

Technical Report

Evan Wills, Logan Mueller, Hayden Mays

1. Introduction

With this problem, we had to place ourselves in the time period of 1805 where we must project trajectories and determine the launch angle and initial speed in order to hit a target in some sort of horizontal distance out. We have to solve this by using numerical methods and determining the conditions for how the trajectories will be shot. We were given an initial condition that the target must be able to reach 0.5 miles, which is roughly 804.67 meters. To do this, we used different numerical methods to verify answers such as:

- Euler's Method
- 4th order Runge Kutta (RK4)
- A bisection root-finding (to find v_0 and θ_0)
- A velocity search algorithm (solve_best function)
- Trajectory functions (euler_vec, z_zero, find)

2. Differential Equations Used

Within our code, we solve it based upon the equations $dx/dt = v_x$, $dz/dt = v_z$, $dv_x/dt = -cv_x\sqrt{v_x^2 + v_z^2}$, and $dv_z/dt = -g - cv_z\sqrt{v_x^2 + v_z^2}$. In these equations, $g = 9.81 \text{ m/s}^2$ and $c = 10^{-3} \text{ m}^{-1}$ as the drag constant. Because the drag force is related to the velocity squared, this is a non-linear system, and we must use numerical methods.

3. Numerical Methods Used

3.1 (Euler's Method)

```
def Euler(t_0:float,t_f:float,h:float,S_initial:float,dSdt:callable) -> tuple [np.ndarray:'S values',np.ndarray]:
    ''' Runs Euler method

    Args:
        t_0: the initial time
        t_f: the final time
        h: the step length between the initial and final time
        S_initial: the initial condition
        dSdt: the ODE for S

    Returns:
        tuple of 2 arrays
        (t_span,S)

    ...
    t_span=np.arange(t_0,t_f+h,h) # creates span

    S=np.zeros((len(t_span),4)) # Creates array for outputs
    S[0]=S_initial #Sets initial value

    # Runs euler approximation
    for i in range(len(t_span)-1):
        S[i+1]=S[i]+h*dSdt(t_span[i],S[i])

    # Returns t_span and the outputs
    return t_span,S
```

The Euler's Method we have within our code was used to have fast root-finding, give us trajectory arrays, and show us the landing location. Within our code, we used dt values of 0.01-0.0001, trying to test out different accuracies.

3.2 4th order Runge Kutta (RK4) Method

```
def RK4(y,systemcallable,h,t):
    k1=system(t,y)
    k2=system(t+h/2,y+(k1/2)*h)
    k3=system(t+h/2,y+(k2/2)*h)
    k4=system(t+h,y+(k3)*h)

        # Update y value using the weighted average of the slopes
    y=y+(1/6)*(k1+2*k2+2*k3+k4)*h
    return y
```

The 4th order Runge Kutta (RK4) method we did was implemented manually. RK4 was mainly used in the z_zero function to accurately track where the projectile would land on the ground.

4. Determining the Range Within Our Code

```
while z>0.0:
    sol=RK4(sol,system,t_step,t)
    t+=t_step
    z=sol[1]
if sol[0]<target:
    return None
return t
```

Our code ran the simulation until the height crossed zero. At the end of this code, the sol[0] output gives us our horizontal range.

5. Finding V₀ and θ₀

5.1 Bisection Solver for Velocity

```
# Bisection Method
def bisection(func, a, b, tol=1e-3, max_iter=100):
    fa = func(a)
    fb = func(b)

    if fa * fb > 0:
        raise ValueError("Bisection error: interval does not bracket root")

    for _ in range(max_iter):
        m = 0.5 * (a + b)
        fm = func(m)

        if abs(fm) < tol:
            return m

        if fa * fm < 0:
            b, fb = m, fm
        else:
            a, fa = m, fm

    return 0.5 * (a + b)
```

To solve for v₀, we used the following code above. The function we then passed into the bisection was:

```
def f_v0(v0):
    return compute_range(v0, fixed_theta) - target_range
```

This essentially verified that our $R(v_0, \theta_0) - 804.67$ (distance in meters) = 0.

