MEEN 357 Phase 3 Project Report

Jae Bryson, Andrew Spain, Kayla Waslwil

Collaboration Statement:

Our group did not collaborate with any other group for this phase of the project.

Task 1

- Name: (Default Value) [Units] Description

- edl_system: overall dictionary for project Phase 3
    - altitude: (NaN) State variable of altitude, changes over time
    - velocity: (NaN) State variable of velocity, changes over time
    - num_rockets: (8) [-] Quantity of rockets, system level parameter
    - volume: (150) [m^3] Volume of system, system level parameter
    - parachute: parachute dictionary
        - deployed: (True) True if parachute is deployed, but not ejected
        - ejected: (False) True if parachute is no longer attached to system
        - diameter: (16.25) [m] Diameter of parachute
        - Cd: (0.615) [-] Coefficient of drag of parachute for subsonic
        - mass: (185.0) [kg] Mass of parachute
    - heat_shield: heat shield dictionary
        - ejected: (False) True means heat shield has been ejected
        - mass: (225.0) [kg] Mass of heat shield
        - diameter: (4.5) [m] Diameter of heat shield
        - Cd: (0.35) [-] Coefficient of drag of heat shield
    - rocket: rocket dictionary
        - on: (False) If rocket is turned on
        - structure_mass: (8.0) [kg] Mass of everything minus fuel
        - initial_fuel_mass: (230.0) [kg] Initial mass of fuel
        - fuel_mass: (230.0) [kg] Current mass of fuel
        - effective_exhaust_velocity: (4500.0) [m/s] Velocity of gasses out of exhaust
        - max_thrust: (3100) [N] Max possible thrust
        - min_thrust: (40.0) [N] Min possible thrust
    - speed_control: speed control dictionary
        - on: (False) Indicates whether control mode is activated
        - Kp: (2000) [-] Proportional gain term
        - Kd: (20) [-] Derivative gain term
        - Ki: (50) [-] Integral gain term
        - target_velocity: (-3.0) [m/s] Desired descent speed
    - position_control: position control dictionary
        - on: (False) Indicates whether control mode is activated
        - Kp: (2000) [-] Proportional gain term
        - Kd: (1000) [-] Derivative gain term
        - Ki: (50) [-] Integral gain term
        - target_altitude: (7.6) [m] Desired final altitude with sky crane

    - sky_crane: sky crane dictionary
        - on: (False) True means lowering rover mode
        - danger_altitude: (4.5) [m] Altitude that is considered too low for safe touchdown
        - danger_speed: (-1.0) [m/s] Speed that is considered too fast for safe touchdown
        - mass: (35.0) [kg] Mass of sky crane
        - area: (16.0) [m^2] Frontal area for drag calculation
        - Cd: (0.9) [-] Coefficient of drag
        - max_cable: (7.6) [m] Max length of cable for lowering rover
        - velocity: (-0.1) [m/s] Speed at which the sky crane lowers rover
    - rover: overall rover dictionary
        - wheel_assembly: wheel assembly of rover dictionary
            - wheel: individual wheel dictionary
                - radius: (0.20) [m] Wheel radius
                - mass: (2) [kg] Single wheel mass
            - speed_reducer: speed reducer of rover dictionary
                - type: (‘reverted’) Type of speed reducer
                - diam_pinion: (0.04) [m] Diameter of pinion
                - diam_gear: (0.06) [m] Diameter of gear
                - mass: (1.5) [kg] Mass of speed reducer
            - motor: individual motor dictionary
                - torque_stall: (165) [N*m] Maximum (stall) torque of motor
                - torque_noload: (0) [N*m] Torque when no load is applied
                - speed_noload: (3.85) [rad/s] Speed of motor when no load is applied
                - mass: (5.0) [kg] Mass of a single motor
                - effcy_tau: (np.array([0, 10, 20, 40, 75, 170])) [N*m] Experimental efficiency data (torques)
                - effcy: (np.array([0,.60,.75,.73,.55, .05])) [-] Experimental efficiency data (efficiencies)
        - chassis: rover chassis dictionary
            - mass: (674) [kg] Mass of chassis
        - science_payload: science payload dictionary
            - mass: (80) [kg] Mass of science payload
        - power_subsys: power subsystem dictionary
            - mass: (100) [kg] Mass of power subsystem
        - planet: planet constants dictionary
            - g: (3.72) [m/s^2] Gravitational acceleration of mars
- mars: overall mars atmosphere dictionary

- g: (-3.72) [m/s^2] Gravitational acceleration of mars
- altitude_threshold: (7000) [m] Altitude value seperating low and high altitude
- low_altitude
    - temperature: (function) [C] Temperature as a function of altitude
    - pressure: (function) [kPa] Pressure as a function of altitude
- high_altitude
    - temperature: (function) [C] Temperature as a function of altitude
    - pressure: (function) [kPa] Pressure as a function of altitude
- density: (function) [kg/m^3] Density as a function of pressure and temperature

Task 2

Function Name:
get_mass_rover
Calling Syntax:
m = get_mass_rover(edl_system)
Description:
This function computes the mass of the rover defined in the rover field of the EDL system structure. This function assumes that the rover is defined as a dictionary corresponding to the specification of project phase 1.
Input Arguments:

edl_system	dict	Data structure containing EDL system parameters

Output Arguments:

m	scalar	Mass of rover [kg]

Function Name:
get_mass_rockets
Calling Syntax:
m = get_mass_rockets(edl_system)
Description:
This function returns the current total mass of all rockets on the EDL system.
Input Arguments:

edl_system	dict	Data structure containing EDLl system parameters

Output Arguments:

m	scalar	Mass of all rockets [kg]

Function Name:

get_mass_edl

Calling Syntax:

m = get_mass_edl(edl_system)

Description:

This function returns the total current mass of the EDL system. This function should call get_mass_rover and get_mass_rockets

Input Arguments:

edl_system	dict	Data structure containing EDL system parameters

Output Arguments:

m	scalar	Mass of EDLl system [kg]

Function Name:

get_local_atm_properties

Calling Syntax:

density, temperature, pressure = get_local_atm_properties(planet, altitude)

Description:

This function returns the local atmospheric properties at a given altitude. This function is not vectorized and will not accept a vector of given altitudes. If the altitude is greater than the altitude threshold, it will use the high altitude, otherwise it will use the low altitude to determine temperature and pressure.

