Reflection on group assignment 3:

Entire instructions:

1. Setting Up Docker:

- **Install Docker:**
- If Docker is not installed on your system, download and install it from the official Docker website.
- **Start Docker:**
  Copy codedocker build -t your_image_name .
- Run the Docker Container:
- Start a Docker container from the image you built:arduin

  Copy codedocker run -it --name your_container_name your_image_name

Entering Docker Container Example:

Entering Docker Container
Enter docker desktop and click “run” to start image

```
docker ps
```

```
CONTAINER ID   IMAGE              COMMAND                  CREATED       STATUS
PORTS     NAMES
0abd7d539bf3   ros:noetic-robot   "/ros_entrypoint.sh …"   10 days ago   Up 38 minutes
        suspicious_jones
```

```
(base) Christians-MacBook-Air:~ christianjoserojas$ docker exec -it suspicious_jones bash
```

- Make sure also to run roscore

2. Generating Code from Simulink:

- **Remember to rosinit in simulink**
- **Generate Code:**
- Open your Simulink model.
- Configure the model settings for code generation, ensuring compatibility with your Docker environment.
- Generate the code by clicking on the "Generate Code" button.
- **Copy Generated Code to Docker:**
- Copy the generated code from your host machine to the Docker container using the **docker cp** command:bash

  Copy codedocker cp path_to_generated_code your_container_name:/ path_in_container

Notes from generating code:

```
Mkdir ros (creates ros directory)
```

Cd (change directory)

```
root@0abd7d539bf3:/ros# source /ros_entrypoint.sh
root@0abd7d539bf3:/ros# catkin_make
```

source ../../ros_entrypoint.sh (if you make a mistake or a typo, you must "reenter" the container)
Cd .. Returns to source folder

3. Running Simulations:

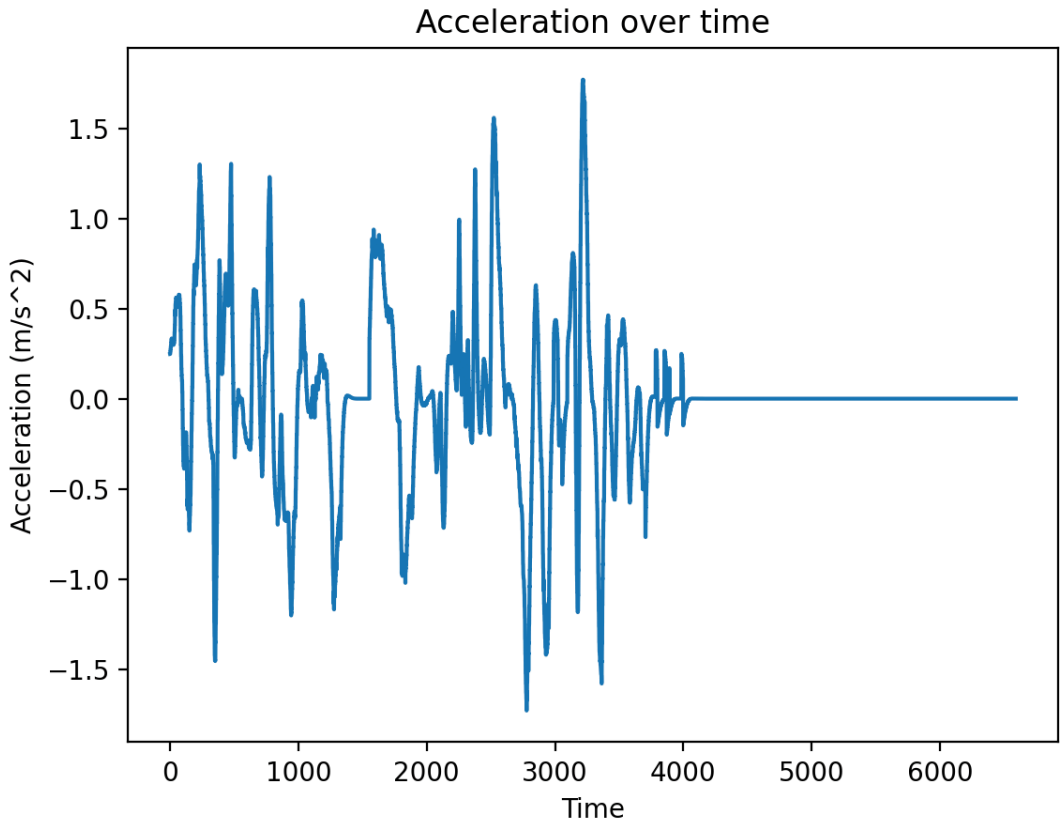
- **Software-in-the-Loop (SWIL) Simulation:**
- Inside the Docker container, navigate to the directory containing the generated code.
- Run the SWIL simulation. This might involve executing a script or running a command, depending on how your simulation is set up.
- **Playing a Bagfile:**
- use the **rosbag play** command:
  Copy rosbag play your_bagfile.bag
- Rosrun anson1031 Anson1031 (in my case)
- Check the results by opening up another terminal, run rostopic list and rosecho the acceleration variable

