FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Project MULE

Moon Utility Loader for Exploration

Space Exploration Project Winter Term 2023/2024

FH Aachen
March X, 2024

Team Members

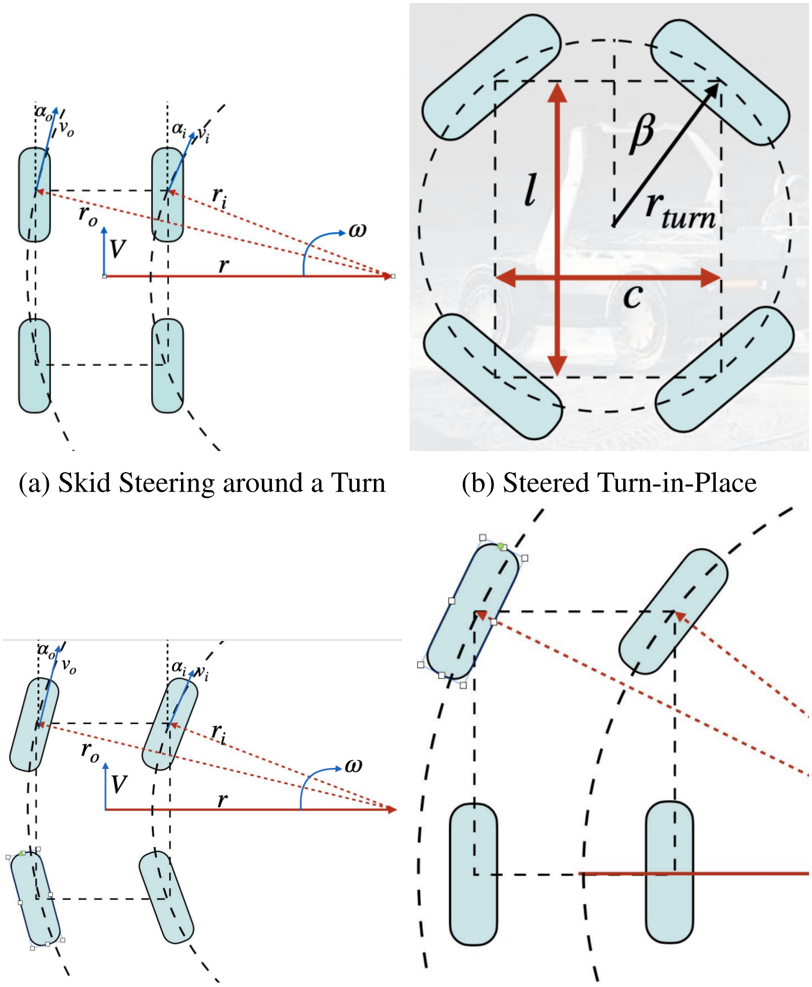
Sebastian Bartel	sebastian.bartel@alumni.fh-aachen.de
Nils Berndt	nils.berndt@alumni.fh-aachen.de
Raffaele Curia	raffaele.curia@alumni.fh-aachen.de
Lukas Germund	lukas.germund@alumni.fh-aachen.de
Michael Hermann	michael.hermann@alumni.fh-aachen.de
Benedikt Kampmann	benedikt.kampmann@alumni.fh-aachen.de
Adrian Klein	adrian.klein@alumni.fh-aachen.de
Hanna Kloy	hanna.kloy@alumni.fh-aachen.de
David Krieger	david-rene.krieger@alumni.fh-aachen.de
Alwin Meyer	alwin.meyer@alumni.fh-aachen.de
Jan Meyne	jan.meyne@alumni.fh-aachen.de
Pascal Müllender	pascal.muellender@alumni.fh-aachen.de
Sven Nebendahl	sven.nebendahl@alumni.fh-aachen.de
Clara Offenhäusser	clara.offenhaeusser@alumni.fh-aachen.de
Marius Ronshausen	marius.ronshausen@alumni.fh-aachen.de
André Röthling	andre.roethling@alumni.fh-aachen.de
Jonas Schumacher	jonas-willi.schumacher@alumni.fh-aachen.de
Alexander Stumpf	alexander.stumpf1@alumni.fh-aachen.de
Maximilian Sturm	maximilian.sturm@alumni.fh-aachen.de
Hannah Walther	hannah.walther@alumni.fh-aachen.de
Dennis Westermann	dennis.westermann@alumni.fh-aachen.de
Daniel Zinken	daniel.zinken@alumni.fh-aachen.de

Index

List of Figures

1.7 Steering

Steering is performed using Skid Steering around a Turn as shown in Figure 1.11a.Other steering methods such as a steered Turn-in-Place or Double- or Single Ackermann Steering would have required additional actuators and mechanisms, increasing complexity and adding possible failure points.



