Ahmad Suleman AAI & DS

Research Interests

Deep Reinforcement Learning
Machine Learning
Locomotion
Robotics

Pakistan Zindaa Baa..d

Intelligent Robotics Pipeline

Simulation

- Operating System
- GPU
  CUDA Drivers
  Pytorch, TF
- ROS
- Gazebo
- Description
  - URDF.xacro
  - Plugins_Actuator.xacro
  - Plugins_Sensor.xacro
  - Physics Engine Interface
- Reinforcement learning
  - Interface Node Class
  - Wrapper For RL
  - MODEL, ALGO, Buffer
- Simulation
- Physics Engine

Deployment

- Jetson
  RaspberryPi
- ROS
- Description
  - URDF.xacro
  - Parameters →
    Dims, Friction, PID etc.
- Driver Package
  - Interface with I/O
    Sensors & Actuators
- AI Drivers
  Pytorch, TF, GPU ↔ AI Package
  - Trained Model
  - Sens. Actu. Interface

Autonomous Artificial Intelligence and Decision Support Lab

ROS

Introduction to Simulation

Definition: "simulation is the act of using a computer to generate a solution for a problem that otherwise could not be solved by traditional mathematics" [1]

Benefits:

- Test and prove theoretical methods initially or solely in a simulator as robots themselves are oftentimes expensive, fragile and scarce.
- An environment that is cheap and allows users access to a variety of desired robots
- Simulation can run faster than real time & facilitates parallel instances.
- Importance of simulation and further studies [5,6]

Simulator of Interest ➔ Gazebo ☺

Gazebo Simulator:

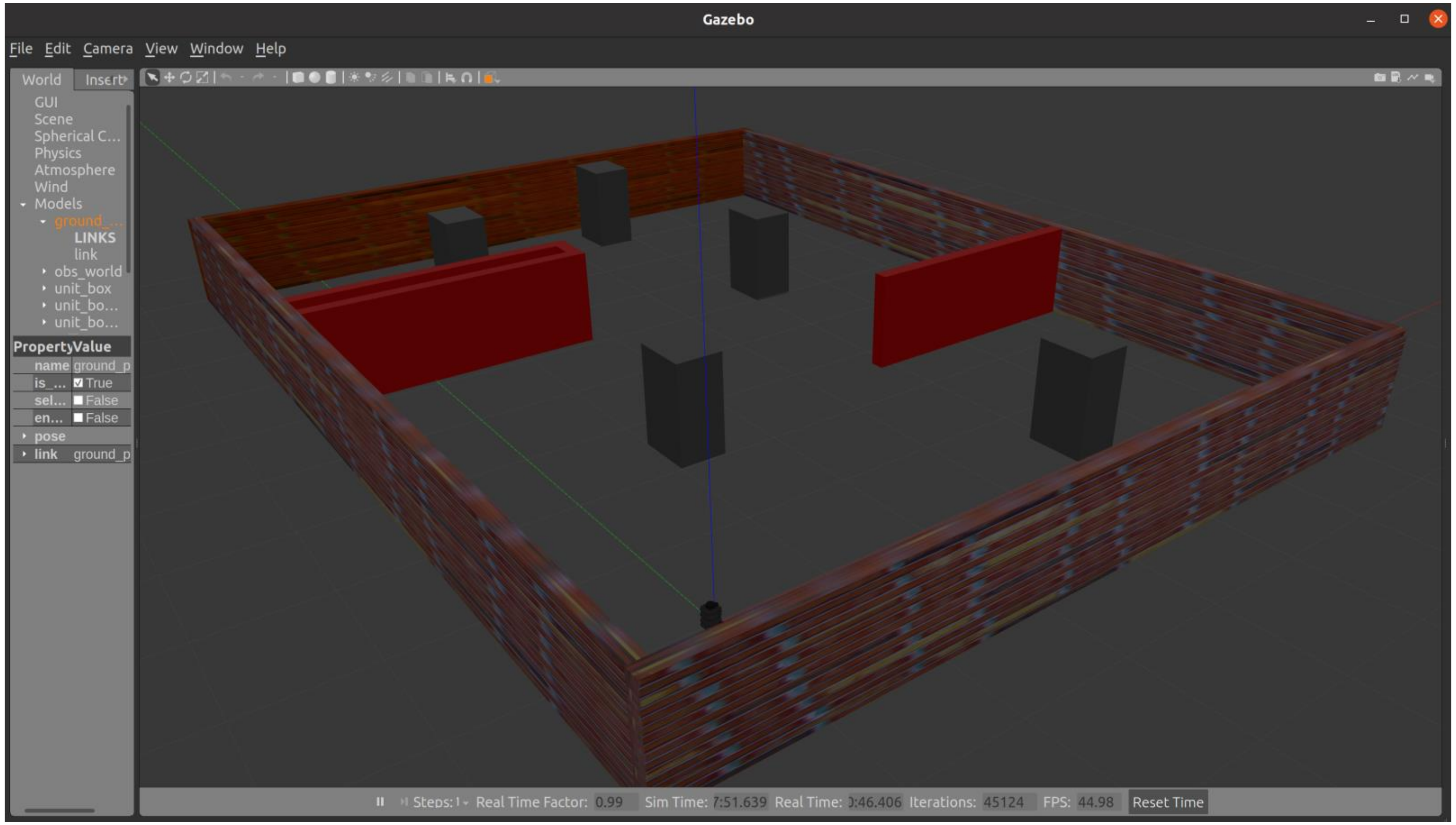
- Gazebo is a popular rigid body robotics simulator used in research, for both legged [15], [16] and wheeled [17] robots.

