ECE 484: Principles of Safe Autonomy Fall 2023

Written by: ECE484 team *HW0, MP0: Reachability, Safety, Invariants and Design*

Website 

Due Date: 9/8/23

1 Introduction

Welcome! In this first assignment, you get to practice an invariance proof, play with simple simulations for testing a safety requirement, and design using these ideas.

In part one (Section 2), you will use the inductive method to prove safety of an Automatic Emergency Braking scenario. You will need to show that, under some assumptions, the system is indeed safe, i.e., the vehicle will never hit the pedestrian. In Section 3, you will create a decision logic, which helps the vehicle avoid collisions with pedestrians and maximize average speeds.

Problems 1-4 must be done individually and Problem 5 can be done in groups of 2-3 people. All the regulations for academic integrity and plagiarism spelled out in the student code apply.

Learning objectives

- Know and use inductive method for proving invariants
- Basics of reachability analysis
- Exercise safe design thinking with tests and reachability analysis

System requirements

- Ubuntu 20.04, Python 3.8 or above, Verse library

2 Safety Analysis of Automatic Emergency Braking

We define an explicit model of scenario involving a vehicle and a pedestrian as shown in Figure 1. In this model, x_1, v_1, x_2, and v_2 are state variables. x_1 and v_1 correspond to the position and velocity of our vehicle. x_2 and v_2 correspond to the position and velocity of the pedestrian. We use the convention that $d(t)$ is the *valuation* of the state variable at time t. That is, $d(0) = x_2(0) - x_1(0) = x_{20} - x_{10}$, $d(1)$ is the value of d after the program is executed once, $d(2)$ after the program is executed a second time, and so on. Similarly, we can refer to other state variables in the same manner e.g. $x_1(t)$ and $x_1(t+1)$ refer to the valuation of x_1 in two different time instances. D_{sense} is a constant, which is the sensing distance: if $d(t) \leq D_{sense}$, the vehicle applies the brakes and decelerates.

```
1  SimpleCar(D_sense, v_0, x_10, x_20, a_b), x_20 > x_10
2  Initially: x_1(0) = x_10, x_2(0) = x_20, v_1(0) = v_0, v_2(0) = 0
3  s(0) = 0, timer(0) = 0, timer2(0) = 0
4  d(t) = x_2(t) − x_1(t)
5  if d(t) ≤ D_sense
6      s(t + 1) = 1
7      if v_1(t) ≥ a_b
```

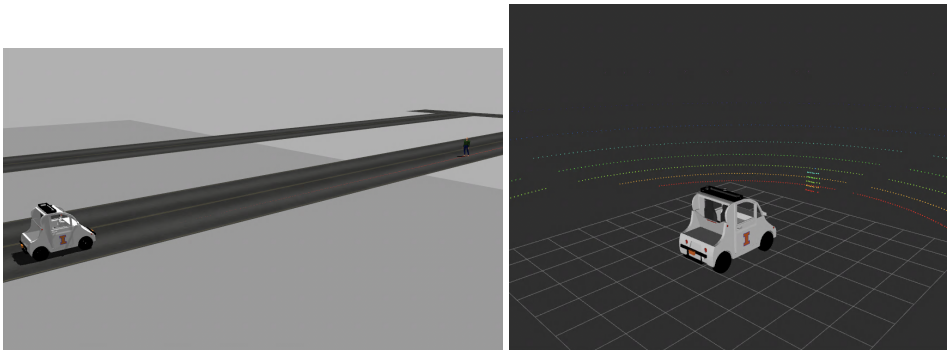


Figure 1: Vehicle and pedestrian on a single lane. *Left:* Initial positions of the two are at x_{10} and x_{20}, respectively. *Right:* Vehicle undergoes constant deceleration a_b once the pedestrian is detected by the vehicle.

```
                $v_1(t+1) = v_1(t) - a_b$
                $timer(t+1) = timer(t) + 1$
                $timer2(t+1) = timer2(t)$
            else
                $v_1(t+1) = 0$
                $timer(t+1) = timer(t)$
                $timer2(t+1) = timer2(t)$
        else
            $s(t+1) = 0$
            $v_1(t+1) = v_1(t)$
            $timer(t+1) = timer(t)$
            $timer2(t+1) = timer2(t) + 1$
        $x_1(t+1) = x_1(t) + v_1(t)$
```

