ECE 484: Principles of Safe Autonomy	Spring 2025
Written by: ECE484 team	*MP0B: Part 1*
Website 	*Due Date: February 7, 2025*

1 Introduction

In this MP, you must complete two design problems using a tool called Verse [1]. This MP is part of a research project in which we are trying to evaluate Verse, and therefore, we will perform a user study. If you choose to participate in this user study, you must complete a consent form and 4 surveys. **The MP is mandatory regardless of whether or not you chooose to participate in the survey**

Your answers in the surveys will not impact your grades in ECE 484 in any way. We are asking for your name and email for possibly conducting a follow-up interview. Once collected the data will be anonymized. None of your personal information will be published or available to anyone outside of the small group of researchers conducting this study.

The design tool Verse has several functions that can be useful for designing vehicle controllers. Please note that your assignment may differ from your classmates. **Please work by yourself and only use the functions of the tool provided for the problem you are currently working on**.

Along with this MP, each student must individually complete Homework 0 regardless of whether they choose to participate in the user study. HW 0 is due at the same time as the MP.

2 Homework 0: Safety Analysis of Automatic Emergency Braking

HW Problem 1 (5 points) **[Individual]** For an automaton $\mathcal{A} = \langle Q, Q_0, D \rangle$, and an unsafe set $U \subseteq Q$, (a) Is Q an inductive invariant? (b) Write down the conditions for an set $I \subseteq Q$ to be an inductive invariant that helps prove safety with respect to U.

For the remaining problems, we define an automaton model $\mathcal{A}$ 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 $x_1(t)$ to refer to the value of x_1 at time t along a fixed execution, and similarly for other variables. So, $d(0) = x_2(0) - x_1(0) = x_{20} - x_{10}$, $d(1)$ 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 sensing distance: if $d(t) \leq D_{sense}$, the vehicle applies the brakes and decelerates.

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

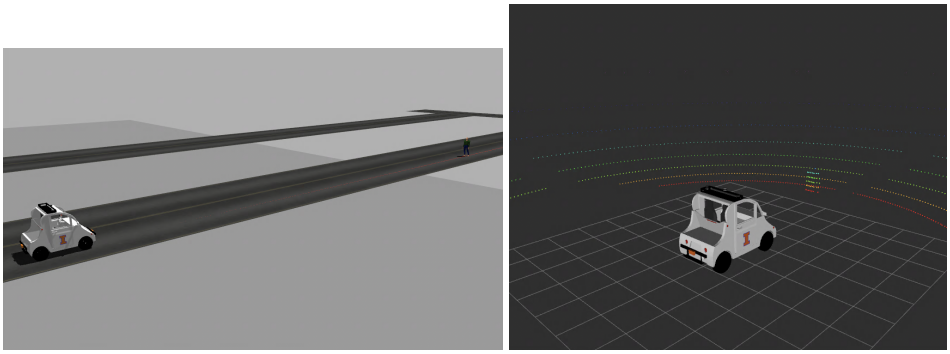


Figure 1: Vehicle and pedestrian on a single lane. *Left:* Initial positions of the two are at x_{10} and x_{20}, respectively. *Right:* Vehicle undergoes constant deceleration a_b once the vehicle detects the pedestrian.

```
                timer(t+1) = timer(t)
                timer2(t+1) = timer2(t)
    else
        s(t+1) = 0
        v1(t+1) = v1(t)
        timer(t+1) = timer(t)
        timer2(t+1) = timer2(t) + 1
     x1(t+1) = x1(t) + v1(t+1)
```

Consider the following invariant for $\mathcal{A}$:

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

HW Problem 1 (5 points) **[Individual]** Is $0 \leq v_1(t) \leq v_0$ an invariant of $\mathcal{A}$? No need to write a complete proof; a two-sentence argument would suffice.

HW Problem 2 (10 points) **[Individual]** 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.

