Dynamics Systems Lab: Modeling Car Coasting

Jordan Hardy and

Introduction

When a car coasts without engine power, its acceleration is determined only by environmental factors, such as gravity, rolling friction, and aerodynamic drag. These forces act against the vehicle's motion and cause it to decelerate over time. By analyzing this deceleration, it is possible to infer key parameters that characterize the vehicle's interaction with its environment. Given experimental data taken from a 2016 Aston Martin DB9 rolling up and down a relatively steady incline, we determined the values of the coefficients of kinetic friction and drag. These parameters are important for understanding how external forces influence vehicle deceleration.

Methods

To analyze the motion of a car coasting both uphill and downhill on an inclined plane, we began by constructing free-body diagrams for each scenario. These diagrams (included in the Appendix) account for the force of gravity, the normal force, aerodynamic drag, and rolling friction. Using the free-body and corresponding kinetic diagrams, we derived the equations of motion by applying Newton's second law in the direction of motion. The resulting second-order differential equation is:

$$\ddot{x} = \frac{1}{2m}(C_d A_p \rho_f)\,\dot{x}^2 - \mu_k g \cos(\theta) \pm g \sin(\theta)$$

The plus-minus term accounts for the direction of motion, where uphill uses the minus sign and downhill uses the plus sign.

To calculate the aerodynamic drag force, we modeled the car's frontal area as a rectangle using the provided dimensions. With this, and the equation above, we obtained a system with two unknown parameters: the coefficient of drag C_d and the coefficient of friction μ_k.

We then converted each second-order equation into a system of two first-order ordinary differential equations, suitable for numerical integration. Using Python and the scipy.integrate.odeint

solver, we numerically solved the system for both the uphill and downhill coasting scenarios. We overlaid the resulting velocity-time curves on top of the experimental data, shown in Figure 1, and manually adjusted the values of C_d and μ_k to minimize the difference between our model and the experimental data.

From here, we calculated the power required to maintain constant speeds of 55 mph and 100 mph on a flat surface. This was done using the following power equation, with velocities that had been converted to meters per second:

$$P = v\,(F_{drag} + F_{friction}) = \frac{1}{2}C_d A_p \rho_f v^3 + \mu mgv$$

The computed power values (in watts) were then converted to horsepower for reporting.

Results

The best-fit parameters that matched the experimental data were $C_d = 0.3283$ and $\mu_k = 0.0170$. Using these values, the modeled curves closely aligned with the measured data, as shown in Figure 1. The power calculated using these values that is required to drive 55 mph is 18.66 hp, whereas the calculated power required to go 100 mph is 60.36 hp.

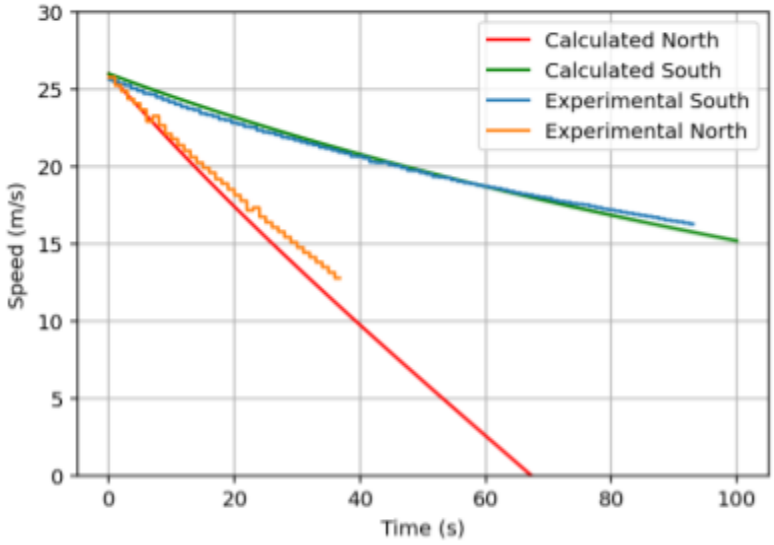


Figure 1. A graph showing the velocity vs time of the car coasting up and down the slope, including both experimental car data and calculated car data.

Discussion

Our fitted values of produced simulated velocity curves that closely match the experimental data for both uphill and downhill coast-down tests. While the agreement is strong, it is important to recognize

that these parameters are not exact and depend on a variety of modeling assumptions and experimental uncertainties.