We will use the above model and we call it model $\mathcal{A}$. Consider the following invariant:

Invariant 1. Over all executions of $\mathcal{A}$, $timer(t) + v_1(t)/a_b \leq v_0/a_b$.

Problem 1 (30 points)**.** Is $0 \leq v(t) \leq v_0$ an invariant of $\mathcal{A}$? No need to write a complete proof; a two sentence argument would suffice.

Problem 2 (30 points)**.** Is $timer(t) \leq v_0/a_b$ an invariant of $\mathcal{A}$? Explain why. Can we use the induction method to prove this invariant? If so, present your proof.

Hint: You may find the usage of other invariants handy in your proof.

Problem 3 (40 points)**.** Let us now introduce some delay in the sensing-computation-actuation pipeline, say T_{react}. This could model cognitive delay of a human driver or processing delay in electronics and computers. Assume we have exactly T_{react} seconds delay between the sensing of the pedestrian and the application of the brakes (the **start** of the deceleration). Moreover, let us also introduce acceleration to the vehicle: whenever the vehicle is outside the sensing distance, the vehicle undergoes constant acceleration a_s. Rewrite the new updated model with the sensing delay and vehicle acceleration.

Problem 4 (10 bonus points)**.** Identify additional assumptions on $x_{20}, x_{10}, D_{sense}$ under which the system is safe. That is, come up with an new invariant such that you can prove this invariant using the method

of induction, the assumptions, and the previously proved invariants. (Note: This question is much more difficult than previous one. If you get stuck with it, try to solve coding section first then get back to this one)

Hint: Try the assumptions: $x_{20} - x_{10} \geq D_{sense}$ and $D_{sense} > v_0^2/a_b + 2v_0$ and the invariant $d(t) > 0$.

Submission Instructions Problems 1-4 must be done individually (Homework 0). Write solutions to each problem in a file named `<netid>_ECE484_HW0.pdf` and upload the document in Canvas. Include your name and netid in the pdf. This should be individual work. You may discuss concepts with others, but not use written notes from those discussions. If you use/read any material outside of those provided for this class to help grapple with the problem, you should cite them explicitly.

3 Testing Automatic Emergency Braking with Simulations

You are in charge of designing the *autmatic emergency braking (AEB) decision logic (DL)* for a car. You will write the DL code and test it. You will have to provide evidence that will convince auditors that your design is safe. The design should not be too conservative, i.e., hard-brake all the time, and try to achieve high average speed.

3.1 Scenario Description and Assumptions

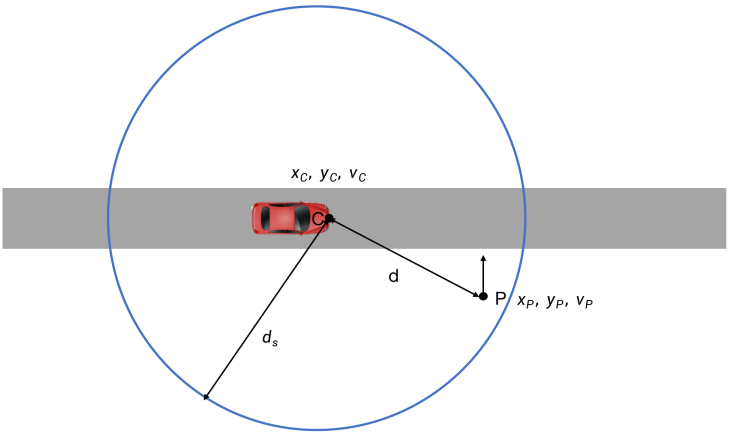


Figure 2: The vehicle-pedestrian scenario illustration.

