ECE 484: Principles of Safe Autonomy	Fall 2025
Written by: ECE 484 team	*MP0*
Website 	*Due Date: September 12, 2025*

1 Introduction

In this MP, you must verify several flight collision avoidance scenarios using a tool called Verse [1].

Along with this MP, each student must individually complete Homework 0. HW 0 is due at the same time as the MP.

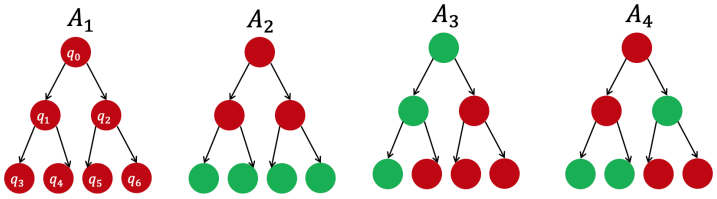
2 Homework 0: Safety and Invariants

HW Problem 1 (15 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.

HW Problem 2 (35 points) **[Individual]**

1. Is $\mathcal{A}_1$ deterministic? Why or why not?
2. How many executions does $\mathcal{A}_2$ have?
3. Consider the following 4 requirements:
   - Always red: $R_1 = \{\alpha \mid \forall i,\ \alpha_i.col = red\}$.
   - Never green: $Unsafe = \{q \mid q.col = green\}$.
   - Eventually green: $R_2 = \{\alpha \mid \exists i,\ \alpha_i.col = green\}$.
   - Never red: $R_3 = \{\alpha \mid \forall i,\ \alpha_i.col \neq red\}$.

For each of the 4 automata $\mathcal{A}_1, \ldots \mathcal{A}_4$ and each of the 4 requirements, say whether the automaton satisfies the requirement or give a counter-example. You can present the answer in the form of a 4×4 table. This question is tricky. Make sure to review the definitions in the slides carefully!



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.

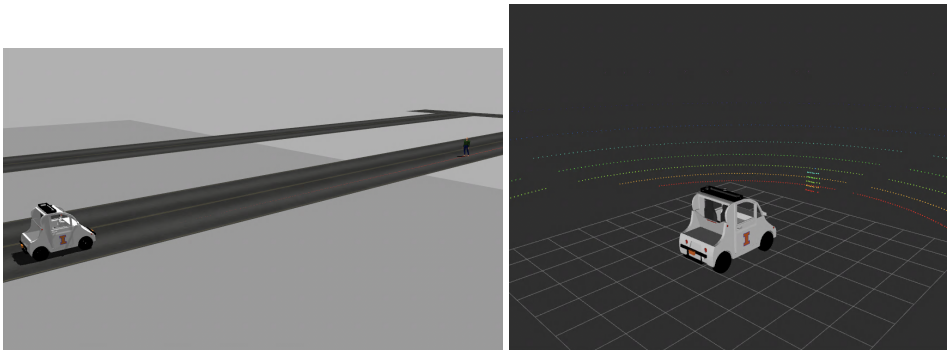


Figure 1: Vehicle and pedestrian on a single lane. *Left:* Initial positions of the two are at x_{10} and x_{20}, respectively. *Right:* Vehicle undergoes constant deceleration a_b once the vehicle detects the pedestrian.

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 $d(t) \leq D_{sense}$
    $s(t+1) = 1$
    if $v_1(t) \geq a_b$
        $v_1(t+1) = v_1(t) - a_b$
        $timer(t+1) = timer(t) + 1$
        $timer2(t+1) = timer2(t)$
    else
        $v_1(t+1) = 0$
        $timer(t+1) = timer(t)$
        $timer2(t+1) = timer2(t)$
else
    $s(t+1) = 0$
    $v_1(t+1) = v_1(t)$
    $timer(t+1) = timer(t)$
    $timer2(t+1) = timer2(t) + 1$
$x_1(t+1) = x_1(t) + v_1(t+1)$

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 3 (10 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 4 (40 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. Make sure to cover ALL branches of logic in your proof!

