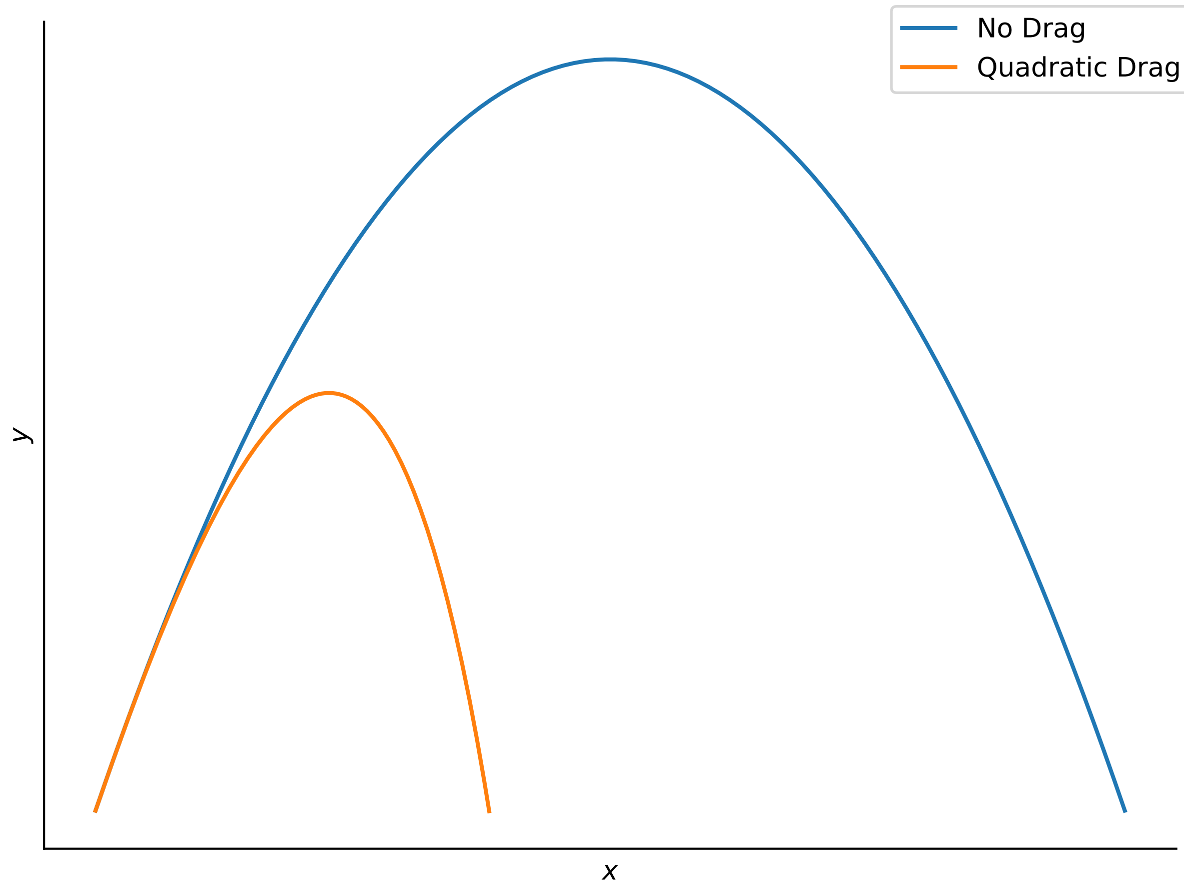
Realistic Projectiles

Vincent Edwards, Julia Corrales, and Rachel Gossard

May 25, 2025



Contents

1. Quadratic Drag Equation

In introductory physics, projectiles are typically modeled as experiencing negligible air drag. For this project, projectiles were modeled as experiencing *quadratic drag*.

$$\frac{d^2\vec{r}}{dt^2} = \vec{g} - kv^2\hat{v} \tag{1}$$

The terms in this equation are as follows:

$$\begin{aligned} \vec{r} &= \begin{pmatrix} x \\ y \end{pmatrix} && \text{(position)} \\ \vec{v} &= \frac{d\vec{r}}{dt} && \text{(velocity)} \\ \vec{g} &= \begin{pmatrix} 0 \\ -g \end{pmatrix} && \text{(gravitation acceleration)} \\ k &= \text{"constant"} && \text{(drag constant)} \end{aligned} \tag{2}$$

The y axis points straight up, and the x axis points horizontally along the plane of motion of the projectile. This keeps the motion in 2 dimensions. Projectiles were started on the ground at $(x, y) = (0, 0)$.

To focus on scale-independent features of the motion, units of distance and time were used such that $g = 1$ and $k = 1$. This makes the terminal speed $v_\infty = 1$.