(a) Skid Steering around a Turn (b) Steered Turn-in-Place

(c) Double Ackermann Steering (d) Single Ackermann Steering

Figure 1.11: Steering Methods, taken from [Akin 2022a]

1.8 Static Stability

$$\theta_{static,pitchover} = \tan^{-1}\left(\frac{L_{wheelbase}}{2 \cdot h_{CoG}}\right) = \tan^{-1}\left(\frac{4.13\,\mathrm{m}}{2 \cdot 1.04\,\mathrm{m}}\right) = 63^\circ > \theta_{max} = 20^\circ \qquad \checkmark \quad (1.28)$$

$$\theta_{static,rollover} = \tan^{-1}\left(\frac{2.8\,\mathrm{m}}{2 \cdot 1.04\,\mathrm{m}}\right) = 53^\circ > \theta_{max} = 20^\circ \qquad \checkmark \quad (1.29)$$

Bibliography

Bibliography

[Aki22a] Dave Akin. *Steering Forces/Slopes and Static Stability*. https://spacecraft.ssl.umd.edu/academics/788XF22/788XF22L10.steering-slopesx.pdf. Accessed: 2023–01-10. 2022.

[Aki22b] Dave Akin. *Terramechanics 1: Passive Rolling Resistance*. https://spacecraft.ssl.umd.edu/academics/788XF22/788XF22L06.terramechanics1_r1x.pdf. Accessed: 2023–01-06. 2022.

[Aki22c] Dave Akin. *Terramechanics 2: Traction*. https://spacecraft.ssl.umd.edu/academics/788XF22/788XF22L07.terramechanics2x.pdf. Accessed: 2023–01-06. 2022.

[Ber+95] Peter Berkelman et al. *Design of a Day / Night Lunar Rover*. Tech. rep. CMU-RI-TR-95-24. Pittsburgh, PA: Carnegie Mellon University, June 1995.

[BWW08] P. W. Bartlett, David S. Wettergreen, and William Whittaker. "Design of the Scarab Rover for Mobility & Drilling in the Lunar Cold Traps". In: 2008. URL: https://api.semanticscholar.org/CorpusID:9428456.

[Din+11] Liang Ding et al. "Experimental study and analysis on driving wheels' performance for planetary exploration rovers moving in deformable soil". In: *Journal of Terramechanics* 48.1 (2011), pp. 27–45. ISSN: 0022-4898. DOI: https://doi.org/10.1016/j.jterra.2010.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0022489810000601.

[Ell15] Alex Ellery. *Planetary Rovers: Robotic Exploration of the Solar System*. 1st. Springer Publishing Company, Incorporated, 2015. ISBN: 3642032583.

[Gar21] John D. Garrett. "garrettj403/SciencePlots". Version 1.0.9. In: (Sept. 2021). DOI: 10.5281/zenodo.4106649. URL: http://doi.org/10.5281/zenodo.4106649.

[GB17] Nikola Georgiev and Joel Burdick. "Design and analysis of planar rotary springs". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 4777–4784. DOI: 10.1109/IROS.2017.8206352.

[Ula+23] Stephan Ulamec et al. "Science objectives of the MMX rover". In: *Acta Astronautica* 210 (2023), pp. 95–101. ISSN: 0094-5765. DOI: https://doi.org/10.1016/j.actaastro.2023.05.012. URL: https://www.sciencedirect.com/science/article/pii/S0094576523002448.

Terrain Simulation

January 9, 2024

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scienceplots
plt.style.use('science')

#plt.rcParams['figure.dpi'] = 600
#plt.rcParams['savefig.dpi'] = 600

n_wheel = 4 # number of wheels
diameter = 0.8
radius = diameter/2
b_width = 0.35 # wheel width in m
h_grouser = 0.03 # grouser height
n_grouser = 8 # number of grousers
eta = 0.7 # gearbox efficiency
power_constant = 150*eta # constant power for the whole locomotion subsystem
v_rover_constant = 0.05 # km/h
c_b = 170 # N/m^2 coefficient of soil/wheel cohesion
n = 1 # exponent of soil deformation
k_c = 1400 # N/m^2 cohesive modulus of soil deformation
k_phi = 830000  # N/m^3 frictional modulus of soil deformation
K = 0.018  # coefficient of soil slip in m
gamma = 2470 # N/m3 lunar soil density
phi = 33 # deg soil internal friction angle
phi_rad = np.deg2rad(phi) # rad soil internal friction angle
mu = np.tan(phi_rad) # static friction coefficient
N_q = 32.23
N_c = 48.09
N_gamma = 33.27
K_c = 33.37
K_phi = 72.77
K_gamma = 81.93
mass = 1800 # mass of the rover
gravity = 1.62 # gravitational constant of the moon
slope = np.linspace(0, 25, 100)
#slope = 0
```