To quantify uncertainty in our fitted parameters, we introduced a ±5% variation in the final velocity and used a root-finding method to identify how much each parameter could vary while still producing a statistically distinguishable result. Specifically, we fixed one parameter and adjusted the other until the resulting percent error increased by 5%. This analysis yielded confidence intervals for both C_d and μ_k in both uphill and downhill cases, shown in Table 1. The results showed a smaller confidence interval in the downhill (southbound) run for the coefficient of drag, indicating greater sensitivity and thus higher certainty in that direction's model.

Table 1. Confidence intervals for the coefficients of drag and friction for a coasting car, based on the direction in which it travels

Direction	Coefficient of Drag (C_d)	Coefficient of Friction(μ_k)
North (Up)	0.3283 ± 0.0907	0.0170 ± 0.00145
South (Down)	0.3283 ± 0.0443	0.0170 ± 0.0019

The coefficient of drag C_d primarily influences vehicle deceleration at higher speeds due to its quadratic relationship with velocity, making it more dominant during the early phase of coast-down. In contrast, the rolling friction coefficient μ_k exerts a near-constant effect and is more significant at lower speeds. Because of this, the effects of C_d and μ_k can be distinguished by examining how quickly velocity drops early in the coast versus how it behaves near the end. However, there is still some flexibility — it is possible to slightly increase one parameter while decreasing the other and still achieve a visually similar fit. This limits the precision with which either parameter can be determined individually.

The uncertainty in our model comes from several factors, including inaccuracies in the way we modeled the surface area of the car, small errors in the assumed incline angle, and measurement noise in

the GPS-derived velocity data. An overestimate of our area, A_p, would result in an underestimate of C_d, and vice versa. Similarly, the incline angle directly affects the gravitational component of acceleration, so an incorrect angle could lead to a misestimation of the frictional force. Repeating the test under slightly different environmental conditions, such as changes in road surface texture or the presence of wind, could yield slightly different results. However, our confidence intervals suggest that the model is reasonably reliable.

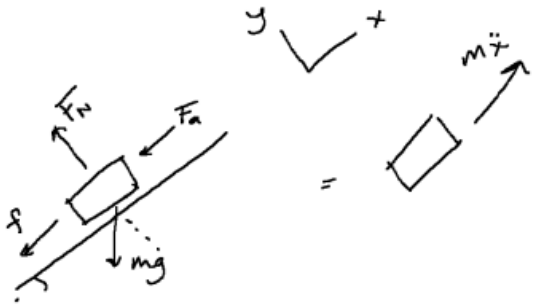
These uncertainties also propagate into our power calculations. Since the power required to maintain a steady velocity is directly dependent on both C_d and μ_k, any variation in these inputs will affect the result. At lower speeds (e.g., 55 mph), power is more sensitive to rolling resistance, while at higher speeds (e.g., 100 mph), aerodynamic drag dominates. While the results are reasonable and consistent with the expected performance of the vehicle, they should be viewed as approximations rather than precise values.

Appendix

Hand Calculations:

$$F_a = \frac{1}{2} C_d A_p \rho_f v^2$$
$$F_a = \frac{1}{2} C_d A_p \rho_f \dot{x}^2$$

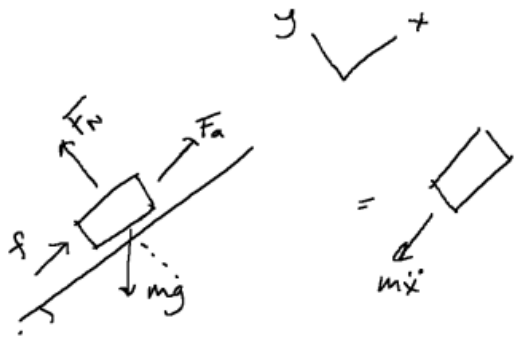
Traveling North



$$\Sigma F_x = m\ddot{x} = -F_a - f - mg\sin\theta$$
$$\Sigma F_y = 0 = F_N - mg\cos\theta$$
$$F_N = mg\cos\theta$$

$$m\ddot{x} + F_a + \mu F_N + mg\sin\theta = 0$$
$$m\ddot{x} + \frac{1}{2} C_d A_p \rho_f \dot{x}^2 + \mu mg\cos\theta + mg\sin\theta = 0$$
$$\ddot{x} = -g(\sin\theta + \mu\cos\theta) - \frac{1}{2m} C_d A_p \rho_f \dot{x}^2$$