2. Runge-Kutta Four (RK4) Method for Systems

To solve the system of differential equations, the RK4 method for systems was used.

$$\begin{aligned} \frac{d\vec{u}}{dt} &= \vec{f}(t, \vec{u}) \\ \vec{k}_1 &= \vec{f}(t_i, \vec{u}_i) \\ \vec{k}_2 &= \vec{f}\left(t_i + \frac{h}{2}, \vec{u}_i + \frac{h}{2}\vec{k}_1\right) \\ \vec{k}_3 &= \vec{f}\left(t_i + \frac{h}{2}, \vec{u}_i + \frac{h}{2}\vec{k}_2\right) \\ \vec{k}_4 &= \vec{f}\left(t_i + h, \vec{u}_i + h\vec{k}_3\right) \\ \vec{u}_{i+1} &= \vec{u}_i + \frac{h}{6}\left(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4\right) \end{aligned} \tag{3}$$

The `rk4.py` file contains a `calculate()` function that implements the RK4 method for systems in Python. `calculate()` returns arrays containing t and $\vec{u}$ values, and it takes the following parameters:

- `t_0`: a starting t value
- `u_0`: an array containing the initial value for each variable in the system $\vec{u}_0$
- `h`: a step size
- `diff`: a function that takes `t` and `u` as inputs and returns an array containing the result of the differential equation $\vec{f}(t, \vec{u})$

- `should_exit`: a function that takes `t` and `u` as inputs and returns `True` when the iterations should stop

rk4.py

```
import numpy as np

def calculate(t_0, u_0, h, diff, should_exit):
    """
    Calculate t values & u vectors using the vector RK4 method on a system of 1st
order ODEs.
    Return an array of t values, and an array of u vectors.

    - t_0: starting t value
    - u_0: starting u vector
    - h: step size
    - diff: function that takes t and u as arguments and returns du/dt
    - should_exit: function that takes t and u as arguments and returns True if no
more steps should be taken
    """
    t = [t_0]
    u = [u_0]

    while not should_exit(t[-1], u[-1]):
        k_1 = diff(t[-1], u[-1])
        k_2 = diff(t[-1] + h/2, u[-1] + h/2 * k_1)
        k_3 = diff(t[-1] + h/2, u[-1] + h/2 * k_2)
        k_4 = diff(t[-1] + h, u[-1] + h * k_3)

        u_next = u[-1] + h/6 * (k_1 + 2*k_2 + 2*k_3 + k_4)
        u.append(u_next)
        t_next = t[-1] + h
        t.append(t_next)

    return np.array(t), np.array(u)
```

The `projectile.py` file contains functions to help simulate the motion of a projectile experiencing quadratic drag. The `u_prime()` function implements the system of differential equations that describe the motion of the projectile.

$$\begin{aligned} \vec{u} &= \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix} \\ \frac{d\vec{u}}{dt} &= \begin{pmatrix} v_x \\ v_y \\ -kvv_x \\ -g - kvv_y \end{pmatrix} \end{aligned} \tag{4}$$

The `launch()` function simulates launching a projectile from the origin with the given initial velocity `v_0`, and it returns arrays containing t and $\vec{u}$ values. By default the `should_exit` parameter is set to the `below_ground()` function, which returns `True` when the projectile falls below the ground ($y < 0$).

The `horizontal_range()` function was used to determine the horizontal range for a projectile launched with the given initial velocity `v_0`. It does so by calculating the intersection between the ground and the line connecting the last two points of the projectile's motion. If the last point is labeled (x_{-1}, y_{-1}) and the second to last point is labeled (x_{-2}, y_{-2}), then the slope of the line connecting them is

$$m = \frac{y_{-2} - y_{-1}}{x_{-2} - x_{-1}} \tag{5}$$

The equation for that line can be written as

$$y = m(x - x_{-1}) + y_{-1} \tag{6}$$

The x-value where the line intersects the ground ($y = 0$) corresponds to the range R. Solving for that intersection yields

$$\begin{aligned} 0 &= m(R - x_{-1}) + y_{-1} \\ R &= -\frac{y_{-1}}{m} + x_{-1} \end{aligned} \tag{7}$$

Note that if the line is vertical, which occurs when $x_{-2} = x_{-1}$, then the range is simply equal to the x-value of either of the last two points.