```python
slip = 0.0353 + 1/(1+67*np.exp(-0.32*slope))
#slip = 1
k = (k_c / b_width) + k_phi
c_f = 0.05 # internal rolling friction coefficient (bearings)

W_vehicle = mass * gravity
W_vehicle_normal = W_vehicle * np.cos(np.deg2rad(slope))
W_wheel = mass * gravity / n_wheel
W_wheel_normal = W_wheel * np.cos(np.deg2rad(slope))

z_sinkage = (3 * W_wheel_normal / ((3 - n) * (k_c + b_width * k_phi) * np.sqrt(diameter))) ** (2/(2*n+1))
pressure = k * z_sinkage ** n

theta_1 = np.arccos(1 - z_sinkage / radius)
theta_m = (0.45 + 0.24 * slip) * theta_1
lever_e = radius * np.sin(theta_1 - theta_m)
lever_l = radius * np.cos(theta_1 - theta_m)
l_contact = (diameter / 2) * np.arccos(1 - (2 * z_sinkage / diameter))
A = b_width * l_contact
alpha = np.arccos(1 - 2 * z_sinkage / diameter) # angle of approach
l_0 = z_sinkage * np.tan(np.pi / 4 - phi_rad / 2) ** 2 # distance of rupture

R_compression = ((b_width * k) / (n+1)) * z_sinkage ** (n+1)
R_bulldozing = (b_width*np.sin(alpha+phi_rad) / (2*np.sin(alpha)*np.cos(phi_rad))) * (2*z_sinkage*c_b*K_c + gamma*K_gamma*z_sinkage**2) + ((gamma*l_0**3)/3) * (np.pi/2 - phi_rad) + (c_b*l_0**2) * (1+np.tan(np.pi/4 + phi_rad/2))
R_gravitational = W_wheel * np.sin(np.deg2rad(slope))
R_rolling = W_wheel_normal * c_f # internal (bearings)

sum_R = 4* R_compression + 2* R_bulldozing + 4* R_gravitational + 4* R_rolling
n_grouser_contact = 2 * np.pi * radius / (n_grouser * l_contact)

sum_H_grouser = 4* (A*c_b*(1+2*h_grouser/b_width)*n_grouser_contact + W_wheel_normal*mu*(1 + (0.64*h_grouser/b_width) * np.arctan(b_width/h_grouser))) * (1-np.exp(-slip * l_contact / K))

W_wheel_safe = A * (c_b*N_c + gamma*z_sinkage*N_q + gamma*b_width*N_gamma/2)
pressure_soil_safe = W_wheel_safe / A
pressure_real = W_wheel / A
pressure_margin = pressure_real / pressure_soil_safe
weight_margin = W_wheel / W_wheel_safe

drawbar_pull = sum_H_grouser - sum_R
torque = (((drawbar_pull * lever_l) + (W_vehicle_normal * lever_e))/n_wheel) + 28
```

```
omega = power_constant / (torque * n_wheel)
rpm = omega * 60/ (2 * np.pi)
v_rover = omega * radius * 3.6 * (1-slip)
power_variable = (v_rover_constant * torque * n_wheel) / (radius * 3.6 *␣
↪(1-slip))

theta_max = np.rad2deg(np.arctan(drawbar_pull / W_vehicle))
```

[ ]:
```
plt.plot(slope, v_rover)
plt.xlabel('Slope $\\theta$ [$^\circ$]')
plt.ylabel('Max Rover velocity [$\\frac{km}{h}$]')
plt.xlim(0, 25)
plt.ylim(0,)
plt.title('Constant Power = {:.0f} W'.format(power_constant/eta))
plt.grid()
plt.savefig("max_rover_velocity_slope.pdf")
plt.show()
```

[ ]:
```
plt.plot(slope, power_variable)
plt.xlabel('Slope $\\theta$ [$^\circ$]')
plt.ylabel('Variable Power [W]')
plt.xlim(0,)
plt.ylim(0,150)
plt.grid()
plt.savefig("power_variable_slope.pdf")
plt.show()
```

[ ]:
```
# Find the index of the maximum torque
max_torque_index = np.argmax(torque)
max_torque_value = torque[max_torque_index]
max_torque_theta = slope[max_torque_index]
plt.plot(max_torque_theta, max_torque_value, 'o', color=(255/255, 120/255, 0/
↪255), markersize=3)
plt.plot(slope, torque)
plt.xlabel('Slope $\\theta$ [$^\circ$]')
plt.ylabel('Torque [Nm]')
plt.xlim(0, 25)
plt.ylim(0,)
plt.grid()
plt.legend([f'Max Wheel Torque: {max_torque_value:.2f} Nm at {max_torque_theta:.
↪2f}$^\circ$'], loc='lower right', fontsize='small')
plt.savefig("max_torque_slope.pdf")
plt.show()
```

[ ]:
```
# Find the index of the maximum torque
max_dp_index = np.argmax(drawbar_pull)
max_dp_value = drawbar_pull[max_dp_index]/n_wheel
```