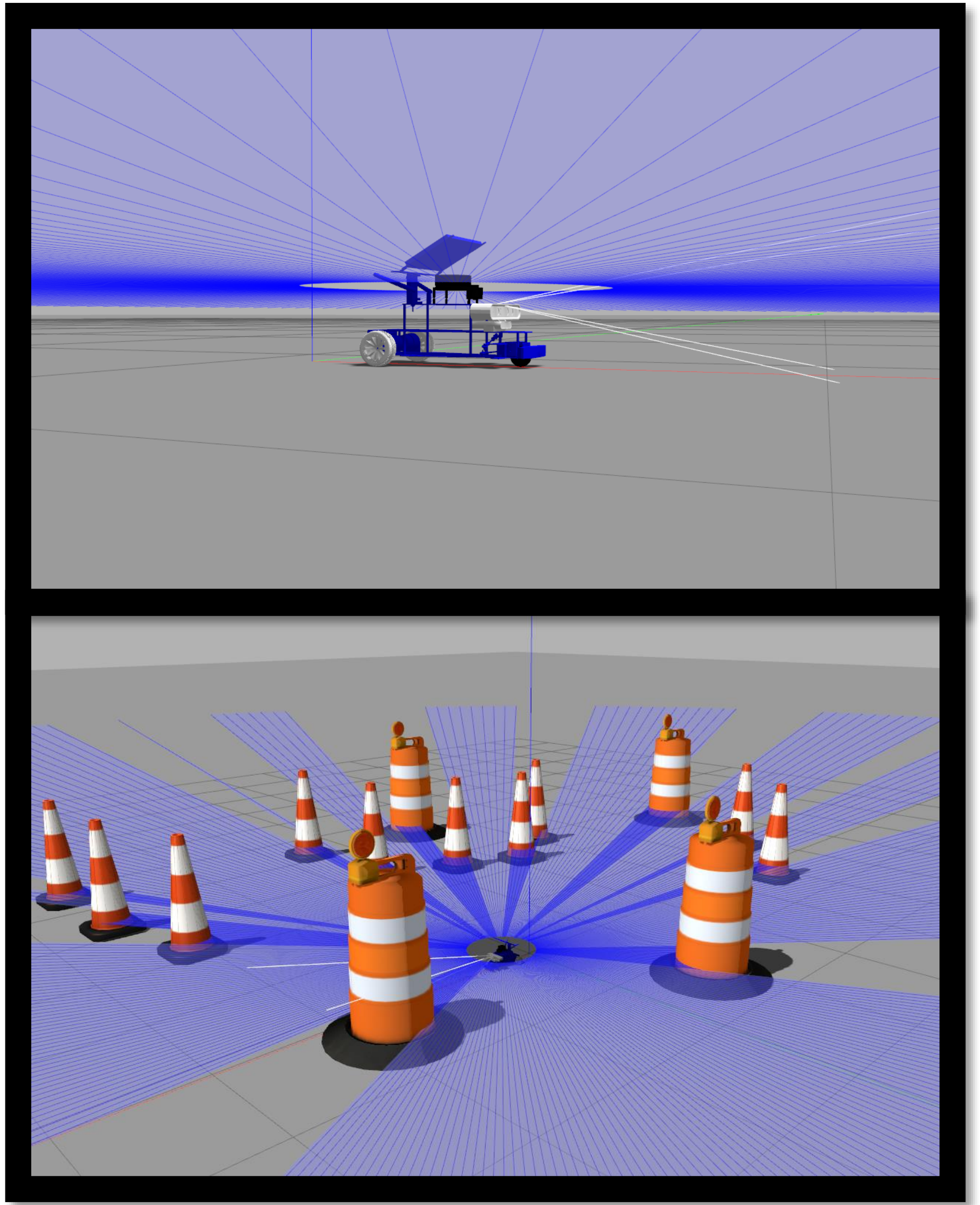
ROS-Gazebo Interface

- The ROS interface provided by Gazebo simplifies the process of testing in simulation and transferring it onto the physical system.
- Gazebo also offers model library for sensors such as camera, LiDAR, GPS, and IMU.
- Gazebo provides capability to import environments & robot models from well defined file formats. E.g. URDF, SDF, & OpenStreetMap
- The simulator runs quickly and can simulate multiple robots in real-time.

Gazebo for simulation

- Scenarios as per requirements can be designed.





Information Gathering

Motivation

To detect the surrounding environment and obstacles, autonomous vehicle perception sensing requires the collection of data from vehicle sensors.

- Radar, Camera, and LiDAR being the most prominent technologies in use.