Input Arguments:

planet	dict	Data structure containing planet parameters.
altitude	scalar	Given altitude [m]

Output Arguments:

density	scalar	Atmospheric density [kg/m^3]
temperature	scalar	Local temperature [C]
pressure	scalar	Local pressure [KPa]

Function Name:

F_buoyancy_descent

Calling Syntax:

F = F_buoyancy_descent(edl_system, planet, altitude)

Description:

This function computes the net buoyancy force, accounting for the direction of the force of gravity. This function should call get_local_atm_

Input Arguments:

edl_system	dict	Data structure containing EDL system parameters

planet	dict	Data structure containing planet parameters
altitude	scalar	Given altitude [m]

Output Arguments:

F	scalar	Force of buoyancy on EDL system [N]

Function Name:

F_drag_descent

Calling Syntax:

F = F_drag_descent(edl_system, planet, altitude, velocity)

Description:

This function computes the net drag force on the EDL system. It assumes the heat shield as the drag contributor if it has not been ejected, otherwise the sky crane is the drag contributor. It also accounts for the parachute area if it is in the deployed state. This function should call get_local_atm_properties

Input Arguments:

edl_system	dict	Data structure containing EDL system parameters
planet	dict	Data structure containing planet parameters
altitude	scalar	Given altitude [m]
velocity	scalar	Given velocity [m/s]

Output Arguments:

F	scalar	Net force of drag on EDL system [N]

Function Name:
F_gravity_descent
Calling Syntax:
F = F_gravity_descent(edl_system, planet)
Description:
This function computes the gravitational force acting on the EDL system. This function should call get_mass_edl
Input Arguments:

edl_system	dict	Data structure containing EDL system parameters
planet	dict	Data structure containing planet parameters

Output Arguments:

F	scalar	Force of buoyancy on EDL system [N]

Function Name:
v2M_Mars
Calling Syntax:
M = v2M_Mars(v, a)
Description:
This function returns the absolute value of the Mach number by converting the descent speed to Mach number on Mars as a function of altitude. It uses a set array of SPD data values and the pchip function to do so. It will return the absolute value of the Mach number.
Input Arguments:

v	scalar	Given descent speed of system [m/s]
a	scalar	Given altitude of system [m]

Output Arguments:

M	scalar	Absolute value Mach number [dimensionless]

Function Name:

thrust_controller

Calling Syntax:

edl_system = thrust_controller(edl_system, planet)

Description:

This function implements a PID controller for the EDL system. It uses the edl_system and planet structures to create a modified edl_system structure. It will modify fields in rocket and telemetry substructures. It should check that both inputs are dictionaries.

If the rocket is on and control is activated, the function will run as intended, otherwise will append 0 to the telemetry error and will set the rocket thrust as the fixed thrust if the control is not activated. If the rocket is not on, a value of 0 will be appended to the thrust.

This function will apply corrections as needed due to saturating rocket capabilities.

Input Arguments:

edl_system	dict	Data structure containing EDL system parameters.
planet	dict	Data structure containing planet parameters.

Output Arguments:

edl_system	dict	Updated data structure containing implemented PID controller EDL system parameters.

Function Name:
edl_events
Calling Syntax:
events = edl_events(edl_system, mission_events)
Description:
This function defines events that occur in the EDL system simulation. It will create a total of nine events, as follows, based on a list containing system values defined as y. This function will create a lambda function for each event and set the terminal value to True.

event0	Reached altitude to eject heat shield
event1	Reached altitude to eject parachute
event2	Reached altitude to turn on rockets
event3	Reached altitude to turn on crane and altitude control
event4	Recognizes when system is out of fuel
event5	Recognizes when altitude is zero, therefore crashing
event6	Reached speed at which speed-controlled descent is required
event7	Reached position at which altitude control is required
event8	Recognizes that rover has touched down on Mars

Input Arguments:

edl_system	dict	Data structure containing EDL system parameters.
mission_events	dict	Data structure containing information regarding the values of mission event parameters.

Output Arguments:

events	list	Nine element list containing each event as an element.

Function Name:
edl_dynamics
Calling Syntax:
dydt = edl_dynamics(t, y, edl_system, planet)
Description:
This function describes the dynamics of EDL as it descends and lowers the rover to the planet surface. The EDL altitude, velocity and acceleration are absolute while rover values are relative to EDL. The fuel mass should be the total over all the rockets. The momentum contributions from both the EDL system and the propellant should be considered. The calculations of dydt will depend on both the state of the EDL system and the state of the sky crane. This function will call get_mass_edl, F_gravity_descent, F_buoyancy_descent, and F_drag_descent. The elements of y should be described as follows: y = [vel_edl;pos_edl;fuel_mass;ei_vel;ei_pos;vel_rov;pos_rov] ydot = [accel_edl;vel_edl;dmdt;e_vel;e_pos;accel_rov;vel_rov]
Input Arguments:

t	Numpy array	Time array relative to y [s]
y	Numpy array	State vector seven element array of EDL and rover characteristics
edl_system	dict	Data structure containing EDL system parameters.
planet	dict	Data structure containing planet parameters.