```
max_dp_slip = slip[max_dp_index]*100
plt.plot(max_dp_slip, max_dp_value, 'o', color=(255/255, 120/255, 0/255),␣
↪markersize=3)
plt.plot(slip*100, drawbar_pull/n_wheel)
plt.xlabel('Slip [\%]')
plt.ylabel('Wheel DP [N]')
plt.grid()
plt.legend([f'Max Wheel DP: {max_dp_value:.2f} N at {max_dp_slip:.2f}$\%$'],␣
↪loc='lower right', fontsize='small')
plt.savefig("max_drawbar_pull_slip.pdf")
plt.show()
```

[ ]:
```
# Find the index of the maximum torque
max_torque_index = np.argmax(torque)
max_torque_value = torque[max_torque_index]
max_torque_slip = slip[max_torque_index]*100
plt.plot(max_torque_slip, max_torque_value, 'o', color=(255/255, 120/255, 0/
↪255), markersize=3)
plt.plot(slip*100, torque)
plt.xlabel('Slip [\%]')
plt.ylabel('Wheel Torque [Nm]')
plt.grid()
plt.ylim(0,)
plt.legend([f'Max Wheel Torque: {max_torque_value:.2f} Nm at {max_torque_slip:.
↪2f}$\%$'], loc='lower right', fontsize='small')
plt.savefig("max_torque_slip.pdf")
plt.show()
```

[ ]:
```
# Find the index of the maximum torque
max_dp_index = np.argmax(drawbar_pull)
max_dp_value = drawbar_pull[max_dp_index]/n_wheel
max_dp_theta = slope[max_dp_index]
plt.plot(max_dp_theta, max_dp_value, 'o', color=(255/255, 120/255, 0/255),␣
↪markersize=3)
plt.plot(slope, drawbar_pull/n_wheel)
plt.xlabel('Slope $\\theta$ [$^\circ$]')
plt.ylabel('Wheel DP [N]')
plt.xlim(0, 25)
plt.grid()
plt.legend([f'Max Wheel DP: {max_dp_value:.2f} N at {max_dp_theta:.
↪2f}$^\circ$'], loc='lower right', fontsize='small')
plt.savefig("max_drawbar_pull_slope.pdf")
plt.show()
```

[ ]:
```
plt.plot(0, rpm[0], 'o', color=(255/255, 120/255, 0/255), markersize=3)
plt.plot(slope, rpm)
plt.xlabel('Slope $\\theta$ [$^\circ$]')
```

```
plt.ylabel('RPM [$\\frac{1}{min}$]')
plt.xlim(0,25)
plt.ylim(0,)
plt.grid()
plt.legend([f'RPM($0^\circ$): {rpm[0]:.4f} [1/min]'], loc='lower right',␣
↪fontsize='small')
plt.savefig("max_rpm_slope.pdf")
plt.show()
```

[ ]:
```
plt.plot(slope, slip*100)
plt.xlabel('Slope $\\theta$ [$^\circ$]')
plt.ylabel('Slip ratio s [\%]')
plt.xlim(0, 25)
plt.ylim(0, 110)
plt.grid()
plt.savefig("sigmoid_slip.pdf")
plt.show()
```

[ ]:
```
data = pd.read_csv('slope.csv')
distance = data['position']
longitude = -data['lon']
latitude = -data['lat']
slope = data['TerrainSlope']

def func(slope, v_rover_constant, mass, diameter, power_constant, b_width,␣
↪gravity, h_grouser, n_grouser):
    n_wheel = 4 # number of wheels
    #diameter = 0.8
    radius = diameter/2
    #b_width = 0.35 # wheel width in m
    #h_grouser = 0.03 # grouser height
    #n_grouser = 8 # number of grousers
    eta = 0.7 # gearbox efficiency
    #power_constant = 150*eta # constant power for the whole locomotion␣
↪subsystem
    #v_rover_constant = 0.05 # km/h
    c_b = 170 # N/m^2 coefficient of soil/wheel cohesion
    n = 1 # exponent of soil deformation
    k_c = 1400 # N/m^2 cohesive modulus of soil deformation
    k_phi = 830000  # N/m^3 frictional modulus of soil deformation
    K = 0.018  # coefficient of soil slip in m
    gamma = 2470 # N/m3 lunar soil density
    phi = 33 # deg soil internal friction angle
    phi_rad = np.deg2rad(phi) # rad soil internal friction angle
    mu = np.tan(phi_rad) # static friction coefficient
    N_q = 32.23
    N_c = 48.09
```