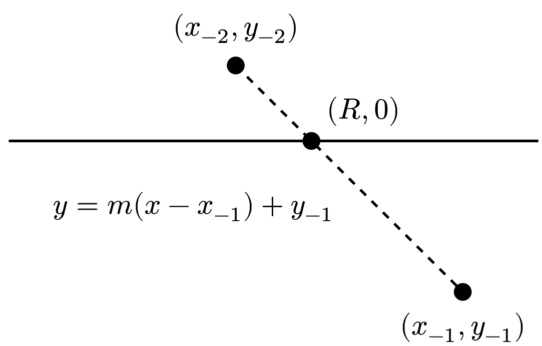


Figure 1: Intersection Between Ground and Last Two Points

projectile.py

```
import rk4
import numpy as np

def u_prime(t, u):
    """
    Return an array of derivatives with respect to t for each component of the vector
u.
    u consists of x, y, v_x, and v_y.
    """
    k = 1
```

```
    g = 1

    x, y, v_x, v_y = u
    speed = np.sqrt(v_x**2 + v_y**2);
    drag_part = k * speed
    if speed == 0:
        drag_x = 0
        drag_y = 0
    else:
        drag_x = drag_part * v_x
        drag_y = drag_part * v_y

    return np.array([
        v_x,
        v_y,
        -drag_x,
        -g - drag_y,
    ])

def below_ground(t, u):
    y = u[1]
    return y < 0

def launch(v_0, should_exit=below_ground):
    """
    Launch a projectile from the origin with the given launch velocity.
    By default, stop after the projectile hits the ground (when y < 0).
    If desired, an alternate function of t and u can be passed.
    This function should return True when the exit condition is met.

    Return the arrays of t and u.
    Note that u consists of x, y, v_x, and v_y.
    """
    t_0 = 0.0
    h = 0.001
    v_x, v_y = v_0
    u_0 = np.array([0, 0, v_x, v_y])

    t, u = rk4.calculate(t_0, u_0, h, u_prime, should_exit)

    return t, u

def horizontal_range(v_0):
    """
    Launch a projectile from the origin with the given launch velocity.
    Return the horizontal range of the projectile.
    Approximate the range as the x-intercept of the line connecting the last two
points of the projectile's path.
    """
    t, u = launch(v_0, below_ground)
    x = u[:, 0]
```

```
    y = u[:, 1]

    if x[-2] == x[-1]:
        distance = x[-1]
    else:
        slope = (y[-2] - y[-1]) / (x[-2] - x[-1])
        distance = -y[-1]/slope + x[-1]
    return distance
```

3. Interdependence of Horizontal and Vertical Motion

When modeling projectiles with no drag or linear drag, one property that emerges is the independence of horizontal and vertical motion. This occurs because $\frac{dv_x}{dt}$ does not depend on y or v_y, and similarly $\frac{dv_y}{dt}$ does not depend on x or v_x.

With the quadratic drag model, $\frac{dv_x}{dt}$ depends on v_y.

$$\frac{dv_x}{dt} = -kvv_x = -kv_x\sqrt{v_x^2 + v_y^2} \quad (8)$$

Similarly, $\frac{dv_y}{dt}$ depends on v_x.

$$\frac{dv_y}{dt} = -g - kvv_y = -g - kv_y\sqrt{v_x^2 + v_y^2} \quad (9)$$

Increasing v_x or v_y causes the drag experienced in both the x and y directions to increase. This leads to the interdependence of horizontal and vertical motion.

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

# Plot horizontal position over time as vertical velocity changes
v_0x = 0.5
v_0y_values = np.linspace(0, 1.5, 7)
t_f = 2.0

fig, ax = plt.subplots()
ax.set(ylabel="$x$", xlabel="$t$", title=f"$v_{{0x}}$ = {v_0x:.2f}")

for v_0y in v_0y_values:
    t, u = projectile.launch([v_0x, v_0y], lambda t, u: t >= t_f)
    x = u[:, 0]
    ax.plot(t, x)

fig.legend(v_0y_values, title="$v_{0y}$", loc="center right")
fig.tight_layout()
fig.savefig("media/x_vs_t.svg")


# Plot vertical position over time as horizontal velocity changes
v_0y = 0.5
```

```
v_0x_values = np.linspace(0, 1.5, 7)

fig, ax = plt.subplots()
ax.set(ylabel="$y$", xlabel="$t$", title=f"$v_{{0y}}$ = {v_0y:.2f}")

for v_0x in v_0x_values:
    t, u = projectile.launch([v_0x, v_0y])
    y = u[:, 1]
    ax.plot(t, y)