HW Problem 5 (10 BONUS points) **[Individual]** Is $d(t) > 0$ an invariant of $\mathcal{A}$ assuming $x_{20} - x_{10} \geq D_{sense}$

and $D_{sense} > v_0^2/a_b + 2v_0$.

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 verse installation. It is also recommended to create a virtual environment to make package installation easier.

To get the code for MP0, run the following command:

```
git clone https://gitlab.engr.illinois.edu/GolfCar/mp-release-fa25.git
```

Listing 1: Retrieving MP0 Code

ECE 484: Principles of Safe Autonomy	Fall 2025
Written by: ECE484 team	*MP0*
Website 	*Due Date: September 12, 2025*

4 Verification of air-traffic control via reachability analysis

4.1 Overview

Airtaxis can transform how people travel in crowded metropolitan areas. One of the key challenges in building airtaxi systems is the problem of traffic management and collision avoidance. There are numerous reports from NASA and AIAA on this topic (See, for example, this report [3]). In this MP, you will learn how to use reachability analysis through Verse to analyze and verify a simplified air collision avoidance protocol.

4.2 Problem Description

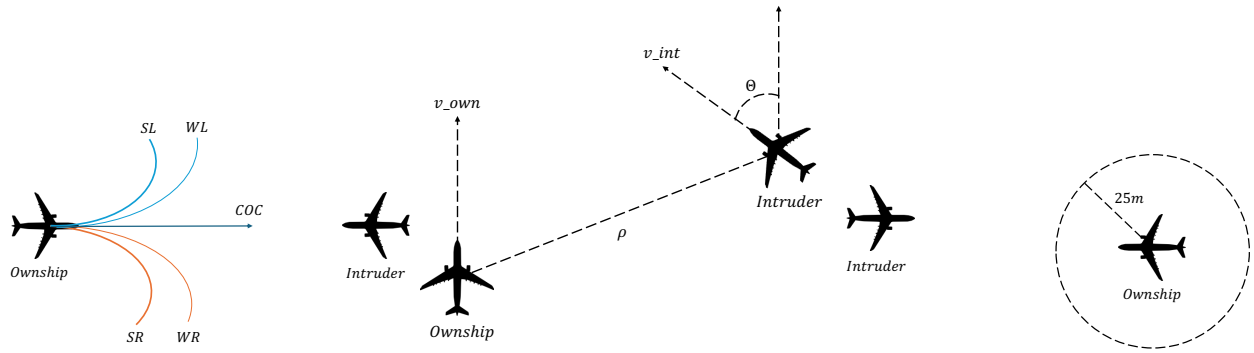
You are given three different scenarios inspired by ACAS-Xu (**A**irborne **C**ollision **A**voidance **S**ystem for Unmanned Aircraft)—a collision avoidance protocol for unmanned aircraft [2]. In each scenario, 2 airplanes are flying on the (x, y) plane—an *ownship* **O** (modelled with Dubin's plane model Link) and an *intruder* aircraft **I** that simply flies straight. **O** will follow *advisories* from ACAS-Xu to detect and avoid collisions with **I**. There are five possible advisories that the protocol can generate:

- **COC**: Fly straight, no turn
- **SL** : Turn a strong left
- **SR** : Turn a strong right
- **WL** : Turn a weak left
- **WR** : Turn a weak right

Each aircraft model has 4 different variables: **x,y**: are the (x,y) coordinates in meters (m), θ is the heading (measured in radians w.r.t. the x-axis), and **v** is the speed in m/s.

This controller will give an advisory to the ownship aircraft based on the following variables:

- ρ (m): The distance between the ownship and intruder aircraft
- Θ (radian): The angle made by the heading angle of the ownership and the line made connecting the position of ownship and intruder aircraft.