The scenario is shown in Figure 2. There are two agents: the car (**C**) and the pedestrian (**P**). **C** is moving down the straight road with its initial position (x_c, y_c) and initial velocity v_c. **P** is moving across the road with its position (x_p, y_p) and its velocity v_p. The dynamics of **C** is the kinematic bicycle model. **C** has a sensor that detects **P** when the two agents are within $d_s (= 60m)$ distance of each other. When **P** is within d_s distance, the DL function will have the exact euclidean distance between the **C** and **P**. Otherwise, it gets an arbitrary large value.

C can run in four modes: `Normal` mode keeps the velocity constant; `Brake` and `HardBrake` induce braking with different deceleration a; `Accel` speeds-up **C**.

The DL sets the mode for **C** based on different conditions on the variables. For example:

```
def decisionLogic(ego: State, other: State):
    output = copy.deepcopy(ego)
    ## When C and P <  12 meters away, then Brake:
```

```
    if ego.agent_mode == VehicleMode.Normal and other.dist < 12:
        output.agent_mode = VehicleMode.Brake
    ## Safety assertion:
    assert other.dist > 2.0
    return output
```

Listing 1: Baseline Decision Logic

This DL establishes that when **C** is in Normal mode and the distance between **C** and **P** is smaller than 12m (note 12m is less than d_s), **C** will start to brake. The `assert` is a standard way to state requirements in Python. Here the safety requirement is the **C** has to be at least 2m away from the **P**. Whenever the `assert` is violated, the simulation will stop.

3.2 Design Targets

Create a single DL so that the system is *safe* starting from any initial condition in a given range **R**, up to a time horizon $T_s = 50s$.

Given a deterministic DL and a particular initial condition in **R**, the complete system has a unique *execution*. An execution is *safe* if **P** is never within 2m distance from **C** within T_s. For this MP, there are three **R**'s of increasing difficulty; you have to submit a single DL.

- $\mathbf{R_1}$: $x_c, y_c \in [-5, 5]$; $v_c = 8$; $x_p = 175$; $y_p = -55$; $v_p = 3$.
- $\mathbf{R_2}$: $x_c, y_c \in [-5, 5]$; $v_c \in [5, 10]$; $x_p = 175$; $y_P = -55$; $v_P = 3$.
- $\mathbf{R_3}$: $x_c, y_c \in [-5, 5]$; $v_c \in [5, 10]$; $x_p \in [165, 175]$; $y_p \in [-55, -50]$; $v_P = 3$.

Design the DL to ensure that all executions starting from $\mathbf{R_i}$ are safe as well as to maximize the average speed of **C** within T_s. Of course, designing for $\mathbf{R_2}$ is harder than that for $\mathbf{R_1}$ and so forth. So, you get higher points for a design that works for a higher $\mathbf{R_i}$. The rest of the document describes the files you have to work with, the submission process, and some of the functions you can use for testing.

3.3 Documentation of Provided Files

We have pre-installed necessary libraries in the lab machines, so you don't have to install any additional packages. To get MP0's source code, run the below commands:

```
git clone https://gitlab.engr.illinois.edu/GolfCar/mp-release-23fa.git
```

Listing 2: Retrieving MP0 Code

The supporting code is available from this git repo in `./src/mp0`. The important files are:

The file **vehicle_controller.py** contains (1) the definition of modes of agents (do not change); (2) definition of state variables of agents (do not change); (3) decision logic for automatic emergency braking. You will edit this last part only.

What you can and cannot write in DL. You will write a DL of the same type as in Listing 1 within the function `decisionLogic`. The inputs to the function are `ego`, the full state of **C** (`ego.x`, `ego.y`, `ego.v`) and `other`, the sensed state of **P** (`other.dist`).