fig.legend(v_0x_values, title="$v_{0x}$")
fig.tight_layout()
fig.savefig("media/y_vs_t.svg")
```

Figure 2 plots the horizontal position of the projectile over time as the initial vertical velocity varies. The initial horizontal velocity was kept constant. Each launch was kept going for the same amount of time. If x and y motion were independent, then each plot for a different v_{0y} value would be identical. Since the plots vary as v_{0y} changes, this demonstrates the interdependence of x and y motion.

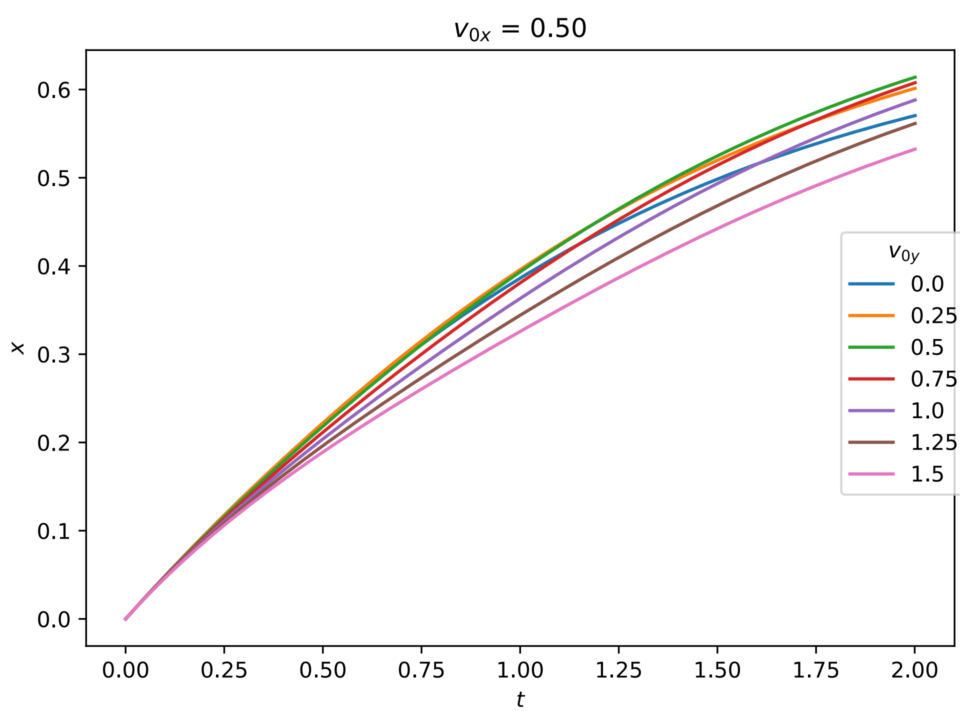


Figure 2: x vs t as v_{0y} Varies

Figure 3 plots the vertical position of the projectile over time as the initial horizontal velocity varies. The initial vertical velocity was kept constant. Each launch was kept going until the projectile hit the ground ($y = 0$). Notice that the max height and time in the air decrease as v_{0x} increases. If x and y motion were independent, then each plot for a different v_{0x} value would be identical. Since the plots vary as v_{0x} changes, this demonstrates the interdependence of x and y motion.

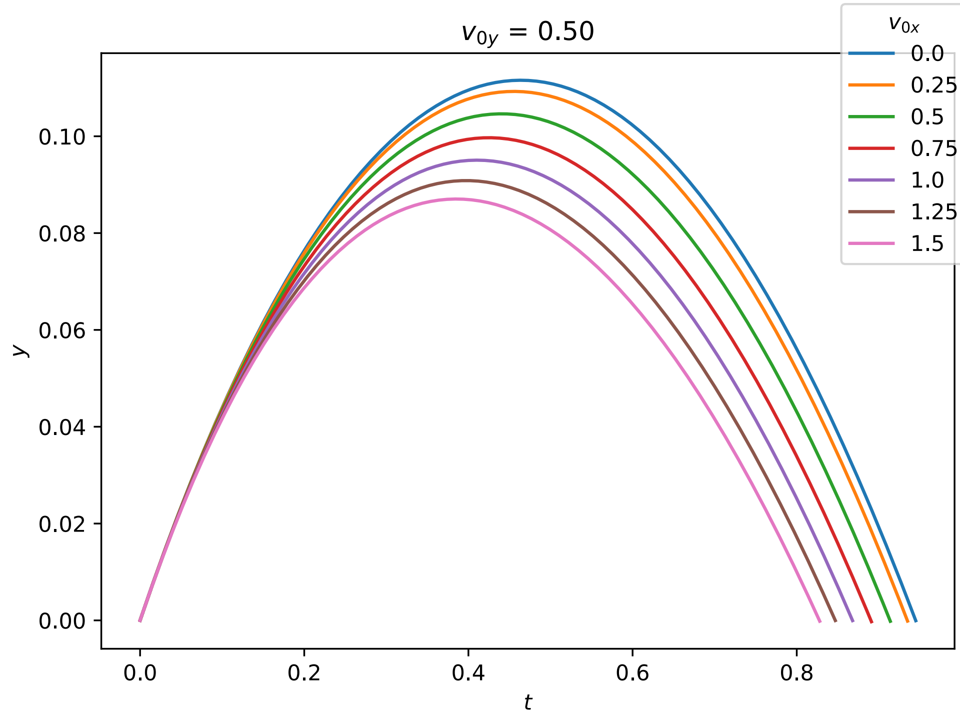


Figure 3: y vs t as v_{0x} Varies

4. Trajectory Shapes

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

# Plot trajectories as launch angle changes
v_0 = 1.5
deg_theta_values = np.linspace(0, 90, 7)

fig, ax = plt.subplots()
ax.set(ylabel="$y$", xlabel="$x$", title=f"$v_0$ = {v_0:.2f}")

for rad_theta in np.radians(deg_theta_values):
    v_x = v_0 * np.cos(rad_theta)
    v_y = v_0 * np.sin(rad_theta)
    t, u = projectile.launch([v_x, v_y])
    x = u[:, 0]
    y = u[:, 1]
    ax.plot(x, y)

fig.legend(deg_theta_values, title="$\\theta$ (°)")
fig.tight_layout()
fig.savefig("media/xy_vs_theta.svg")
```

```
# Plot trajectories as launch speed changes
deg_theta = 45
rad_theta = np.radians(deg_theta)
v_0_values = np.linspace(0, 1.5, 7)

fig, ax = plt.subplots()
ax.set(ylabel="$y$", xlabel="$x$", title=f"$\\theta$ = {deg_theta}°")

for v_0 in v_0_values:
    v_x = v_0 * np.cos(rad_theta)
    v_y = v_0 * np.sin(rad_theta)
    t, u = projectile.launch([v_x, v_y])
    x = u[:, 0]
    y = u[:, 1]
    ax.plot(x, y)

fig.legend(v_0_values, title="$v_0$")
fig.tight_layout()
fig.savefig("media/xy_vs_v.svg")
```