(a) (b) (c)

Figure 2: 2a) Illustration of possible advisories (Strong Left (SL), Weak Left (WL) , Clear of Conflict (COC), Strong Right (SR), Weak Right (WR)), 2b) Partitioning based on intruder position, relative distance ρ, and relative angle Θ between 2 planes, 2c) Safe region ensured by 25m separation between ownship and intruder.

Given a particular initial condition in **R**, the complete system has a unique *execution*. An execution is *safe* if **O** is always greater than 25m distance from **I** within T_s. To put everything in perspective, you can take a look at Figure 2.

Your task is to prove the scenario is *safe* or *unsafe* from any initial condition in a given range **R**, in a given time T_s = 80, and with a given controller. In Python, the initial set is defined as [lowerBound, upperBound] where each bound is a state in the format of [x, y, θ, v].

Here are the following initial conditions that are included in the MP:

- $\mathbf{R_1}$: $x_{own} \in [-4100, -3100], y_{own} \in [-5000, -4950], \theta_{own} \in [0, 0], v_{own} \in [120, 120], x_{int} \in [-1450, -1250], y_{int} \in [1500, 2000], \theta_{int} \in [-\pi/2, -\pi/2], v_{int} \in [100, 100]$.
- $\mathbf{R_2}$: $x_{own} \in [-2200, -1200], y_{own} \in [-2000, -950], \theta_{own} \in [-\pi/2, -\pi/2], v_{own} \in [120, 120], x_{int} \in [-100, 100], y_{int} \in [1500, 2000], \theta_{int} \in [-3\pi/4 + 1.4, -\pi/12 + 1.4], v_{int} \in [100, 100]$.
- $\mathbf{R_3}$: $x_{own} \in [-7500, -7000], y_{own} \in [-6000, -5950], \theta_{own} \in [0, 0], v_{own} \in [120, 120], x_{int} \in [-100, 100], y_{int} \in [1500, 2000], \theta_{int} \in [-\pi/2, -\pi/2], v_{int} \in [100, 100]$.

The rest of the section describes the files you have to work with.

4.3 Documentation of Provided Files

The important files are:

dubins_controller.py This file contains the decision logic to control the ownship aircraft to avoid collision with the intruder.

dubins_agent.py This file contains the implementation of Dubin's plane model, including dynamics and modes, and the always-go-straight plane model.

dubins_sensor.py This file contains the implementation of processing the information of the initial states to return information that will be used to give advisory for the aircraft.

dubins_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 and modify this file for the task.**

5 Verse Functions

5.1 Simulation

Verse provides simulation and reachability functions. The simulation function generates a random initial state within the specified ranges and computes an execution of the system (with 2 planes) from this initial state. If unsafety is detected, from any point in the initial set, the scenario is sure to be unsafe. Therefore, it may be wise to run simulations in regions of the initial set likely to be unsafe.

This function runs in a few seconds by itself, but running it thousands of times will take much longer.

`trace = scenario.simulate(ownship_aircraft_init, intruder_aircraft_init, time_horizon, time_step, ax)`: 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 a red message `assert hit` in the terminal.

- `ownship_aircraft_init`: List, the initial state of the ownship (R1-R3).
- `intruder_aircraft_init`: List, the initial state of the intruder plane (R1-R3).
- `time_horizon`: Float, the total time for the scenario to evolve.
- `time_step`: Float, the time sampling period.
- `ax`: Plotter, the pyvista plotter for visulization
- `trace`: AnalysisTree object containing the simulated trajectory. The AnalysisTree object contains a list of AnalysisTreeNode that contains the execution of the scenario for each mode of **C**.