The DL has to be straight-line if-then-else code. In the `if` condition, you can only write linear inequalities or equalities, and Python logical operators. In the `if` body, you can assign the `output.agent_mode` to one of `Normal`, `Brake`, `HardBrake`, and `Accel` .

vehicle_pedestrian_scenario.py This file contains the mechanism to create different vehicle-pedestrian scenarios. The three different initial ranges $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$ are provided in this file. In addition, we provide function `sample_init` for randomly sampling points from the initial range to perform different simulations of the scenario. You can change the number of samples to simulate or modify how the initial points are generated in this function.

3.4 Simulations and Testing

Run **vehicle_pedestrian_scenario.py** to generate simulations.

```
cd mp-release-23fa/mp0/src/
python3 vehicle_pedestrian_scenario.py
```

Listing 3: Command to run simulations

Here are the key functions.

`trace = scenario.simulate(time_horizon, time_step)`: Runs a single simulation from a randomly sampled initial point chosen from **R**. When the simulation is unsafe, you will get message `assert hit for car` in the terminal.

- `time_horizon`: Float, the total time for the scenario to evolve.
- `time_step`: Float, the time sampling period.
- `trace`: AnalysisTree object containing the simulated trajectory. The trajectory can be visualized using the `simulation_tree_3d` function. The AnalysisTree object holds a list of AnalysisTreeNode that contains execution of the scenario for each mode of **C**. To access the exact trajectory of **C** in a mode, one can do `AnalysisTree.nodes[i].trace['car']`. The trace will be in the form of a numpy array.

`init_list=sample_init(scenario, num_sample)`: Samples initial points from the scenario's initial range.

- `scenario`: the current scenario to sample initial point from.
- `num_sample`: Int, the number of initial point sampled from initial range.
- `init_list`: List[Dict], a list of dictionary containing different initial points to perform simulation. The key of the dictionary is `car` and `pedestrian` and the value of the dictionary is a list of Float specifying the initial point for **C** or **P** sampled from the initial range of the input scenario.

`traces = scenario.simulate_multi(time_horizon, time_step, init_dict_list)`: Perform multiple simulations starting from different initial points.

- `time_horizon`: Float, the total time for the scenario to evolve.
- `time_step`: Float, the time period that the state of the scenario is sampled.
- `init_dict_list`: List[Dict], the initial dict generated by the `sample_init` function
- `trace`: List[AnalysisTree], a list of AnalysisTrees containing the simulated trajectory for each of the inital point.

`fig = simulation_tree_3d(tree, fig, x, x_title, y, y_title, z, z_title)`: Visualize result from `scenario.simulate` and `scenario.simulate_multi`.

- `tree`: The resulting trajectory generated by the `scenario.simulate` or `scenario.simulate_multi` functions.
- `fig`: Figure object, the figure object to plot on.

- `x,y,z`: Int, the index of x (or y, z) dimension to be plotted.
- `x_title, y_title, z_title`: Str, the x, y, z axis label
- `fig`: Figure object. A 3d plot with tree axes as time and x,y position of **C** and **P**. The plot can be shown using the `fig.show()` function as shown in Listing 4 and Listing 5

`avg_vel, unsafe_frac, unsafe_init = eval_velocity(traces)`: Compute average velocity and count the number of unsafe simulations.

- `traces`: List, the list of simulation trajectories generated by `scenario.simulate_multi` function
- `avg_vel`: Float, the average velocity of **C** among all simulations.
- `unsafe_frac`: Float, the fraction of unsafe simulation among total number of simulations
- `unsafe_init`: List, the list of initial points that are unsafe.