Sensor of Interest

Lidar

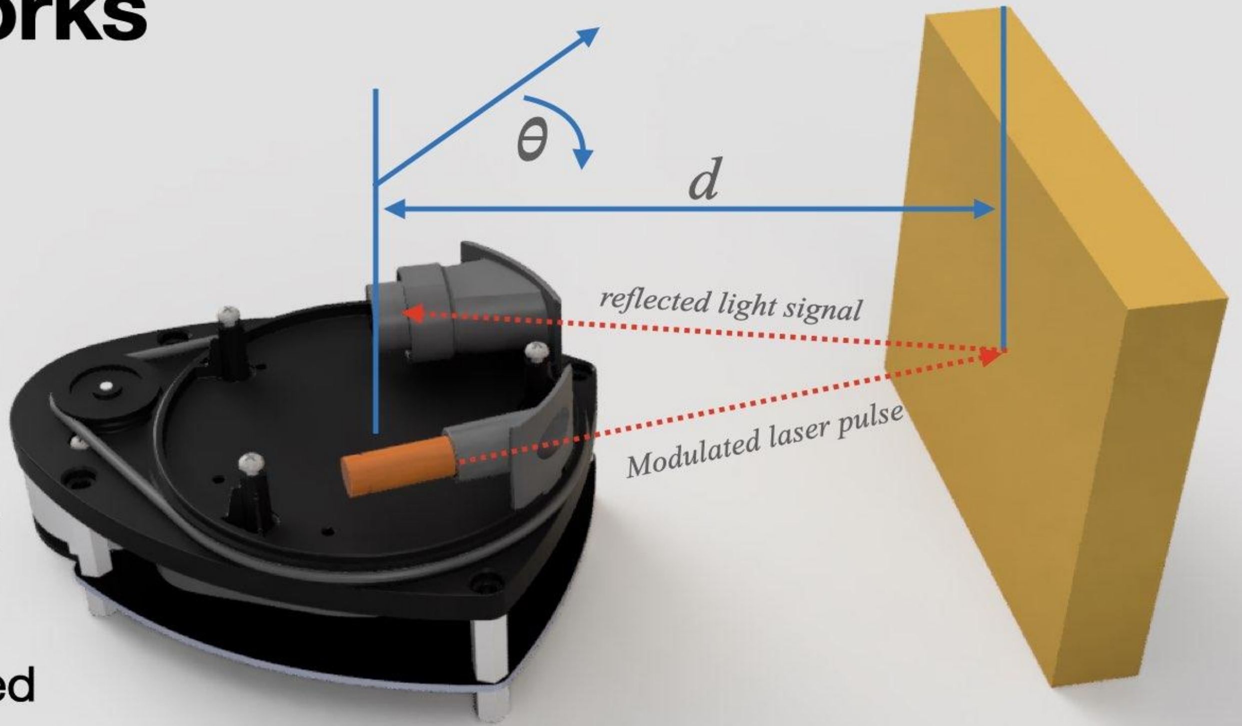
The lidar can measure distances by simply calculating the round-trip time of a laser pulse traveled to the target and back [2]

Lidar Principle

How LiDAR works

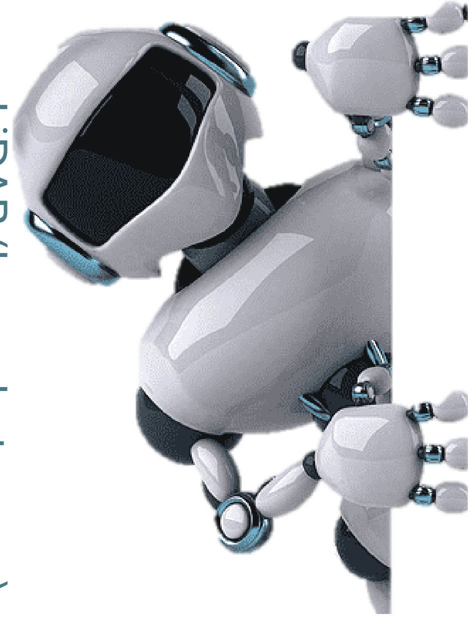
Time of Flight

- The laser sends a modulated pulse
- This is reflected off an object
- The reflected light signal hits the light sensor
- The distance is calculated by taking the speed of light and halving the time it takes from sending to receiving



- This happens so fast the rotation is mostly irrelevant

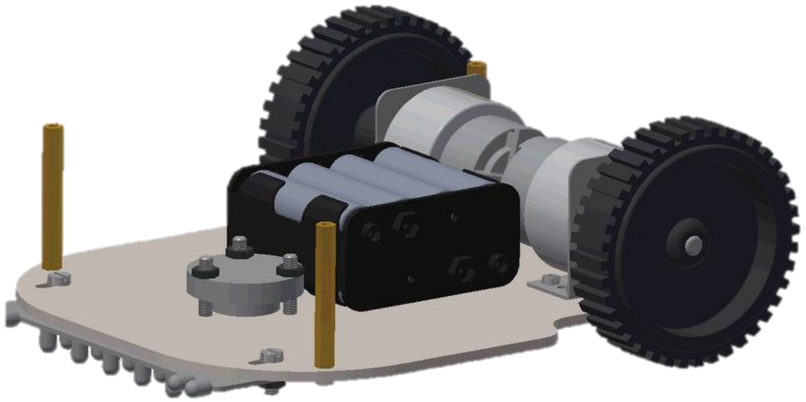
Image Copied from: LiDAR (kevsrobots.com)