Output Arguments:

dydt	Numpy array	Seven element array containing corresponding derivatives for each value in array y.

Function Name:

update_edl_state

Calling Syntax:

edl_system, y0, TERMINATE_SIM = update_edl_state(edl_system, TE, YE, Y, ITER_INFO)

Description:

This function updates the status of the EDL system based on simulation events. It uses the same event convention as described in the edl_events function. This function will also update the rocket mass due to fuel expulsion. This function will default the initial conditions to the prior final conditions. If ITER_INFO is true, the function should print information regarding the EDL system to the console.

Input Arguments:

edl_system	dict	Data structure containing EDL system parameters.
TE	Numpy array	Array containing values for the time of system events.
YE	Numpy array	Array containing values for parameters of system events.
Y	Numpy array	Array containing values of the prior conditions of the system.
ITER_INFO	boolean	Flag to display detailed iteration information (optional)

Output Arguments:

edl_system	dict	Updated data structure containing new EDL system parameters.
y0	Numpy Array	Array containing information on the state of the system
TERMINATE_SIM	boolean	Holds value True when event situation should cause termination of simulation, otherwise False.

Function Name:

simulate_edl

Calling Syntax:

T, Y, edl_system = simulate_edl(edl_system, planet, mission_events, t_max, ITER_INFO)

Description:

This function simulates the EDL system with the solve_ivp function. It requires a definition of the EDL system, the planet, the mission events, a maximum simulation time and should have an optional flag to display detailed iteration information. This function should call edl_events, edl_dynamics, and update_edl_state.

Input Arguments:

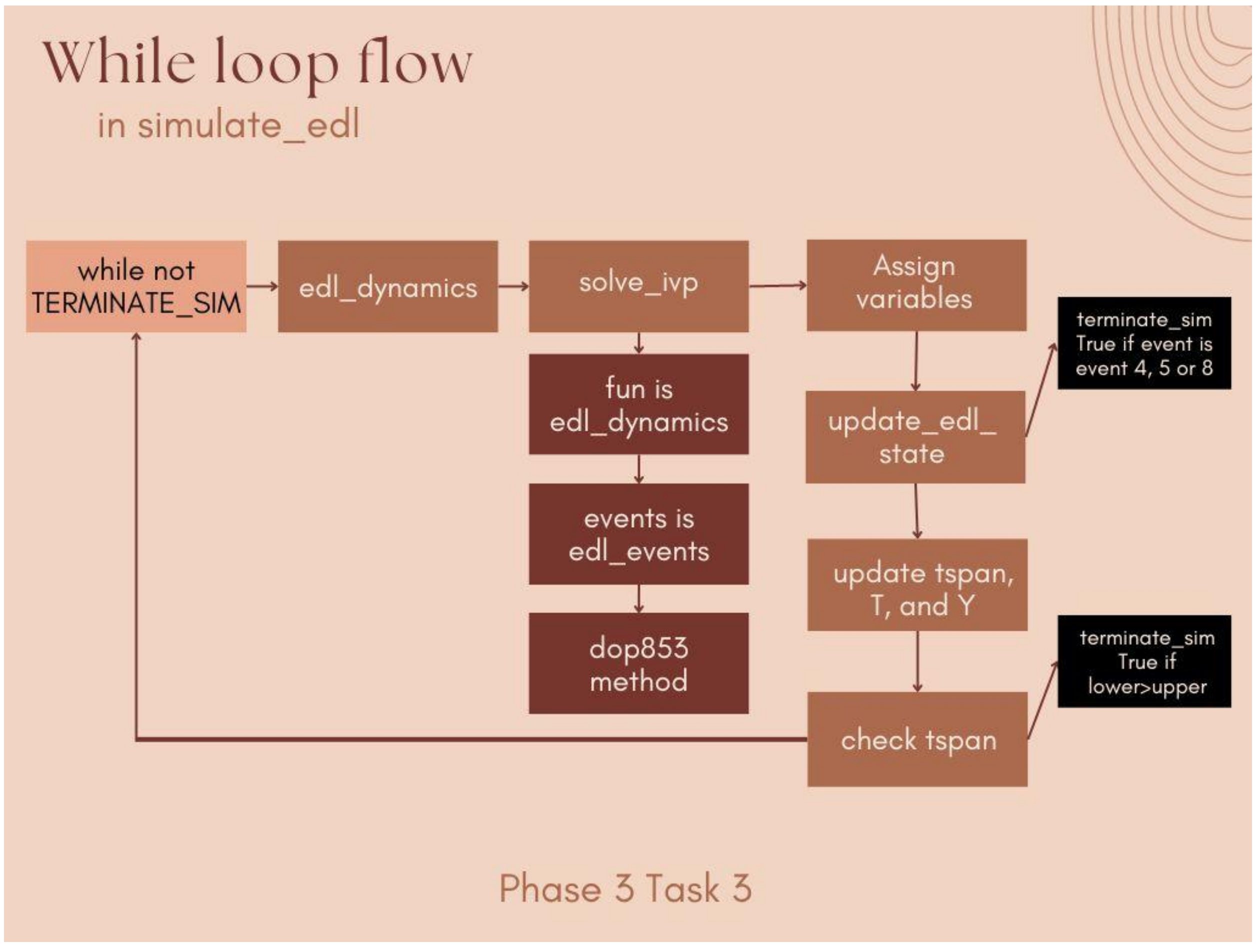
edl_system	dict	Data structure containing EDL system parameters.
planet	dict	Data structure containing planet parameters.
mission_events	dict	Data structure containing information regarding the values of mission event parameters.
t_max	scalar	Maximum simulation time [s]
ITER_INFO	boolean	Flag to display detailed iteration information (optional)

