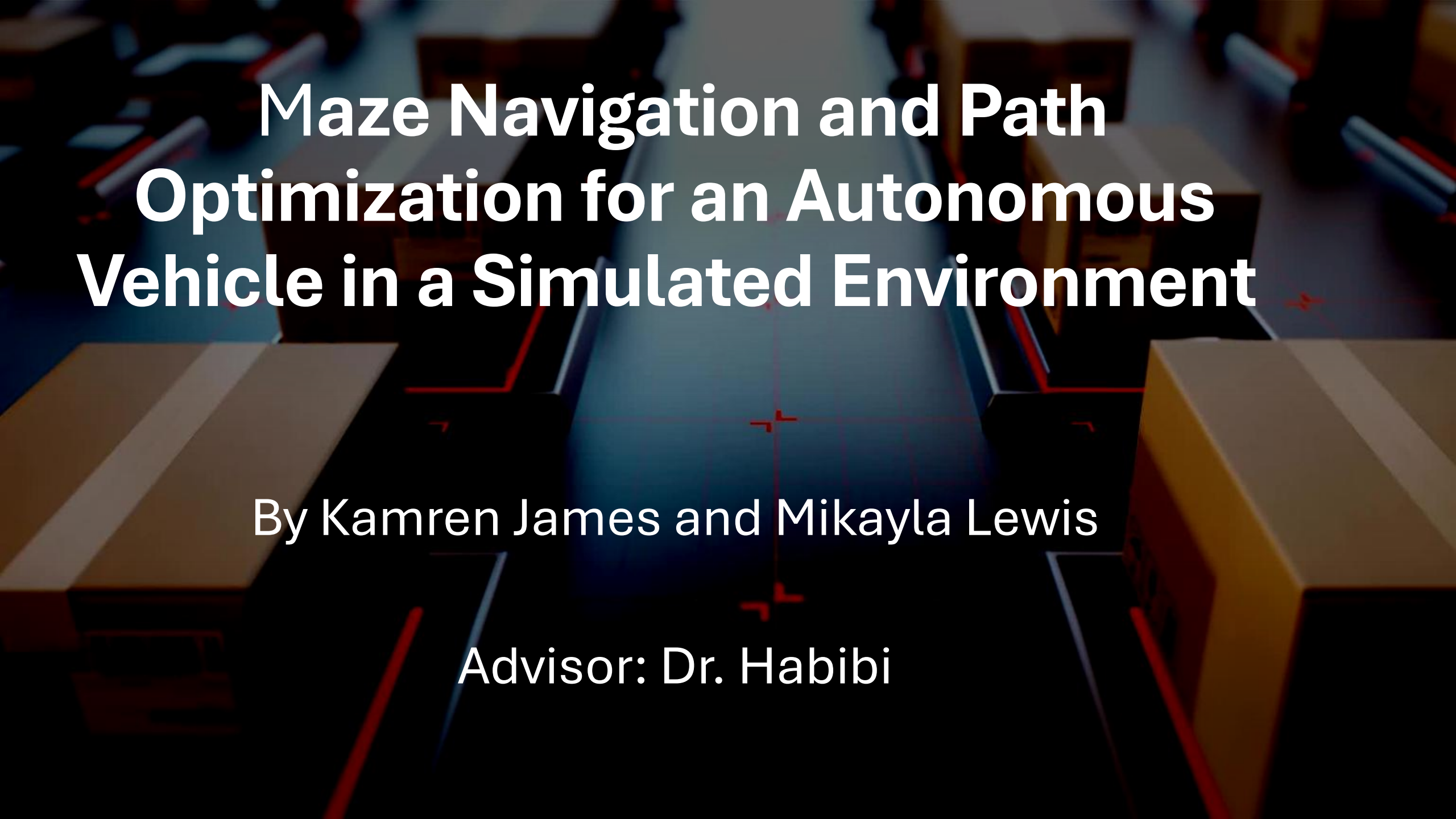
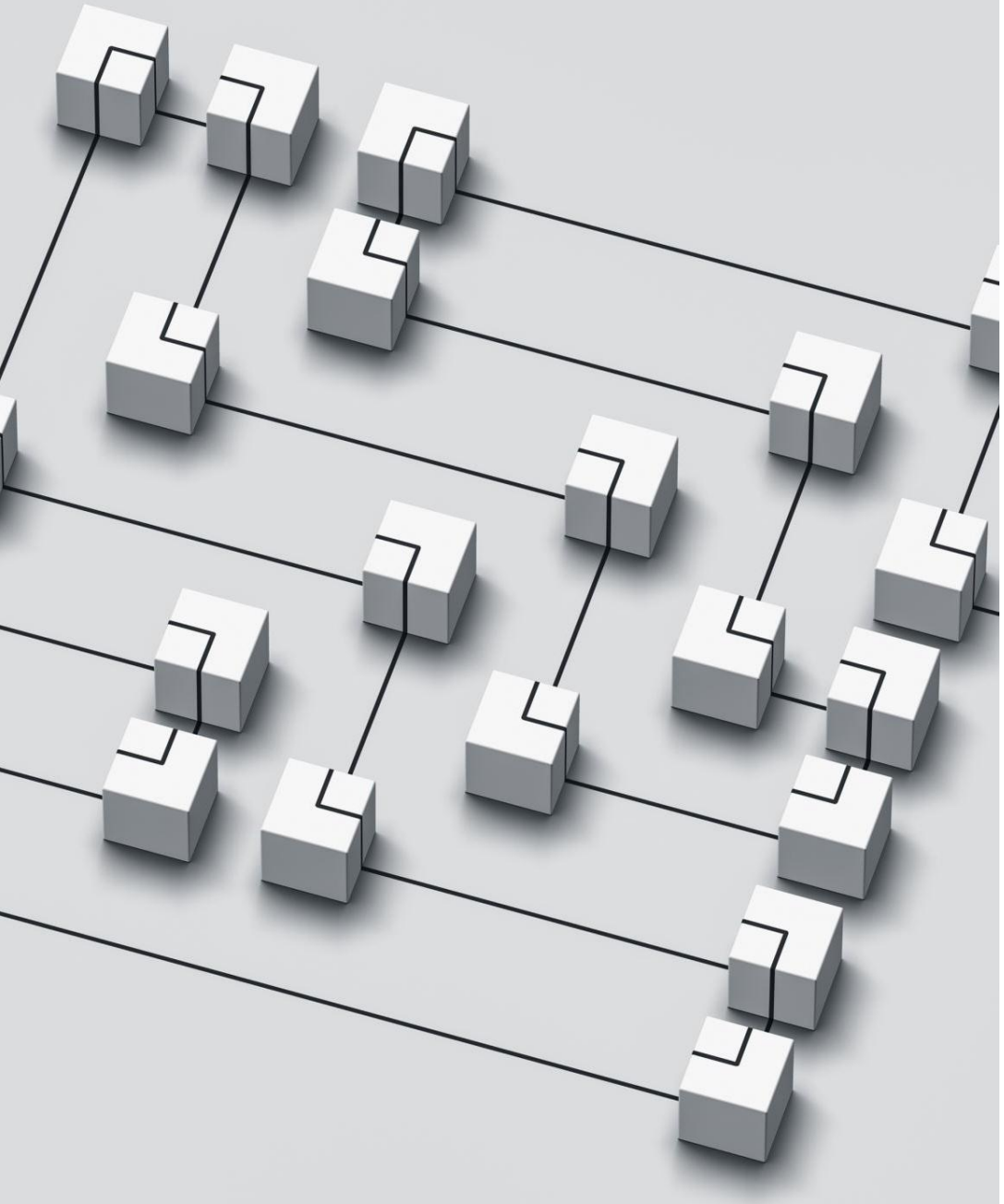


Maze Navigation and Path Optimization for an Autonomous Vehicle in a Simulated Environment

By Kamren James and Mikayla Lewis

Advisor: Dr. Habibi



Overview

- What is the problem?
- Machine Learning
- Software
- Virtual Model
- Hardware
- Prototype/ Model/ Maze
- Calculations and Output
- Conclusions

Introduction

- In recent years, autonomous systems have witnessed significant growth and adoption throughout various industries.
- However, during this period of expansion, robots of this type need to have the capability to work and move independently
- This capstone research on the development of machine learning algorithms and systems for autonomous mapping and path planning, aiming to improve the efficiency, safety, and automation in diverse applications

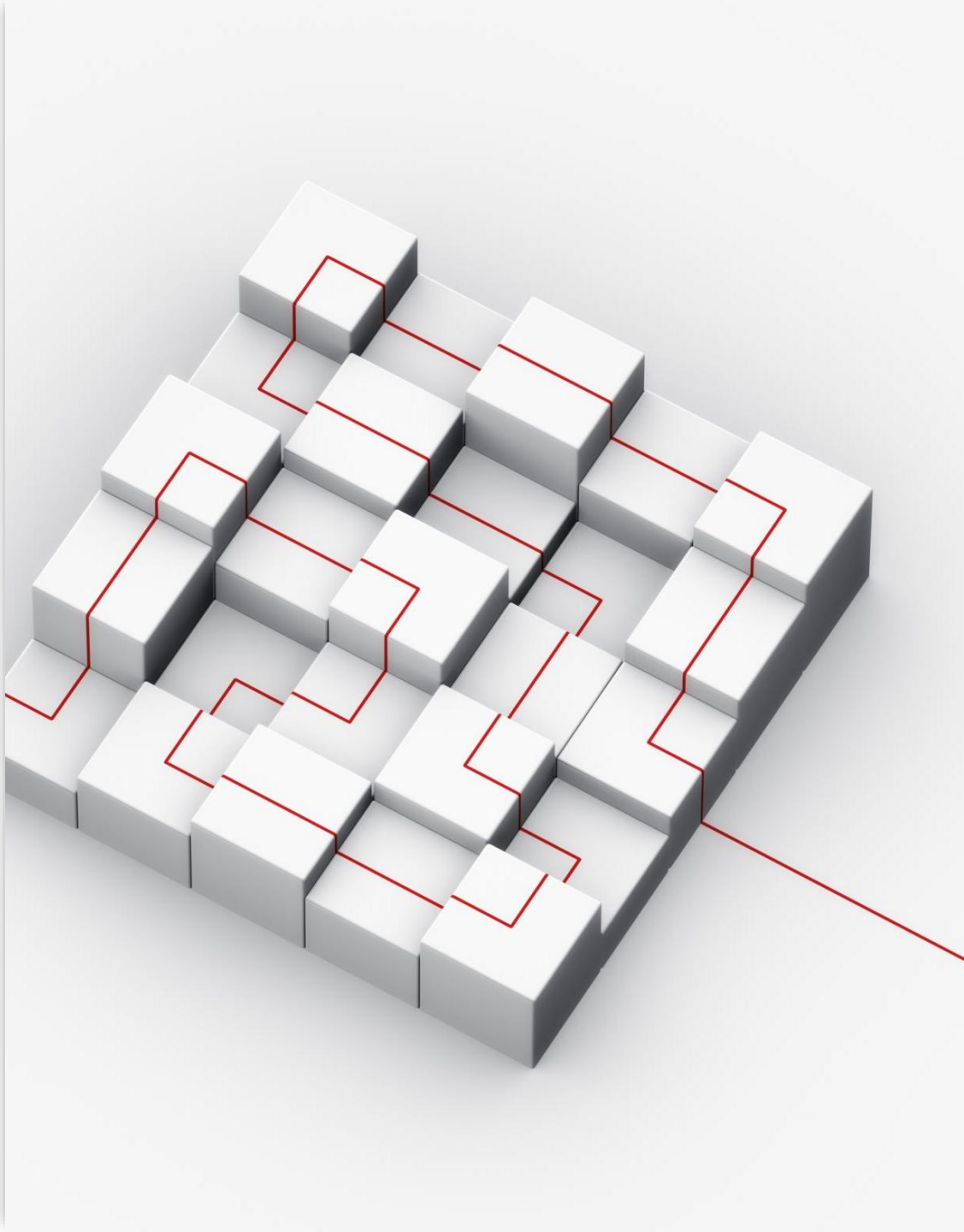
Problem Statement

-Navigating through mazes efficiently is a fundamental challenge in robotics.

-Need for advanced algorithms and strategies to enable robots to navigate through intricate mazes swiftly while optimizing their paths to achieve a desired goal.

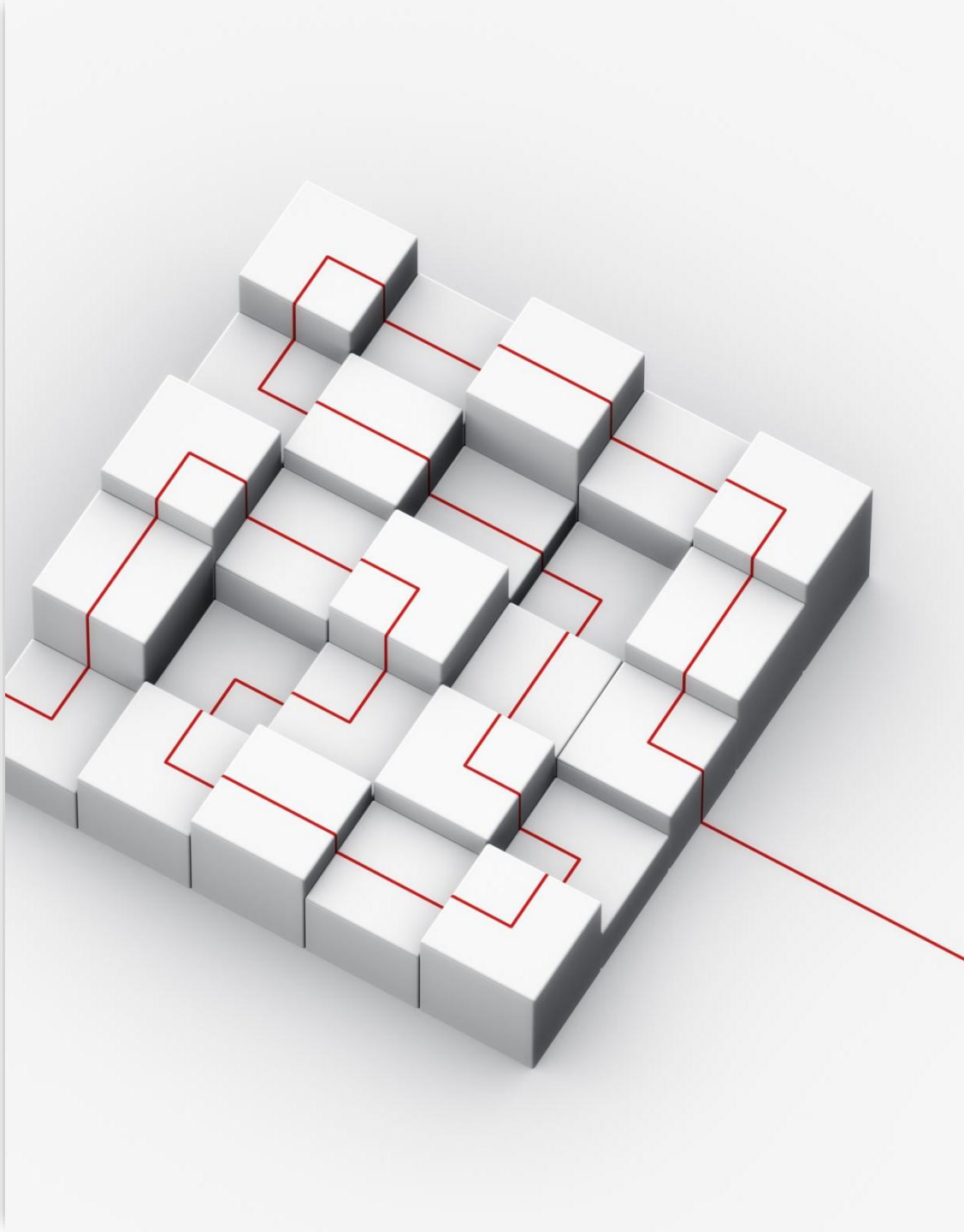
Project Objective

The objective of our project is to design a virtual robot that can autonomously explore and map a maze. Then effectively plan the most efficient path from the starting point to the assigned goal