4. Analyzing Results:

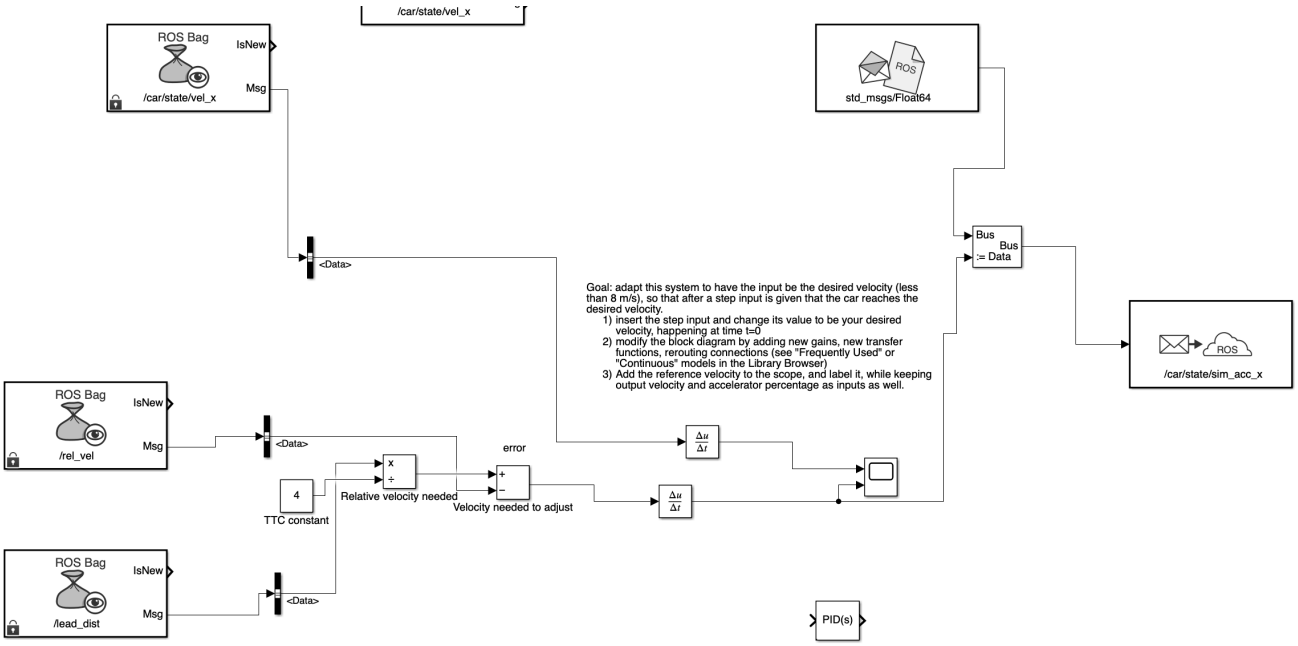
- **Viewing Results:**
- Analyze the results by looking up the terminal of the rosrun

```python
import matplotlib.pyplot as plt

# Replace 'your_file.txt' with the path to your text file
file_path = './data.txt'

# Initialize an empty list to store the numbers
numbers = []

# Open the text file and read the numbers
with open(file_path, 'r') as file:
    for line in file:
        # Split the line into words and iterate through them
        for word in line.split():
            # Try to convert each word to a float
            try:
                number = float(word)
                numbers.append(number)
            except ValueError:
                # If the conversion fails, the word is not a number, so ignore it
                continue

# Plot the numbers
plt.plot(numbers)
plt.title('Acceleration over time')
plt.xlabel('Time')
plt.ylabel('Acceleration (m/s^2)')
plt.show()
```

The following diagram I did not have a chance to talk about it in the video, but we generate put the data that generates from the ros into a txt file and script a python script that is the following: We run the script on terminal and get the following result. The acceleration fluctuates between -1.5 to 1.5m/s^2 as expected. The data cuts off from 4000 because we may only have 4000 data points recorded and there is no more input data from the bag file.