HW Problem 3 (15 points) **[Individual]** Let us introduce some delay in the sensing-computation-actuation pipeline, say T_{react}. This could model the 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. Hint: the vehicle should still accelerate during the delay

HW Problem 4 (5 BONUS points) **[Individual]** Identify additional assumptions on $x_{20}, x_{10}, D_{sense}$ under which the system is safe. That is, come up with a 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 the previous one. If you get stuck with it, try to solve the 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 $d(t) > 0$.

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 file. This should be individual work and you should follow the student code of conduct. You may discuss solutions with others, but do not use written notes from those discussions to write your answers. 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 MP 0 Setup

Verse is installed on all of the ECEB 5072 Ubuntu computers. Because of the large number of students, we recommend doing the MP on your own machine. Visit this gitlab and follow the readme instructions for installation. It is also recommended to create a virtual environment to make package installation easier.

To get the code for the design problems, run the below commands:

```
git clone https://gitlab.engr.illinois.edu/GolfCar/mp0b-fa24.git
```

Listing 1: Retrieving MP0 Code

4 MP 0 Design Problems

You will be given two design problems to complete, Design Problem 1: Emergency Automatic Braking (DP1) and Design Problem 2: Traffic Intersection Controller (DP2). Each problem will require three tasks: (a) design a decision logic, (b) demonstrate safety, and (c) take a survey about your experience. After you finish your code for each decision problem, you should save it for submission, and do not update it once you move on to the next problem. You will also prepare a write-up to demonstrate the safety and submit it as a PDF file. Problems 3 and 5 have questions you will answer in your write-up.

The ordering of the problems in this MP is very important. Please complete the steps in order and do not return to the previous steps.

See Figure 2 below. Finish each checkbox before moving on and do not return to previous steps.

5 Pre-Assignment Forms

If you are participating in the user survey, these forms must be completed before starting any design problems. If you are not participating in the survey, you do not have to fill out any forms

Problem 1. Consent form: You should have already filled out the consent form. You do not need to do this again.

Problem 2. Pre-survey: Fill out the pre-assignment survey.

References

[1] Yangge Li, Haoqing Zhu, Katherine Braught, Keyi Shen, and Sayan Mitra. Verse: A python library for reasoning about multi-agent hybrid system scenarios. In *International Conference on Computer Aided Verification*, pages 351–364. Springer, 2023.

Please do not move on unless you have completed Step 1.

Step 1	☑Compete the Pre-Survey Problems 1 and 2
Step 2	❑DP1 Emergency Automatic Braking Complete Decision Logic code Add Problem 3 to write up
Step 3	❑Compete DP1 Survey Problem 4
Step 4	❑DP2 Traffic Intersection Controller Complete Decision Logic code Add Problem 5 to write up
Step 5	❑Compete DP2 Survey and Final Survey Problems 6 and 7
Step 6	❑Submit writeup to Canvas and upload code

Figure 2: Problem Ordering

6 DP1 : Automatic Emergency Braking

You are in charge of designing the *autmatic emergency braking (AEB) decision logic (DL)* for a car. You will write the DL and provide evidence for its safety. Essentially, you will determine when to switch between the modes (Normal, Brake, Accelerate, Hard Brake). The design should not be too conservative, i.e., hard-brake all the time, and try to achieve a high average speed.

6.1 Scenario Description and Assumptions

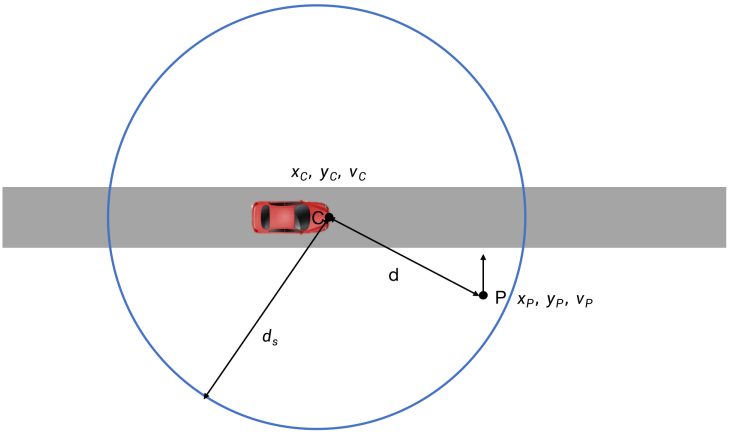


Figure 3: The vehicle-pedestrian scenario illustration.