Examples usages are presented in Listing 4 and 5. The corresponding plots are presented in Figure 3. Note that, although the system is simulated to be safe for $\mathbf{R_1}$, the same decision logic may fail for a different **R** as shown in Figure 4 for $\mathbf{R_2}$.

```
trace = scenario.simulate(50, 0.1)
fig = go.Figure()
fig = simulation_tree_3d(trace, fig, 0,'time', 1,'x',2,'y')
fig.show()
```

Listing 4: Code for a single simulation.

```
init_dict_list= sample_init(scenario.init_dict, num_sample=50)
traces = scenario.simulate_multi(50, 0.1,init_dict_list=init_dict_list)
fig = go.Figure()
for trace in traces:
    fig = simulation_tree_3d(trace, fig,0,'time', 1,'x',2,'y')
fig.show()
avg_vel, unsafe_reac, unsafe_init = eval_velocity(traces)
```

Listing 5: Code for mutiple simulations.

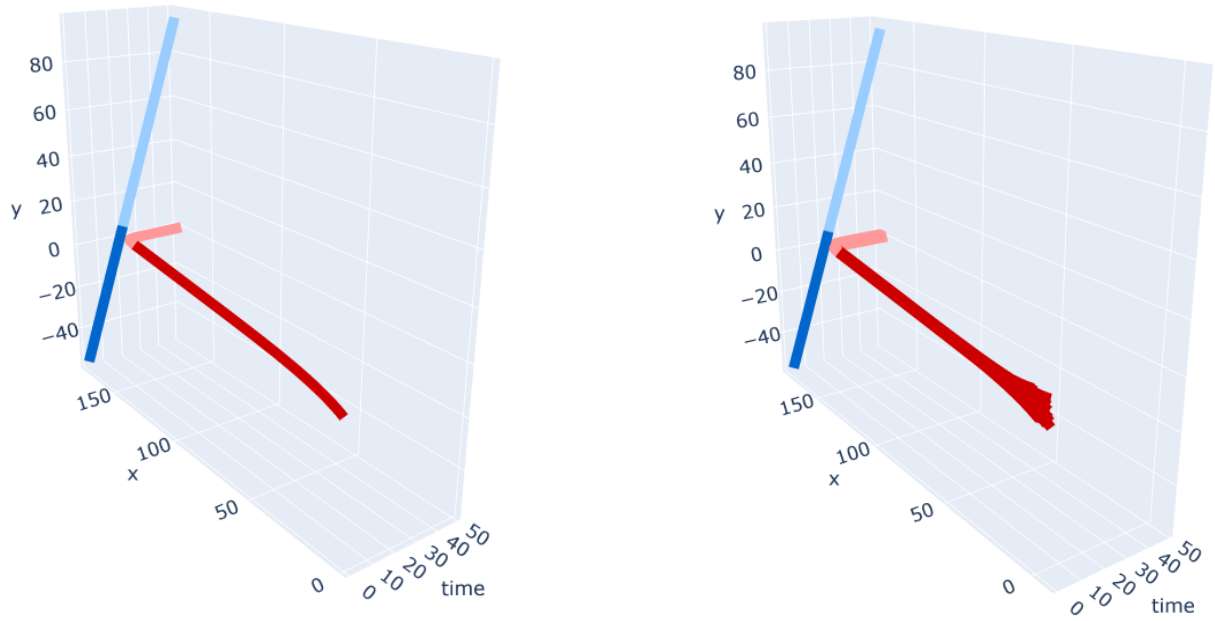
3.5 Reachability analysis

You can use *reachability analysis* to perform more thorough check on your DL. Recall, reachability over-approximates all possible executions, and so, if the analysis says the system is safe then you are guaranteed safety, but if it says unsafe, that does not necessarily mean that there is a counter-example. Reachability analysis from a smaller initial set usually results in a tighter (less conservative) approximation.

If the DL is not safe for entire **R**, then you can manually partition **R** into smaller sub-regions and present evidences of your DL's safety in these sub-regions.

`traces, safe, unsafe = verify_refine(scenario, time_horizon, time_step)`: Compute reachable set and check the safety of the scenario. Due to the over-approximation, reachability analysis may produce spurious counter examples. In this case, `verify_refine` can automatically partition the initial ranges into smaller regions to get more precise reachability analysis result.