```
    N_gamma = 33.27
    K_c = 33.37
    K_phi = 72.77
    K_gamma = 81.93
    #mass = 1800 # mass of the rover
    #gravity = 1.62 # gravitational constant of the moon
    #slope = np.linspace(0, 25, 100)
    #slope = 0
    slip = 0.0353 + 1/(1+67*np.exp(-0.32*slope))
    #slip = 1
    k = (k_c / b_width) + k_phi
    c_f = 0.05 # internal rolling friction coefficient (bearings)

    W_vehicle = mass * gravity
    W_vehicle_normal = W_vehicle * np.cos(np.deg2rad(slope))
    W_wheel = mass * gravity / n_wheel
    W_wheel_normal = W_wheel * np.cos(np.deg2rad(slope))

    z_sinkage = (3 * W_wheel_normal / ((3 - n) * (k_c + b_width * k_phi) * np.sqrt(diameter))) ** (2/(2*n+1))
    pressure = k * z_sinkage ** n

    theta_1 = np.arccos(1 - z_sinkage / radius)
    theta_m = (0.45 + 0.24 * slip) * theta_1
    lever_e = radius * np.sin(theta_1 - theta_m)
    lever_l = radius * np.cos(theta_1 - theta_m)
    l_contact = (diameter / 2) * np.arccos(1 - (2 * z_sinkage / diameter))
    A = b_width * l_contact
    alpha = np.arccos(1 - 2 * z_sinkage / diameter) # angle of approach
    l_0 = z_sinkage * np.tan(np.pi / 4 - phi_rad / 2) ** 2 # distance of rupture

    R_compression = ((b_width * k) / (n+1)) * z_sinkage ** (n+1)
    R_bulldozing = (b_width*np.sin(alpha+phi_rad) / (2*np.sin(alpha)*np.cos(phi_rad))) * (2*z_sinkage*c_b*K_c + gamma*K_gamma*z_sinkage**2) + ((gamma*l_0**3)/3) * (np.pi/2 - phi_rad) + (c_b*l_0**2) * (1+np.tan(np.pi/4 + phi_rad/2))
    R_gravitational = W_wheel * np.sin(np.deg2rad(slope))
    R_rolling = W_wheel_normal * c_f # internal (bearings)

    sum_R = 4* R_compression + 2* R_bulldozing + 4* R_gravitational + 4* R_rolling
    n_grouser_contact = 2 * np.pi * radius / (n_grouser * l_contact)

    sum_H_grouser = 4* (A*c_b*(1+2*h_grouser/b_width)*n_grouser_contact + W_wheel_normal*mu*(1 + (0.64*h_grouser/b_width) * np.arctan(b_width/h_grouser))) * (1-np.exp(-slip * l_contact / K))
```

```
    W_wheel_safe = A * (c_b*N_c + gamma*z_sinkage*N_q + gamma*b_width*N_gamma/2)
    pressure_soil_safe = W_wheel_safe / A
    pressure_real = W_wheel / A
    pressure_margin = pressure_real / pressure_soil_safe
    weight_margin = W_wheel / W_wheel_safe

    drawbar_pull = sum_H_grouser - sum_R
    torque = (((drawbar_pull * lever_l) + (W_vehicle_normal * lever_e))/
 ↪n_wheel) + 28
    omega = power_constant / (torque * n_wheel)
    rpm = omega * 60/ (2 * np.pi)
    v_rover = omega * radius * 3.6 * (1-slip)
    power_variable = (v_rover_constant * torque * n_wheel) / (radius * 3.6 *␣
 ↪(1-slip))
    power_variable_step = np.clip(50 * np.ceil(np.clip(power_variable,30,100) /␣
 ↪30), 50, 100)
    return v_rover, power_variable, power_variable_step, drawbar_pull, torque,␣
 ↪rpm, pressure, pressure_soil_safe, pressure_real

solution = func(slope, v_rover_constant, mass, diameter, power_constant,␣
 ↪b_width, gravity, h_grouser, n_grouser)
v_mean = np.mean(solution[0])
slope_mean = np.mean(slope)
end_distance = distance.iloc[-1]
total_time = end_distance/v_mean
```

[ ]:
```
v_mean
```

[ ]:
```
slope_mean
```

[ ]:
```
end_distance
```

[ ]:
```
total_time
```

[ ]:
```
v_rover_constant = distance.iloc[-1] / ((0.5 - 0.25) * 365 * 24)
v_rover_constant
```

[ ]:
```
plt.plot(distance, func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.
 ↪62, 0.03, 8)[1], linewidth=0.3, label='Min required Power')
plt.plot(distance, func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.
 ↪62, 0.03, 8)[2], color='red',  linewidth=0.3, label='Power States')
plt.xlabel('Mission Distance [km]')
plt.ylabel('Power [W]')
plt.xlim(0, distance.iloc[-1])
plt.ylim(0, 125)
plt.title('Constant Velocity = {:.4f} km/h'.format(v_rover_constant))
plt.grid()
```