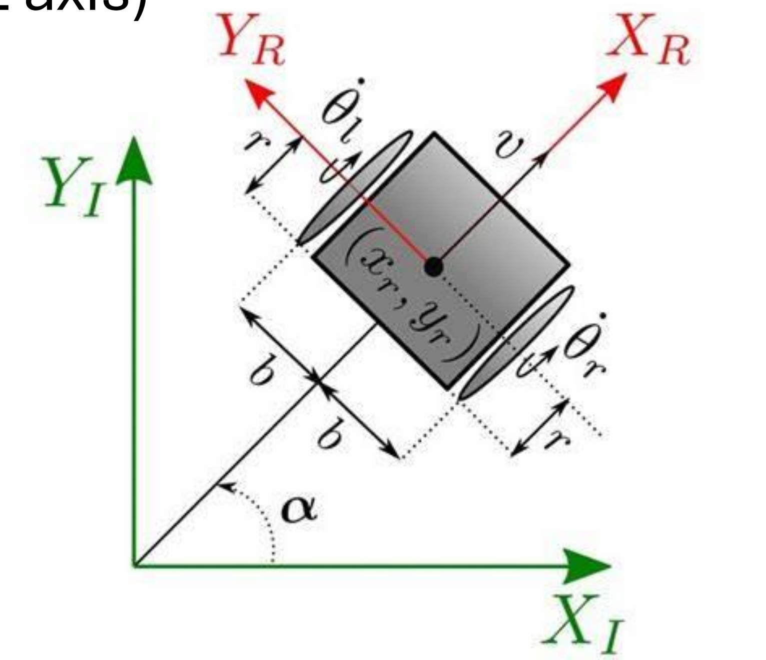


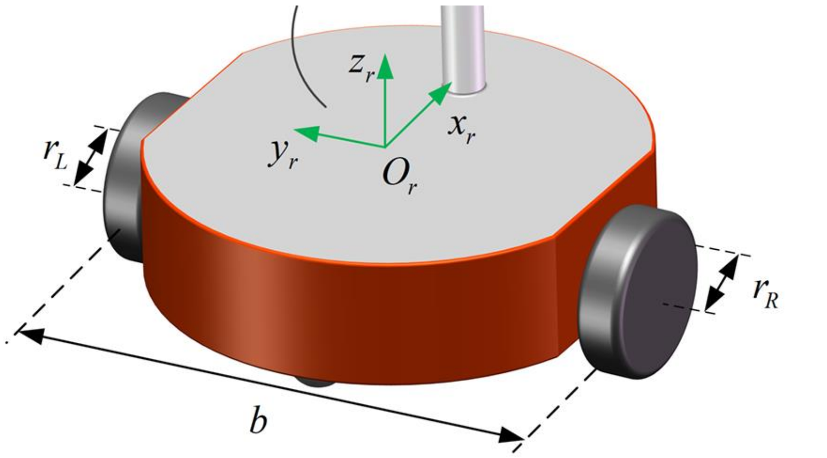
Actuation

Differential Drive Control

- It is a two wheel drive where each wheel can move Independently.
- A caster wheel is also present for balance
- **Controlled by:** (*Topic: cmd_vel*)
    - Translational Velocity (along X, Y coordinates)
    - Rotational Velocity (along Z axis)







Plugins

Plugins: Adding a new piece of code in the host program without altering the functionality of the host program itself.

ROS Plugins :

Gazebo supports several plugin types, and all of them can be connected to ROS, but only a few types can be referenced through a URDF file:

1. ModelPlugins, to provide access to the physics::Model API
2. SensorPlugins, to provide access to the sensors::Sensor API
3. VisualPlugins, to provide access to the rendering::Visual API

A plugin type should be chosen based on the desired functionality.

Sensor Plugin


```
<link  name="laser_link">⋯
</link>
<joint name="laser_joint" type="fixed">⋯
</joint>

<gazebo reference="laser_link">
    <material>Gazebo/Black</material>

    <sensor name="laser" type="ray">
        <pose> 0 0 0 0 0 0 </pose>
        <visualize>true</visualize>
        <update_rate>10</update_rate>
        <ray>
            <scan>⋯
            </scan>
            <range>⋯
            </range>
        </ray>
        <plugin name="laser_controller" filename="libgazebo_ros_ray_sensor.so">
            <ros>
                <argument>~/out:=scan</argument>
            </ros>
            <output_type>sensor_msgs/LaserScan</output_type>
            <frame_name>laser_link</frame_name>
        </plugin>
    </sensor>
</gazebo>
```

JOINT Connector

Visual and Param

Plugin file and topic interface

```
<ros2_control name="GazeboSystem" type="system">
    <hardware>
        <plugin>gazebo_ros2_control/GazeboSystem</plugin>
    </hardware>
    <joint name="back_right_joint">
            <command_interface name="velocity">
                <param name="min">-10</param>
                <param name="max">10</param>
            </command_interface>
            <state_interface name="velocity"/>
            <state_interface name="position"/>
    </joint>
    <joint name="back_left_joint">
            <command_interface name="velocity">
                <param name="min">-10</param>
                <param name="max">10</param>
            </command_interface>
            <state_interface name="velocity"/>
            <state_interface name="position"/>
    </joint>

</ros2_control>

<gazebo>
    <plugin name="gazebo_ros2_control" filename="libgazebo_ros2_control.so">
        <parameters>$(find articubot_one)/config/my_controllers.yaml</parameters>

    </plugin>
</gazebo>
```