- `scenario`: the scenario to verify.
- `time_horizon`: Float, the total time to perform reachability analysis.
- `time_step`: Float, the time period that the reachable set is sampled.
- `traces`: AnalysisTree, the computed reachtube for the scenario. The data structure is the same as that produced by `scenario.simulate`.



(a) Plot for a single simulation.

(b) Plot for multiple simulations.

Figure 3: Simulation of scenario starting from $\mathbf{R_1}$. The red curve represents $\mathbf{C}$'s trajectory, while the blue represents the pedestrian's. We can see from both plots that as $\mathbf{C}$ approaches $\mathbf{P}$, it will slow down and eventually stop as described in the decision logic in Listing 1.

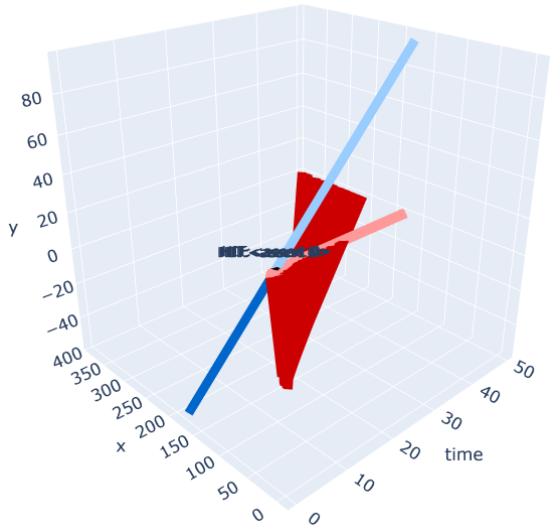
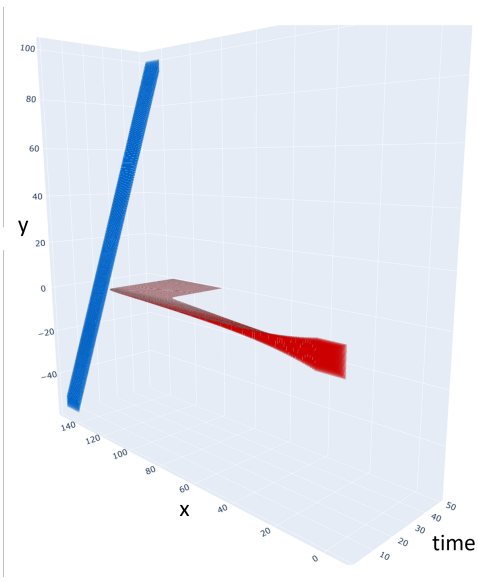
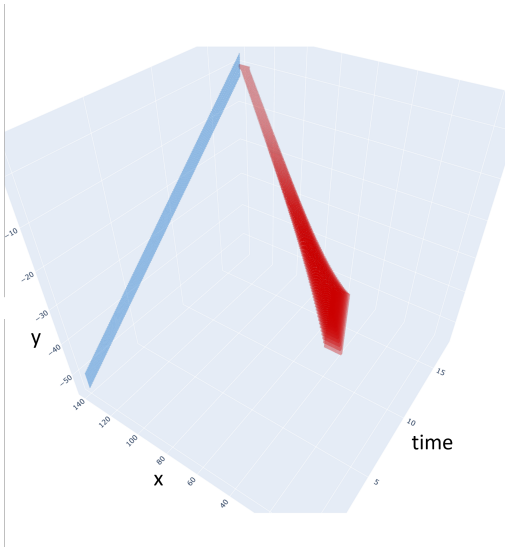


Figure 4: The baseline decision logic in Listing 1 when simulated on $\mathbf{R_2}$ leads to unsafe behavior as $\mathbf{C}$ is not able to brake in time to avoid collision with $\mathbf{P}$ (note the difference in velocity range of $\mathbf{C}$ between $\mathbf{R_2}$ and $\mathbf{R_1}$).