5.2 Bisection Solver for Angle

After had checked for the velocity, this is when we ran the second bisection in our code:

```
def f_theta(theta):
    return compute_range(v0_solution, theta) - target_range
```

5.3 Velocity Search (Using best_solve algorithm)

```
def Solve_Best(target_x, angles=np.arange(0, 80, 5)):
    best = None

    for theta in angles:
        _,v0,tf = solve_v0(target_x,0,0.01,theta,0,500)
        if v0 != None and (best is None or v0 < best[1]):
            best = (theta, v0, tf)
```

What this essentially did was it searched for the smallest velocity to allow the projectile to reach the target distance. This produced the outputs of:

```
Best Tested Values:
Angle:40.00
Velocity:128.19
Initail:
X: 0.00 m | Z: 0.00 m | V:128.19 m/s

Finial:
X:804.53 m | Z: -0.57 m | V: 68.72 m/s
```

This was used to confirm the results of the bisection solver.

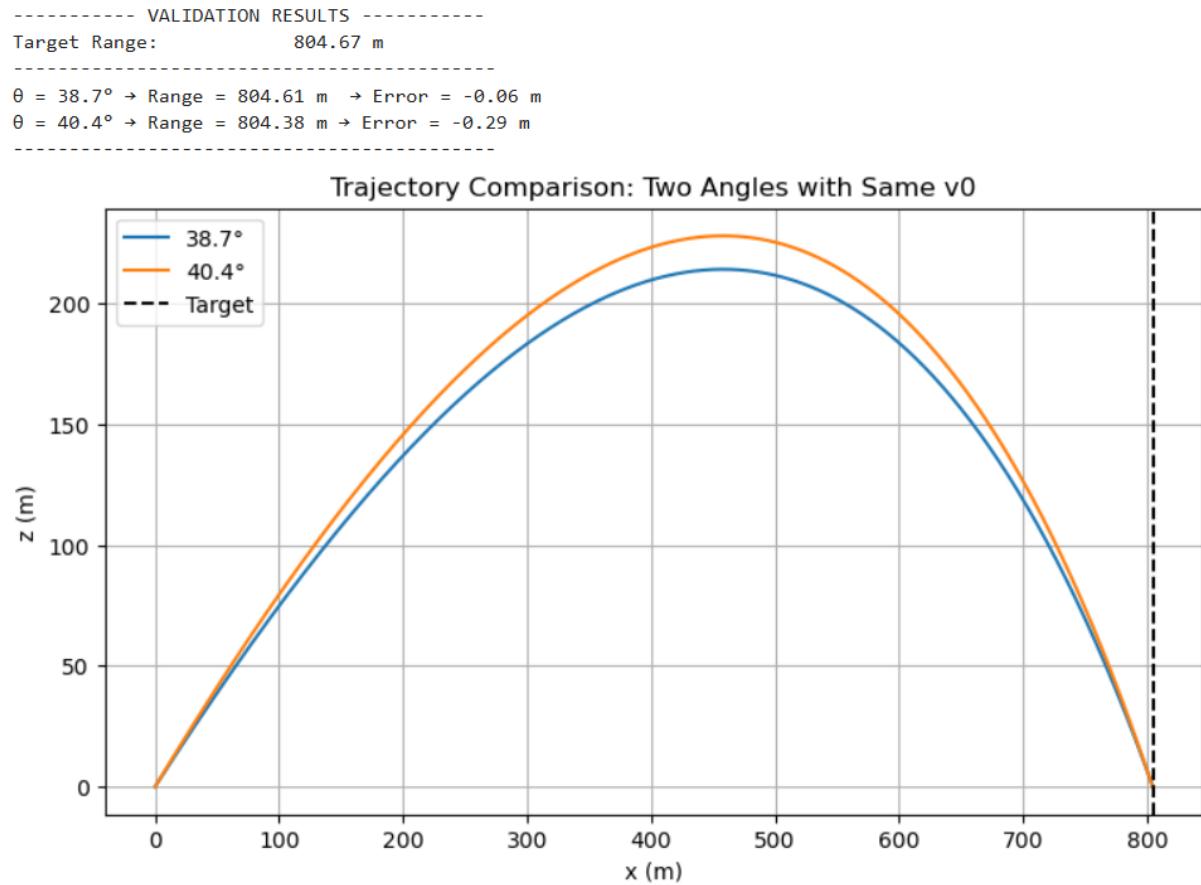
6. Results for Part A

Using the findings from all the previous messages, our solutions came together to find very similar results. Our v_0 came out to be roughly 128.2-128.3 m/s, and the θ_0 was roughly 38.7-40 degrees. This essentially came together to show that for the simulated range of 804.65m, it had a peak height of 214m, and a flight time of roughly 13.1 seconds

based on those conditions. This was within centimeters of the actual target which verifies the accuracy even more.

7. Different Angles Giving Same Result