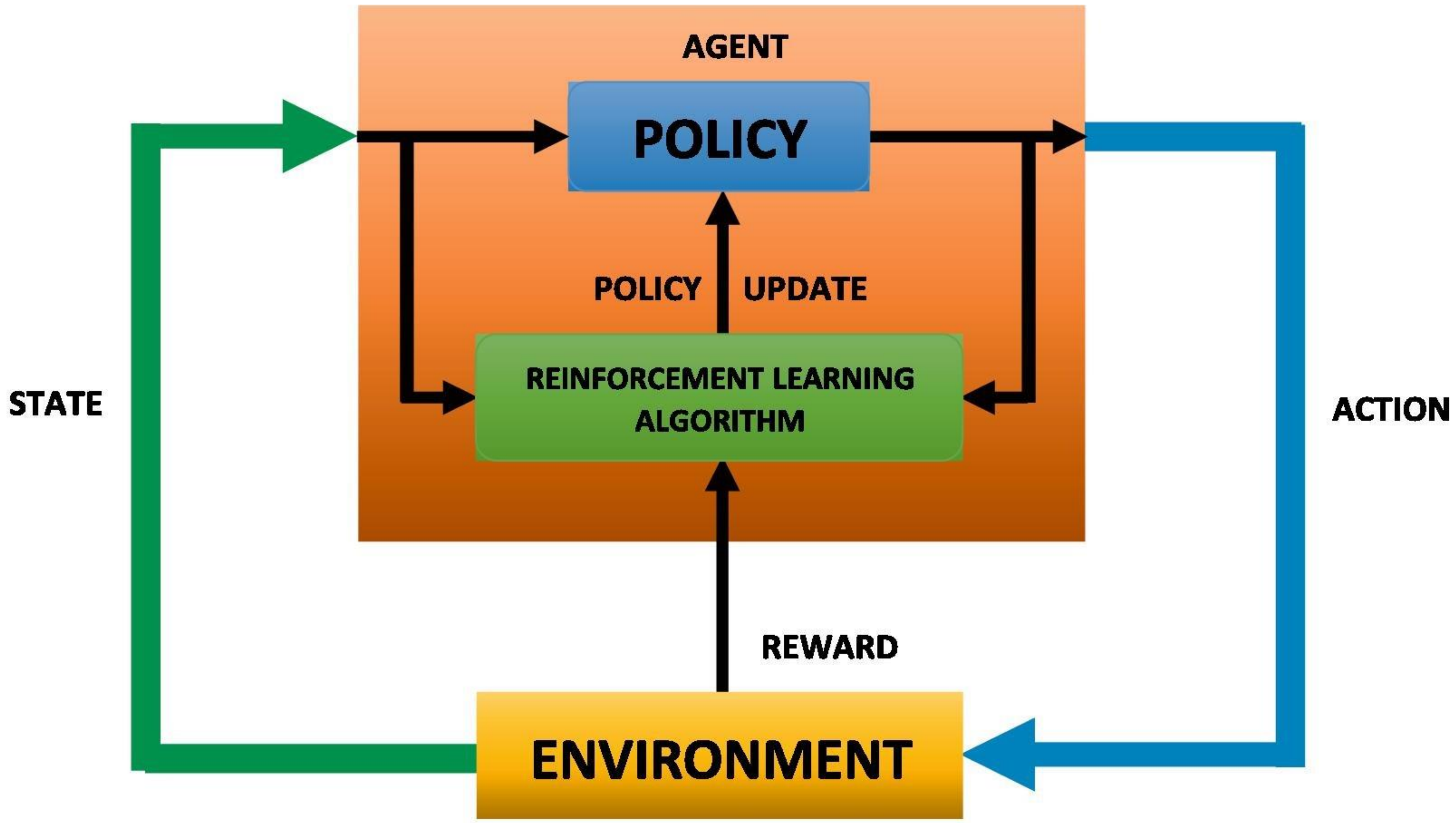
Intelligent Robotics

```yaml
controller_manager:
  ros__parameters:
    update_rate: 30
    use_sim_time: true

    diff_cont:
      type: diff_drive_controller/DiffDriveController

    joint_broad:
      type: joint_state_broadcaster/JointStateBroadcaster

diff_cont:
  ros__parameters:

    publish_rate: 50.0

    base_frame_id: base_link

    left_wheel_names: ['back_left_joint']
    right_wheel_names: ['back_right_joint']
    wheel_separation: 0.1685
    wheel_radius: 0.065

    use_stamped_vel: false
```

Robotics solutions with RL

- RL in robotics is a "*sequential decision problems under uncertainty*".
- So! We need Markov decision process (MDP) which satisfies Markov property.
- **Markov Property:** the effects of an action taken in a state depend only on that state and not on the prior history
- An MDP can be described as a controlled Markov chain, where the control is given at each step by the ***chosen action***. The process then visits a ***sequence of states*** and can be evaluated through the ***observed rewards***.
- Solving an MDP consists of controlling the agent in order to reach an optimal behavior, i.e. expected utility maximization.

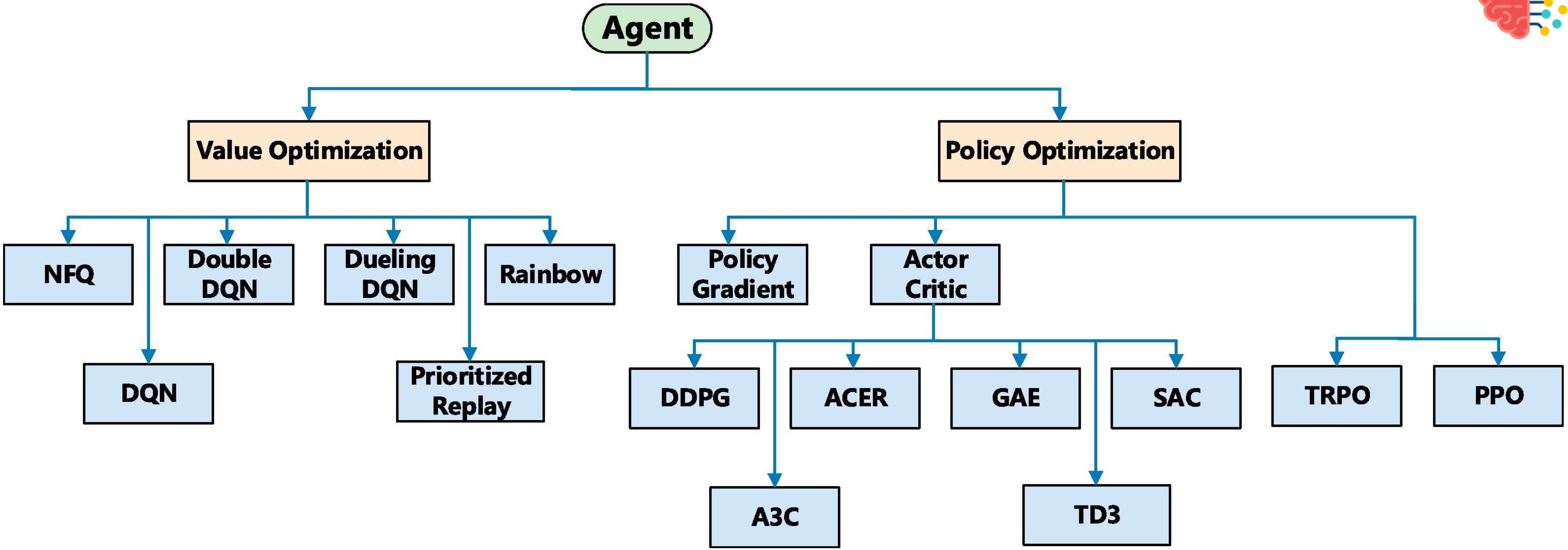
Reinforcement Learning (RL)



Reinforcement Learning (RL)

- **1911** – Trial and Error Learning
- **1950s** – Optimal Control, Dynamic Programming, Bellman Equation, MDP
- **1960s** – Introduction of RL, Temporal Difference Learning
- **1980s** – Actor Critic architecture, Q-Learning
- **1992** – TD Gammon
- **2000** – Policy Gradient Method with Function Approximation
- **2015** – DQN, AlphaGo

Reinforcement Learning (RL)



Intelligent Robotics

DeepMind Control Suite (Tassa et al., 2018)

- Provides a learning platform for a set of continuous control tasks
- Created with the help of Python and MuJoCo physics engine
- Freely available at GitHub.