Travelling South



$$-m\ddot{x} = mg\mu\cos\theta + \frac{1}{2} C_d A_p \rho_f \dot{x}^2 - mg\sin\theta$$
$$\ddot{x} = -g\mu\cos\theta - \frac{1}{2m} C_d A_p \rho_f \dot{x}^2 + g\sin\theta$$

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from scipy.optimize import root_scalar
import scipy

#----------------------------------------------------Given Data-----------------------------------------------------#

# Load the .mat file
data = scipy.io.loadmat('AustinMartinData')
# Contains: t1 v1 dR1 elev1 p1 theta_deg1 t2 v2 dR2 elev2 p2 theta_deg2 note units_note

# Extract variables from the loaded data
t1 = data['t1'].flatten()  # flatten to convert 2D arrays to 1D
v1 = data['v1'].flatten()
dR1 = data['dR1'].flatten()
elev1 = data['elev1'].flatten()
p1 = data['p1'].flatten()
theta_deg1 = data['theta_deg1'].flatten()

t2 = data['t2'].flatten()
v2 = data['v2'].flatten()
dR2 = data['dR2'].flatten()
elev2 = data['elev2'].flatten()
p2 = data['p2'].flatten()
theta_deg2 = data['theta_deg2'].flatten()

# Optional: Display any notes from the .mat file
# if 'units_note' in data:
#     print(data['units_note'])

# Plot speed versus time
plt.figure(1)
plt.plot(t1, v1, label='Experimental South')
plt.plot(t2, v2, label='Experimental North')

plt.grid(True)
plt.xlabel('Time (s)')
plt.ylabel('Speed (m/s)')
plt.legend()

# Plot position vs elevation to see road profile
plt.figure(2)
plt.plot(dR1, elev1, label='Experimental South')
plt.plot(dR2, elev2, label='Experimental North')
plt.grid(True)
plt.xlabel('Distance (m)')
plt.ylabel('Elevation (m)')

# Add line showing slope of each
plt.plot(dR1, min(elev1) + (dR1 - min(dR1)) * np.tan(theta_deg1 * np.pi / 180), '--', label=f'Slope {theta_deg1[0]:.2f} deg')
plt.plot(dR2, min(elev2) + (dR2 - min(dR2)) * np.tan(theta_deg2 * np.pi / 180), '--', label=f'Slope {theta_deg2[0]:.2f} deg')

plt.legend()
plt.title('Elevation vs. Distance')

#----------------------------------------------------Equations of Motion--------------------------------------------#

#Equations of motion
def eom_up(z, t, Cd, uk):
    '''Passes in a two by one matrix with initial conditions
    (displacement and velocity), as well as a timespan.
    Outputs a two by one matrix with position and velocity'''

    x, x_dot = z

    x_dt = x_dot
```

```
    x_dot_dt = -(0.5*Cd*Ap*rf*x_dot**2)/m - uk*g*np.cos(theta) - g*np.sin(theta)

    return [x_dt, x_dot_dt]

def eom_down(z, t, Cd, uk):
    '''Passes in a two by one matrix with initial conditions
    (displacement and velocity), as well as a timespan.
    Outputs a two by one matrix with position and velocity'''

    x, x_dot = z

    x_dt = x_dot
    x_dot_dt = -(0.5*Cd*Ap*rf*x_dot**2)/m - uk*g*np.cos(theta) + g*np.sin(theta)

    return [x_dt, x_dot_dt]

#Parameters
m = 1760 + 150 #kg
g = 9.81 #m/s^2
Ap = 2.387092 #m^2
rf = 1.06 #kg/m^3
theta = np.deg2rad(0.91) #radians

#Initial conditions
distance_0 = 0 #m
velocity_0 = 26 #m/s
init_state = [distance_0, velocity_0]

#-------------------------------------------------Creating Comparative Graphs-------------------------------------------------#