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

- `ownship_aircraft_init`: List, the initial state of the ownship (R1-R3).
- `intruder_aircraft_init`: List, the initial state of the intruder plane (R1-R3).
- `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 [{'plane1': [-5,25,6,8], 'plane2':[170,-53,0,3]}] will simulate starting from plane1's state [-5,25,6,8] and plane2's state [170,-53,0,3].
- `ax`: Plotter, the pyvista plotter for visulization
- `traces`: List, list of traces. A trace is explained above in `scenario.simulate`

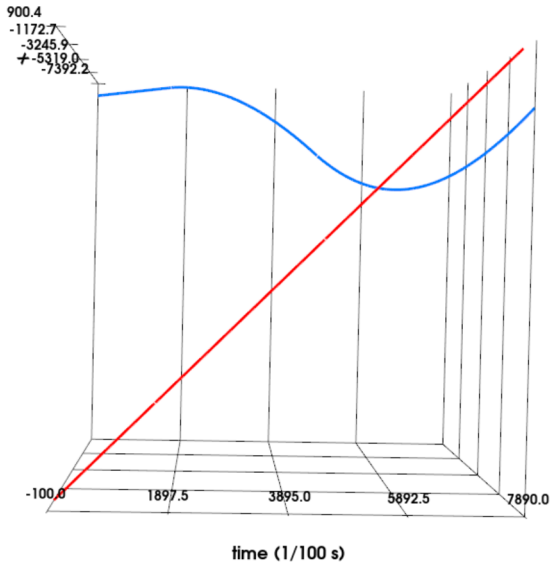


Figure 3: Simulation plot for a single simulation

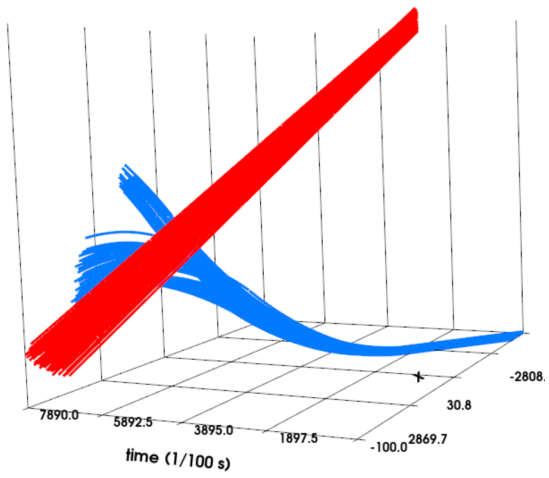


Figure 4: Simulation plot with multiple simulations run

5.2 Reachability

You can also use *reachability analysis*. Recall that reachability over-approximates all possible executions in the initial set. If the analysis determines that the system is safe, then safety is guaranteed. If unsafety is detected, that does not necessarily mean that the scenario is unsafe. The analysis usually generates an approximation of the reachable set larger than the actual reachable set. The reachability analysis could have over-approximated the reachable set such that it intersects with the unsafe set when the actual (smaller) reachable set would not.

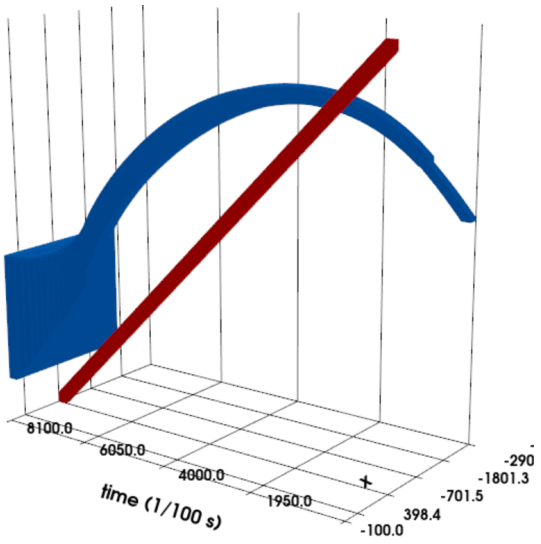
`traces = scenario.verify(ownship_aircraft_init, intruder_aircraft_init, time_horizon, time_step, ax)`: Compute reachable set and check the safety of the scenario.