Output Arguments:

T	Numpy array	Array containing the time points resulting from solving

		the edl_dynamics function.
Y	Numpy array	Array containing the value of the solution at each corresponding time point resulting from solving the edl_dynamics function.
edl_system	dict	Updated data structure containing new EDL system parameters

Task 3



The while loop runs on the boolean logic that if TERMINATE_SIM is False, the code in the loop will run and will continue to loop until TERMINATE_SIM holds the value True. The first part of the loop defines a function with edl_dynamics, and since edl_dynamics returns derivative values that define the motion of the EDL system, the function defined by it can be said to do the same. Next, solve_ivp is used with edl_dynamics as fun, the previously defined time span, y0 array, method DOP853 for accuracy and a max step size of .1. The DOP853 method is an explicit 8th order Runge Kutta method IVP solver. Events are used in this solve_ivp, and are established by using edl_events.

In the edl_events function, a variable is defined for each possible event that is taken into consideration and each event variable is put into a list and that list is returned from the function. Because of the nature of solve_ivp and since each of the event terminal boolean values were set to True, the integration will terminate once the first event occurs.

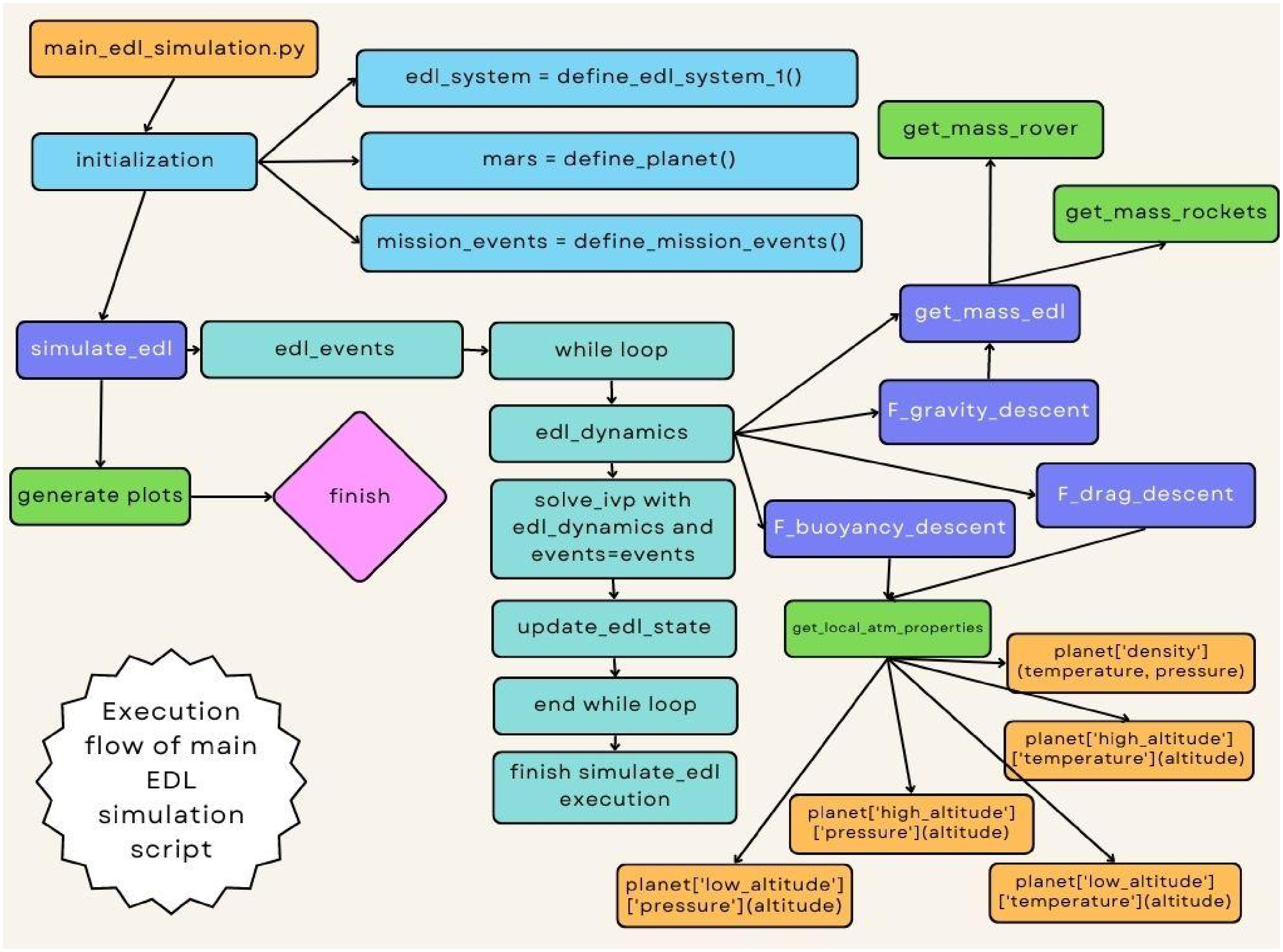
After establishing the solve_ivp parameters, variables are defined for arrays of the time values of the solution, the y values of the solution, the time values at which the event encountered occurred, and the y values at which the event occurred. Each of these are then used as the parameters for the update_edl_state function in addition to edl_system to define the system and ITER_INFO to display iteration information.

