MODULE 5:

TECHNICAL FUNDAMENTALS – IN-DEPTH TECHNOLOGIES AND BASICS OF SIMULATION & ANALYSIS FOR AUTONOMOUS VEHICLES

Nicholas Ho

Institute of System Science, NUS

Contents

1. Self-driving cars' perception and decision-making
2. Basics of Simulation & Analysis
3. Workshop: Introduction to software pertaining to topics on Simulation & Analysis for Autonomous Vehicles

<u>CHAPTER 1:</u>
SELF-DRIVING CARS' PERCEPTION AND DECISION-MAKING

RECAP: How AV Work?

PERCEPTION | **LOCALIZATION** | **PATH PLANNING** | **CONTROL**

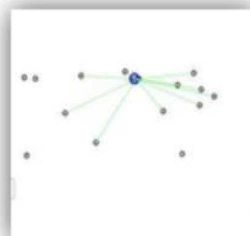
COMPUTER VISION

Detection of roads, signs, lights, objects, …

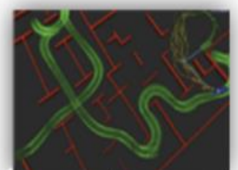
SENSOR FUSION

Detection of vehicles, obstacles, objects…

Understanding the world



Localization & Mapping : « **Where are we ?** »