- `ownship_aircraft_init`: List, the initial state of the ownship (R1-R3).
- `intruder_aircraft_init`: List, the initial state of the intruder plane (R1-R3).
- `time_horizon`: Float, the total time to perform reachability analysis.
- `time_step`: Float, the time period that the reachable set is sampled.
- `ax`: Plotter, the pyvista plotter for visulization
- `traces`: AnalysisTree, the computed reachtube for the scenario. Has similar structure as that produced by `scenario.simulate()`. This reachtube will contain the reachable set for both planes across the entire time horizon

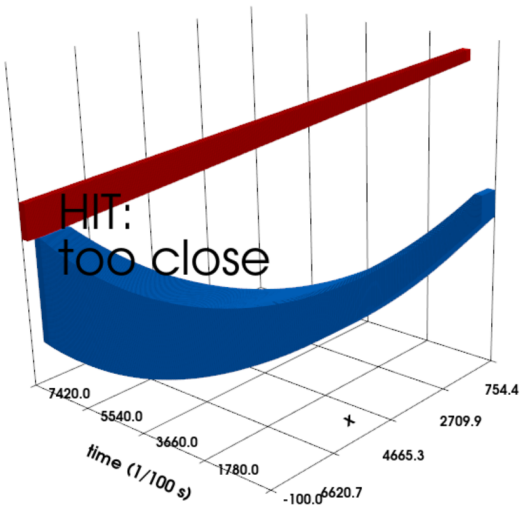
Figure 5 shows an example of performing reachability analysis and generating plots using above functions. If a safety assertion is violated, then a label will appear on the plot and a message will appear in the console.

6 Verification Problem Task

Reachability analysis from a smaller initial set usually results in a tighter (less conservative) approximation. The more conservative the approximation, the less precise it is. Therefore, for a larger initial set, you



(a) Reachtube analysis that guarantees safety.



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

Figure 5: Reachability results with verify()

must determine a strategy to partition the initial set into smaller regions for a more precise reachability analysis.

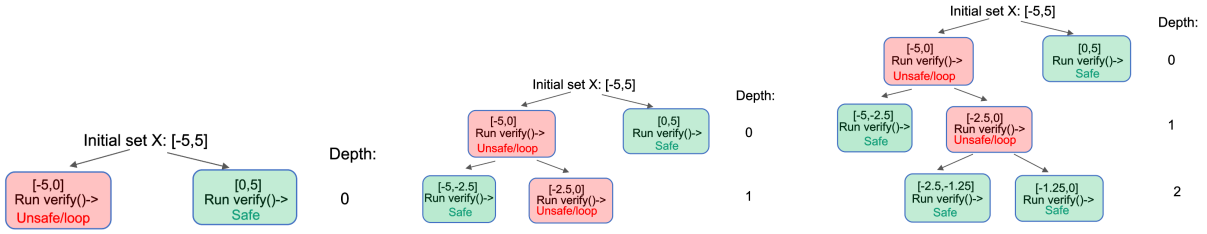
Using the `verify()` function, you will write the function:

`traces = verify_refine(scenario, ownship_aircraft_init, intruder_aircraft_init, time_horizon, time_step, ax)`: Compute reachable set and check the safety of the scenario. `verify_refine()` must partition the initial ranges into smaller regions to get a more precise reachability analysis result.

- `scenario`: the scenario to verify.
- `ownship_aircraft_init`: List, the initial state of the ownship (R1-R3).
- `intruder_aircraft_init`: List, the initial state of the intruder plane (R1-R3).
- `time_horizon`: Float, the total time to perform reachability analysis. (Just pass this into `verify()`)
- `time_step`: Float, the time period that the reachable set is sampled. (Just pass this into `verify()`)
- `ax`: Plotter, the pyvista plotter for visulization (Just pass this into `verify()`)
- `traces`: List[`AnalysisTree`], the computed reachtube for the scenario. List of all traces returned by calls to `verify()`

One way to write this function is to use a DFS (depth first search) or BFS (breadth first search) style recursive traversal: continually partition the initial set into smaller and smaller regions until the result is safe. Each level of partition depth increases the number of calls to `verify()` exponentially.