Project Goal(s)

- Create a virtual environment
- Create virtual model of robot
- Develop code that allows for virtual model to navigate through a maze to a specified goal
- (optional) Create a physical prototype model to test trained network



Method of Investigation

- Research existing solutions to the problem and methods in which chosen to attempt to solve the problem
- Determine investigation method
- Create virtual model of robot
  - Environment
  - Virtual Robot
  - Code
- Create prototype model

Survey Solutions

- Leonardo Da'Vinci's Self-propelled Cart (1478)
- General Motors Firebird prototype (1953)
- Stanford Universities “Stanford Cart” (1979)
- DARPA Grand Challenge vehicles (2004 – 2020)

Software

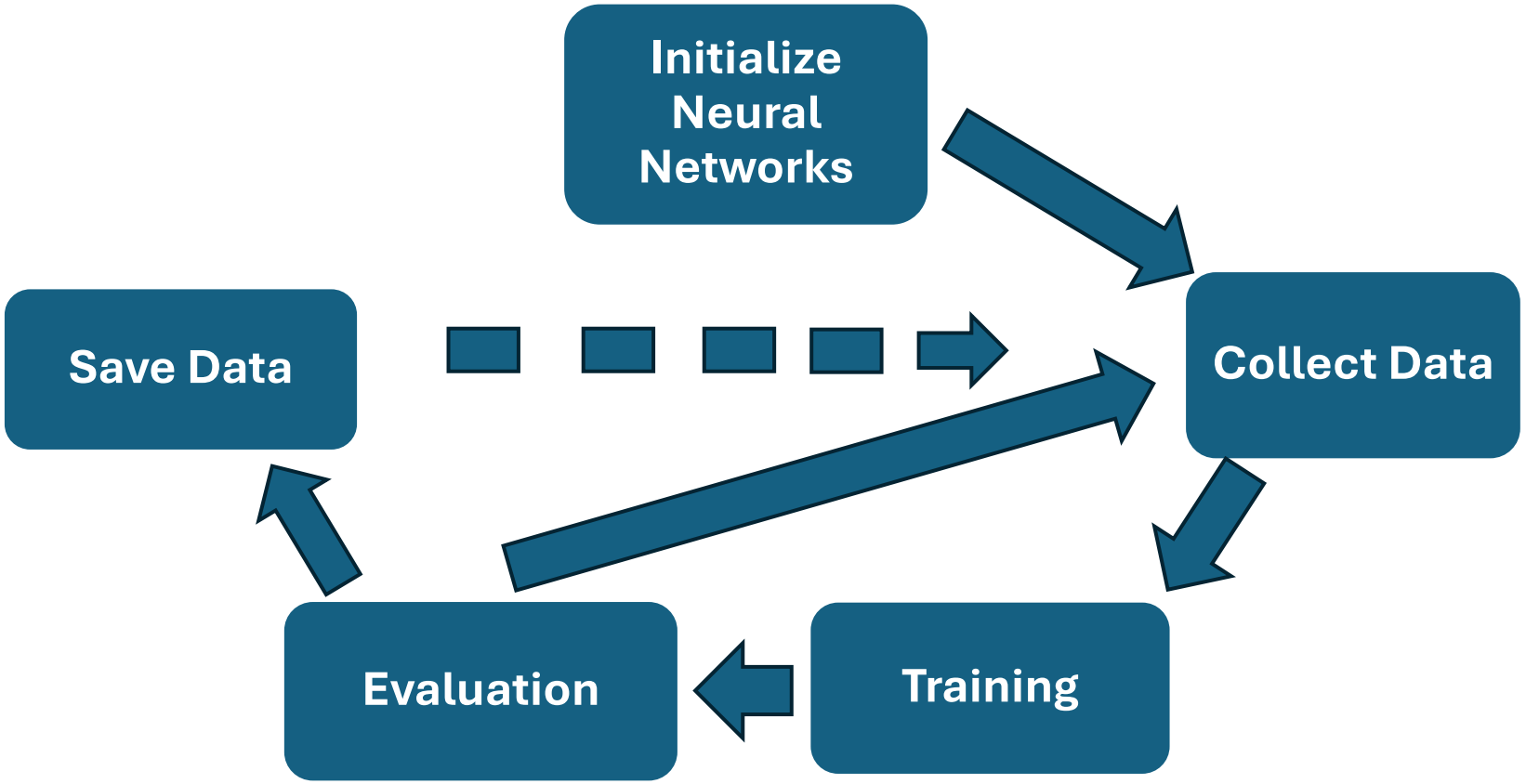
-SLAM
-Graphical Based Methods(Dijkstra, A*)
-Sensor Fusion
-Genetic Algorithms(RRT) Machine learning
-Deep Learning(TD3)
-Python-[PyTorch]

Machine Learning Models vs. Deep Learning Models

- Automatically
- learns and improves from experience
- Learns from data collected/feed
- Works well with smaller data sets and linear problems
- Performance plateaus at a point

- Creates a artificial neural network used to make intelligent decisions
- Learns from data collected/feed to identify features and solutions
- Works better with larger data sets
- Will continue to learn with more data and will become more efficient and accurate

Deep Learning



Software

- Linux Mint (Computer Operating system)
- ROS (Robot Operating System)
- Gazebo (Simulation space)*
- Rviz (3D visualization tool)

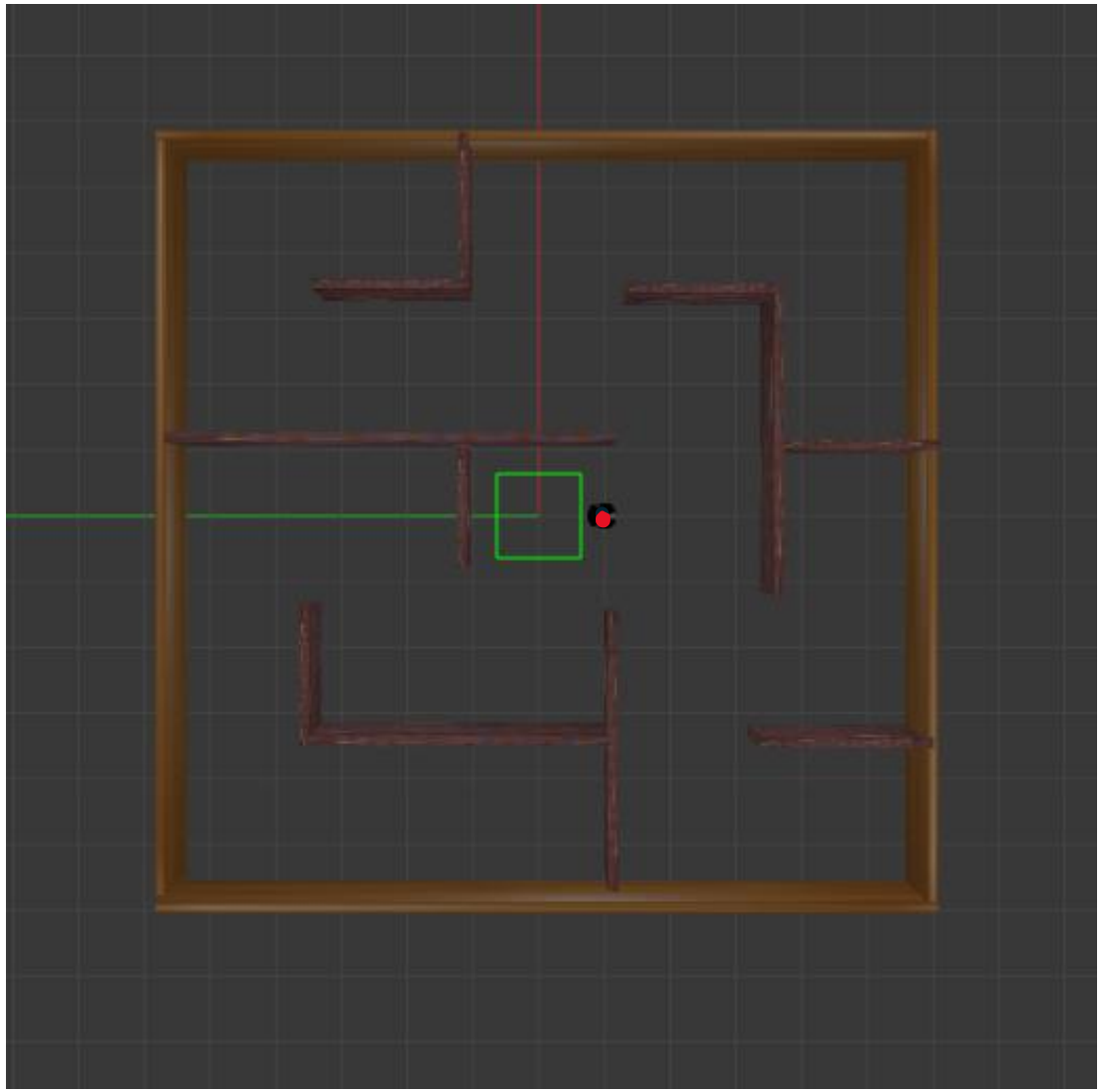
ROS (Robot Operating System)

Has a large list of public software libraries

Integration with Simulators (Gazebo & Rviz)

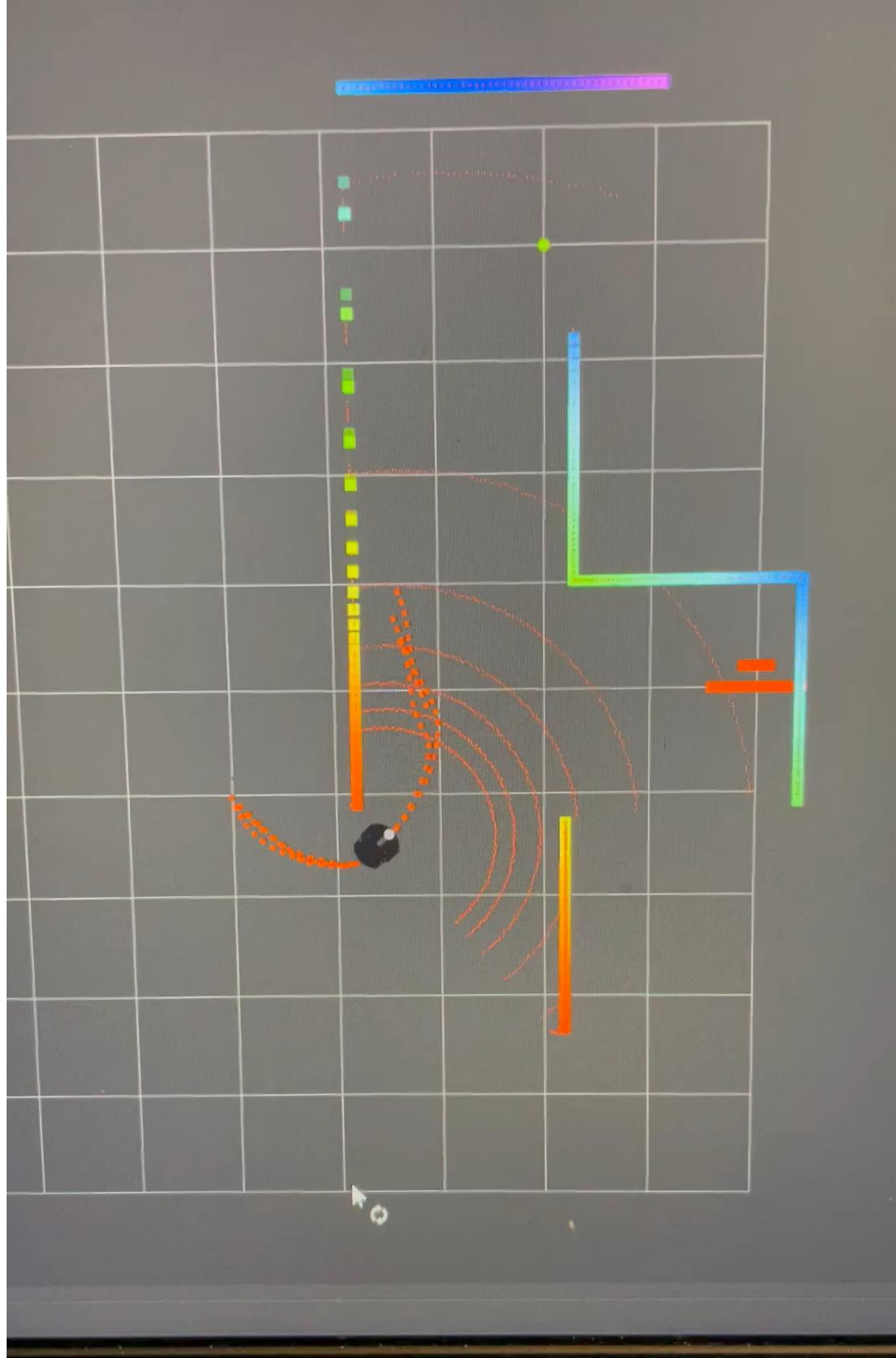
Gazebo

- Offers 3-Dimensional physics simulation, collisions and dynamic movement allowing the virtual robot to mimic real-world physics
- Allows Full integration with ROS