Our code also outputs two valid angles for the velocity. 38.7 degrees hits at 804.61m and 40.4 degrees hits at 804.38m. Both had less than a 0.3m error, which verifies they are both valid. This idea was best shown in the graph below.



8. Relating our Results to the Drag Coefficient

Based on how our code works, if we lower our drag coefficient, the optimal angle would be closer to 45 degrees and require less of a velocity decrease. If the drag coefficient is increased, the required velocity would increase, the optimal angle would shift more downward, and if it gets too high, the target is unreachable.

9. Accounting for Headwind (Part C)

```

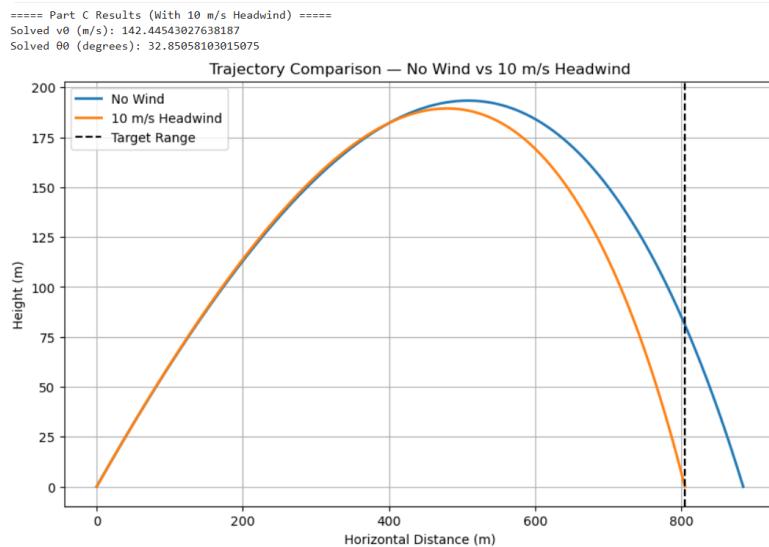
# ODE WITH 10 m/s Headwind
def projectile_ode_wind(t, y):
    x, z, vx, vz = y

    # relative velocity including wind
    vrel_x = vx - wind
    vrel_z = vz
    vrel = np.sqrt(vrel_x**2 + vrel_z**2)

    dvx = -c * vrel_x * vrel
    dvz = -g - c * vrel_z * vrel
    return np.array([vx, vz, dvx, dvz])

```

The solver ended up producing a v_0 of 142.45m/s and a θ_0 of 32.85 degrees. This makes sense because a headwind would increase drag, shorten the range, require an increased velocity, and shift the optimal angle downward. These results were best shown and matched our graph produced as well.



10. Extension: Firing from Different Height

To extend the model beyond what it was, we looked at how the solution would change if the cannon were elevated/shot from a different height. We selected an initial height of $z_0 = 804.5\text{m}$. This matches the horizontal target distance so the horizontal and vertical distances from the target match.