Figure 6 shows partitioning on the dimension x. The final output is considered safe, since all leaf partitions are safe and together form the original initial set. In your scenarios, there are more dimensions than just x. The full state is defined in Section 4.2. You will need to decide which dimensions to partition and which order to partition. One method is to alternate splitting between dimensions. For example, you may split ownship's x dimension in half, then in the next layer of partitioning, split the intruder plane's y dimension in into thirds.



(a) Unsafety in some partitions (b) Partitioning on unsafe regions (c) All leaf partitions safe

Figure 6: Partitioning with tree method from initial set x: [-5,5]

You may also initially divide the initial set into some number of partitions in each dimension. (np.linspace function may be helpful here.) This may be faster than recursion if you try not run the `verify()` function more times than necessary. You may also combine the two approaches and partition the set in the beginning and then do recursive partitioning. We encourage you to try creative partitioning methods for this assignment.

Each call to `verify()` will generate a trace. All traces of partitions that run successfully without unsafety MUST be added to the traces list and returned to be visualized with the given plotting function. This plotting function simply visualizes all traces on one plot. Using this knowledge, you can answer Q4 in the report.

To conclusively prove safety, all of the partitions together MUST form the original initial set. That is, any point in the state space of the original initial set must be present in at least one of the partitions. You must run the `is_refine_complete()` function to validate this. To run this function, you must add your final partitions of the initial set to the partitions list.

We expect an optimal implementation of the function `verify_refine()` to run in 15 min or less for the scenarios provided. However, your `verify_refine()` does not need to be optimal, nor is it needed for all of the scenarios!
Implementation tips:

- Use the function `tree_safe()` to determine if a trace returned by `verify()` is safe.
- Reminder: In Python, lists are mutable objects, meaning that assigning one list to another variable will create a reference to the same object, not a copy. If you need to make a copy of an initial set (e.g., to preserve the original array), please use the `.copy()` method.
- It is also recommended to use a progress bar package such as alive-progress to keep track of what percent of the initial set is verified.
- If you see an "infinite loop detected" message, this essentially means that Verse's over-approximation was too big, and the the reachability analysis cannot continue.

7 Report & Grading

Questions 1–3

For each R1–R3, answer the following:

1) [20 pts] Is the scenario safe or unsafe? Provide proof (images of the plot from multiple angles and console output) with either the `simulate/simulate_multi` or `verify/verify_refine` function that the scenario is either safe or unsafe.

2) [4 pts] Which function did you use to prove this and why?

3) [4 pts] If reachability was used, what was your strategy for partitioning (if partitioning is needed)? If simulation was used, what was your strategy for running simulations (number of simulations, regions to simulate in, etc.)?

Question 4 [6 pts]

Recall that running `verify()` will generate a plot and a reachability analysis result (safe or unsafe). If the different colored regions intersect on the plot, the analysis will always determine unsafety is present in the scenario. Is this necessarily true for a correct implementation of `verify_refine()`? That is, is there some way to partition the initial set such that there is visible intersection on the `verify_refine()` plot, but the analysis determines the scenario is safe? Explain why or why not.

Demo Attendance [10 pts]

Attend your lab session on September 12 to demo your design logic. We will ask questions regarding invariance concepts related to the MP. Additionally, be prepared to show your plots and the result for `is_refine_complete()`.

Submission: Please upload your code to Gradescope in a file named `netid_mp0.py` and your report to a file named `netid_mp0.pdf`

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.

[2] Michael P. Owen, Adam Panken, Robert Moss, Luis Alvarez, and Charles Leeper. Acas xu: Integrated collision avoidance and detect and avoid capability for uas. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2019.

[3] Casey L. Smith and R Conrad Rorie. Helicopter pilot evaluations of the airborne collision avoidance system xr in a high-fidelity motion simulation - nasa technical reports server (ntrs). July 2024. [Online; accessed 2025-08-19].