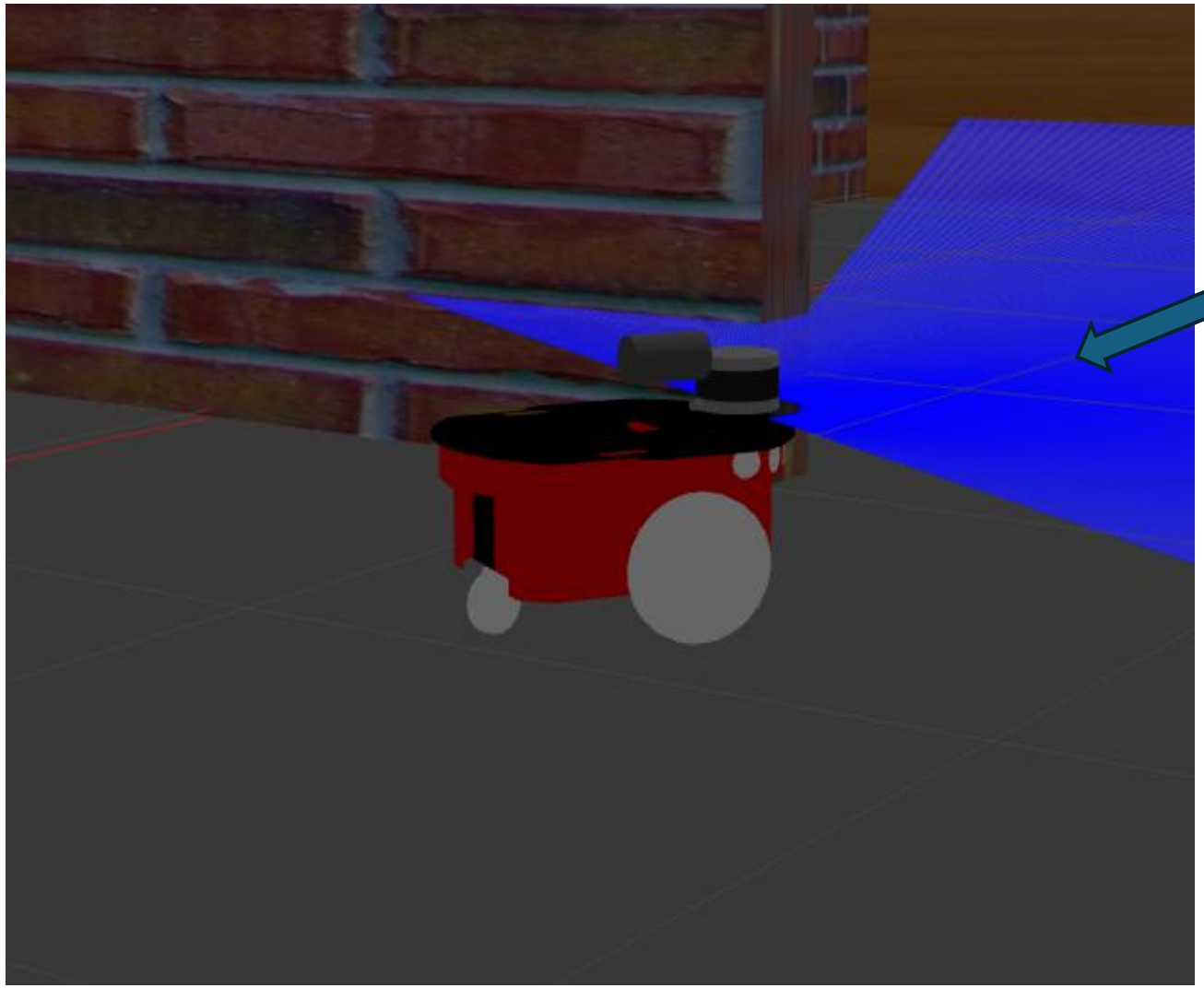


Rviz

- Allows for sensor data visualization
- Supports visualization of trajectory and path plans



Virtual Robot



Initializing Network
Collecting Data
Training Cycle

Initialize Network

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Actor, self).__init__()
        self.layer_1 = nn.Linear(state_dim, 800)
        self.bn1 = nn.BatchNorm1d(800)
        self.layer_2 = nn.Linear(800, 600)
        self.bn2 = nn.BatchNorm1d(600)
        self.layer_3 = nn.Linear(600, action_dim)
        self.tanh = nn.Tanh()

    def forward(self, s):
        if s.size(0) > 1:
            s = F.relu(self.bn1(self.layer_1(s)))
            s = F.relu(self.bn2(self.layer_2(s)))
        else:
            s = F.relu(self.layer_1(s))
            s = F.relu(self.layer_2(s))
        a = self.tanh(self.layer_3(s))
        return a
```

```python
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        # First Critic Network
        self.layer_1 = nn.Linear(state_dim, 800)
        self.bn1 = nn.BatchNorm1d(800)
        self.layer_2_s = nn.Linear(800, 600)
        self.layer_2_a = nn.Linear(action_dim, 600)
        self.bn2 = nn.BatchNorm1d(600)
        self.layer_3 = nn.Linear(600, 1)

        # Second Critic Network
        self.layer_4 = nn.Linear(state_dim, 800)
        self.bn4 = nn.BatchNorm1d(800)
        self.layer_5_s = nn.Linear(800, 600)
        self.layer_5_a = nn.Linear(action_dim, 600)
        self.bn5 = nn.BatchNorm1d(600)
        self.layer_6 = nn.Linear(600, 1)
```

Initialize Network

Initializing Network
Collecting Data
Training Cycle

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Actor, self).__init__()
        self.layer_1 = nn.Linear(state_dim, 800)
        self.bn1 = nn.BatchNorm1d(800)
        self.layer_2 = nn.Linear(800, 600)
        self.bn2 = nn.BatchNorm1d(600)
        self.layer_3 = nn.Linear(600, action_dim)
        self.tanh = nn.Tanh()

    def forward(self, s):
        if s.size(0) > 1:
            s = F.relu(self.bn1(self.layer_1(s)))
            s = F.relu(self.bn2(self.layer_2(s)))
        else:
            s = F.relu(self.layer_1(s))
            s = F.relu(self.layer_2(s))
        a = self.tanh(self.layer_3(s))
        return a
```

```python
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        # First Critic Network
        self.layer_1 = nn.Linear(state_dim, 800)
        self.bn1 = nn.BatchNorm1d(800)
        self.layer_2_s = nn.Linear(800, 600)
        self.layer_2_a = nn.Linear(action_dim, 600)
        self.bn2 = nn.BatchNorm1d(600)
        self.layer_3 = nn.Linear(600, 1)

        # Second Critic Network
        self.layer_4 = nn.Linear(state_dim, 800)
        self.bn4 = nn.BatchNorm1d(8
        self.layer_5_s = nn.Linear(
        self.layer_5_a = nn.Linear(action_dim, 600)
        self.bn5 = nn.BatchNorm1d(600)
        self.layer_6 = nn.Linear(600, 1)
```

(parameter) action_dim: Any

Initializing Network
Collecting Data
Training Cycle

Initialize Network

```python
def forward(self, s, a):
    # Applying batch normalization only if batch size > 1 for the first critic network
    if s.size(0) > 1:
        s1 = F.relu(self.bn1(self.layer_1(s)))
        s1 = F.relu(self.bn2(self.layer_2_s(s1) + self.layer_2_a(a)))
    else:
        s1 = F.relu(self.layer_1(s))
        s1 = F.relu(self.layer_2_s(s1) + self.layer_2_a(a))
    q1 = self.layer_3(s1)

    # Applying batch normalization only if batch size > 1 for the second critic network
    if s.size(0) > 1:
        s2 = F.relu(self.bn4(self.layer_4(s)))
        s2 = F.relu(self.bn5(self.layer_5_s(s2) + self.layer_5_a(a)))
    else:
        s2 = F.relu(self.layer_4(s))
        s2 = F.relu(self.layer_5_s(s2) + self.layer_5_a(a))
    q2 = self.layer_6(s2)

    return q1, q2
```

Collecting Data

Initializing Network

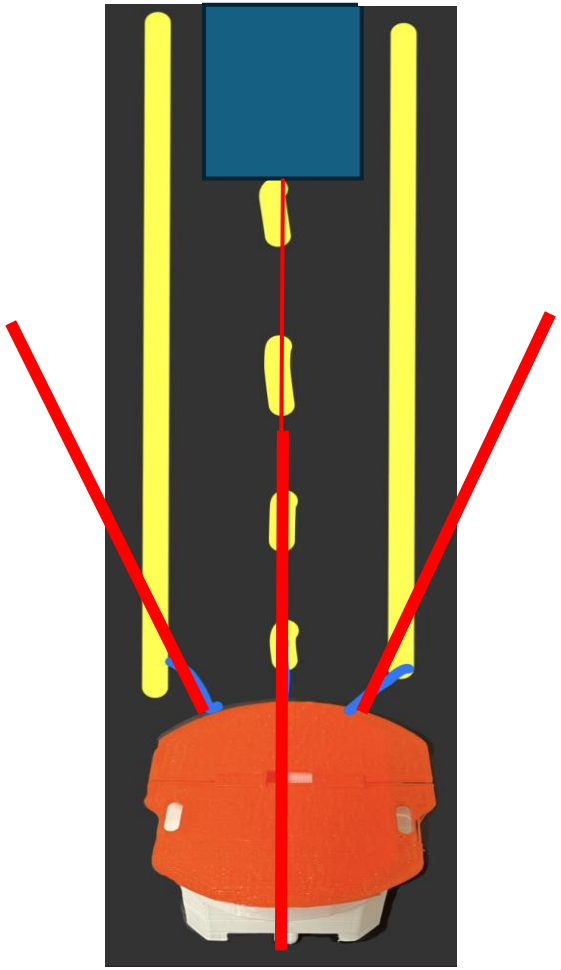
Collecting Data

Training Cycle

```python
class GazeboEnv(Node):
    def step(self, action):

        # read velodyne laser state
        done, collision, min_laser = self.observe_collision(velodyne_data)
        v_state = []
        v_state[:] = velodyne_data[:]
        laser_state = [v_state]

        # Calculate robot heading from odometry data
        self.odom_x = last_odom.pose.pose.position.x
        self.odom_y = last_odom.pose.pose.position.y
        quaternion = Quaternion(
            last_odom.pose.pose.orientation.w,
            last_odom.pose.pose.orientation.x,
            last_odom.pose.pose.orientation.y,
            last_odom.pose.pose.orientation.z,
        )
        euler = quaternion.to_euler(degrees=False)
        angle = round(euler[2], 4)
```



Initializing Network
Collecting Data
Training Cycle

Collecting Data

```
# Calculate distance to the goal from the robot
distance = np.linalg.norm(
    [self.odom_x - self.goal_x, self.odom_y - self.goal_y]
)

# Calculate the relative angle between the robots heading and heading toward the goal
skew_x = self.goal_x - self.odom_x
skew_y = self.goal_y - self.odom_y
dot = skew_x * 1 + skew_y * 0
mag1 = math.sqrt(math.pow(skew_x, 2) + math.pow(skew_y, 2))
mag2 = math.sqrt(math.pow(1, 2) + math.pow(0, 2))
beta = math.acos(dot / (mag1 * mag2))
```


```
skew_x = self.goal_x - self.odom_x
skew_y = self.goal_y - self.odom_y
dot = skew_x * 1 + skew_y * 0
mag1 = math.sqrt(math.pow(skew_x, 2) + math.pow(skew_y, 2))
mag2 = math.sqrt(math.pow(1, 2) + math.pow(0, 2))
beta = math.acos(dot / (mag1 * mag2))
```


```
    theta = np.pi - theta
```

[self.goal_x,
self.goal_y]

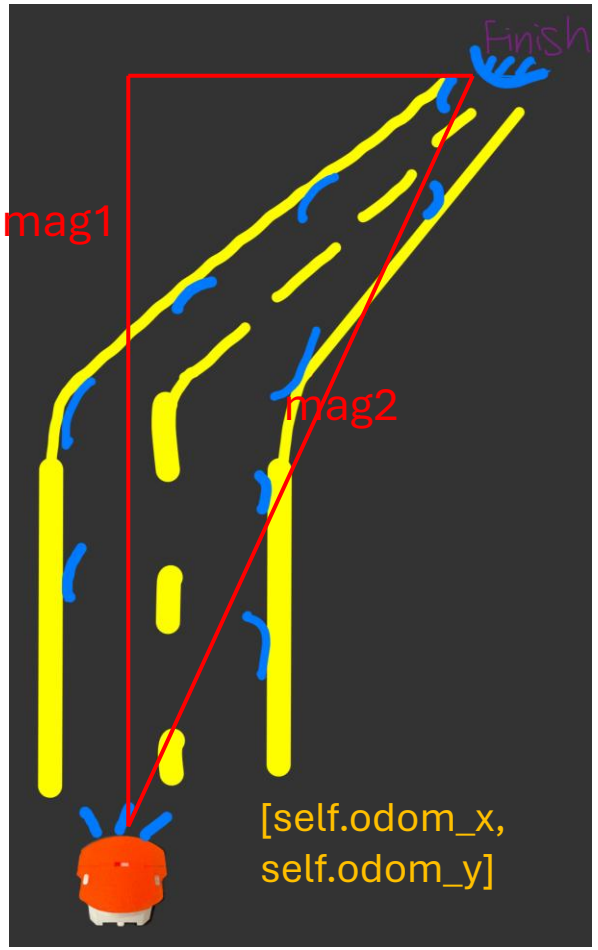
Collecting Data

Initializing Network
Collecting Data
Training Cycle

```python
# Calculate distance to the goal from the robot
distance = np.linalg.norm(
    [self.odom_x - self.goal_x, self.odom_y - self.goal_y]
)

# Calculate the relative angle between the robots heading and heading toward the goal
skew_x = self.goal_x - self.odom_x
skew_y = self.goal_y - self.odom_y
dot = skew_x * 1 + skew_y * 0
mag1 = math.sqrt(math.pow(skew_x, 2) + math.pow(skew_y, 2))
mag2 = math.sqrt(math.pow(1, 2) + m
beta = math.acos(dot / (mag1 * mag2
if skew_y < 0:
    if skew_x < 0:
        beta = -beta
    else:
        beta = 0 - beta
theta = beta - angle
if theta > np.pi:
    theta = np.pi - theta
    theta = -np.pi - theta
if theta < -np.pi:
    theta = -np.pi - theta
    theta = np.pi - theta
```


```python
if skew_y < 0:
    if skew_x < 0:
        beta = -beta
    else:
        beta = 0 - beta
theta = beta - angle
```

[self.goal_x, self.goal_y]

Initializing Network
Collecting Data
Training Cycle

Collecting Data

```
# Detect if the goal has been reached and give a large positive reward
if distance < GOAL_REACHED_DIST:
    env.get_logger().info("GOAL is reached!")
    target = True
    done = True

robot_state = [distance, theta, action[0], action[1]]
state = np.append(laser_state, robot_state)
reward = self.get_reward(target, collision, action, min_laser)
return state, reward, done, target
```

Collecting Data

- Initializing Network
- Collecting Data
- Training Cycle

```
# Detect if the goal has been reached and give a large positive reward
if distance < GOAL_REACHED_DIST:
    env.get_logger().info("GOAL is reached!")
    target = True
    done = True

robot_state = [distance, theta, action[0], action[1]]
state = np.append(laser_state, robot_state)
reward = self.get_reward(target, collision, action, min_laser)
return state, reward, done, target
```

Initializing Network
Collecting Data
Training Cycle

Training cycle

```
# training cycle
def train(
    self,
    replay_buffer,
    iterations,
    batch_size=100,
    discount=1,
    tau=0.005,
    policy_noise=0.2,  # discount=0.99
    noise_clip=0.5,
    policy_freq=2,
):
    av_Q = 0
    max_Q = -inf
    av_loss = 0
    for it in range(iterations):
        # sample a batch from the replay buffer
        batch_states, batch_actions, batch_rewards, batc
        #     batch_next_states = batch_next_states.resh

        if batch_next_states.shape != (batch_size, 24):
            print("Incorrect shape of batch_next_states,
            batch_next_states = batch_next_states.reshap
        # else:
            # print("Correct shape of batch_next_states"
```

```
def train(
    self,
    replay_buffer,
    iterations,
    batch_size=100,
    discount=1,
    tau=0.005,
    policy_noise=0.2,  # discount=0.99
    noise_clip=0.5,
    policy_freq=2,
```

Initializing Network
Collecting Data
Training Cycle

Training Cycle

```
state = torch.Tensor(batch_states).to(device)
next_state = torch.Tensor(batch_next_states).to(device)
action = torch.Tensor(batch_actions).to(device)
reward = torch.Tensor(batch_rewards).to(device)
done = torch.Tensor(batch_dones).to(device)

# Obtain the estimated action from the next state by using the actor-target
next_action = self.actor_target(next_state)

# Add noise to the action
noise = torch.Tensor(batch_actions).data.normal_(0, policy_noise).to(device)
noise = noise.clamp(-noise_clip, noise_clip)
```