In the update_edl_state function, each possible event is checked and if there is information at the corresponding location for the time event, meaning that the event did happen, the edl_system dictionary will be updated with new information about the event. Since the while loop can only check for one event at a time, it will only update for the current event it found in that iteration of the loop. If the event that occurred is the rocket running out of fuel, the EDL crashing before the sky crane was activated, or the rover landing either with damage or without damage, the TERMINATE_SIM value will be assigned True, therefore ending the while loop after it finishes its current iteration.

After updating the dictionary, the lower boundary of time span is updated with the lowest array value of time from the solution. Additionally, the t and y values from the solution are appended to two separate previously defined lists to work around the adaptive step size. Lastly, the time span is checked to ensure that the lower boundary is not greater than the upper boundary. If it is then the terminate sim boolean is changed to True, therefore ending the loop after its current iteration.

If after all this, the terminate sim variable still holds the value false the loop will iterate again, this time using the updated variables and dictionaries from previous runs as the new initial conditions. With the updated information, the loop will detect the next event that occurs and continue to go through the events and update the information until an event or other source causes the terminate sim value to become True. If at the end of the loop the terminate sim value is true, the loop will not run again and the function will return its output values.

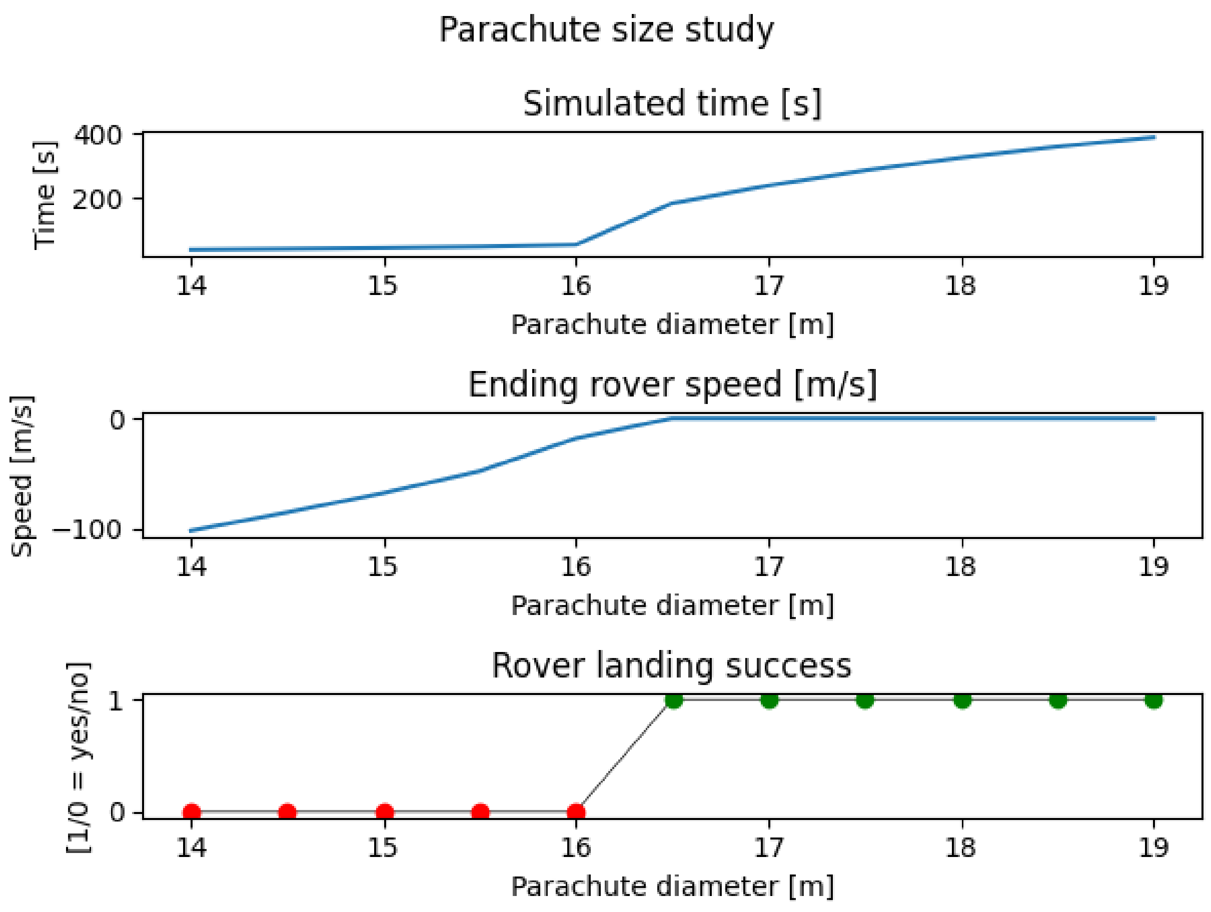
Task 4


main_edl_simulation.py
initialization
edl_system = define_edl_system_1()
mars = define_planet()
mission_events = define_mission_events()
simulate_edl
edl_events
while loop
edl_dynamics
solve_ivp with edl_dynamics and events=events
update_edl_state
end while loop
finish simulate_edl execution
generate plots
finish
get_mass_rover
get_mass_rockets
get_mass_edl
F_gravity_descent
F_drag_descent
F_buoyancy_descent
get_local_atm_properties
planet['density'] (temperature, pressure)
planet['high_altitude']['temperature'](altitude)
planet['high_altitude']['pressure'](altitude)
planet['low_altitude']['pressure'](altitude)
planet['low_altitude']['temperature'](altitude)
Execution flow of main EDL simulation script