The scenario is shown in Figure 3. 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. **C** follows the kinematic bicycle model and 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 access to 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 2: 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 way to state safety requirements. Here the requirement is that **C** has to be at least 2m away from the **P**. Whenever the `assert` is violated, the simulation will stop. **Do not change the safety assertion or modify output.x/output.y You will be severely penalized if you do so.**

6.2 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 task, there are three **R**'s of increasing difficulty; you have to submit a single DL.

- $\mathbf{R_1}$: $x_c, y_c \in [-5, 5]$; $v_c = 8$; $x_p = 175$; $y_p = -55$; $v_p = 3$.
- $\mathbf{R_2}$: $x_c, y_c \in [-5, 5]$; $v_c \in [7.5, 8.5]$; $x_p = 175$; $y_P = -55$; $v_P = 3$.
- $\mathbf{R_3}$: $x_c, y_c \in [-5, 5]$; $v_c \in [7.5, 8.5]$; $x_p \in [173, 176]$; $y_p \in [-55, -53]$; $v_P = 3$.

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. $\mathbf{R_3}$ is harder than $\mathbf{R_2}$ which is harder than $\mathbf{R_1}$. So a design that works for a higher i will reward you with more points in both the autograder and report. The rest of the section describes the files you have to work with.

6.3 Documentation of Provided Files

The code for this problem is in `./highway_brake`. 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 2 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`). You may not access the full state of the **P**, only the sensed distance.

The DL has to be straight-line if-then 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`. You may not directly change the state variables of `ego` or `other`.You also may not directly change `output.x` or `output.y`.

In short, you may only use:

- if statements
- arithmetic operators (+,-,/,*)
- logical operators (and, or , not)
- comparison operators (<=, <, ==, ! =, >, >=)
- functions

- assertions (for defining safety)
- "any" keyword
- "all" keyword

This means that the following are not supported (don't even try it):

- print statements
- numpy functions
- exponents/roots
- loops (except any and all statements)
- variables declared behind an if statement

  It is also highly recommended not to use else/elif statements

The DL is written like python code but, it is not actually being run by the python interpreter. Instead, the code is parsed and analyzed separately by Verse without running the code. This is why print statements do not work! The DL is analyzed in the following way:

Each if statement creates a logical branch. At the end of each branch, there needs to be a mode transition or a state change. All branches that logically evaluate to true will be verified simultaneously. If 2 branches evaluate to true, then they are both analyzed at the same time, creating a "fork" in the plot.

More Verse Troubleshooting Please visit the Verse help and troubleshooting page (especially if you run into an infinite loop)

For more examples of how to write/format the decision logic, visit the demo folder. You may run any of the scenario files to get a sense of how Verse is organized.

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. **You will run this file to test your DL.**

7 DP1 Design Tool Functions: Simple Simulation and Reachability

You can use *reachability analysis* to examine the behavior of your DL. Recall that reachability over-approximates all possible executions in the initial set. If the analysis says the system is safe, then you are guaranteed safety. If unsafety is detected, that does not necessarily mean the scenario is unsafe. The reachability analysis could have over-approximated the reachable set too much. It is possible that the tool generated an approximation of the reachable set that is much larger than the actual reachable set. Reachability analysis from a smaller initial set usually results in a tighter (less conservative) approximation. The more conservative the approximation is, the less precise it is.

You may also use the simulation function to directly run your decision logic and compute an average velocity of your vehicle.

We expect the function `verify_refine()` to run for a very long time, 40 min+, for larger sets such as $\mathbf{R_3}$. If nothing is verified safe after a partition depth of around 5, it is recommended to abort and adjust your decision logic. All other functions should run in a few minutes or seconds. You may consider consulting the troubleshooting page if an infinite loop appears.