#Solve for both directions and seperate the dispalcement and velocity
#The 'args' function in odeint is Cd first and uk second, args = (Cd, uk)
solution_up = odeint(eom_up, init_state, t2, args = (0.3283, 0.0170))
distance_up = solution_up[:,0]
velocity_up = solution_up[:,1]

solution_down = odeint(eom_down, init_state, t1, args = (0.3283, 0.0170))
distance_down = solution_down[:,0]
velocity_down = solution_down[:,1]

#Plot the velocities
plt.figure(1)
plt.ylim(0,30)
plt.plot(t2, velocity_up, color='red', label = 'Calculated North')
plt.plot(t1, velocity_down, color='green', label = 'Calculated South')
plt.xlabel('Time (s)')
plt.ylabel('Velocity (m/s)')
plt.legend()


#-------------------------------------------------Finding Percent Error Equations-------------------------------------------------#

def perc_error(real, calc):
    '''Find percent error of two numbers given'''
    error = abs( (calc-real)/real ) * 100
    return error

v1f_exp = v1[-1] #Real values of the final velocity
v2f_exp = v2[-1]

#Equations for up error
def eom_up_err(Cd, uk):
    '''Takes in a coeffiecient a coeffiecient of friction of drag and finds the percent error'''
    solution_up = odeint(eom_up, init_state, t2, args= (Cd, uk))
    velocity_up = solution_up[:,1]
    v2f_calc = velocity_up[-1]

    error = perc_error(v2f_exp, v2f_calc)
    return error
```

```
def find_lowu_error(Cd, uk):
    '''Subtracts an error value from a given error you are looking for'''
    p = 11.367 + 5 #Percent error to try and calculate
    error = eom_up_err(Cd, uk)
    return (p - error)

#Equations for down error
def eom_down_err(Cd, uk):
    '''Takes in a coeffiecient a coeffiecient of friction of drag and finds the percent error'''
    solution_down = odeint(eom_down, init_state, t1, args= (Cd, uk))
    velocity_down = solution_down[:,1]
    v1f_calc = velocity_down[-1]

    error = perc_error(v1f_exp, v1f_calc)
    return error

def find_lowd_error(Cd, uk):
    '''Subtracts an error value from a given error you are looking for'''
    p = 0.3716720519270135 + 5 #Percent error to try and calculate
    error = eom_down_err(Cd, uk)
    return (error - p)

#-------------------------------------------------Running Percent Error Equations-----------------------------------------------#

#Find a Cd with p% error given a uk, up
sol = root_scalar(lambda Cd: find_lowu_error(Cd, 0.0170), x0 = 0.4)
print('Cd for a given uk, up:', sol.root)
print('Error, up:', eom_up_err(sol.root, 0.0170))

#Find a Cd with p% error given a uk, down
sol = root_scalar(lambda Cd: find_lowd_error(Cd, 0.0170), x0 = 0.07)
print('\nCd for a given uk, down:', sol.root)
print('Error, down:', eom_down_err(sol.root, 0.0170))

#Find a uk with p% error given a Cd, up
sol = root_scalar(lambda uk: find_lowu_error(0.3283, uk), x0=0.4)
print('\nuk for a given Cd, down:', sol.root)
print('Error, down:', eom_up_err(0.3283, sol.root))

#Find a uk with p% error given a Cd, down
sol = root_scalar(lambda uk: find_lowd_error(0.3283, uk), x0 = 0.04)
print('\nuk for a given Cd:', sol.root)
print('Error, up:', eom_down_err(0.3283, sol.root))

#-------------------------------------------------Finding Power-----------------------------------------------------------------#

#P = F * v
vel1 = 24.5872 #m/s, equivalent to 55 mph
vel2 = 44.704 #m/s, equivalent to 100 mph

Cd_guess = 0.3283
uk_guess = 0.0170

def find_power(v):
    '''Finds the horsepower necessary to go a given velocity (m/s)'''
    F =  (0.5*Cd_guess*Ap*rf*v**2) + (uk_guess*m*g) #Adds the drag force and the friction force on the axles
    p_watts = F*v #Converts to watts by multiplying by velocity
    p_hp = p_watts / 745.7 #Converts to horsepower
    return p_hp

print(f"\nThe horsepower needed to go 55 mph is {find_power(vel1)} hp")
print(f"The horsepower needed to go 100 mph is {find_power(vel2)} hp")
```