Soft Save


```
state = torch.Tensor(batch_states).to(device)
next_state = torch.Tensor(batch_next_states).to(device)
action = torch.Tensor(batch_actions).to(device)
reward = torch.Tensor(batch_rewards).to(device)
done = torch.Tensor(batch_dones).to(device)
```


```
# Calculate the loss between the current Q value and the target Q value
loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q)
```

- Initializing Network
- Collecting Data
- Training Cycle

Training Cycle

```
state = torch.Tensor(batch_states).to(device)
next_state = torch.Tensor(batch_next_states).to(device)
action = torch.Tensor(batch_actions).to(device)
reward = torch.Tensor(batch_rewards).to(device)
done = torch.Tensor(batch_dones).to(device)
```

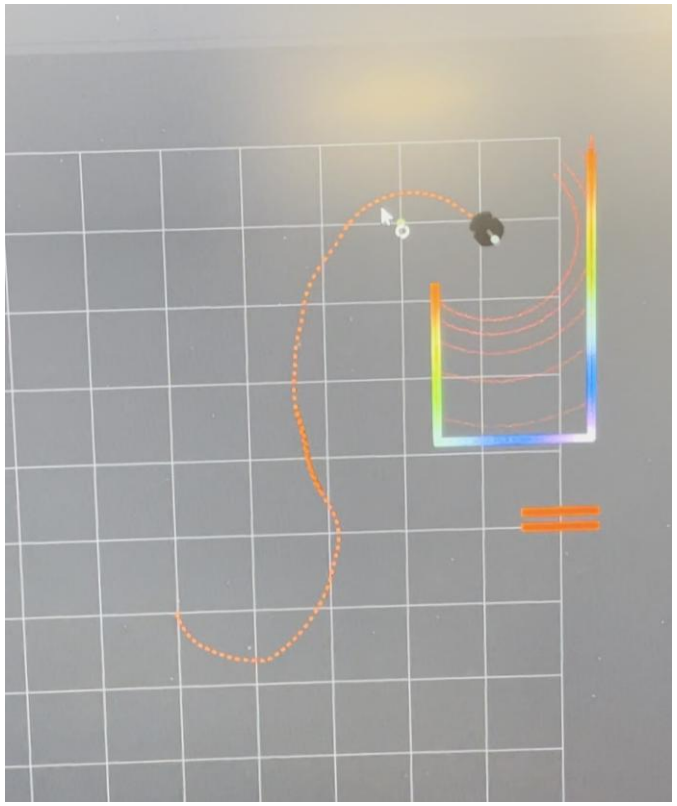
```
# Select the minimal Q value from the 2 calculated values
target_Q = torch.min(target_Q1, target_Q2)
av_Q += torch.mean(target_Q)
max_Q = max(max_Q, torch.max(target_Q))
```

```
target_Q1, target_Q2 = self.critic_target(next_state, next_action)

# Select the minimal Q value from the 2 calculated values
target_Q = torch.min(target_Q1, target_Q2)
av_Q += torch.mean(target_Q)
max_Q = max(max_Q, torch.max(target_Q))
# Calculate the final Q value from the target network parameters by using Bellman equation
target_Q = reward + ((1 - done) * discount * target_Q).detach()

# Get the Q values of the basis networks with the current parameters
current_Q1, current_Q2 = self.critic(state, action)

# Calculate the loss between the current Q value and the target Q value
loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q)
```



```
if it % policy_freq == 0:
    # Maximize the actor output value by performing gradient descent on negative Q
    # (essentially perform gradient ascent)
    actor_grad, _ = self.critic(state, self.actor(state))
    actor_grad = -actor_grad.mean()
    self.actor_optimizer.zero_grad()
    actor_grad.backward()
    self.actor_optimizer.step()

    # Use soft update to update the actor-target network parameters by
    # infusing small amount of current parameters
    for param, target_param in zip(
        self.actor.parameters(), self.actor_target.parameters()
    ):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )
    # Use soft update to update the critic-target network parameters by infusing
    # small amount of current parameters
    for param, target_param in zip(
        self.critic.parameters(), self.critic_target.parameters()
    ):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )

av_loss += loss
```

Training Cycle

Initializing Network

Collecting Data

Training Cycle

Evaluation

```python
def evaluate(network, epoch, eval_episodes=10, logging_frequency=10):
    avg_reward = 0.0
    col = 0
    for episode in range(eval_episodes):
        if episode % logging_frequency == 0:
            env.get_logger().info(f"evaluating episode {episode}")
        count = 0
        state = env.reset()
        done = False
        while not done and count < 501:
            action = network.get_action(np.array(state))
            if count % logging_frequency == 0:
                env.get_logger().info(f"action : {action}")
            a_in = [(action[0] + 1) / 2, action[1]]
            state, reward, done, _ = env.step(a_in)
            avg_reward += reward
            count += 1
            if reward < -90:
                col += 1
    avg_reward /= eval_episodes
    avg_col = col / eval_episodes
    if logging_frequency >= 1:
        env.get_logger().info(
            "Average Reward over %i Evaluation Episodes, Epoch %i: avg_reward %f, avg_col %f"
            % (eval_episodes, epoch, avg_reward, avg_col)
        )
    return avg_reward
```

Evaluation

```
def evaluate(network, epoch, eval_episodes=10, logging_frequency=10):
    avg_reward = 0.0
    col = 0
    ...
            avg_reward += reward
            count += 1
            if reward < -90:
                col += 1
    avg_reward /= eval_episodes
    avg_col = col / eval_episodes
    if logging_frequency >= 1:
        env.get_logger().info(
            "Average Reward over %i Evaluation Episodes, Epoch %i: avg_reward %f, avg_col %f"
            % (eval_episodes, epoch, avg_reward, avg_col)
        )
    return avg_reward
```


```
avg_reward /= eval_episodes
avg_col = col / eval_episodes
if logging_frequency >= 1:
    env.get_logger().info(
        "Average Reward over %i Evaluation Episodes, Epoch %i: avg_reward %f, avg_col %f"
        % (eval_episodes, epoch, avg_reward, avg_col)
    )
return avg_reward
```

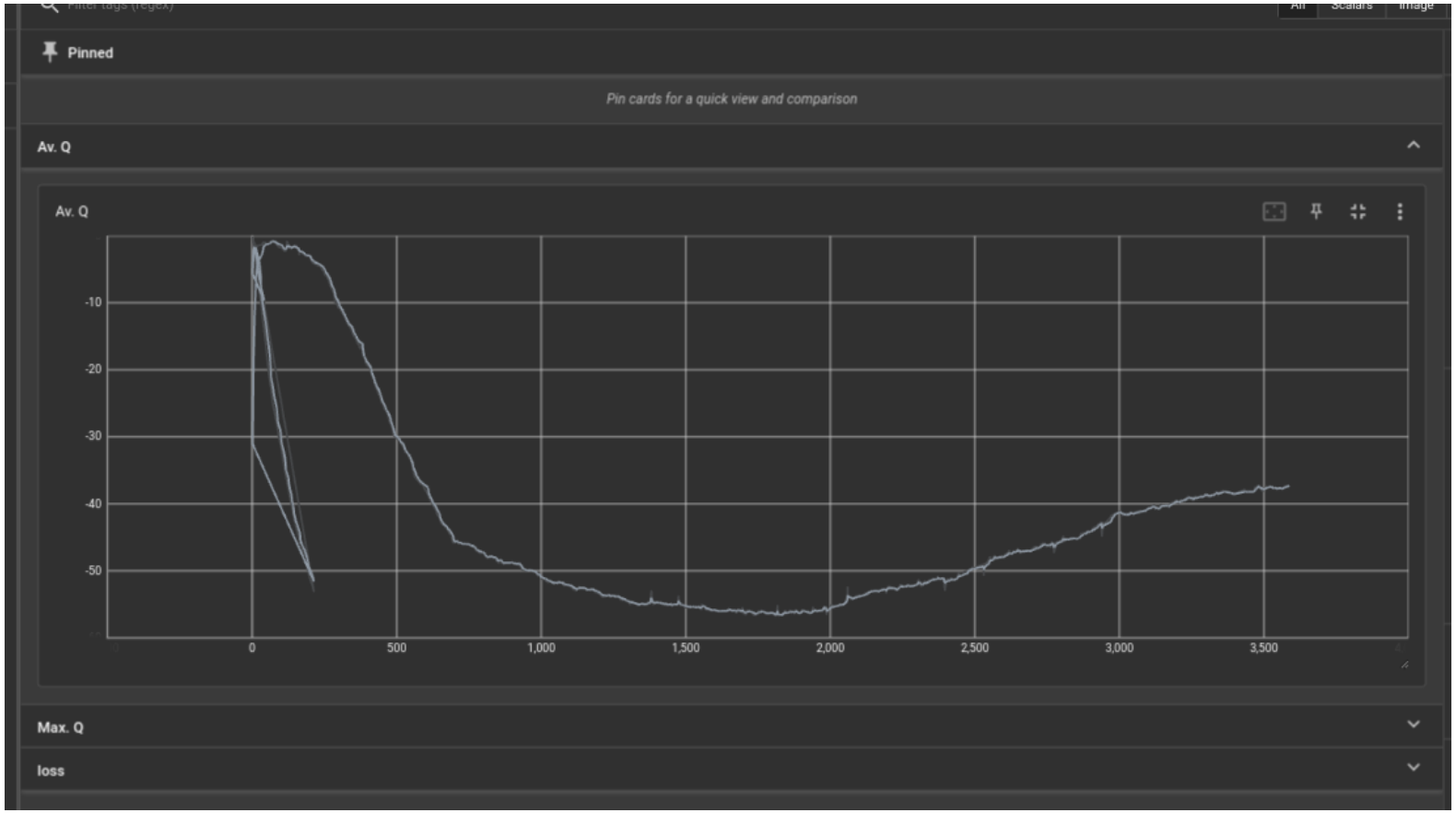
Save Data

```
def save(self, filename, directory):
    torch.save(self.actor.state_dict(), "%s/%s_actor.pth" % (directory, filename))
    torch.save(self.critic.state_dict(), "%s/%s_critic.pth" % (directory, filename))
```

```
if timesteps_since_eval >= eval_freq:
    env.get_logger().info("Validating")
    timesteps_since_eval %= eval_freq
    evaluations.append(
        evaluate(network=network, epoch=epoch, eval_episodes=eval_ep)
    )

    network.save(file_name, directory="/home/kam/capstone_ws/src/td3/runs/train/pytorch_models")
    np.save("/home/kam/capstone_ws/src/td3/runs/train/results/%s" % (file_name), evaluations)
    epoch += 1
```

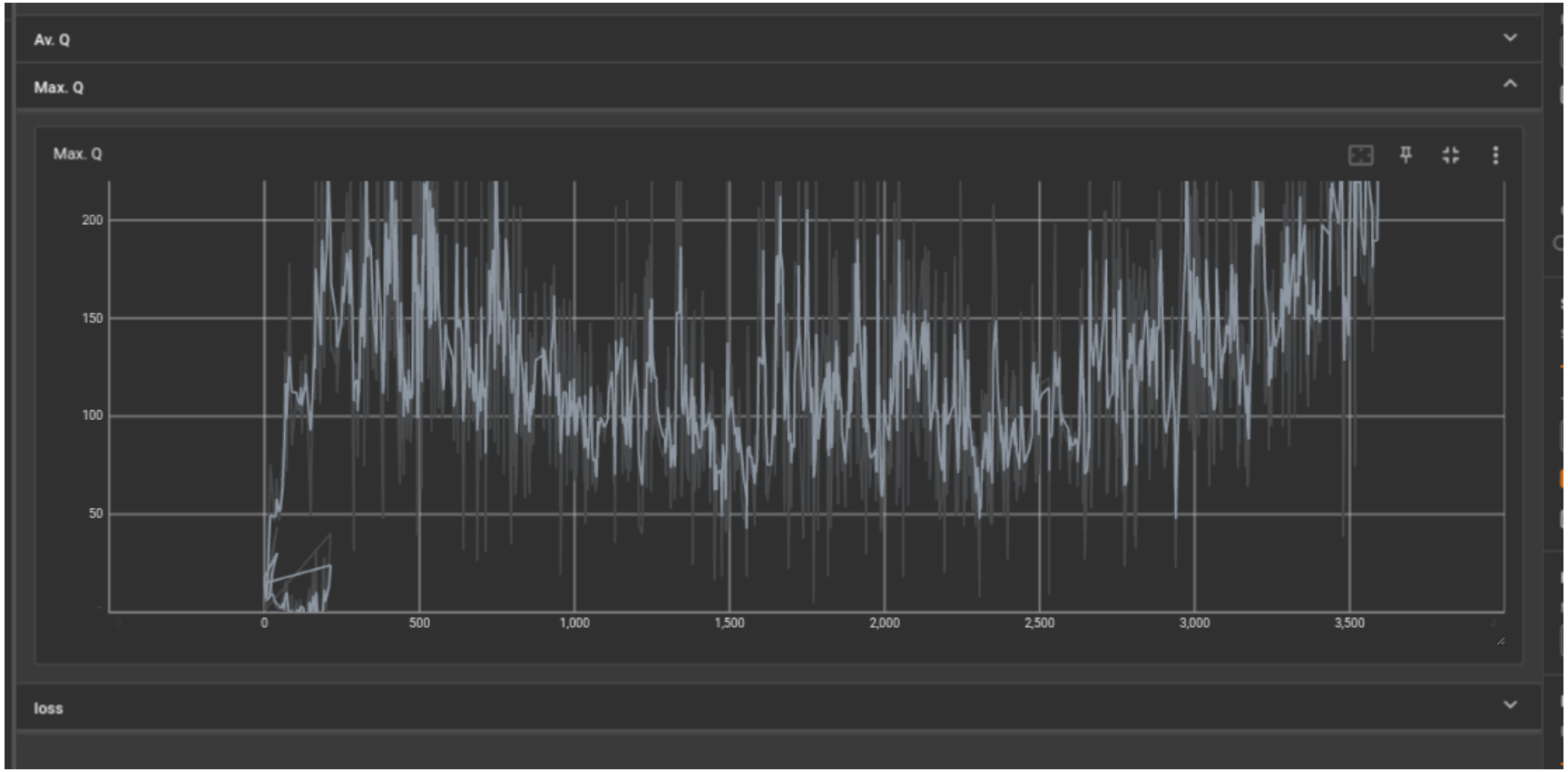
Virtual Model Results



Avg. Q

- Shows a high initial goal expectation, however the after 2000 iterations there is a slight improvement
- Growth and learning is displayed