The above flow chart represents the execution flow of the main EDL simulation script. The script first begins by establishing the dictionaries that will be used throughout the script and assigning them variables. Then it calls the simulate_edl function. Within the simulate_edl function, the edl_events function is run to assign its return values to the variable “events”, and then within the while loop in simulate edl, the edl_dynamics function is called and assigned the variable “fun”. Within edl dynamics, get_mass_edl calls get_mass_rover and get_mass_rockets, F_gravity_descent calls get_mass_edl, F_drag_descent calls get_local_atm_properties, and F_buoyancy_descent calls get_local_atm_properties. Within this, get_local_atm_properties calls five lambda functions within the planet dictionary. After edl dynamics has executed, fun and events are used with the imported function solve_ivp, and then the update_edl_state function is called in order to update the dictionaries with new information. Once the while loop has ended, the simulate edl function has executed and so the flow returns to the script, where the script then generates the plots it was intended to.

Task 5

The script “study_parachute_size.py” simulates the rover’s landing for a multiple with varying parachute diameters. Here’s a plot of the results:



The bottom plot shows that all the simulations with a parachute diameter 16m or less were failures, while any 16.5m or above were successes. This lines up with the intuition that larger parachutes are safer. The results here suggest that the minimum parachute diameter to ensure a successful landing is somewhere between 16m and 16.5m. The top plot shows that it takes longer to land as the parachute size increases, which also makes sense.

The optimal parachute for our purposes is one that ensures landing success, while also minimizing time. We want a parachute that is as small as possible while still being safe. “Safe” here is subjective, it could mean a parachute that our model predicts as a success, but is likely better to include a cushion in our decision to reduce the chance of failure. If we assumed our model was perfect, then 16.5m would be the optimal choice, because it minimizes time spent while also ensuring success, but our model is not perfect.

I would suggest a diameter between 16.5m and 17.5m. The bigger the cost of air time is, the more advisable it is to pick a diameter on the low edge of that range.

Task 6

The new drag model was created in the functions *__get_coefficients_for_drag_model* and *__full_get_MEF* and *get_MEF* in the script *updated_drag_model.py*

1. The method in which we chose to model the experimental data continuously was through linear splines. This was done primarily because of its simplicity and speed, both in creating the code and in solving. Doing this with other interpolation methods would have resulted in much larger arrays to be solved, and would only marginally increase the accuracy. We can prove how much of an accuracy difference there would be using preexisting cubic and quadratic interpolation functions. Figure 2 shows what these interpolations would look like, and there are a few points of note. Firstly, there are oscillations between Mach 0.25 and 0.65. This should not be happening since we know drag is linear up until near Mach 1. Another point of note is beyond Mach 2, the Mach efficiency factor decreases linearly in all interpolations, so calculating cubic splines for these points is unnecessary. Both of these takeaways can be seen in Figure 3, which plots the relative error between cubic and quadratic against linear. While the relative error commonly increases in Mach 1, the error increases above Mach 2 and below Mach 0.5. All of this is to say that linear splines might seem like a non-ideal option, but it makes sense for some of the scenarios presented.

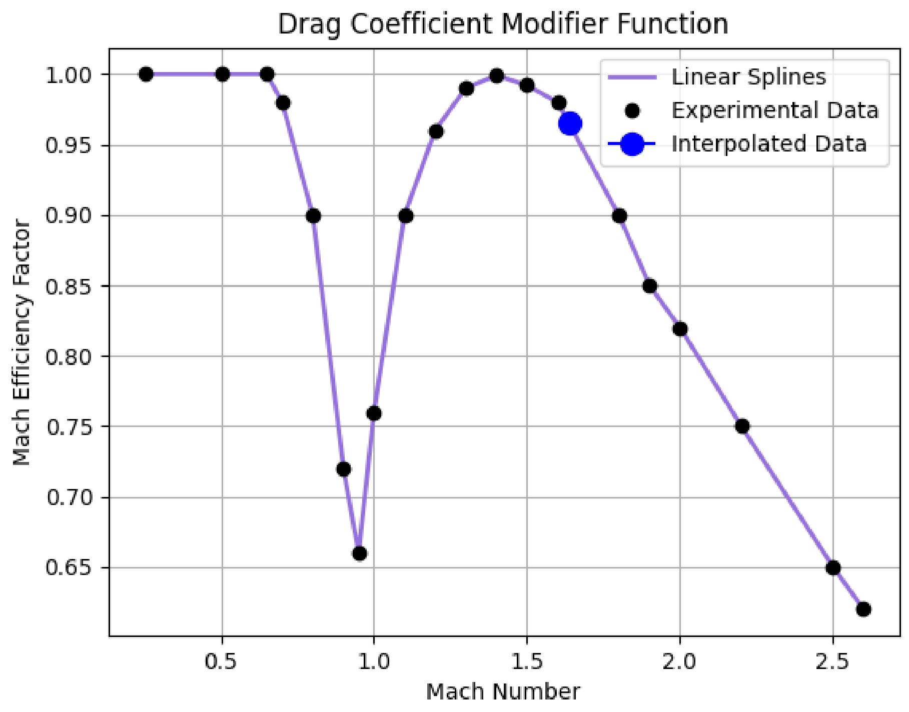


Figure 1: Mach efficiency factor plotted against the Mach number, with the experimental data, linear splines, and an example interpolated data point.