Google Dopamine (Castro et al., 2018)

- TensorFlow based platform that offers reproducibility, flexibility, and reliability to research ideas
- Freely available at GitHub.

Open AI Gym (Brockman et al., 2016)

- One of the most popular and effective learning platforms for developing and comparing RL algorithms
- Created in Python and provides an interface with other important Python libraries such as Keras, TensorFlow, Theano, and Scikit-learn

DeepMind Lab (Beattie et al., 2016)

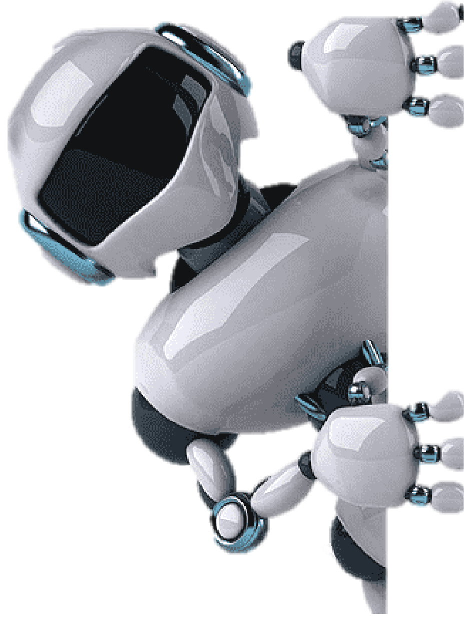
- For training AI agents with effective cognitive skills
- Offers rich 3D game-like simulated environments for efficient learning
- Provides learning for extendable and customizable 3D navigation tasks

Arcade Learning Environment (ALE) (Bellemare et al., 2013)

- One of the most popular RL evaluation platforms
- Allows RL agents to interface with a number of Atari 2600 games.

D4RL (Fu et al., 2021)

- Open source benchmark for offline RL
- Primarily offers learning environments for the fields of robotics, autonomous vehicles, and traffic management.



Autonomous Artificial Intelligence and Decision Support Lab

ROS

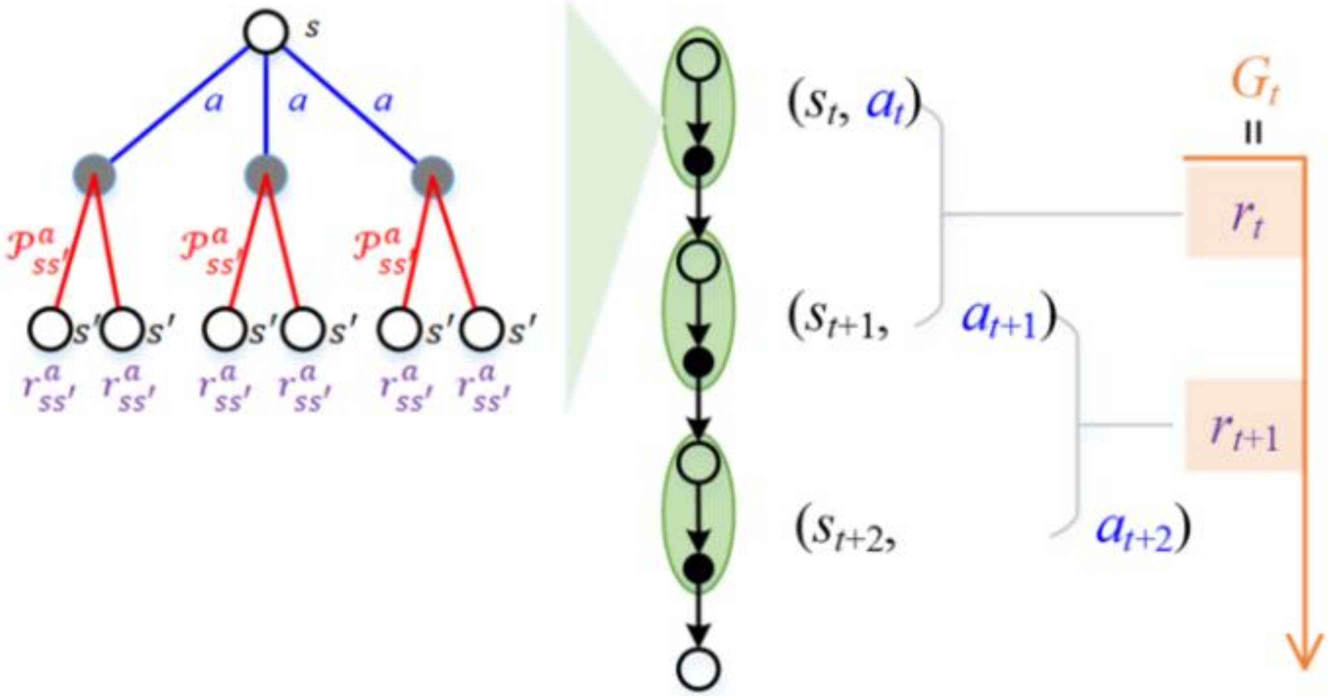
Intelligent Robotics

Autonomous Artificial Intelligence and Decision Support Lab

MDP Environment for Robotics

A Markov Decision Process (MDP) model that contains

1. A set of possible world states S → State Space
2. A set of possible actions A → Action Space
3. A real valued reward function R(s,a) → Reward Function
4. Transition Probability for Model based scenario.