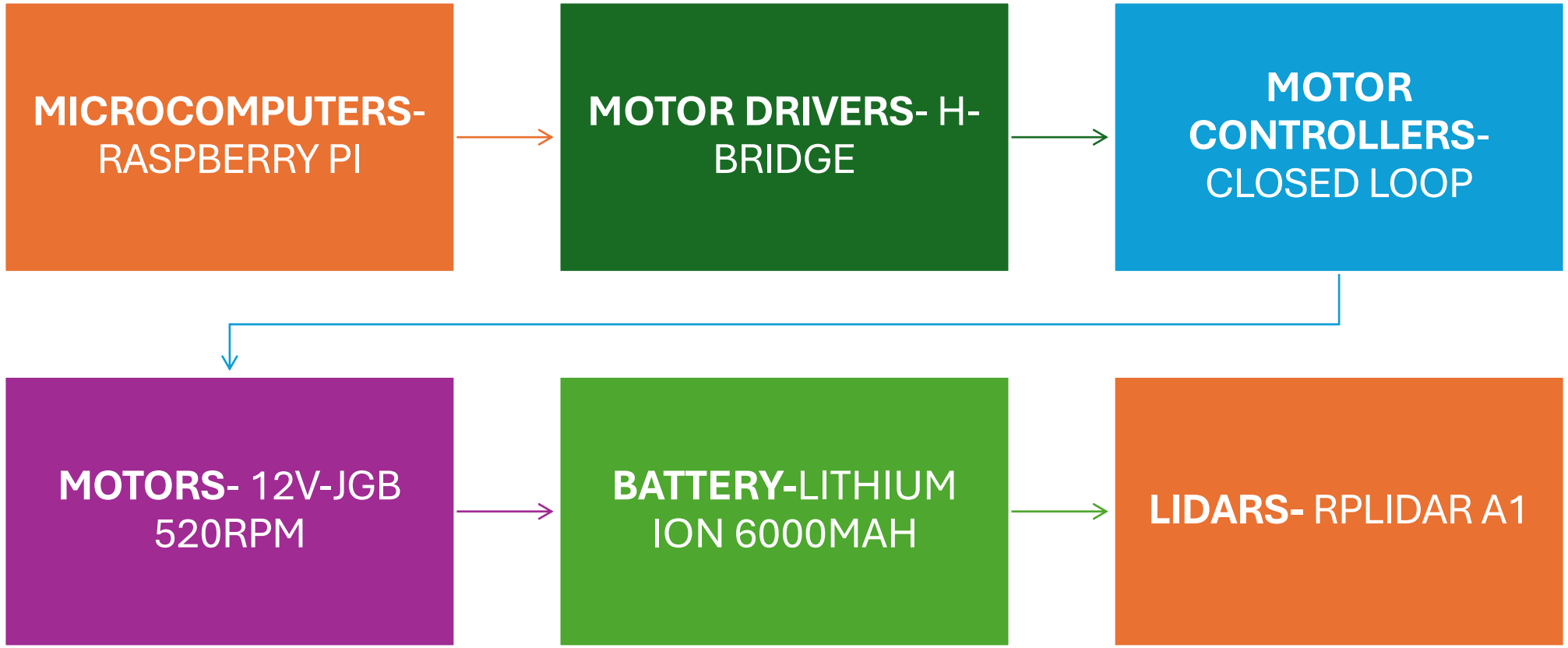
Virtual Model Results



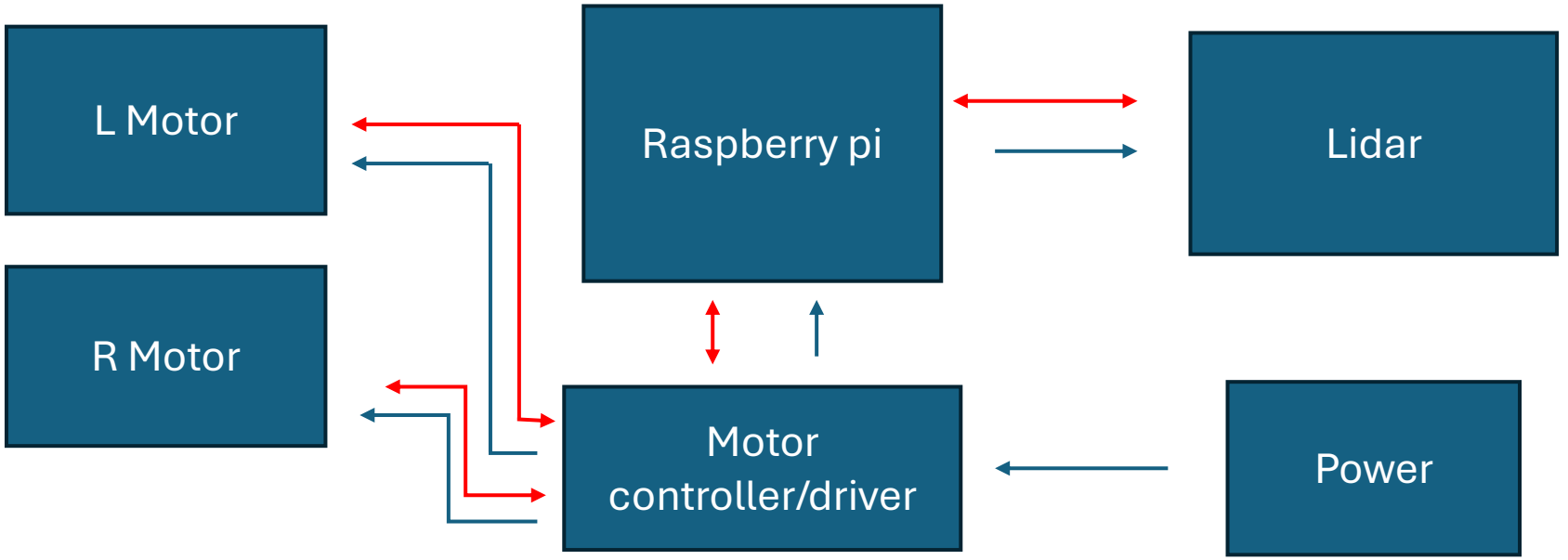
Max Q.

Shows within the 1.227 days of training there was no identifiable pattern found, leading to poor initial performance.

Hardware


MICROCOMPUTERS- RASPBERRY PI
MOTOR DRIVERS- H-BRIDGE
MOTOR CONTROLLERS- CLOSED LOOP
MOTORS- 12V-JGB 520RPM
BATTERY-LITHIUM ION 6000MAH
LIDARS- RPLIDAR A1

Connection Guide



Calculations and Output

Components

- Raspberry Pi 4b~ 3.4W→7.6W with an average of 5.5W (0.46A)
- RPLidar A1~5W (0.42A)
- (2) 12V 520 RPM JGB-520 Motors ~0.6A No load 1.5A underload 12V*1.5A=18W
- Motor driver/controller ~2W

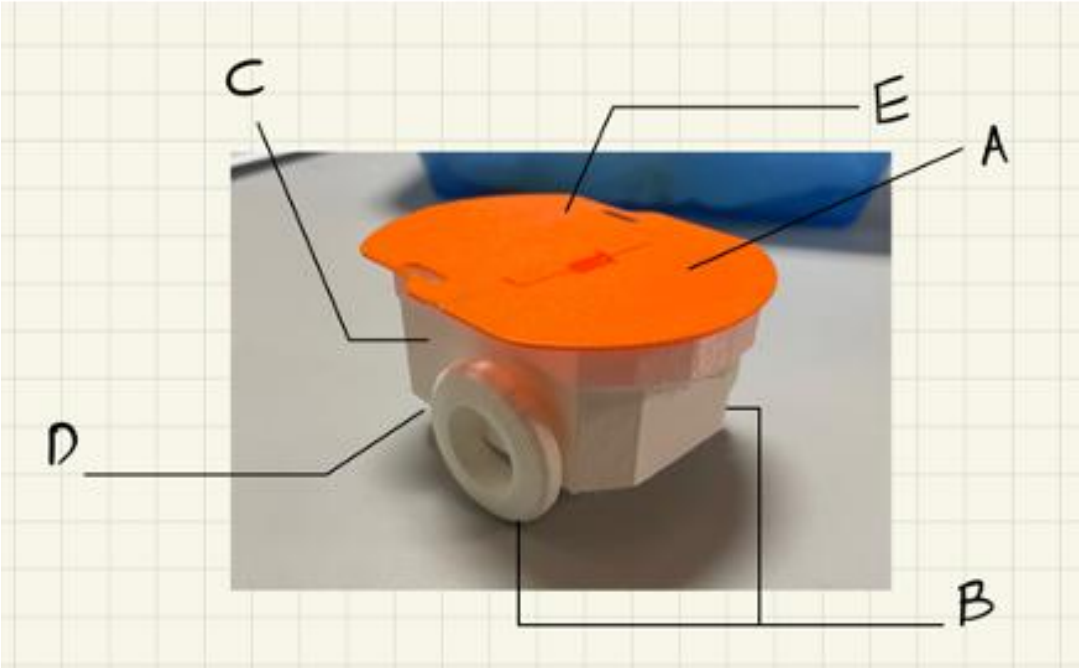
Total= 5.5W +5W+(2)18W=46.5W
Run time =30 min= 0.5hr
Battery Capacity= 46.5 W* 0.5hr= 23.25Whr

Amp hours = $\frac{Wh}{V} = \frac{23.25}{12V} = 1.94Ah = 1940mAh$

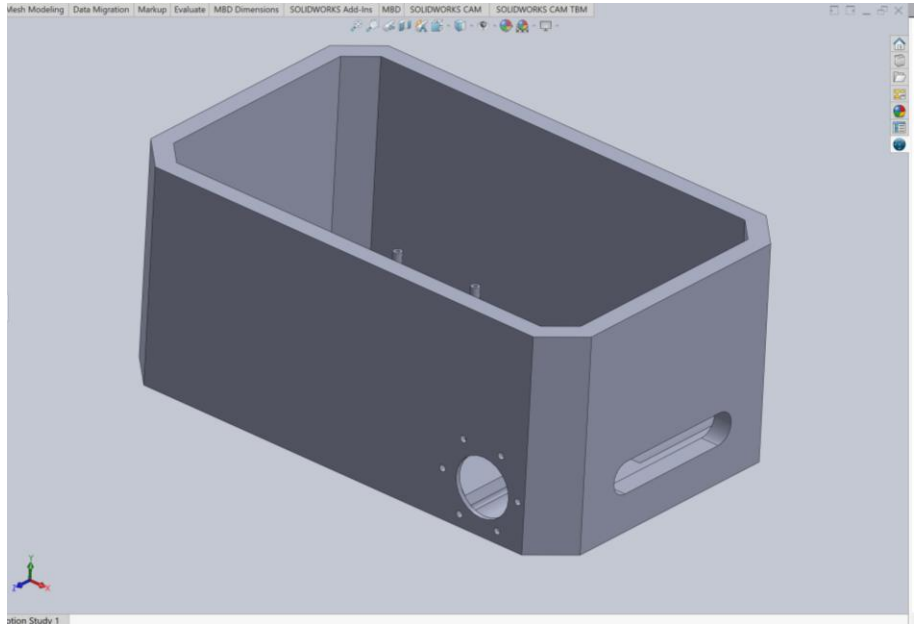
Current Battery =6000 mAh
Operation time of battery= $\frac{6ah}{8.88A} = 1.55hrs \approx 93\ minutes\ of\ Operation$

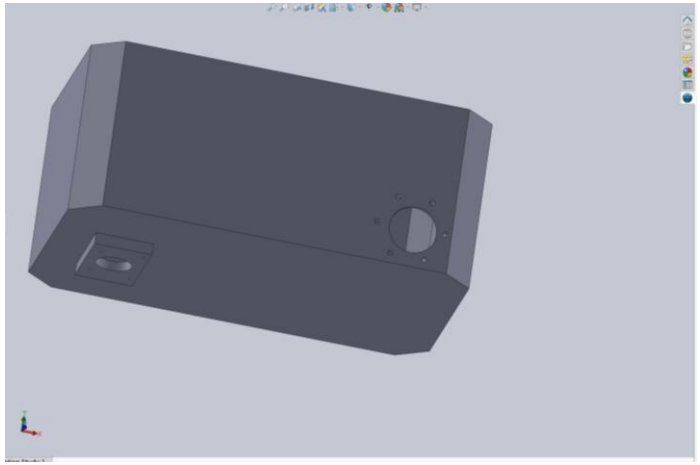
Prototype Development – Part Identification

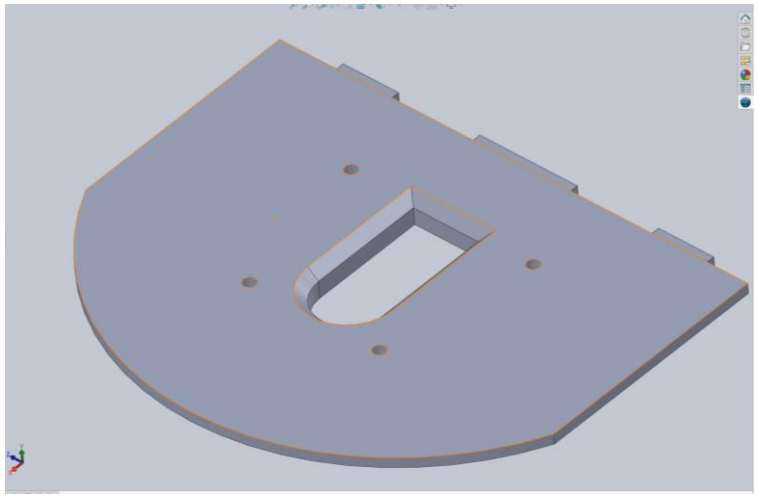
	Parts List
A	Mounting Bracket for Lidar
B	Motor Placement & Wheel Location
C	Robot Chasis(Body)
D	Caster Ball
E	Detachable Cover

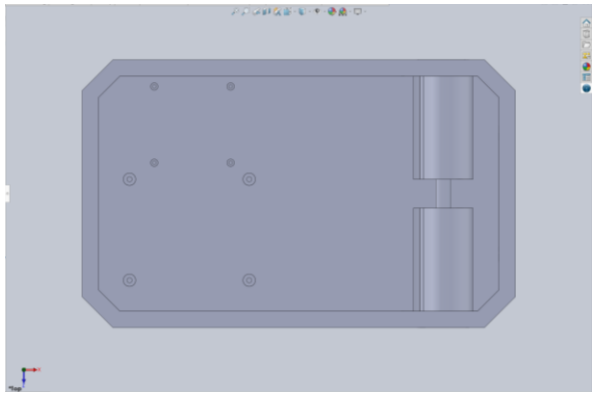


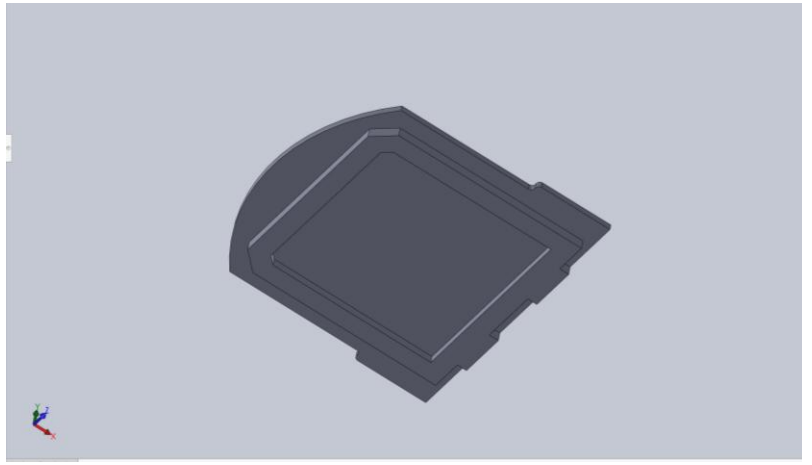
Prototype Development– Section(s)