```
plt.legend(loc='upper left', fontsize='small')
plt.savefig("rover_power_over_mission_distance.pdf")
plt.show()
```

[ ]:
```
power_states = func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.62, 0.
↪03, 8)[2]
plt.plot(distance, func(slope, v_rover_constant, 1800, 0.8, power_states*0.7, 0.
↪35, 1.62, 0.03, 8)[0],  linewidth=0.3, label='Power States')
plt.plot(distance, func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.
↪62, 0.03, 8)[0], color='orange', linewidth=0.3, label='Constant 150W')
plt.xlabel('Mission Distance [km]')
plt.ylabel('Velocity [$\\frac{km}{h}$]')
plt.xlim(0, distance.iloc[-1])
plt.ylim(0, 0.8)
plt.title('Max Velocity with Power States')
plt.grid()
plt.legend(loc='upper left', fontsize='small')
plt.savefig("power_states_over_mission_distance.pdf")
plt.show()
```

[ ]:
```
slip = 0.0353 + 1/(1+67*np.exp(-0.32*slope))
plt.plot(distance, slip*100, linewidth=0.3, color='green')
plt.xlabel('Mission Distance [km]')
plt.ylabel('Slip [\%]')
plt.xlim(0, distance.iloc[-1])
plt.ylim(0, 100)
plt.title('Constant Velocity = {:.4f} km/h'.format(v_rover_constant))
plt.grid()
plt.savefig("slip_over_mission_distance.pdf")
plt.show()
```

[ ]:
```
plt.plot(distance, func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.
↪62, 0.03, 8)[6]/1000, linewidth=1, label='Ground Pressure')
plt.plot(distance, func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.
↪62, 0.03, 8)[7]/1000, linewidth=1, label='Safe Pressure')
plt.xlabel('Mission Distance [km]')
plt.ylabel('Pressure [kPa]')
plt.xlim(0, distance.iloc[-1])
plt.ylim(0, 30)
plt.title('Constant Velocity = {:.4f} km/h'.format(v_rover_constant))
plt.grid()
plt.legend()
plt.savefig("pressure_over_mission_distance.pdf")
plt.show()
```

[ ]:
```
slope = np.linspace(0, 25, 100) # range of slope values
```

```
# Define values for mass, slip, radius, and power
mass_values = [600, 1000, 1300, 1600, 1700, 1800]
h_grouser_values = [0.03, 0.05, 0.07, 0.1, 0.12, 0.14]
diameter_values = [0.4, 0.6, 0.8, 1]
power_values = [ 50, 100, 150, 200, 250]
width_values = [0.1, 0.2, 0.3, 0.4, 0.5]
gravity_values = [1.62, 3.71, 9.81]
gravity_name_values = ['Moon', 'Mars', 'Earth']
n_grouser_values = [8, 16, 24, 32]

# Create subplots for Varying Mass
fig1, ax1 = plt.subplots(figsize=(6, 6))
for mass_n in mass_values:
    ax1.plot(slope, func(slope, v_rover_constant, mass_n, diameter, 
 ↪power_constant, b_width, gravity, h_grouser, n_grouser)[0], label=f"{mass_n} 
 ↪kg")
ax1.legend()
ax1.set_title("Varying Mass")
ax1.set_xlabel('Slope $\\theta$ [$^\circ$]')
ax1.set_ylabel('Speed [$\\frac{km}{h}$]')
ax1.grid(True)
ax1.set_xlim(0, 25)
ax1.set_ylim(0, 1.5)
plt.savefig("tire_sim_varying_mass.pdf")
plt.show()

# Create subplots for Varying Grouser Height
fig2, ax2 = plt.subplots(figsize=(6, 6))
for h_grouser_n in h_grouser_values:
    ax2.plot(slope, func(slope, v_rover_constant, mass, diameter, 
 ↪power_constant, b_width, gravity, h_grouser_n, n_grouser)[0], 
 ↪label=f"{h_grouser_n} m")
ax2.legend()
ax2.set_title("Varying Wheel Grouser Height")
ax2.set_xlabel('Slope $\\theta$ [$^\circ$]')
ax2.set_ylabel('Speed [$\\frac{km}{h}$]')
ax2.grid(True)
ax2.set_xlim(0, 25)
ax2.set_ylim(0, 1.5)
plt.savefig("tire_sim_varying_grouser_height.pdf")
plt.show()

