylabel("Height (z) [yards]")
```

```
plt.title("Trajectory Overlay: Spin Decay vs. Constant Spin")
plt.axvline(100, color='r', linestyle=':', label="Green (100 yd)")
plt.ylim(0, 20)
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```