You may **only** using the following functions of the design tool to test your DL. You may also write your own functions that use these functions.

`trace = scenario.simulate(time_horizon, time_step)`: Runs a single simulation from a randomly sampled initial point chosen from **R** and returns the trajectory. 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 contains a list of AnalysisTreeNode that contains the 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.

`traces = scenario.simulate_multi(time_horizon, time_step, init_dict_list)` : simulate from all initial points in the init_dict_list instead of randomly choosing a point from the initial set. The output will also be a list of traces instead of just one trace.

- `time_horizon`: Float, the total time for the scenario to evolve.
- `time_step`: Float, the time sampling period.
- `init_dict_list`: List[Dictionary], list of initial points to simulate from. For example, an init_dict_list of [{'car': [-5,25,6,8], 'pedestrian':[170,-53,0,3]}] will simulate starting from the car state [-5,25,6,8] and pedestrian state [170,-53,0,3].
- `traces`: List, list of traces. A trace is explained above in `scenario.simulate`

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

- `tree`: The resulting trajectory generated by the `scenario.simulate_simple` function.
- `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 **L**. The plot can be shown using the `fig.show()` function as shown in Listing 3.

`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. A single simulation is generated by `scenario.simulate_simple` 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.

`traces = scenario.verify(time_horizon, time_step)`: Compute reachable set and check the safety of the scenario.

- `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. Has similar structure as that produced by `scenario.simulate_simple`.

`traces = verify_refine(scenario, time_horizon, time_step)`: Compute reachable set and check the safety of the scenario. Due to the over-approximation, the `verify()` function may create a reachable set that is too big. In this case, `verify_refine()` can partition the initial ranges into smaller regions to get a more precise reachability analysis result. It will attempt to continually partition the initial set into smaller and smaller regions until the result is safe. Each level of the partition depth increases the number of calls to `verify()` exponentially. Thus, this function will take much longer to run than `verify()`, but is more precise and may resolve issues related to infinite looping.

- `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`: List[AnalysisTree], the computed reachtube for the scenario. Has similar structure as that produced by `scenario.simulate_simple`.

`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 the car and the traffic light. The plot can be shown using the `fig.show()` function as shown in Listing 3.

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

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

Listing 3: Code for the simple simulation and reachtube generation.

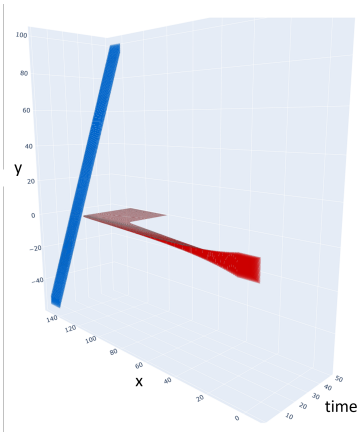
Listing 3 shows an example of performing reachability analysis and generating plots using above functions. If a safety assertion is violated, then a label will appear on the plot such as in Figure 4b and a message will appear in the console.

8 DP1 Design Outcomes & Grading

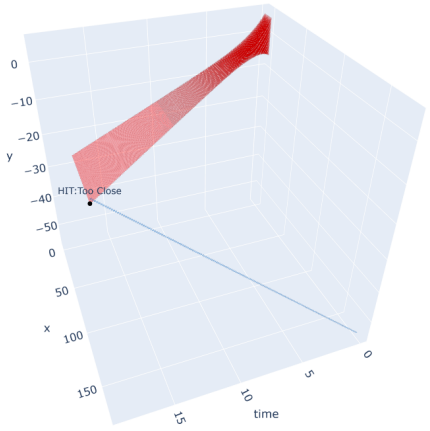
Problem 3 (15 points)**.** (a) (5 points) What is the highest $\mathbf{R_i}$ that you claim you DL is safe for? What is the evidence for safety? Evidence should be provided using reachability analysis. You can provide this evidence using the `verify() or verify_refine()` function. Please provide an image of the reachability plot with no safety violations.