ROS

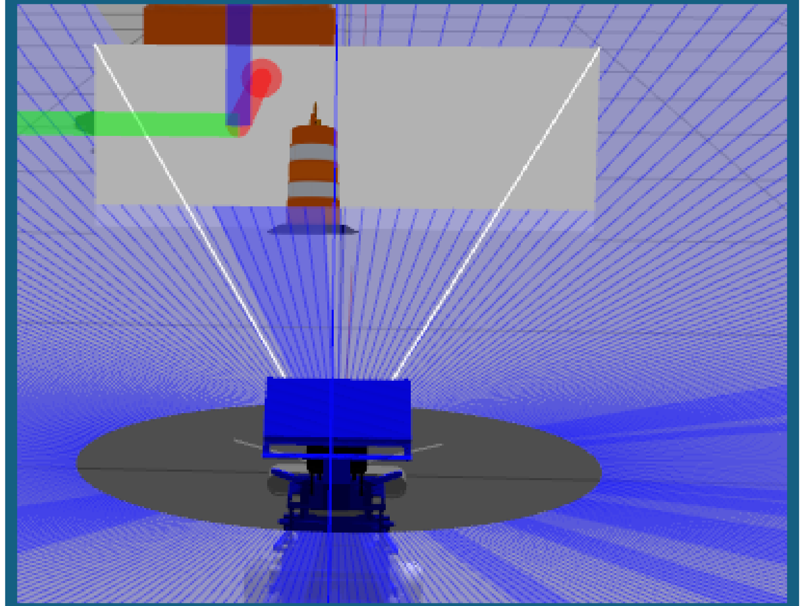
MDP Coding for Robotics


```python
class RoboticsEnv(gym.Env):
    """Custom Environment that follows gym interface."""

    metadata = {"render_modes": ["human"], "render_fps": 30}

    def __init__(self):
        super().__init__()
        self.gym_node = fusion_env()
        # Define action and observation space
        # They must be gym.spaces objects
        # Example when using discrete actions:
        self.action_space = spaces.Discrete(N_DISCRETE_ACTIONS)
        # Example for using image as input (channel-first; channel-last also works):
        self.observation_space = spaces.Box(low=-30, high=30,
                                            shape=(STATE_SHAPE,), dtype=np.float32)

        self.distance_kpi = []
```

```python
def step(self, action):
        #self.gym_node.unpauseSim()
        self.gym_node.take_action(action)
        observation = self.gym_node .get_state()
        reward, terminated = self.gym_node .compute_reward()
        truncated = False
        #self.distance_kpi.append(self.distance_covered)
        info = {}
        #self.gym_node.pauseSim()
        return observation, reward, terminated, truncated, info

def reset(self, seed=None, options=None):
    self.distance_kpi = []
    observation = self.gym_node.reset()
    info = {}
    return observation, info

def render(self):
    pass

def close(self):
    pass
```

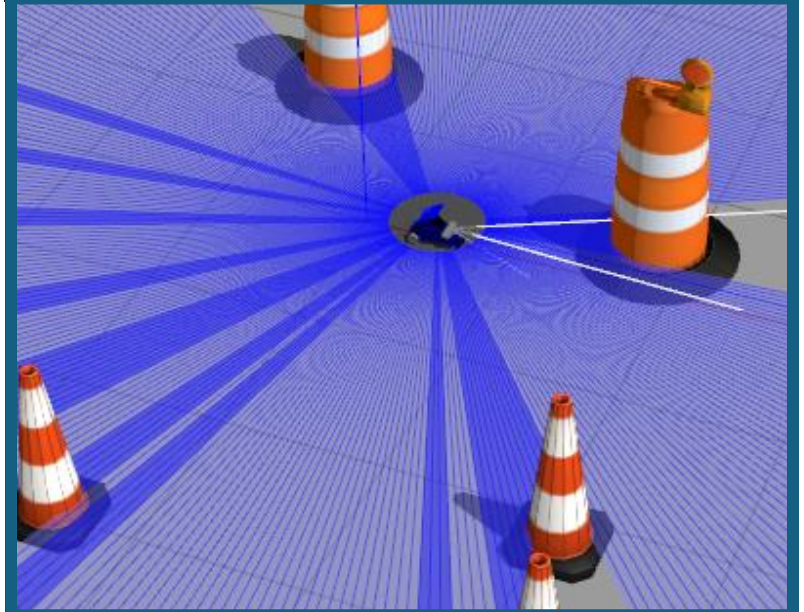
MDP Environment for Robotics

State Space



Camera, Lidar, & Odom data

Action Space



Frwd, Bckwrd, Left & Right
Translat. Vel & Rot. Vel

Reward Function

Issue +ve for desired behavior for example reaching the goal, destination, or maintaining safe distance from obstacles.

Issue –ve for bad behavior. e.g. obstacle collision, time wasting, going away from the goal.

Think of a feedback for desired behavior.

State Space Development

- Lidar range data gathering i.e. laser_scan message subscription.
- It is a list type data with each value at an angle given by the required resolution
- Is this information enough ???
- Lidar data feature extraction
  - Max. distance & angle from the obstacles in range.
  - Min. distance & angle form obstacles in range.
  - Maximum vacant area & minimum vacant area.
- Current position from odometer data. (Subscribe /odom topic)
- Distance from the destination. (if possible)
- Direct camera feed can also be used as state space or features can be extracted.