# Create subplots for Varying Radius
fig3, ax3 = plt.subplots(figsize=(6, 6))
for diameter_n in diameter_values:
```

```
    ax3.plot(slope, func(slope, v_rover_constant, mass, diameter_n, 
 ↪power_constant, b_width, gravity, h_grouser, n_grouser)[0], 
 ↪label=f"{diameter_n} m")
ax3.legend()
ax3.set_title("Varying Wheel Diameter")
ax3.set_xlabel('Slope $\\theta$ [$^\circ$]')
ax3.set_ylabel('Speed [$\\frac{km}{h}$]')
ax3.grid(True)
ax3.set_xlim(0, 25)
ax3.set_ylim(0, 1.5)
plt.savefig("tire_sim_varying_diameter.pdf")
plt.show()

# Create subplots for Varying Power
fig4, ax4 = plt.subplots(figsize=(6, 6))
for power_n in power_values:
    ax4.plot(slope, func(slope, v_rover_constant, mass, diameter, power_n, 
 ↪b_width, gravity, h_grouser, n_grouser)[0], label=f"{power_n} W")
ax4.legend()
ax4.set_title("Varying Subsystem Electrical Power")
ax4.set_xlabel('Slope $\\theta$ [$^\circ$]')
ax4.set_ylabel('Speed [$\\frac{km}{h}$]')
ax4.grid(True)
ax4.set_xlim(0, 25)
ax4.set_ylim(0, 1.5)
plt.savefig("tire_sim_varying_power.pdf")
plt.show()

# Create subplots for Varying Width
fig5, ax5 = plt.subplots(figsize=(6, 6))
for width_n in width_values:
    ax5.plot(slope, func(slope, v_rover_constant, mass, diameter, 
 ↪power_constant, width_n, gravity, h_grouser, n_grouser)[0], 
 ↪label=f"{width_n} m")
ax5.legend()
ax5.set_title("Varying Wheel Width")
ax5.set_xlabel('Slope $\\theta$ [$^\circ$]')
ax5.set_ylabel('Speed [$\\frac{km}{h}$]')
ax5.grid(True)
ax5.set_xlim(0, 25)
ax5.set_ylim(0, 1.5)
plt.savefig("tire_sim_varying_width.pdf")
plt.show()

# Create subplots for Varying Planet
fig6, ax6 = plt.subplots(figsize=(6, 6))
```

```
for gravity_val, gravity_name in zip(gravity_values, gravity_name_values):
    ax6.plot(slope, func(slope, v_rover_constant, mass, diameter,␣
 ↪power_constant, b_width, gravity_val, h_grouser, n_grouser)[0],␣
 ↪label=f"{gravity_name}")
ax6.legend()
ax6.set_title("Varying Astronomical Body")
ax6.set_xlabel('Slope $\\theta$ [$^\circ$]')
ax6.set_ylabel('Speed [$\\frac{km}{h}$]')
ax6.grid(True)
ax6.set_xlim(0, 25)
ax6.set_ylim(0, 1.5)
plt.savefig("tire_sim_varying_gravity.pdf")
plt.show()

# Create subplots for Varying Number of Grousers
fig5, ax7 = plt.subplots(figsize=(6, 6))
for n_grouser_n in n_grouser_values:
    ax7.plot(slope, func(slope, v_rover_constant, mass, diameter,␣
 ↪power_constant, b_width, gravity, h_grouser, n_grouser_n)[0],␣
 ↪label=f"{n_grouser_n}")
ax7.legend()
ax7.set_title("Varying Number of Grousers")
ax7.set_xlabel('Slope $\\theta$ [$^\circ$]')
ax7.set_ylabel('Speed [$\\frac{km}{h}$]')
ax7.grid(True)
ax7.set_xlim(0, 25)
ax7.set_ylim(0, 1.5)
plt.savefig("tire_sim_varying_number_of_grousers.pdf")
plt.show()
```

[ ]:
```
slope = np.linspace(0, 20, 100)
solution = func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.62, 0.03,␣
 ↪8)
# Calculate required values
slope_min, slope_min, slope_max = np.min(slope), np.mean(slope), np.max(slope)
drawbar_pull_min, drawbar_pull_mean, drawbar_pull_max = np.min(solution[3]), np.
 ↪mean(solution[3]), np.max(solution[3])
v_min, v_mean, v_max = np.min(solution[0]), np.mean(solution[0]), np.
 ↪max(solution[0])
torque_min, torque_mean, torque_max = np.min(solution[4]), np.
 ↪mean(solution[4]), np.max(solution[4])
P_min, P_mean, P_max = np.min(solution[1]), np.mean(solution[1]), np.
 ↪max(solution[1])
rpm_min, rpm_mean, rpm_max = np.min(solution[5]), np.mean(solution[5]), np.
 ↪max(solution[5])
```