(b) (5 points) What is the average speed you achieve over this $\mathbf{R_i}$? How do you justify this answer?

(c) (5 points) In 3 or fewer sentences, describe the basic idea of your DL design for achieving safety and high average speed.



(a) Reachtube analysis that guarantees safety.



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

Figure 4: Reachability results

Auto-grading Score: 15 pts. Specifically, there are 5 pts on $\mathbf{R_1}$, 5 pts on $\mathbf{R_2}$, and 5 pts on $\mathbf{R_3}$. The percentage of score you can get on each $\mathbf{R_i}$ will be calculated as Listing **??**. **We use 7 m/s as a threshold to evaluate your DL's performance on speeds.** The autograder will run simulate() on your decision logic many times, including some edge cases. Reachability is not run by the autograder. As you can see, the speed penalty gradually increases, but the penalty for any unsafety detected is very harsh (at least 50%). Be sure to test your DL thoroughly.

```
autograded_part_score = min(average simulation vel/ threshold, 1) * (1 - penalty)
penalty = (0.5 + min(% of failed simulation test cases, 0.5) ) if any unsafety detected else 0
```

Listing 4: Auto-grading Scoring Formula.

Please add Problem 3 to your write up and save your code for submission.

Demo Attendance: 10 pts. Attend your lab session on February 7 to demo your design logic. We will ask questions regarding your DL and your understanding of it.

9 DP1 Survey

After you complete your DL and write-up, you will complete a survey about your experience.

You should complete all your code for DP1 and the write-up for problem 3 before completing the survey.

Problem 4. Consider your experience using the verify function to complete this design task for the automatic braking scenario. Complete the Midpoint Survey (B1).

Please do not move on unless you have completed Steps 1-3. If you are not participating in the study, you do not need to fill out any forms

Step 1	☑Compete the Pre-Survey Problems 1 and 2
Step 2	☑DP1 Emergency Automatic Braking Complete Decision Logic code Add Problem 3 to write up
Step 3	☑Compete DP1 Survey Problem 4
Step 4	❑DP2 Traffic Intersection Controller Complete Decision Logic code Add Problem 5 to write up
Step 5	❑Compete DP2 Survey and Final Survey Problems 6 and 7
Step 6	❑Submit writeup to Canvas and upload code

10 DP2 Traffic Intersection Controller

You will design a safe decision logic (DL) for a car (**C**) to drive through a simple, straight line, intersection with a traffic light (**L**). You will have to provide evidence that will convince auditors that your design is safe. The setup is at follows: There is a traffic light that cycles through the lights, green, yellow, red, green,. . . , with dwell times, $t_g, t_a, t_r, t_g, \ldots$ respectively. There are two regions associated with the light that are important for safety:

- *Entrance* $[d - d_{in}, d - d_{out}]$. The car cannot be in this region if the light is red.
- *Exit* $[d - d_{out}, d]$. The car cannot be slow in this region if the light is red.

The scenario is shown in Figure 5.

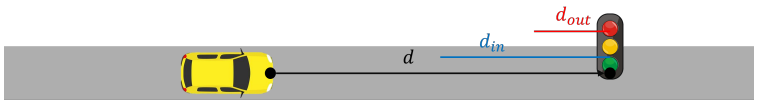


Figure 5: The traffic scenario illustration.

10.1 Scenario Description and Assumptions

You will have to write the decision logic for the car as in Listing 5. In addition to meeting these safety requirements, the car should also minimize travel time over a given distance D that lies beyond the intersection. You vehicle should try and maximize velocity when not stopping for a red light.

```
def decisionLogic(ego: State, other: State):
    output = copy.deepcopy(ego)
    ## Use other.signal_mode == TLMode.RED, TLMode.AMBER, TLMode.GREEN to check status of light
    ## Use other.x for position of light
    <Write your logic here>

    assert ## Safety: No entering when red
    not (other.signal_mode == TLMode.RED and (ego.x > other.x-d_in and ego.x < other.x-d_out))
    assert ## Safety: Fast exit when red
    not (other.signal_mode == TLMode.RED and (ego.x>other.x-d_out and ego.x < other.x) and ego.v
    < vth)
    return output
```

Listing 5: Baseline Decision Logic

10.2 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 $T_s = 80s$.