State Space Coding


```python
def get_state(self):
    while not rospy.is_shutdown():
        self.laser_range_data = None
        while self.laser_range_data is None:
            try:
                self.laser_range_data = rospy.wait_for_message("/fusion_bot/scan", LaserScan, timeout=5)
            except:
                #pass
                rospy.logerr("Laser Scan Message Not Received")
                break
        self.pose = None
        while self.pose is None:
            try:
                self.pose = rospy.wait_for_message("/odom", Odometry, timeout=5)
            except:
                #pass
                rospy.logerr("Odom Message Not Received")
                break
        break
```

Action Space

- The control variables in the case of robotics it can be driving actions.
- It is the main control that will be learned by the AI model / RL agent.
- In coding it will be a simple publisher.

```
vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)
```

- Publish on cmd_vel topic (or the topic name defined in the controller plugin file)
- Action Discretization (Based on the control requirements)
-

```
def take_action(self, action):
    self.step_count += 1
    #self.unpauseSim()
    msg = Twist()
    if action == 0: #Move Forward
        msg.linear.x = 0.5
        msg.linear.y = 0
        msg.angular.z = 0

    if action == 1: #Move Backward
        msg.linear.x = -0.5
        msg.linear.y = 0
        msg.angular.z = 0

    if action == 2: #Turn Right
        msg.linear.x = 0.5
        msg.linear.y = 0
        msg.angular.z = 0.5
    if action == 3: #Turn Left
        msg.linear.x = 0.5
        msg.linear.y = 0
        msg.angular.z = -0.5
    if action == 4: # stop
        msg.linear.x = 0.0
        msg.linear.y = 0
        msg.angular.z = 0.0

    self.action_last = action
    self.publish_action.publish(msg)
    if not rospy.is_shutdown():
        #pass
        self.rate.sleep()
```

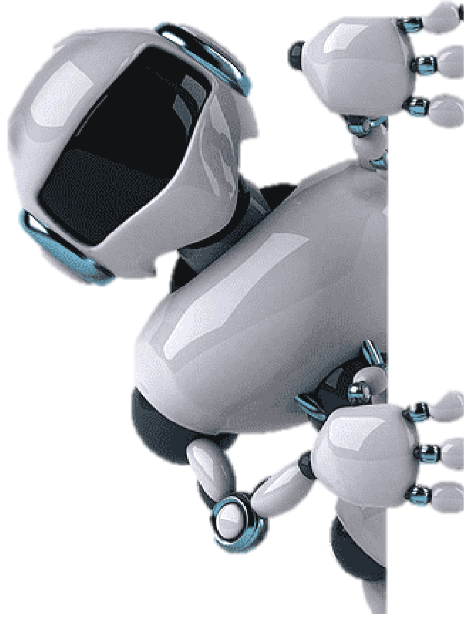
Actual Action

```
# twist
state.twist.linear.x = 0
state.twist.linear.y = 0
state.twist.linear.z = 0
state.twist.angular.x = 0
state.twist.angular.y = 0
state.twist.angular.z = 0
```

Actions Considered

Discretization

Action Space Coding



Reward Function


```python
def compute_reward(self):
    #self.dis_from_target
    self.done = False
    reward = 0

    dist_reward = 1.0 - self.dis_from_target*0.1
    scan_data = self.state_space[0:-5]
    scan_reward = 0
    for d in scan_data:
        if d <= 0.75:
            scan_reward -= 0.05
    reward = dist_reward + scan_reward
    if self.dis_from_target <= 1.0:
        self.done = True
        reward = 50
    if self.collision_occured:
        self.done = True
        reward = -50
    if self.step_count >= 1000:
        self.done = True
        reward = -self.dis_from_target*2
    #print(self.distance_covered, self.dis_from_target, reward)

    return reward, self.done
```

RL Algorithm DQN


```
states, actions, rewards, next_states, dones = self.exp_buffer.sample(batch_size=self.batch_size)
states = torch.tensor(states, device=self.device, dtype=torch.float)
rewards = torch.tensor(rewards, device=self.device, dtype=torch.float)
next_states = torch.tensor(next_states, device=self.device, dtype=torch.float)
self.target_net.eval()
if not np.squeeze(dones):
    with torch.no_grad():
        q_next_pred = self.target_net(next_states).max(1).values
        #print(q_next_pred.shape)
else:
    q_next_pred = rewards
expected_q_value = rewards + q_next_pred*self.GAMMA

target_q_value = np.asarray([0.0, 0.0, 0.0, 0.0], dtype=np.float32)
expected_q_value = expected_q_value.to("cpu").detach().numpy()
target_q_value[actions[0]] = float(expected_q_value[0])
#print(pred_q_value, target_q_value, expected_q_value, actions)
# Compute Huber loss
target_q_value = torch.tensor(target_q_value, device=self.device, dtype=torch.float).unsqueeze(0)
```

Loss Backpropagation

```python
self.main_net.train(True)
self.optimizer.zero_grad()
pred_q_value = self.main_net(states)
loss = self.loss_criterion(pred_q_value, target_q_value)
loss.backward()
# In-place gradient clipping
#torch.nn.utils.clip_grad_value_(self.main_net.parameters(), 100)
self.optimizer.step()
self.train_count += 1
```

Hyperparameters


```
if episode%50==0:
        self.target_net.load_state_dict(self.main_net.state_dict())
        print("Target Network Updated")
```

```
(self, train_episodes=1000,gamma=0.98,use_pre_trained=False,Testing=False, learning_rate = 1e-4,
 epsilon_start=0.5, batch_size = 1, exploration_decay = 0.0008, trained_weights=None):
```

```
self.step_count = 0
self.batch_size = batch_size
self.decay_in_exploration = exploration_decay
self.steps_per_episode = 5001
self.GAMMA = gamma
```

Intelligent Robotics