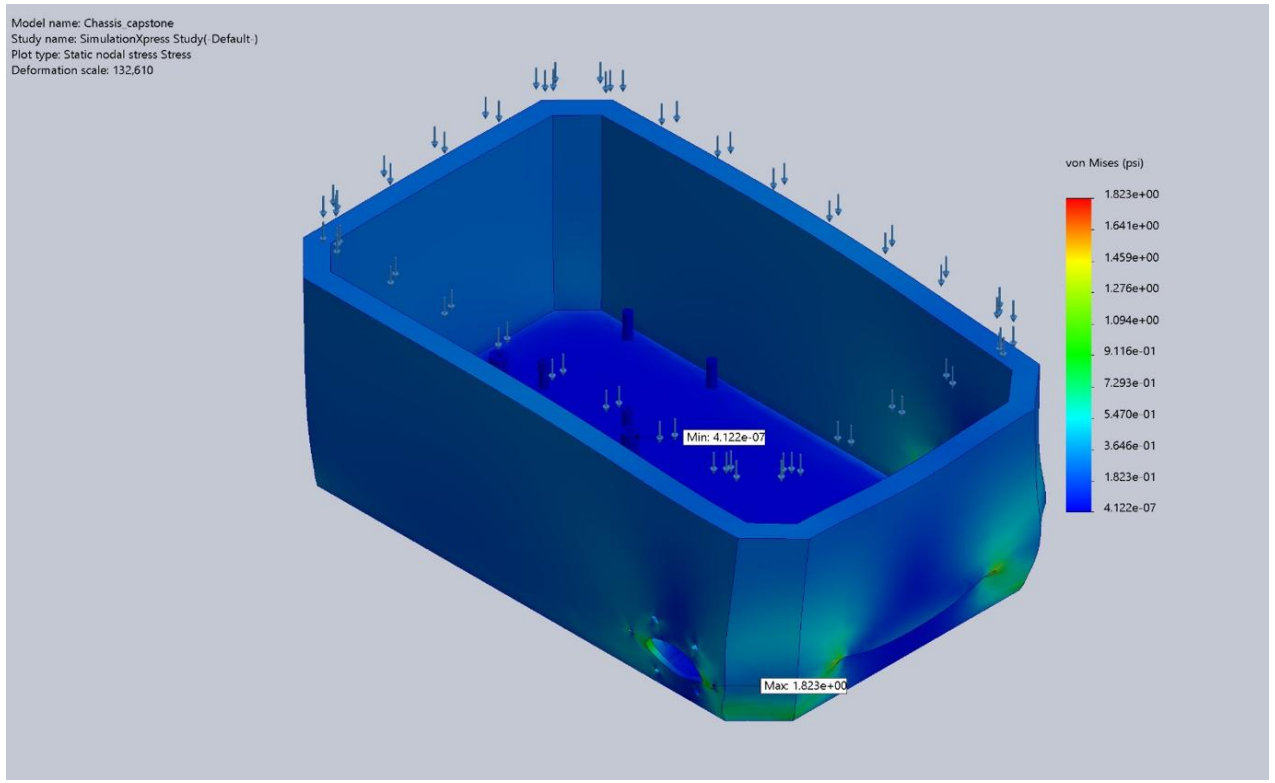






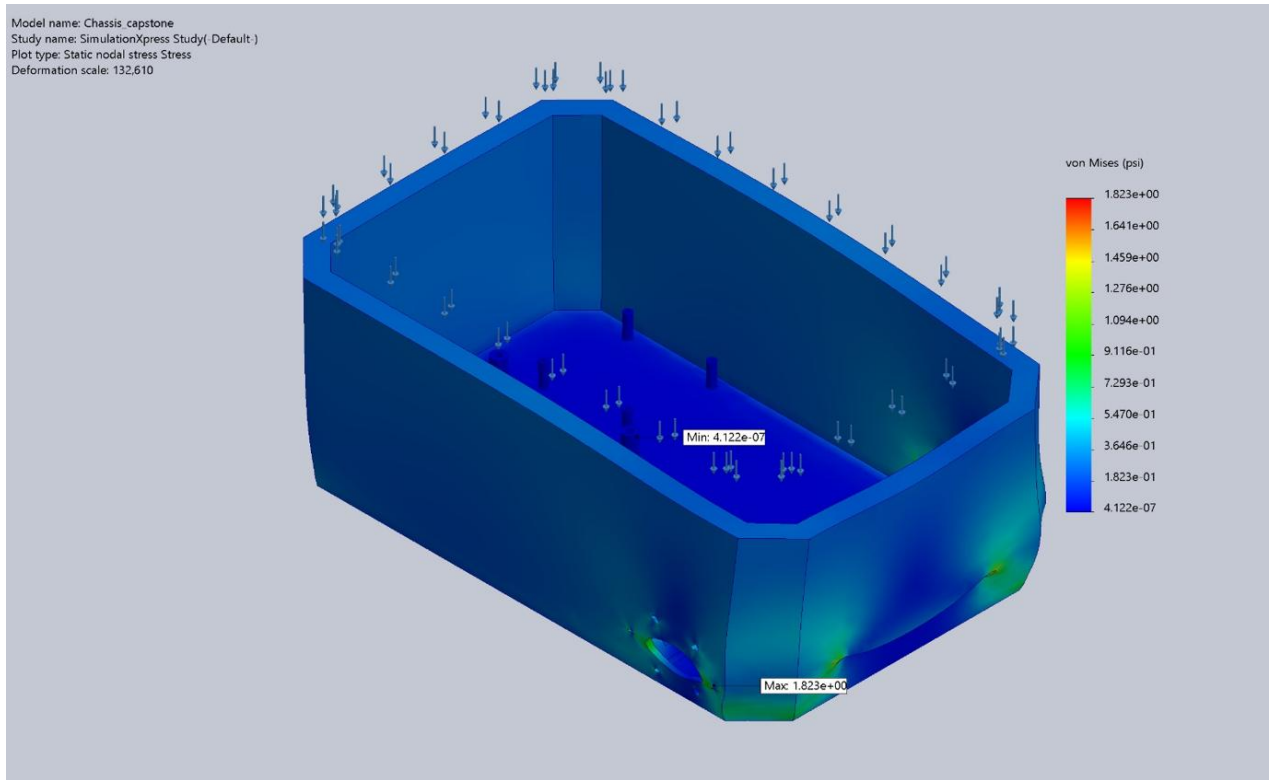


Chassis Stress Calculation



- All parts subjected to a load of 2 lbs
- Max stress: 1.823 psi
- Factor of Safety: $\frac{3782.74\ psi}{1.823\ psi} = 2075$

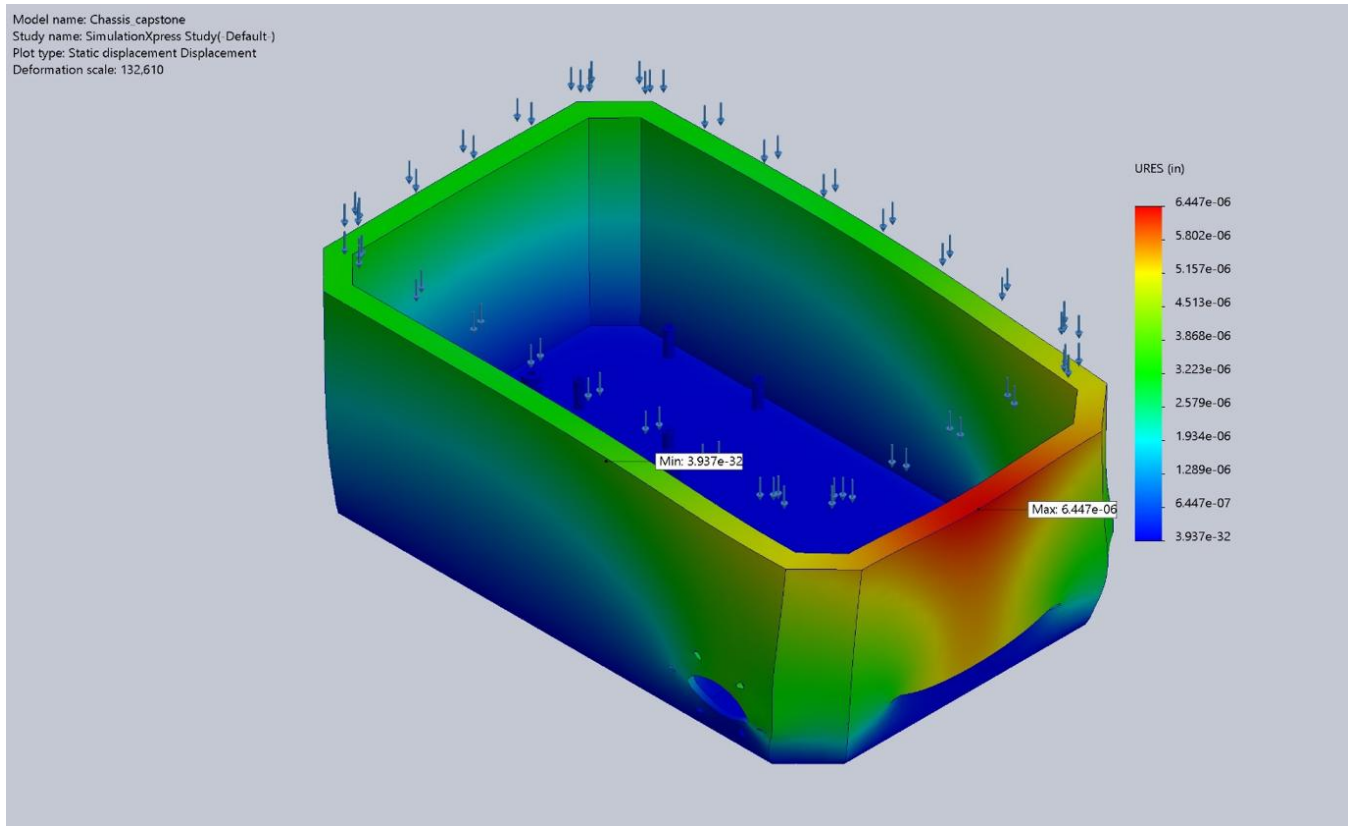
Chassis Stress Calculation



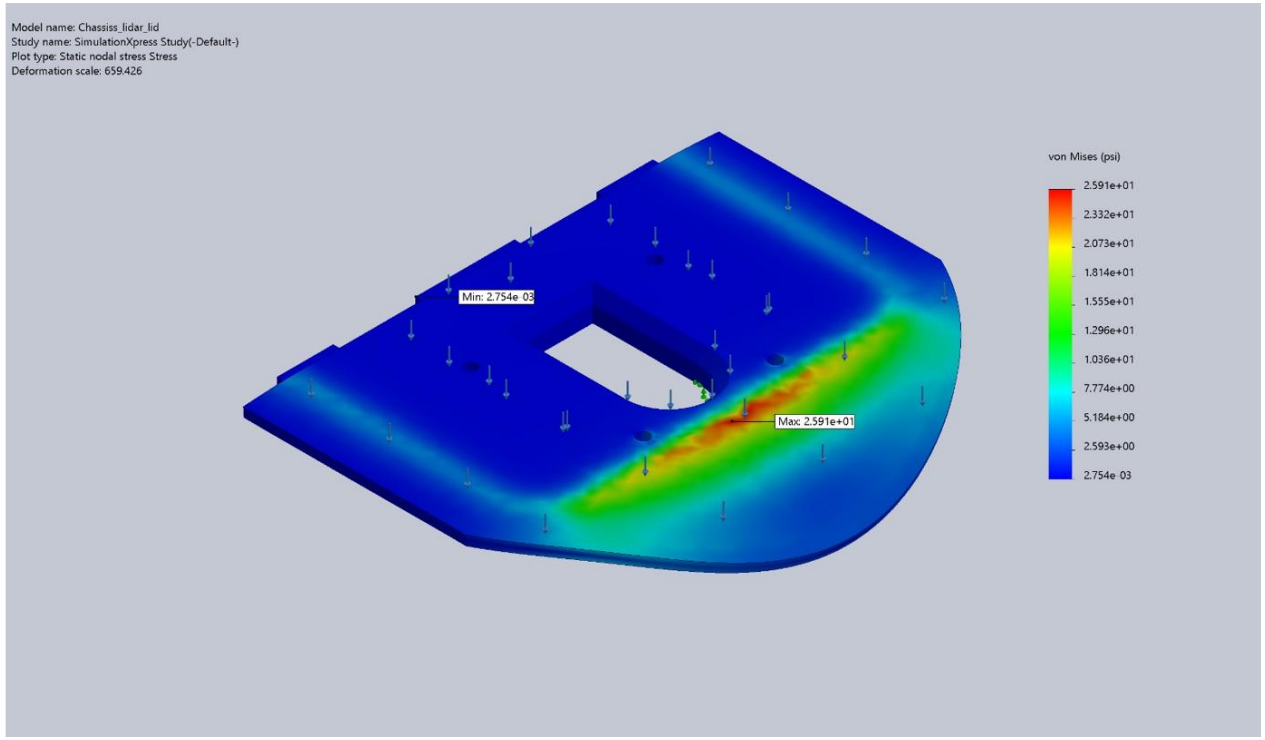
- Max stress: 1.823 psi
- Factor of Safety: $\frac{3782.74\ psi}{1.823\ psi} = 2075$

Chassis Load Calculation

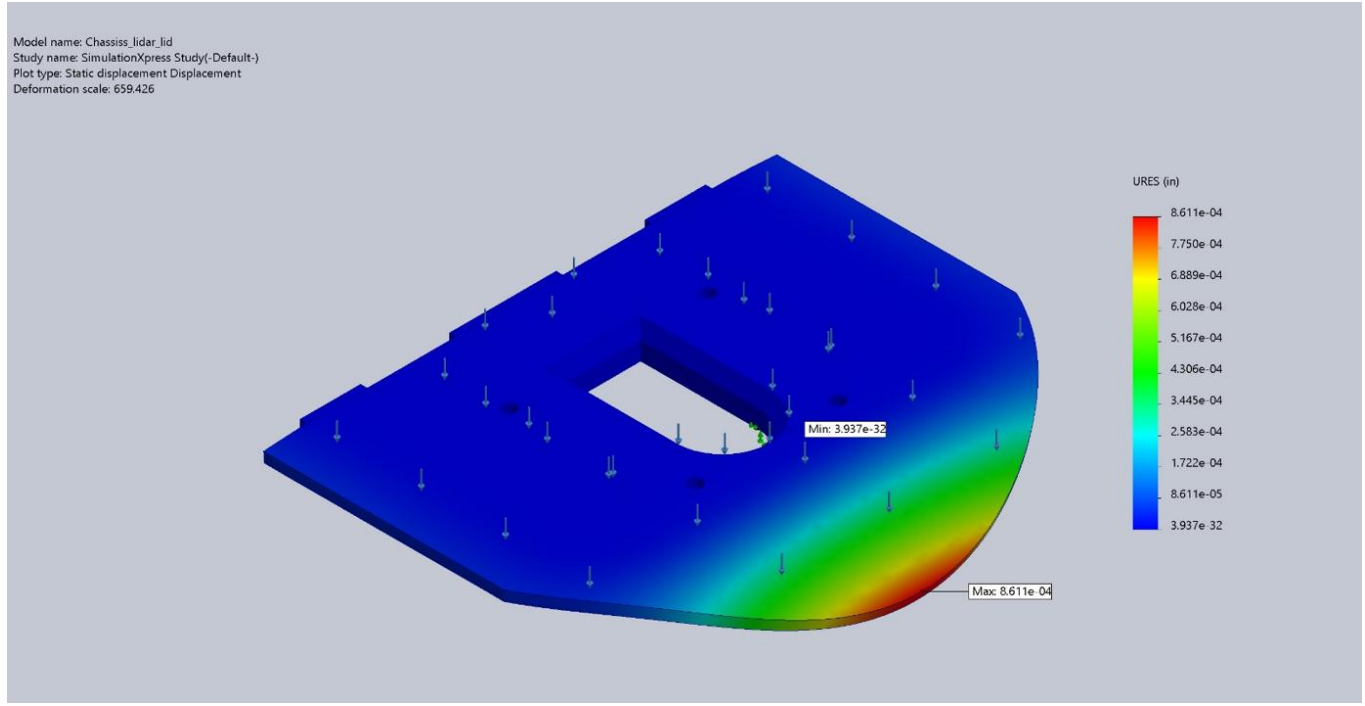
- Max deformation: 6.447e-04 in.