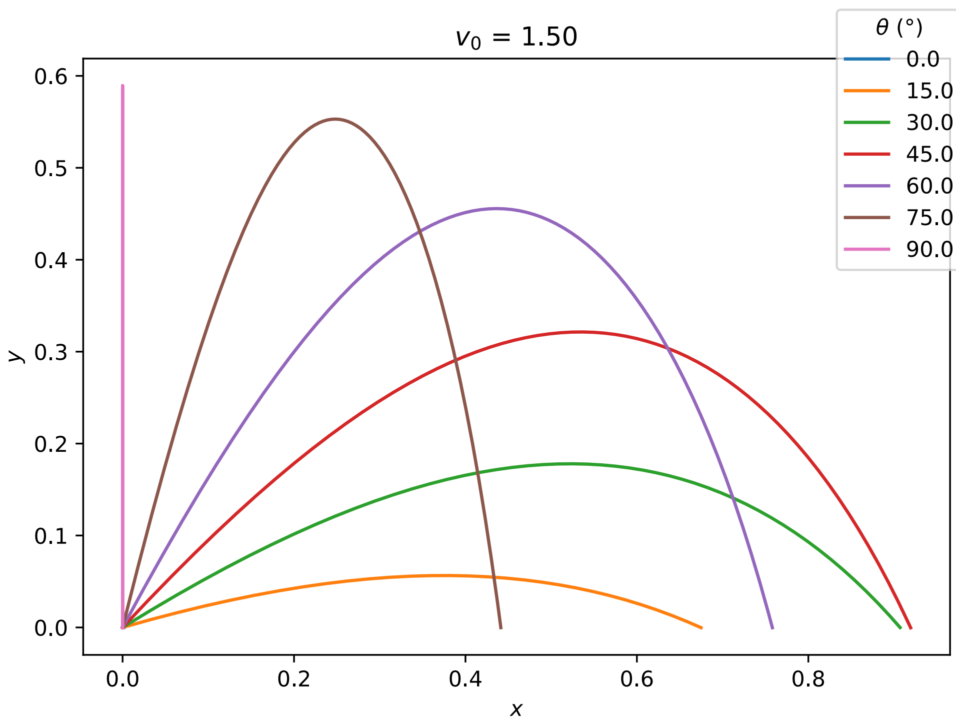


Figure 4: Trajectory as θ Varies

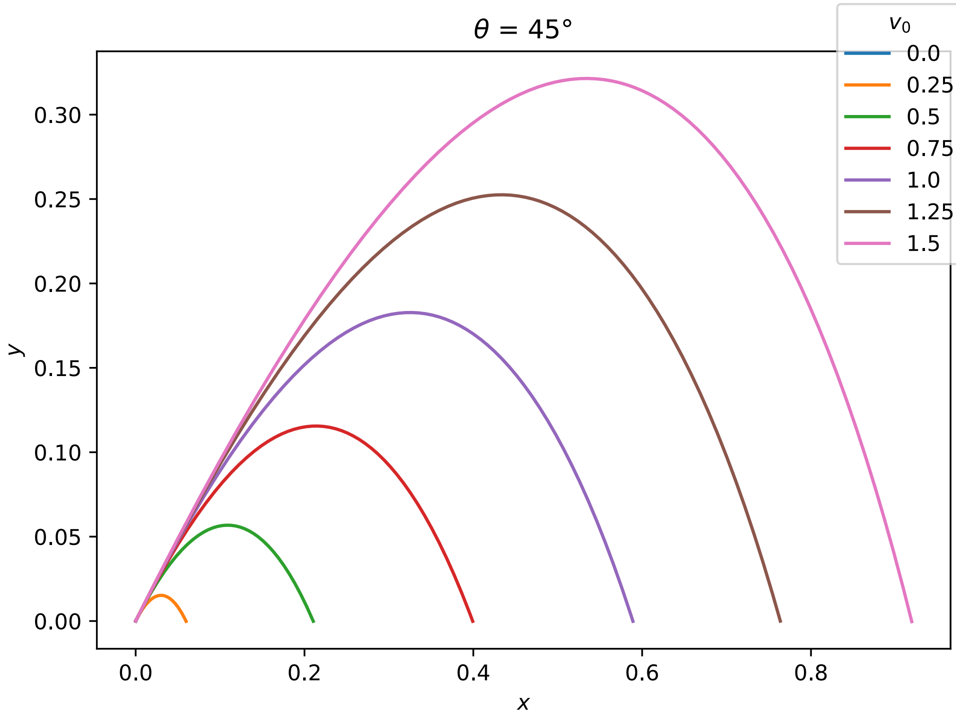


Figure 5: Trajectory as v_0 Varies

5. Firing Range

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

N = 100

# Plot firing range vs launch angle as launch speed changes
v_0_values = np.linspace(0, 2.0, 9)
deg_theta_values = np.linspace(0, 90, N)