(a) Reachtube analysis that guarantees safety.



(b) Reachtube analysis that can't guarantee safety.

- `safe`: List, the subregions that are proved to be safe after partition.
- `unsafe`: List, the subregions that are still unsafe.

`fig = reachtube_tree_3d(tree, fig, x, x_title, y, y_title, z, z_title)`: Visualize reachtubes computed by `verify_refine`.

- `tree`: The resulting trajectory generated by the `verify_refine` function.
- `fig`: Figure object, the figure object to plot on.
- `x, y, z`: Int, the index of x (or y) dimension to be plotted.
- `x_title, y_title, z_title`: Str, the x, y, z axis label
- `fig`: Figure object. A 3d plot with tree axes as time and x,y position of **C** and **P**. The plot can be shown using the `fig.show()` function as shown in Listing 6.

```
traces = verify_refine(scenario, 50, 0.1)
fig = go.Figure()
fig = reachtube_tree_3d(traces, fig, 0,'time', 1,'x',2,'y')
fig.show()
```

Listing 6: Code for the reachtube simulation.

Listing 6 shows an example of performing reachability analysis and generating plots using above functions. As we can see from Figure 5a, the reachtubes of **C** and **P** don't overlap with each other by any part. This means the trajectories of **C** and **P** can't intersect under all possible scenarios, and thus safety gets guaranteed. In comparison, the two reachtubes touch each other in Figure 5b, which means the trajectories of **C** and **P** can potentially intersect. Hence, safety cannot be guaranteed.

3.6 Design Outcomes & Grading

Problem 5 (45 points)**.** (a) (15 points) In your write-up, present your evidence for safety of your design of the DL. Which $\mathbf{R_i}$ do you claim you DL is safe for? What is the evidence for safety? Evidence could be a set

of tests or simulations. It could also be the computation or over-approximation of the reachable states from $\mathbf{R_i}$, or it could be an invariant proof with appropriately stated assumptions.

(b) (15 points) What is the average speed you achieve over $\mathbf{R_i}$? How do you justify this answer?

(c) (15 points) In 3 or fewer sentences, describe the basic idea of your DL design for achieving safety and high average speed.

Auto-grading Score: 45 pts. Specifically, there are 10 pts on $\mathbf{R_1}$, 15 pts on $\mathbf{R_2}$, and 20 pts on $\mathbf{R_3}$. The percentage of score you can get on each $\mathbf{R_i}$ will be calculated as Listing 7. As you can see, we use 7 m/s as a threshold to evaluate your DL's performance on speeds.

```
autograded_part_score = % of successful hidden simulation tests * (1 - penalty)
penalty = 0.5 if avg_vel <= 7(threshold) else 0
```

Listing 7: Auto-grading Scoring Formula.

Problem 5 can be done in groups of 2-3. Submit a single report with name `MP0_<groupname>.pdf` to Canvas. Please include the names and netids of all the group members and cite any external resources you may have used in your solutions.

For DL, rename your `vehicle_controller.py` to `netid1_netid2_netid3.py` and submit ONLY that python file to auto-grading site. You can submit for an unlimited times before the deadline, and we will grade the latest version.

Demo Attendance: 10 pts. Attend your lab session on September 8th to demo your design logic. Every group member needs to show up. We will ask questions regarding your DL, and every group member should show his or her understanding of it.

3.7 Submissions

The maximum one can get from this HW is 110pts (including 10pts bonus). The maximum one can get from this MP is 100 pts.

1. (110pts) Everyone submits individual `<netid>_ECE484_HW0.pdf` on Canvas
2. (45pts) Only one group member needs to submit `MP0_<groupname>.pdf` on Canvas
3. (45pts) Only one group member needs to submit `netid1_netid2_netid3.py` on auto-grading site
4. (10pts) Everyone needs to attend demo, and this part will be graded individually

1-3 are due 11:59pm CST 9/8. 4 is due by the time of your discussion section