Lidar Mount Stress Calculation



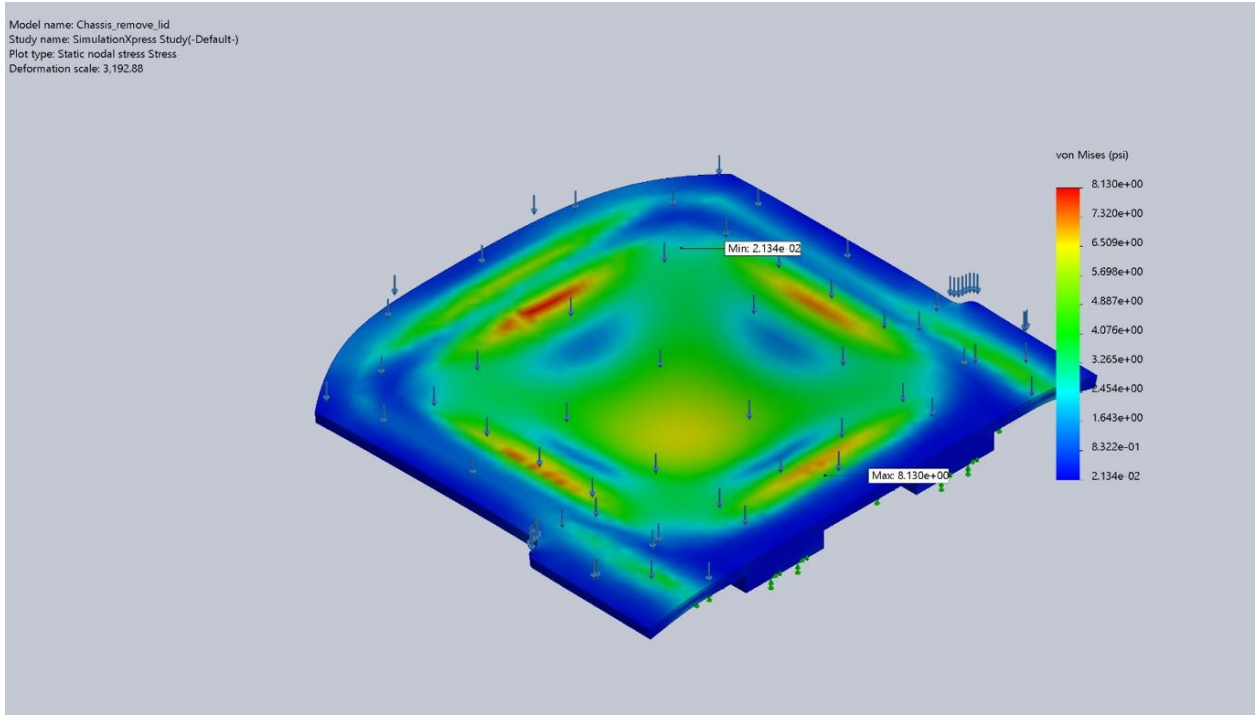
- Max Stress: 25.91 psi
- Factor of Safety:
$$\frac{3782.74\ psi}{25.91\ psi} = 145.995$$



Lidar Mount Load Calculation

- Max Deformation: 8.61e-04 in.

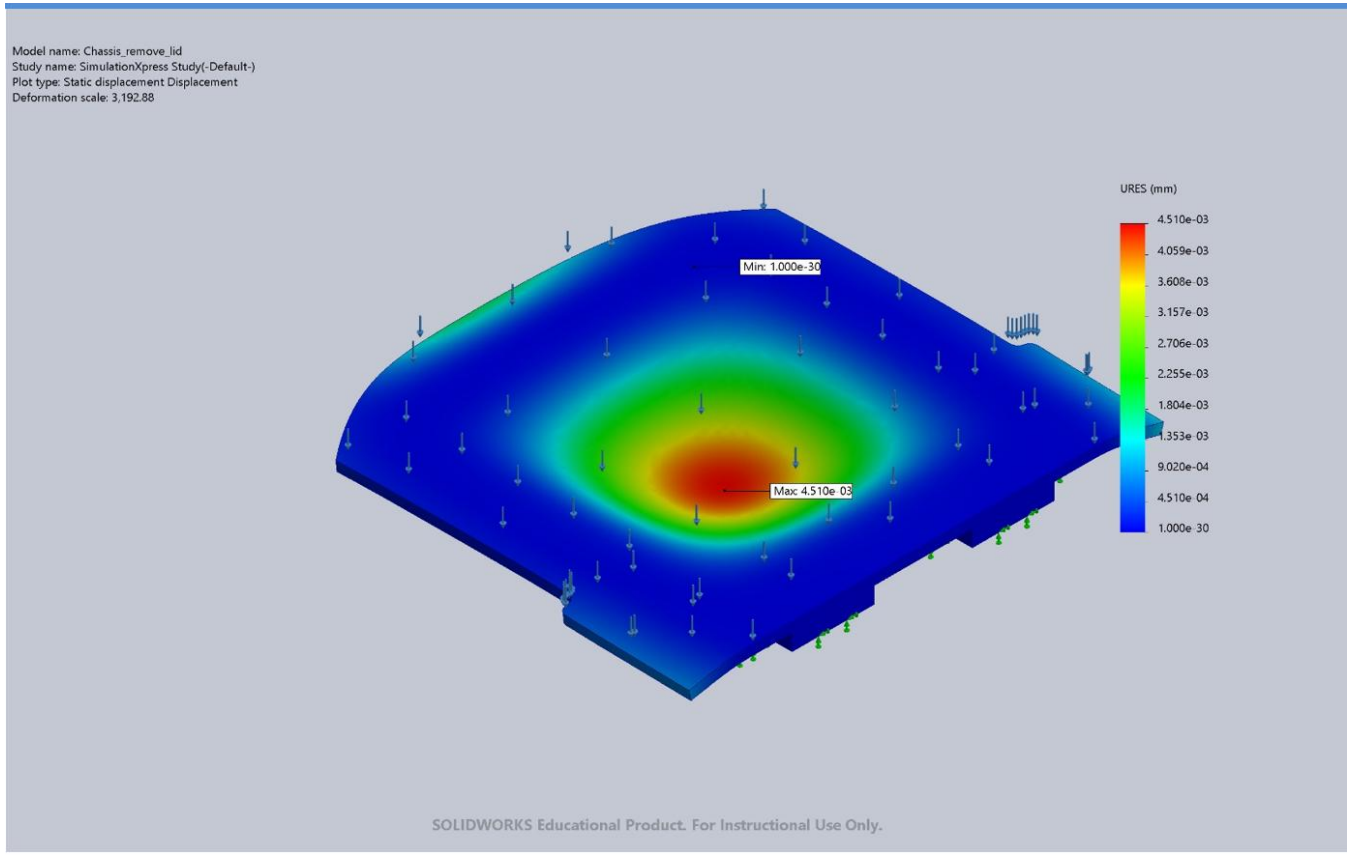
Back Plate Stress Calculation



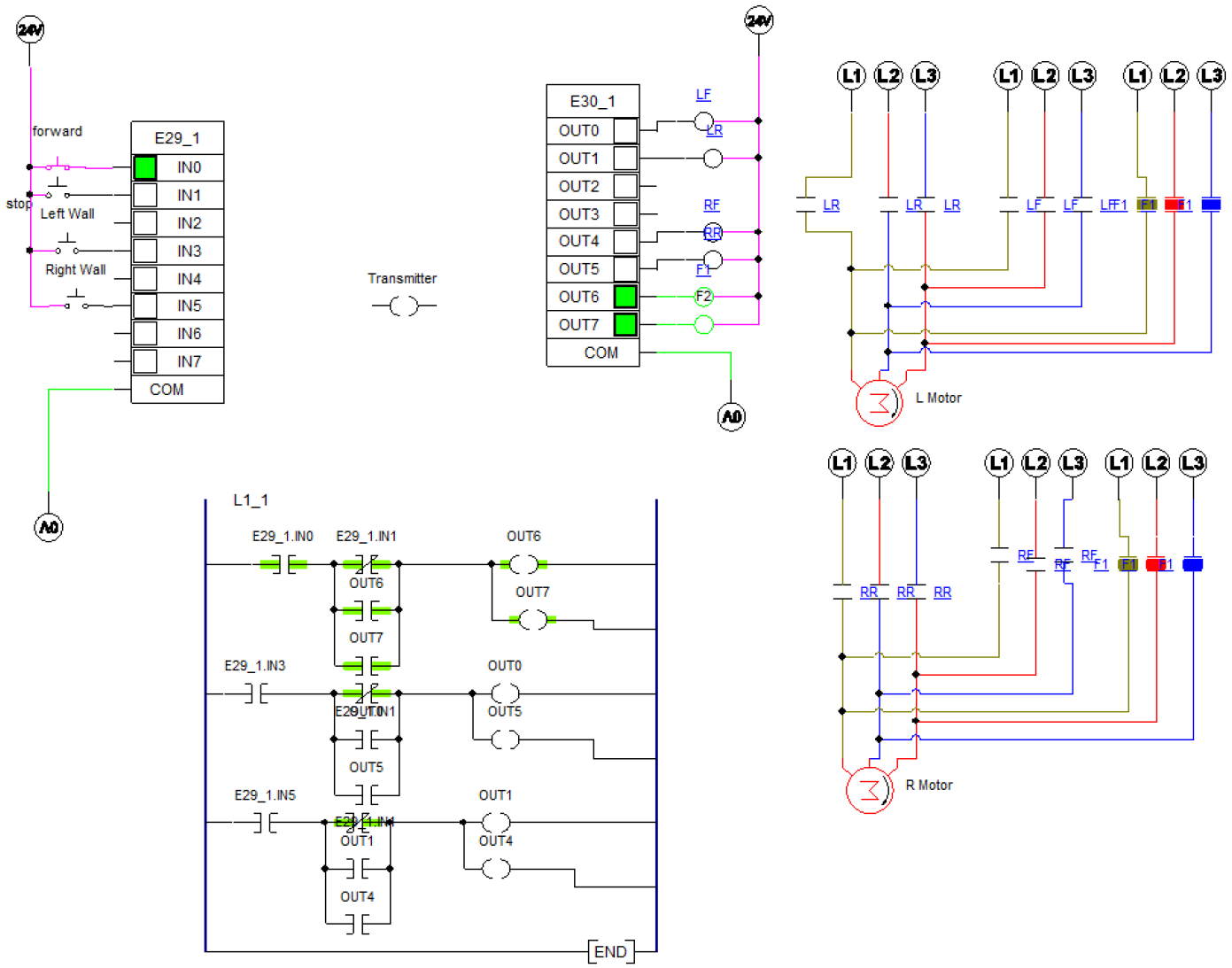
- Max stress: 8.13 psi
- Factor of Safety:
$$\frac{3782.74\ psi}{8.13\ psi} = 465.28$$

Back Plate Load Calculation

- Max displacement: 4.513-03 in.