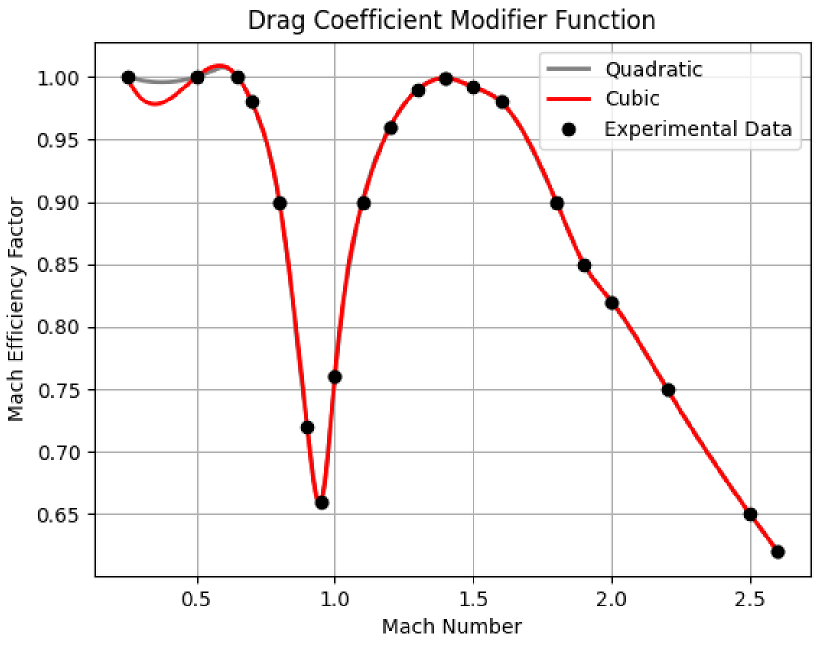


Figure 2: Mach efficiency factor plotted against Mach number using scipy.interpolation functions with cubic and quadratic splines

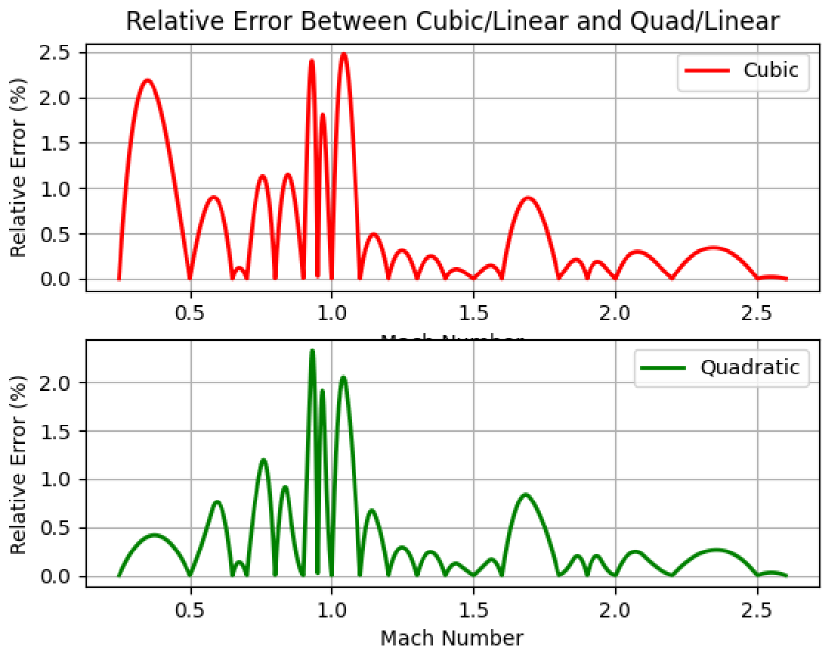


Figure 3: Relative error (%) between Cubic and Linear interpolation, as well as Quadratic and Linear interpolation against Mach number

2. The parachute diameter recommendation does require modifications after accounting for the updated drag model. In the original model (seen in Figure 4), a diameter of 16.5 meters was recommended to ensure a successful landing. After accounting for the new drag model (seen in Figure 5), the diameter recommendation increased to 17.5 meters to ensure a successful landing.