```
# Print optimized results with \n after each value
print(f"slope_min: {slope_min:.2f}\nslope_mean: {slope_min:.2f}\nslope_max:␣
↪{slope_max:.2f}\n"
      f"drawbar_pull_min: {drawbar_pull_min:.2f}\ndrawbar_pull_mean:␣
↪{drawbar_pull_mean:.2f}\ndrawbar_pull_max: {drawbar_pull_max:.2f}\n"
      f"v_min: {v_min:.4f}\nv_mean: {v_mean:.4f}\nv_max: {v_max:.4f}\n"
      f"torque_min: {torque_min:.2f}\ntorque_mean: {torque_mean:.
↪2f}\ntorque_max: {torque_max:.2f}\n"
      f"P_min: {P_min:.2f}\nP_mean: {P_mean:.2f}\nP_max: {P_max:.2f}\n"
      f"rpm_min: {rpm_min:.4f}\nrpm_mean: {rpm_mean:.4f}\nrpm_max: {rpm_max:.
↪4f}")
```

[ ]:
```
from scipy.interpolate import griddata
data = pd.read_csv('slope.csv')

distance = data['position']
longitude = -data['lon']
latitude = -data['lat']
slope = data['TerrainSlope']

# Create a grid spherical
xi_sphere = np.linspace(min(longitude), max(longitude), len(longitude))
yi_sphere = np.linspace(min(latitude), max(latitude), len(latitude))
xi_sphere, yi_sphere = np.meshgrid(xi_sphere, yi_sphere)
zi_sphere = griddata((longitude, latitude), slope, (xi_sphere, yi_sphere),␣
↪method='nearest')

solution = func(slope, v_rover_constant, 1800, 0.8, 150*0.7, 0.35, 1.62, 0.03,␣
↪8)
v_mean = np.mean(solution[0])
end_distance = distance.iloc[-1]
total_time = end_distance/v_mean
v_min = np.min(solution[0])
v_max = np.max(solution[0])

fig, (ax1) = plt.subplots(1, 1, figsize=(6, 6))
ax1.grid(True)
contour1 = ax1.contourf(xi_sphere, yi_sphere, zi_sphere, cmap='viridis')
fig.colorbar(contour1, ax=ax1, label='Terrain Slope [°]')
scatter1 = ax1.scatter(longitude, latitude, c=solution[0], linewidths=0.1,␣
↪marker='o', s=5, cmap='autumn')
fig.colorbar(scatter1, ax=ax1, label='Speed [km/h]')
ax1.axis('auto')
margin = 0.2
ax1.set_xlim(min(longitude) - margin, max(longitude) + margin)
ax1.set_ylim(min(latitude) - margin/5, max(latitude) + margin/5)
```

```python
# Add markers for every k hours
k = 50  # hours
tolerance = 0.1  # Adjust this value based on your desired tolerance
marker_added = False  # Reset the marker flag at the start

for i in range(len(distance)):
    if i > 0:
        v_mean_up_to = np.cumsum(solution[0])[i] / i
        time_passed = distance[i] / v_mean_up_to
        # Check if the current time_passed is within a new k-hour interval
        if abs((time_passed % k) / k - 1) < tolerance and not marker_added:
            label_text = '{}h'.format(round(time_passed / k) * k)
            #ax1.plot(longitude[i], latitude[i], 'o', color='white', markersize=6, markeredgecolor='black')
            ax1.text(longitude[i], latitude[i], label_text, ha='center', va='center', color='white', fontsize=8, alpha=1)
            marker_added = True  # Set the flag to True after adding the marker and label
        elif abs((time_passed % k) / k - 1) >= tolerance:
            marker_added = False  # Reset the flag if outside the tolerance

ax1.set_xlabel('Longitude [°]')
ax1.set_ylabel('Latitude [°]')
ax1.set_title('Constant Power Variable Velocity\n'
              'Mean Velocity = {:.2f} km/h\n'
              'Min Velocity = {:.2f} km/h\n'
              'Max Velocity = {:.2f} km/h\n'
              'Total time = {:.2f} h\n'
              'Total Gearbox and Motor efficiencies $\\eta$ = {:.2f}\n'
              'Constant Electrical Power = {:.2f} W\n'
              'Mass = {:.2f} kg\n'
              'Radius = {:.2f} cm\n'
              'Width = {:.2f} cm'
              .format(v_mean, v_min, v_max, total_time, eta, power_constant/eta, mass, radius, b_width))
plt.tight_layout()
plt.savefig("tire_sim_constant_power_variable_velocity.png")
plt.show()
```