fig, ax = plt.subplots()
ax.set(ylabel="$R$", xlabel="$\\theta$ (°)")

for v_0 in v_0_values:
    firing_range_values = []
    for rad_theta in np.radians(deg_theta_values):
        v_x = v_0 * np.cos(rad_theta)
        v_y = v_0 * np.sin(rad_theta)
        firing_range = projectile.horizontal_range([v_x, v_y])
        firing_range_values.append(firing_range)
    ax.plot(deg_theta_values, firing_range_values)

fig.legend(v_0_values, title="$v_0$")
fig.tight_layout()
```

```
fig.savefig("media/R_vs_theta.svg")


# Plot firing range vs launch speed as launch angle changes
deg_theta_values = np.linspace(0, 90, 7)
v_0_values = np.linspace(0, 4, N)

fig, ax = plt.subplots()
ax.set(ylabel="$R$", xlabel="$v_0$")

for rad_theta in np.radians(deg_theta_values):
    firing_range_values = []
    for v_0 in v_0_values:
        v_x = v_0 * np.cos(rad_theta)
        v_y = v_0 * np.sin(rad_theta)
        firing_range = projectile.horizontal_range([v_x, v_y])
        firing_range_values.append(firing_range)
    ax.plot(v_0_values, firing_range_values)

fig.legend(deg_theta_values, title="$\\theta$ (°)", ncols=2, loc="upper center")
fig.tight_layout()
fig.savefig("media/R_vs_v.svg")
```

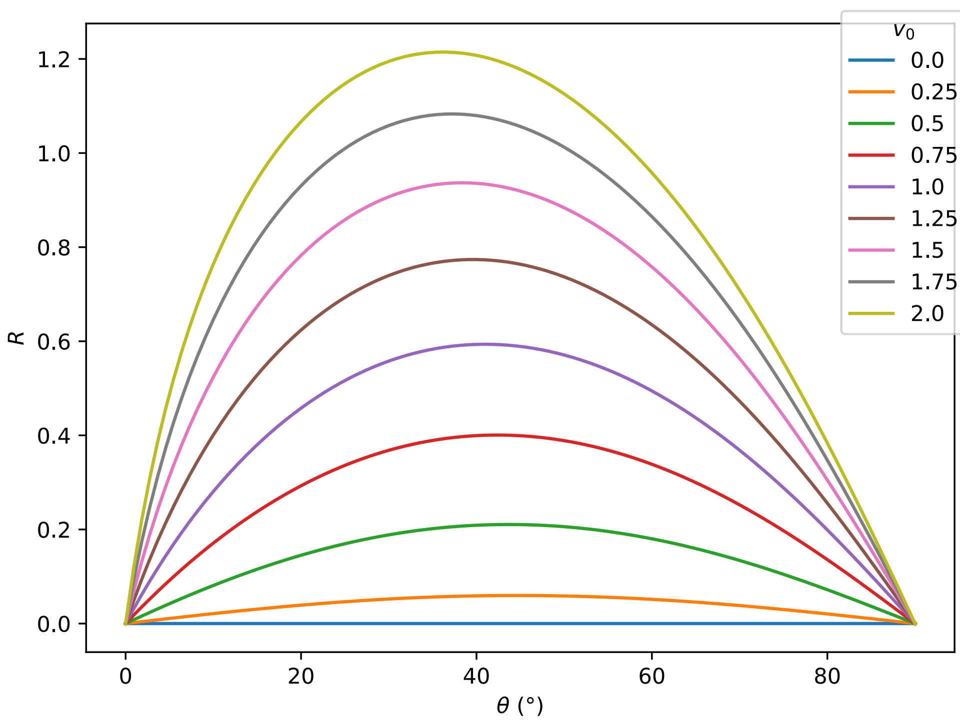


Figure 6: R vs θ as v_0 Varies

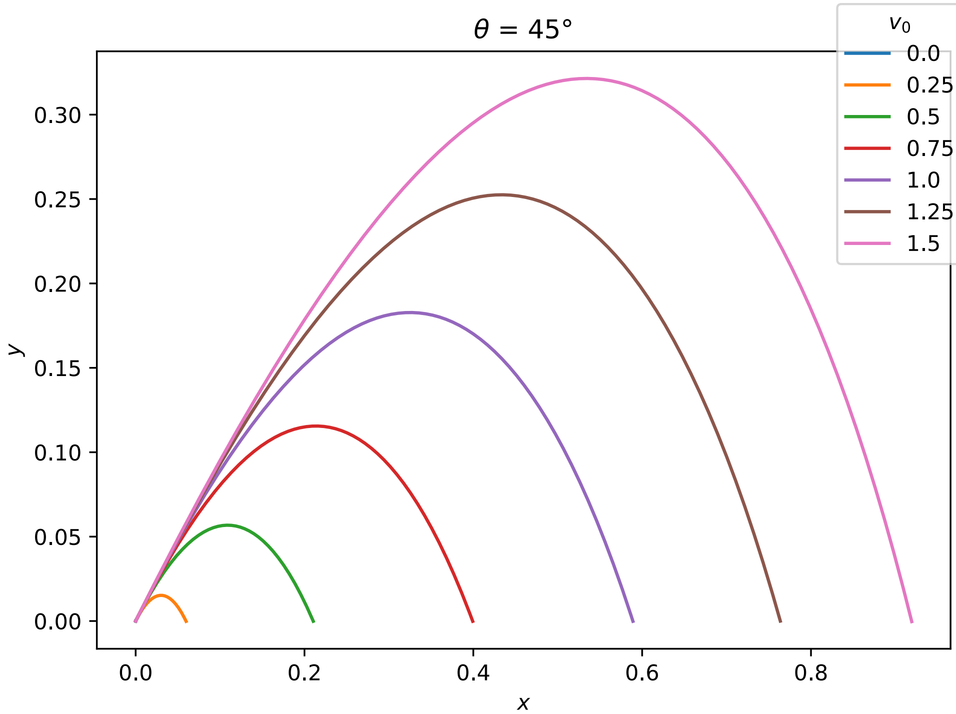


Figure 7: R vs v_0 as θ Varies

6. Hitting a Fixed Target

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

def distance_from_target(v_0, target_pos):
    """
    Launch a projectile from the origin with the given launch velocity.
    Stop simulating when the projectile falls below the line of sight to the target.
    Return position of the projectile relative to the target when it crossed the line
of sight.
    Approximate where the projectile crossed the line of sight as the intersection
between that line and the line connecting the last two points of the projectile's
path.

    If the target is straight up, then return the distance of the projectile from the
target when it was at its peak.
    If the target is straight down, then return the distance of the projectile from
the target when it fell below the target.
    """
    target_x, target_y = target_pos

    # If the target is straight up, exit when the projectile turns around
    # If the target is straight down, exit when the projectile falls below the target
    if target_x == 0:
```

```
        if target_y > 0:
            exit_condition = lambda t, u: u[3] < 0
        else:
            exit_condition = lambda t, u: u[1] < target_y

    # Otherwise, exit when the projectile falls below the line of sight
    else:
        target_slope = target_y / target_x
        exit_condition = lambda t, u: u[1] < target_slope * u[0]

    t, u = projectile.launch(v_0, exit_condition)
    x = u[:, 0]
    y = u[:, 1]

    if target_x == 0:
        if target_y > 0:
            distance_y = y[-1] - target_y
            distance = np.sign(distance_y) * np.sqrt((distance_y)**2 + (x[-1])**2)
        else:
            if x[-2] == x[-1]:
                distance = x[-1]
            else:
                projectile_slope = (y[-2] - y[-1]) / (x[-2] - x[-1])
                distance = (target_y - y[-1])/projectile_slope + x[-1]
    else:
        target_slope = target_y / target_x
        if x[-2] == x[-1]:
            intersection_x = x[-1]
        else:
            projectile_slope = (y[-2] - y[-1]) / (x[-2] - x[-1])