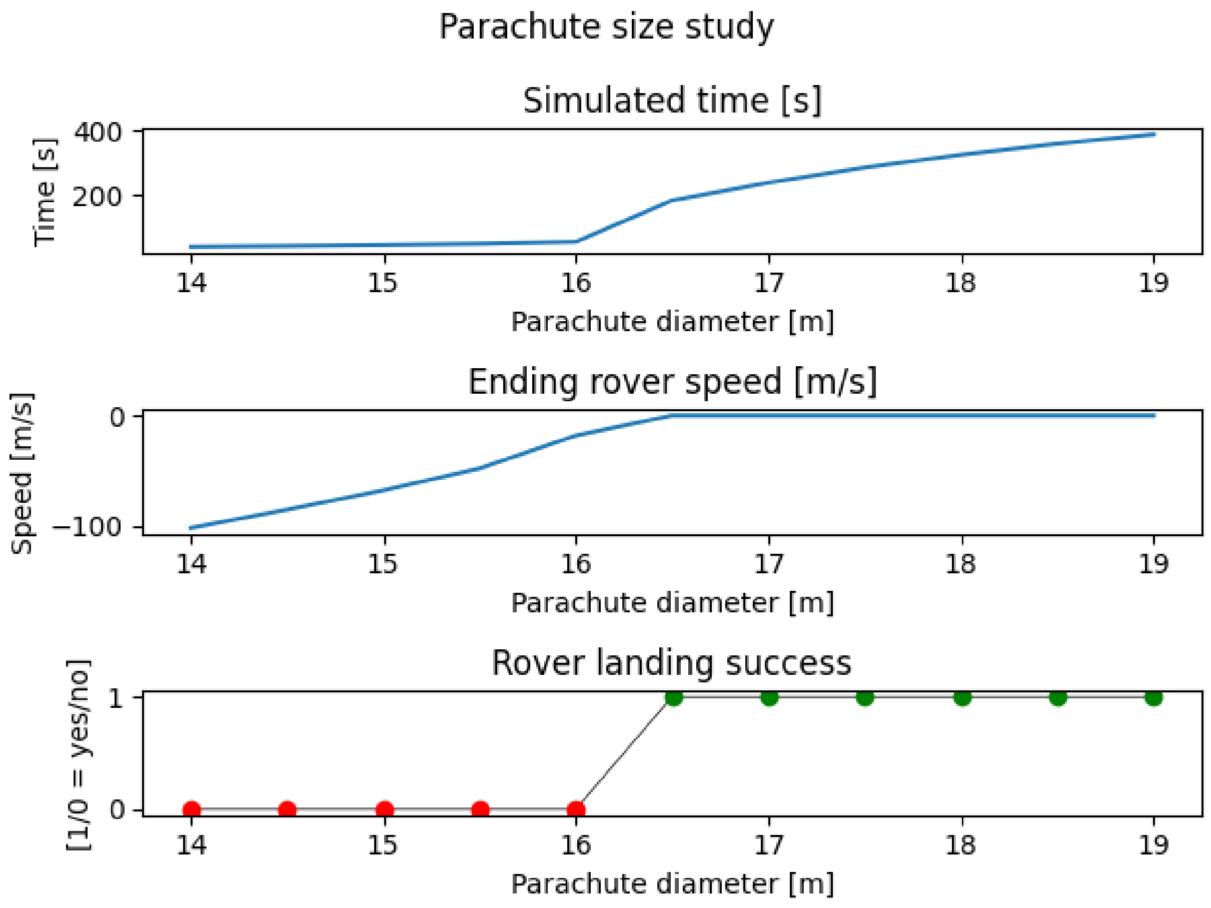


Figure 4: Parachute size study using the old drag model

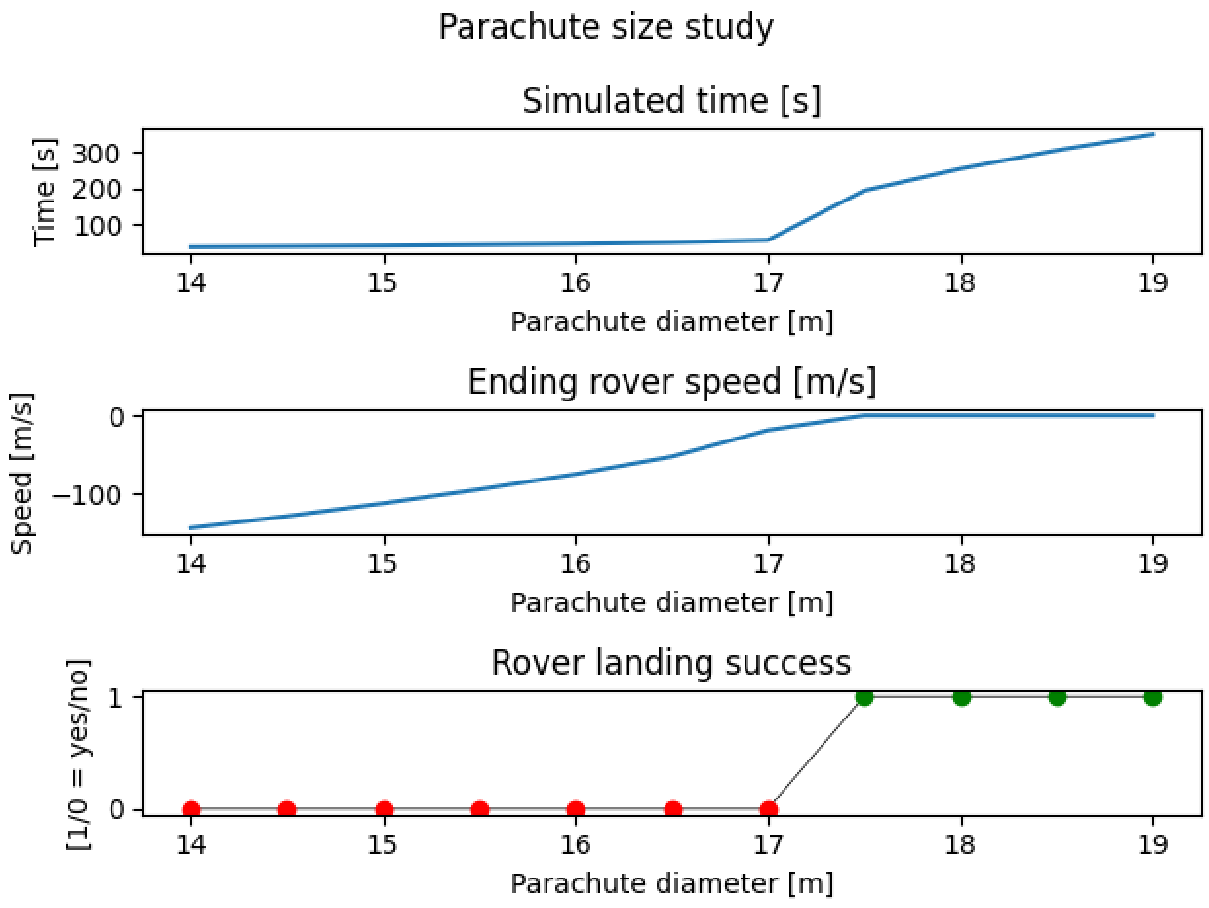


Figure 5: Parachute size study using the new drag model