Given a DL and a particular initial condition in **R**, the complete system has a unique *execution*. An execution is *safe* if when the traffic light is red, the car is not at *Entrance* region and the car is not slow in *Exit* region. For this MP, we let $d = 300m$, $d_{in} = 20m$, $d_{out} = 15m$. The dwell time for the traffic lights are $t_g = 20s$, $t_a = 5s$, $t_r = 20s$. There are three **R**s of increasing difficulty; you have to submit a single DL for all three **R**s.

- $\mathbf{R_1}$: $x_c \in [0, 50]$; $y_c \in [-5, 5]$; $v_c = 5$.
- $\mathbf{R_2}$: $x_c \in [0, 75]$; $y_c \in [-5, 5]$; $v_c = 5$.
- $\mathbf{R_3}$: $x_c \in [0, 75]$; $y_c \in [-5, 5]$; $v_c \in [3, 7]$.

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. Designing for $\mathbf{R_3}$ is harder than $\mathbf{R_2}$ and so forth. You get more points for a design that works for a higher i. The rest of the section describes the files you have to work with for this problem

10.3 Documentation of Provided Files

The code for this problem is in `./traffic_signal`.

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 vehicle. 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 5 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 position and signal of the traffic light (`other.x`, `other.signal_mode`).

The other requirements for DL remains the same as the first design problem.

traffic_light_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. **You will run this file in python to test your DL.**

11 DP2 Design Tool Functions: Simulation

Verse provides a simulation function to test the behavior of your decision logic. Essentially, this simulate function will generate a random initial state within the specified ranges. It will then run or "simulate" an execution of the vehicle and pedestrian starting from this initial state.

These functions should run in a few seconds. If it takes longer, you have run into an infinite loop.

You may **only** using the following functions of the design tool to test your DL (So, you may not use the verify function from your first design problem). You may also write your own functions that use these functions. For example, you may write a function that runs multiple simulations.

`trace = scenario.simulate(time_horizon, time_step)`: Runs a single simulation from a randomly sampled initial point chosen from **R** and returns the trajectory. 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 contains a list of AnalysisTreeNode that contains the 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.

`traces = scenario.simulate_multi(time_horizon, time_step, init_dict_list)` : simulate from all initial points in the `init_dict_list` instead of randomly choosing a point from the initial set. The output will also be a list of traces instead of just one trace.

- `time_horizon`: Float, the total time for the scenario to evolve.
- `time_step`: Float, the time sampling period.
- `init_dict_list`: List of dictionaries, list of initial points to simulate from. For example, an init_dict_list of [{'car': [-5,25,6,8], 'pedestrian':[170,-53,0,3]}] will simulate starting from the car state [-5,25,6,8] and pedestrian state [170,-53,0,3].
- `traces`: List, list of traces. A trace is explained above in `scenario.simulate`

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

- `tree`: The resulting tree output generated by the `scenario.simulate_simple` function.
- `fig`: Figure object, the figure object to plot on. This fig may be reused to put multiple simulated trajectories onto one plot
- `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 **L**. The plot can be shown using the `fig.show()` function as shown in Listing 6.

`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. A single simulation is generated by `scenario.simulate_simple` 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 6. The plot for `scenario.simulate` is presented in Figure 6. Note that, although the system is simulated to be safe for $\mathbf{R}_1$, the same decision logic may fail for a different **R**. Note that the base DL is not effective because, as shown in Figure 6, the car stops and never passes through the intersection.

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

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

Listing 6: Code for a single simulation.