            intersection_x = (y[-1] - projectile_slope*x[-1]) / (target_slope -
projectile_slope)
        intersection_y = target_slope * intersection_x
        target_r = np.sqrt(target_x**2 + target_y**2)
        intersection_r = np.sqrt(intersection_x**2 + intersection_y**2)
        distance = intersection_r - target_r

    return distance

def bisection(a, b, f, atol=1e-8):
    """
    Use the bisection method to find a root of the given function on the interval [a,
b].
    The root returned will have the given absolute tolerance.
    f(a) and f(b) must have opposite signs.
    """
    f_a = f(a)
    f_b = f(b)
    assert max(f_a, f_b) > 0 and min(f_a, f_b) < 0, "f(a) and f(b) must have opposite
signs"
```

```
    while True:
        error_bound = (b - a) / 2
        mid = (a + b) / 2
        if error_bound < atol:
            return mid

        f_mid = f(mid)
        if (f_a >= 0 and f_mid >= 0) or (f_a <= 0 and f_mid <= 0):
            a = mid
            f_a = f_mid
        else:
            b = mid
            f_b = mid

def find_launch_speed(rad_launch_theta, target_pos, launch_speed_guess=1.0):
    """
    Find the launch speed required to hit the target at the given position using the
given launch angle.
    Optionally provide a launch speed guess to help pinpoint where the needed launch
speed might be.
    """
    target_x, target_y = target_pos
    min_target_theta = np.atan2(target_y, target_x)
    assert rad_launch_theta >= min_target_theta, "The launch angle must be greater
than the line of sight angle"

    v_hat = np.array([np.cos(rad_launch_theta), np.sin(rad_launch_theta)])
    distance_func = lambda v: distance_from_target(v * v_hat, target_pos)

    # Find an upper and lower bound for the needed launch speed
    d_guess = distance_func(launch_speed_guess)
    if d_guess <= 0:
        launch_speed_low = launch_speed_guess
        dist_low = d_guess
        launch_speed_high = launch_speed_guess
        while True:
            launch_speed_high *= 2
            dist_high = distance_func(launch_speed_high)
            if dist_high >= 0:
                break
            launch_speed_low = launch_speed_high
    else:
        launch_speed_high = launch_speed_guess
        dist_high = d_guess
        launch_speed_low = launch_speed_guess
        while True:
            launch_speed_low /= 2
            dist_low = distance_func(launch_speed_low)
            if dist_low <= 0:
                break
            launch_speed_high = launch_speed_low
```

```
    return bisection(launch_speed_low, launch_speed_high, distance_func)

N = 50
angle_padding = 20
deg_theta_values = np.linspace(0 + angle_padding, 90 - angle_padding, N)

target_pos = (2, 1)
deg_target_theta = np.degrees(np.atan2(target_pos[1], target_pos[0]))
deg_theta_above_target = deg_theta_values[deg_theta_values > deg_target_theta]
rad_theta_above_target = np.radians(deg_theta_above_target)

launch_speed_values = []
launch_speed_guess = 1.0
for rad_theta in rad_theta_above_target:
    launch_speed = find_launch_speed(rad_theta, target_pos,
launch_speed_guess=launch_speed_guess)
    launch_speed_values.append(launch_speed)
    launch_speed_guess = launch_speed

fig, ax = plt.subplots()
ax.set(ylabel="$v_0$", xlabel="$\\theta$ (°)", title=f"Target at (x, y) =
{target_pos}")
ax.plot(deg_theta_above_target, launch_speed_values, ".")
fig.savefig("media/test_hitting_target.svg")
```