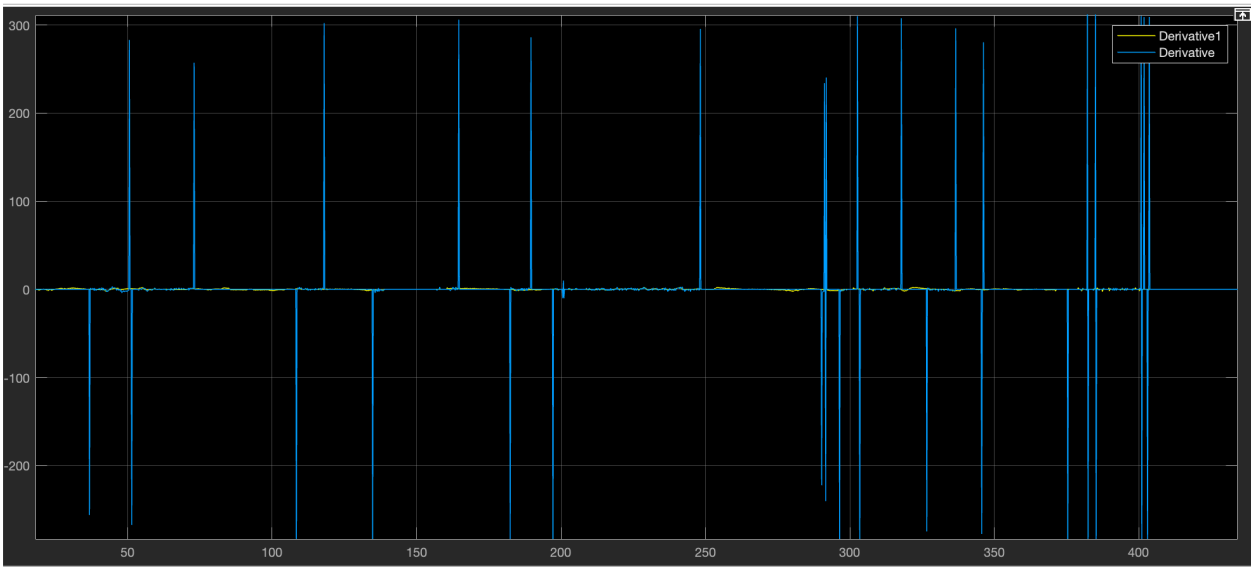
Then we also modify the simulink model to check something more interesting:



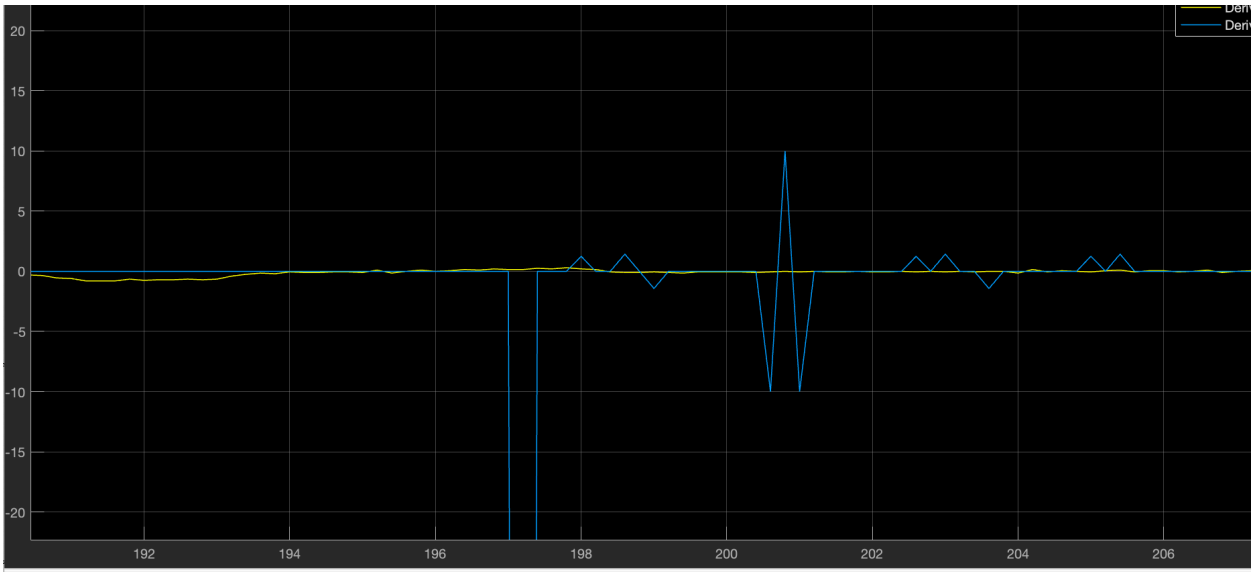
We tried to keep the time to collision (TTC) of the controlled car to 4s to the front car.

The output scope is here:

The blue line shows the commanded acceleration, which goes off the scale pretty much. X axis represents the progression of time.



This is the scope after zoomed in. The yellow line is the current acceleration, which is on the normal scale.



In the second video, I delve into the reasons behind this phenomenon. It occurs because we are altering the vehicle's acceleration instantaneously to attain a Time to Collision (TTC) of 4 seconds, despite the current TTC being 15 seconds. Such a rapid change necessitates a significant acceleration of the vehicle. Conversely, a similar deceleration is required if we need to increase the TTC. We are currently exploring solutions to address this challenge effectively.

Christian and Anson can finish the whole process including the code generation, docker container running, and validation running as well. However, Benjamin is still exploring the technology.

Ways to mitigate structural risk:
Cross-Training:
Organize regular cross-training sessions where team members teach each other their areas of expertise. This ensures that multiple people can handle each task, reducing dependency on any single individual.

Documentation:
Maintain comprehensive documentation for all processes, including code generation, Docker usage, running simulations, and validating results. Documentation should be clear enough for any team member to follow. We also put all of our updated documents on github.

Pair Programming:
Implement pair programming or pair work sessions, especially for critical tasks. This not only facilitates knowledge transfer but also enhances collaboration and code quality.