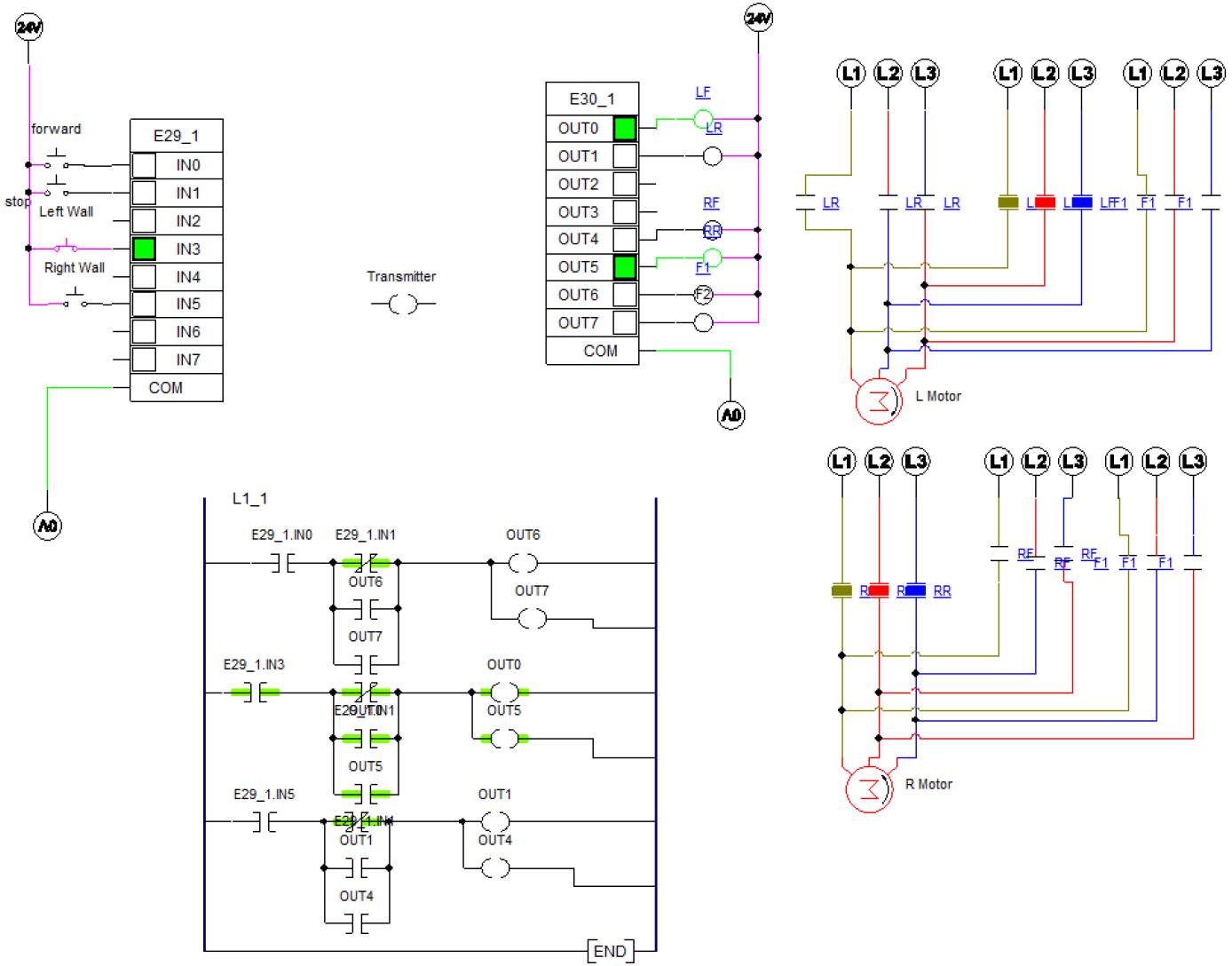


Automation Studio



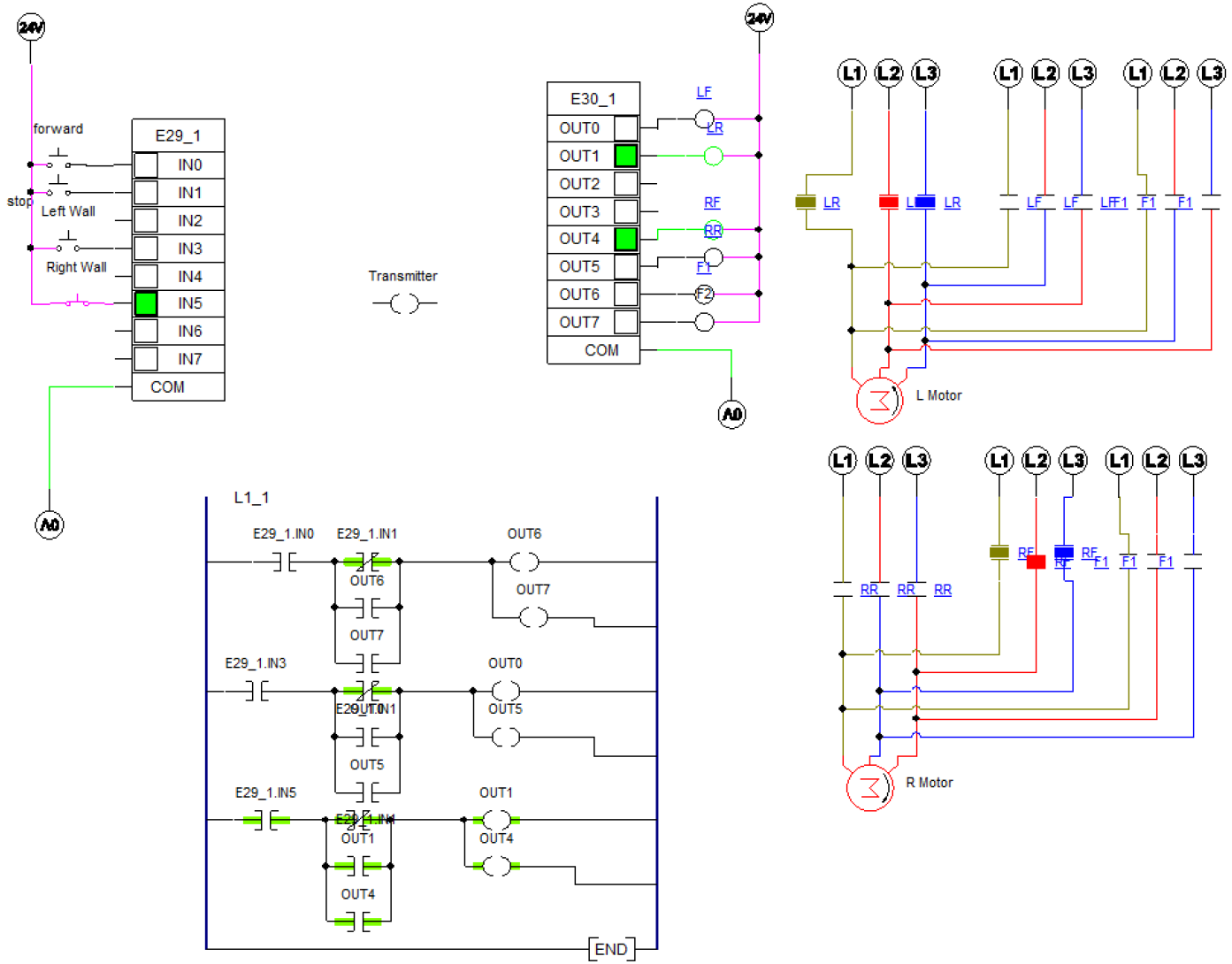
- PLC wired for both motors to move forward

Automation Studio



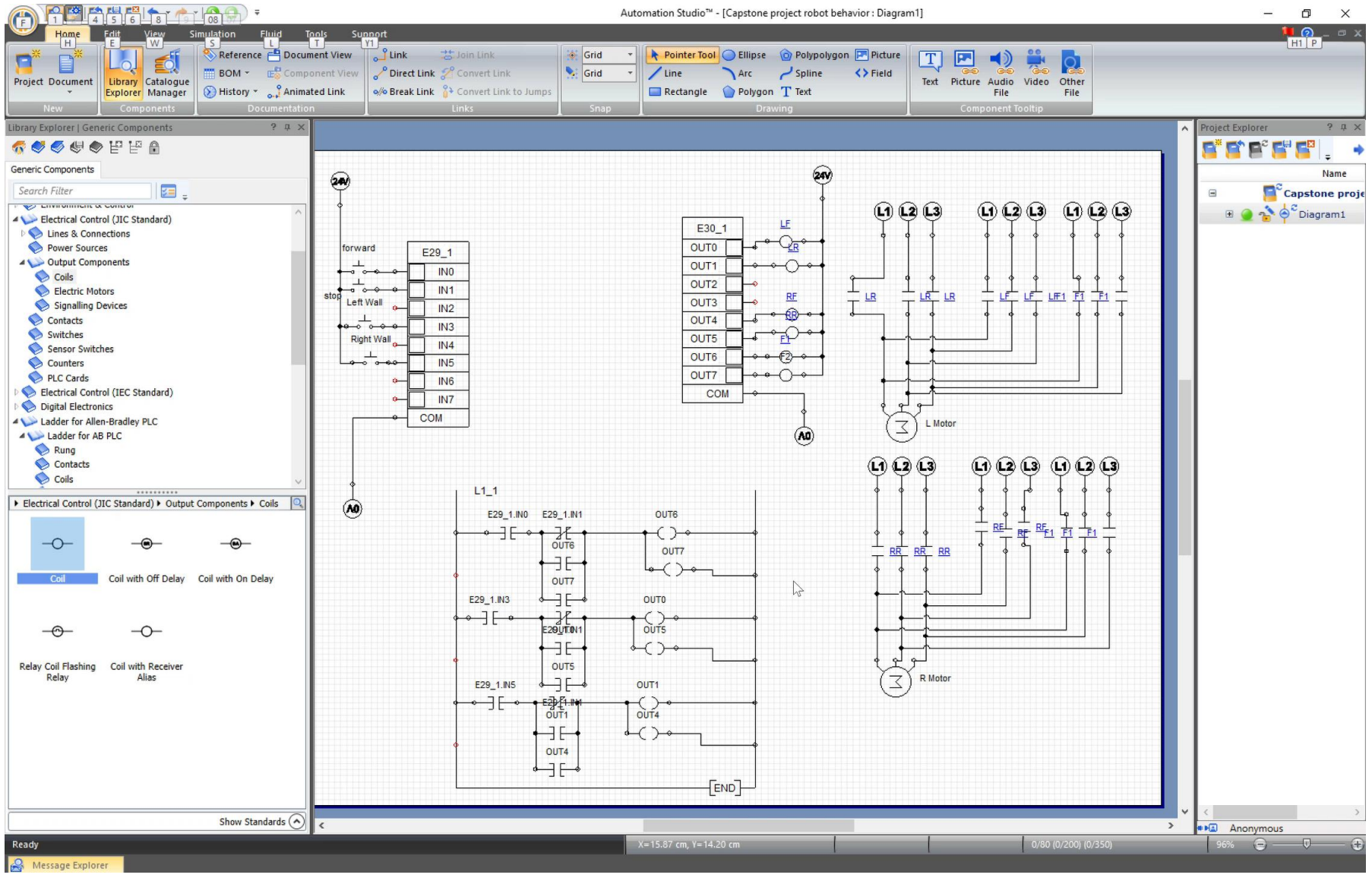
- Modified to account for sensor sensing a object to its left and moving right accordingly

Automation Studio

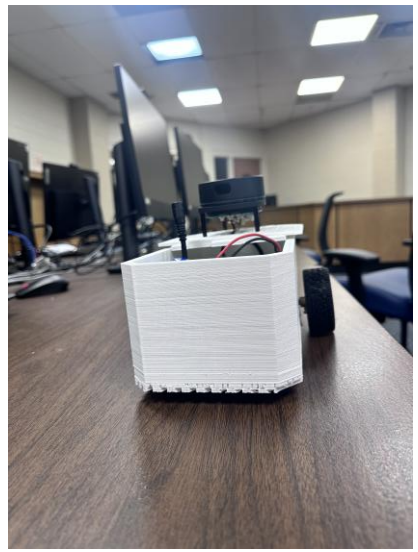


- Final PLC accounts for objects to the right and left of the robot

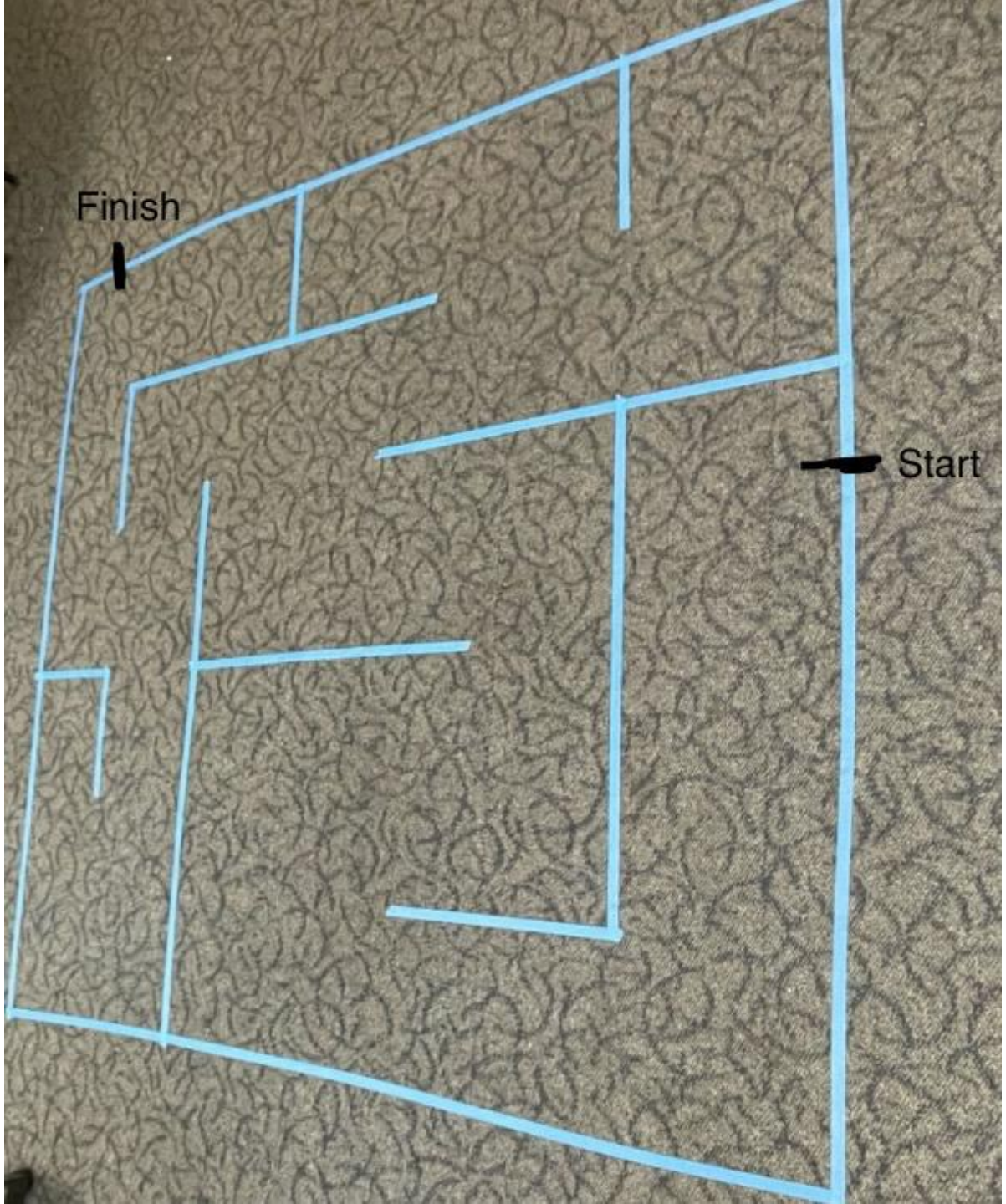
Automation Studio Video

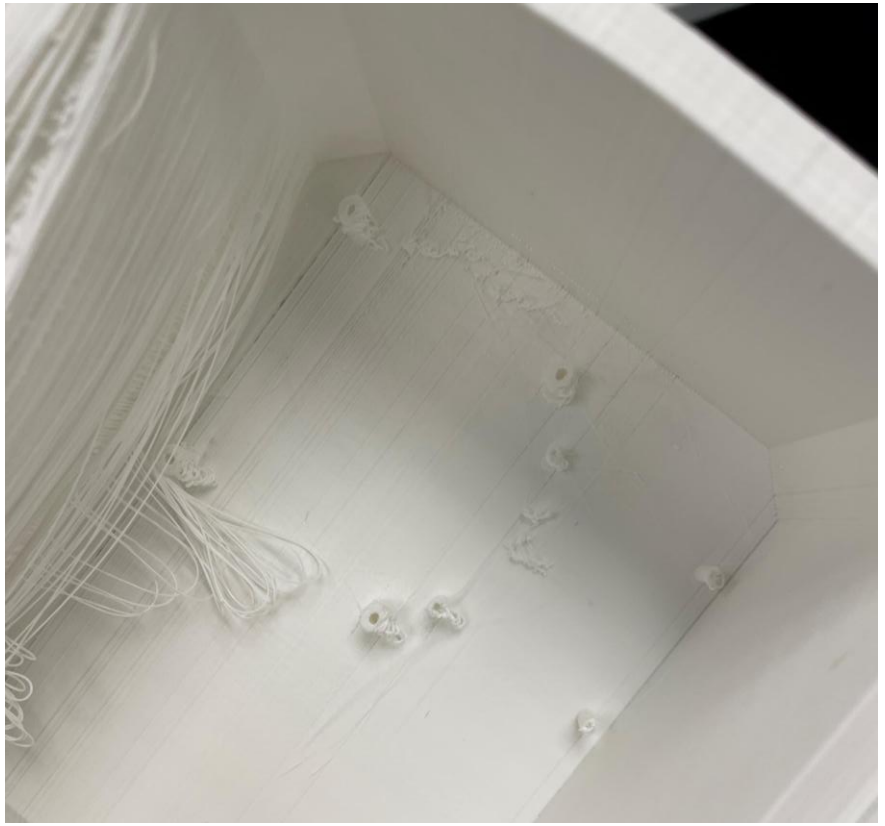


Prototype



Current Maze





Creation Challenges

- 3D Printing Failures
- Uploading the virtual model neural network to the prototype

Implementation Challenges

- Neural network initialization
- Network connection (SSH protocol)
- Data Optimization

Conclusion

- Created a virtual robot that works
- Unable to implement into a physical robot
- Able to create from scratch vs being confined to Amazons server and prefabricated training models
- More cost efficient –AWS Deep Racer base $600 plus $5/hr training
- Deep racer is able to training faster than our model

Future Recommendations

- Implementation of network model on a central server
- Data storage optimization of saved evaluated data

Questions