10.1 Modified Solver

While the original solver assumed $x(0) = 0$ and $z(0) = 0$, so for the new solver we modified the vector inside the solver functions.

```
def z_zero(z0,theta,v,t_initial,target,t_step):
    vx=v*np.cos(np.deg2rad(theta))
    vz=v*np.sin(np.deg2rad(theta))
    sol=np.array([0,z0,vx,vz])
    t=t_initial
    z=1

    while z>0.0:
        sol=RK4(sol,system,t_step,t)
        t+=t_step
        z=sol[1]
    if sol[0]<target:
        return None
    return t
```

This code keeps the same drag model as it was established before. We then reused the velocity-search algorithm and called in the z_0 and best_alt variables:

```
def Solve_Best(target_x,z0=0, angles=np.arange(0, 80 , 5)):
    best = None

    for theta in angles:
        _,v0,tf = solve_v0(target_x,0,0.01,theta,0,500,z0)
        if v0 != None and (best is None or v0 < best[1]):
            best = (theta, v0,tf)

    return best

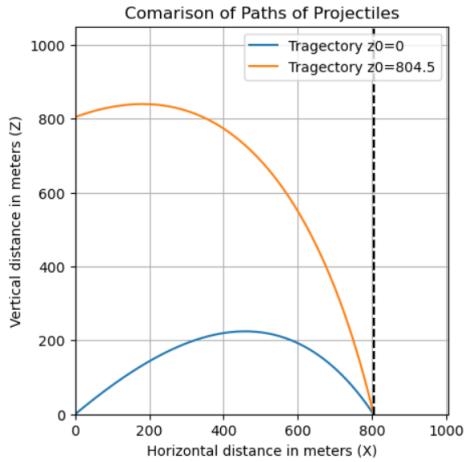
target_alt=804.5
z0=804.5
best_alt=Solve_Best(target_alt,z0)
print(f"Best Tested Values:\nAngle:{best_alt[0]:5.2f}\nVelocity:{best_alt[1]:6.2f}")
```

10.2 Results

The altitude solver produced the results of v_0 of 82.04m/s and θ_0 of 20 degrees. Making note, this is much less than the ground level solution we had found v_0 of 128.3m/s and θ_0 of 39 degrees.

10.3 Interpreting the Results/Graphical Verification

Upon running this simulation, it verified/showed us new findings. If the height is increased, the velocity decreases as gravity assists in the forward motion. The optimal angle also decreases as the height increases as an increased angle uses more energy sending it higher and because it is already higher, we can leverage the height do not have such a high angle. Additionally, as it is fired, it has a much more descending arc which means it will travel further from that alone. These ideas are best represented in the graph below.



From this graph, both landed very close to the 804.5 meters, confirming that the solver was valid. The higher starting altitude shot has a much longer descending arc with less of an initial angle, whereas the ground level does not have as long of an arc but has an increased launch angle. These results are all put together in the table shown below.

Scenario	V_0 (m/s)	θ_0 (degrees)	Behavior
Ground level ($z_0 = 0\text{m}$)	128.3	39	Standard arc
High altitude ($z_0 = 804.5\text{m}$)	82.04	20	Long descending trajectory
Headwind	142.4	32.9	Shorter range, must aim lower

11. Conclusion

Overall, the Python simulation using numerical methods simulated the artillery trajectories under gravity and drag using Euler and RK4 was very successful. Under normal circumstances, we found that to hit the target at 0.5 miles away, a v_0 of 128.3 m/s and θ_0 of 39 degrees was the most optimal. With drag being accounted for, a 10m/s headwind required a higher velocity and decreased angle. This was shown by an output of v_0 being 142.4m/s and a θ_0 of 32.9. A higher altitude launch reduced the launch angle and required velocity. This was shown through the output results of v_0 as 82.04m/s and a θ_0 of 20 degrees. Overall, the graphs matched our results and were consistent with the equations we received as well.