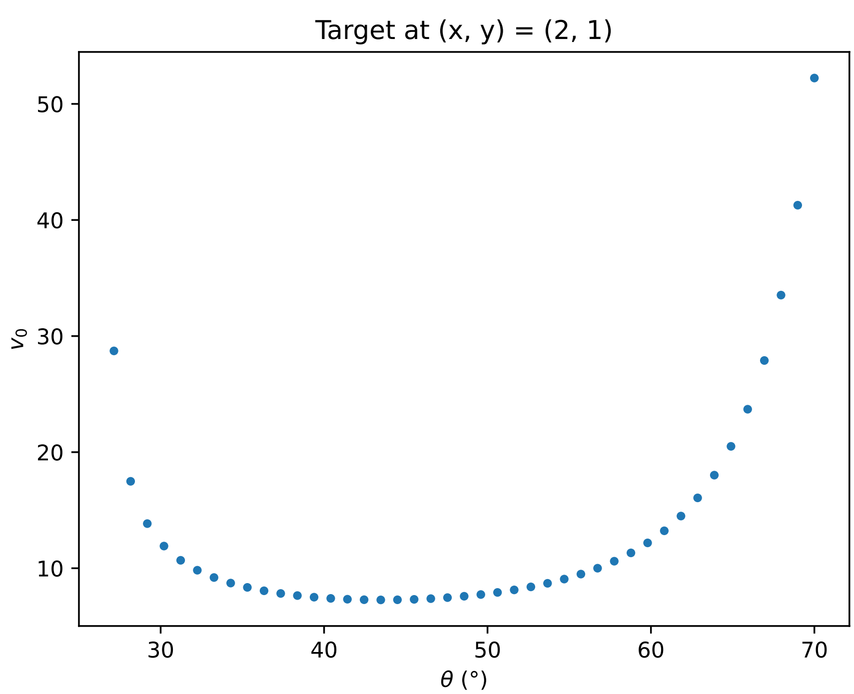


Figure 8: Test Hitting Target

7. Cover Image

The following script was use to create the cover image.

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

g = 1
v_0 = [2.0, 0.75]
v_0x, v_0y = v_0
N = 100

fig, ax = plt.subplots()
ax.set(ylabel="$y$", xlabel="$x$")
ax.tick_params(
    axis="both",
    which="both",
    labelbottom=False,
    bottom=False,
    labelleft=False,
    left=False,
)
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)

# No drag
t_f = 2 * v_0y / g
t = np.linspace(0, t_f, N)
x = v_0x * t
y = v_0y * t - g * t**2 / 2
ax.plot(x, y, label="No Drag")

# Quadratic drag
t, u = projectile.launch(v_0)
x = u[:, 0]
y = u[:, 1]
ax.plot(x, y, label="Quadratic Drag")

fig.legend()
fig.tight_layout()
fig.savefig("media/thumbnail.svg")
```