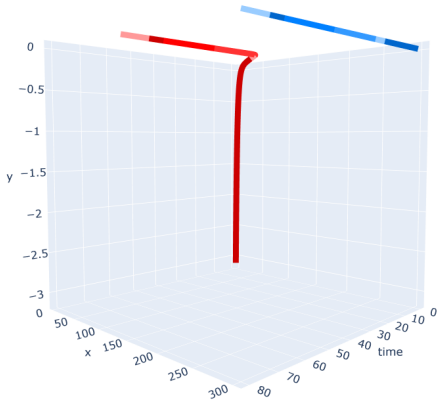


Figure 6: A single simulation of scenario starting from $\mathbf{R_1}$. The red curve represents $\mathbf{C}$'s trajectory, while the blue represents the traffic light's possition.2.

12 DP2 Design Outcomes & Grading

Problem 5 (15 points)**.** (a) (5 points) What is the highest $\mathbf{R_i}$ that you claim you DL is safe for? What is the evidence for safety? Evidence should be a set of tests or simulations with the possibility of manually simulating different edge cases with `simulate_multi()`. Please show an image of the simulation plot with at least a few hundred trajectories plotted and no safety violations. Multiple simulations can be run by calling `simulate()` in a for-loop and outputting all of the traces onto the same figure

(b) (5 points) What is the average speed you achieve over this $\mathbf{R_i}$? How do you justify this answer?

(c) (5 points) In 3 or fewer sentences, describe the basic idea of your DL design for achieving safety and high average speed.

Auto-grading Score: 15 pts. Specifically, there are 5 pts on $\mathbf{R_1}$,5 pts on $\mathbf{R_2}$, and 5 pts on $\mathbf{R_3}$. The percentage of score you can get on each $\mathbf{R_i}$ will be calculated as Listing 7. **We use 6.5 m/s as a threshold to evaluate your DL's performance on speeds.** The autograder will run simulate() on your decision logic many times, including some edge cases. Reachability is not run by the autograder. As you can see, the speed penalty gradually increases, but the penalty for any unsafety detected is very harsh (at least 50%). Be sure to test your DL thoroughly.

```
autograded_part_score = min(average simulation vel/ threshold, 1) * (1 - penalty)
penalty = (0.5 + min(% of failed simulation test cases, 0.5) ) if any unsafety detected else 0
```

Listing 7: Auto-grading Scoring Formula.

Please add Problem 5 to your write up and save your code for submission.

Demo Attendance. This DL will also need to be demoed on February 7th.

13 DP2 Survey and Final Survey

After you complete your DL and write-up, you will complete a survey about your experience on this DL and the entire MP.

You should complete all your code for DP2 and the write-up for problem 5 before completing the survey.

Problem 6. Consider your experience using the simulate function to complete this design task for the traffic light scenario. Complete the Midpoint Survey (B2).

Problem 7. Consider your experiences using both simulate and verify to complete both design tasks. Complete the Final Survey.

14 Submissions

Step 1	☑Compete the Pre-Survey Problems 1 and 2
Step 2	☑DP1 Emergency Automatic Braking Complete Decision Logic code Add Problem 3 to write up
Step 3	☑Compete DP1 Survey Problem 4
Step 4	☑DP2 Traffic Intersection Controller Complete Decision Logic code Add Problem 5 to write up
Step 5	☑Compete DP2 Survey and Final Survey Problems 6 and 7
Step 6	❑Submit writeup to Canvas and upload code

Once you have completed Steps 1-5, you will submit your write-up and code. **Again, if you are not participating in the study, do not fill out any forms**

For DP1 DL file, rename your `vehicle_controller.py` in the format `netid_DP1.py` and submit ONLY that python file to canvas

For DP2 DL file, rename your `vehicle_controller.py` in the format `netid_DP2.py` and submit ONLY that python file to canvas

The write-up should be submitted to Canvas as a file named `<netid>_ECE484_MP0.pdf`. Please include and cite any external resources you may have used in your solutions. Also, please upload your code to some cloud storage like Google Drive, DropBox, or Box; create a shareable link and include it in your write-up.