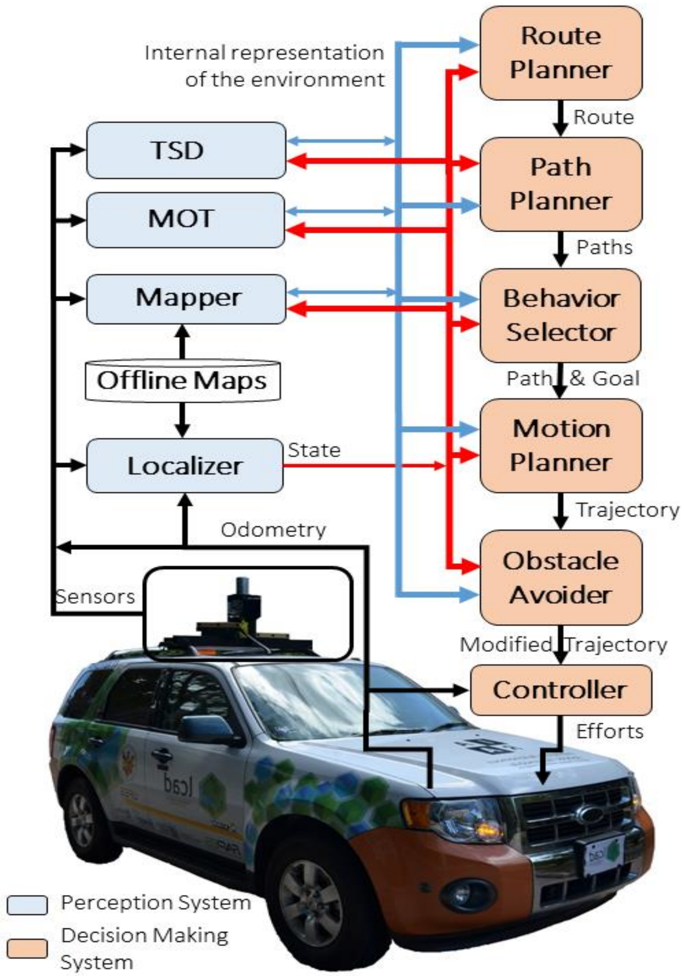


Calculating trajectories from our position and our understanding of the world

Following the calculated trajectory

1. AV PERCEPTION

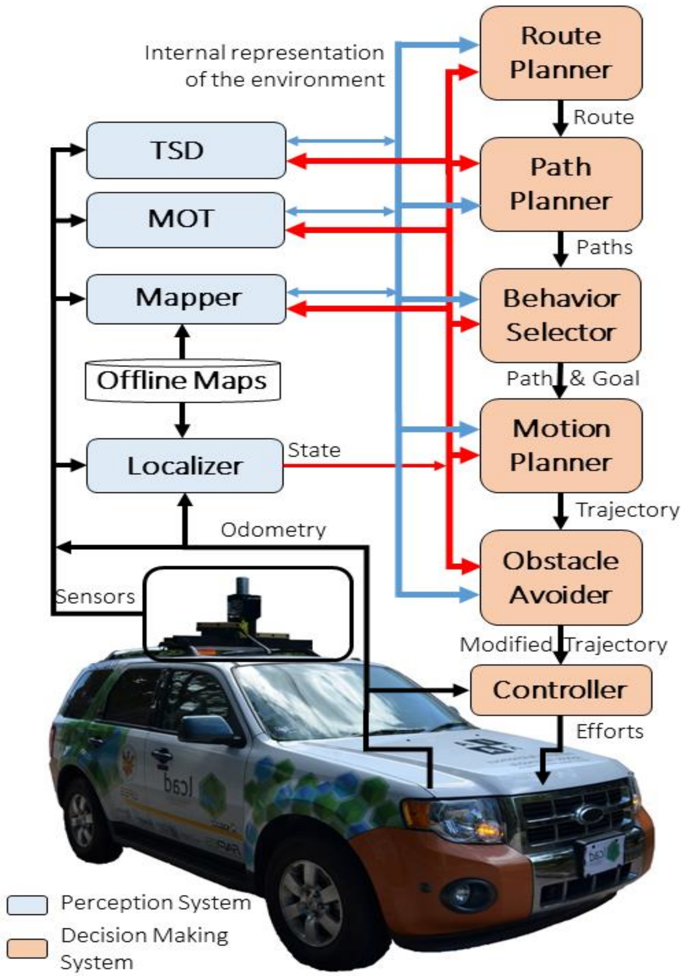
Typical Architecture of Self-Driving Cars



The Perception Subsystems:

- **The Localizer subsystem** is responsible for **estimating the car's State (pose, linear velocities, angular velocities, etc.) in relation to static maps of the environment**
- **Receives inputs from the Offline Maps, Sensors' data and Odometry**
- **Outputs the State of the car**
  - **Offline Maps** (i.e. static maps) are **computed automatically before the autonomous operation**, typically using the sensors of the self-driving car itself; manual annotations or editions may be required

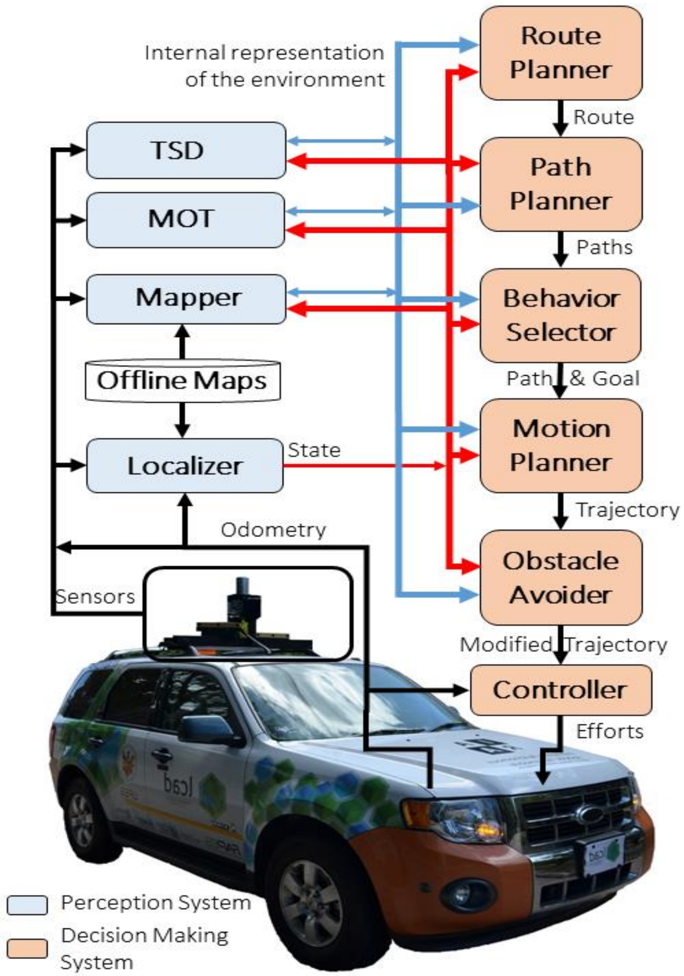
Typical Architecture of Self-Driving Cars



The Perception Subsystems:

- **The Mapper subsystem** receives the **inputs from the Offline Maps database and the self-driving car's State,** and **generates the online map (i.e. real-time maps) as output**
- **The Moving Objects Tracker subsystem (i.e. MOT)** receives the **inputs from the Offline Maps and the self-driving car's State,** and **detects and tracks** (i.e. calculates the pose and velocity of) **the nearest moving obstacles** (e.g. other vehicles, lamp posts, pedestrians)

Typical Architecture of Self-Driving Cars



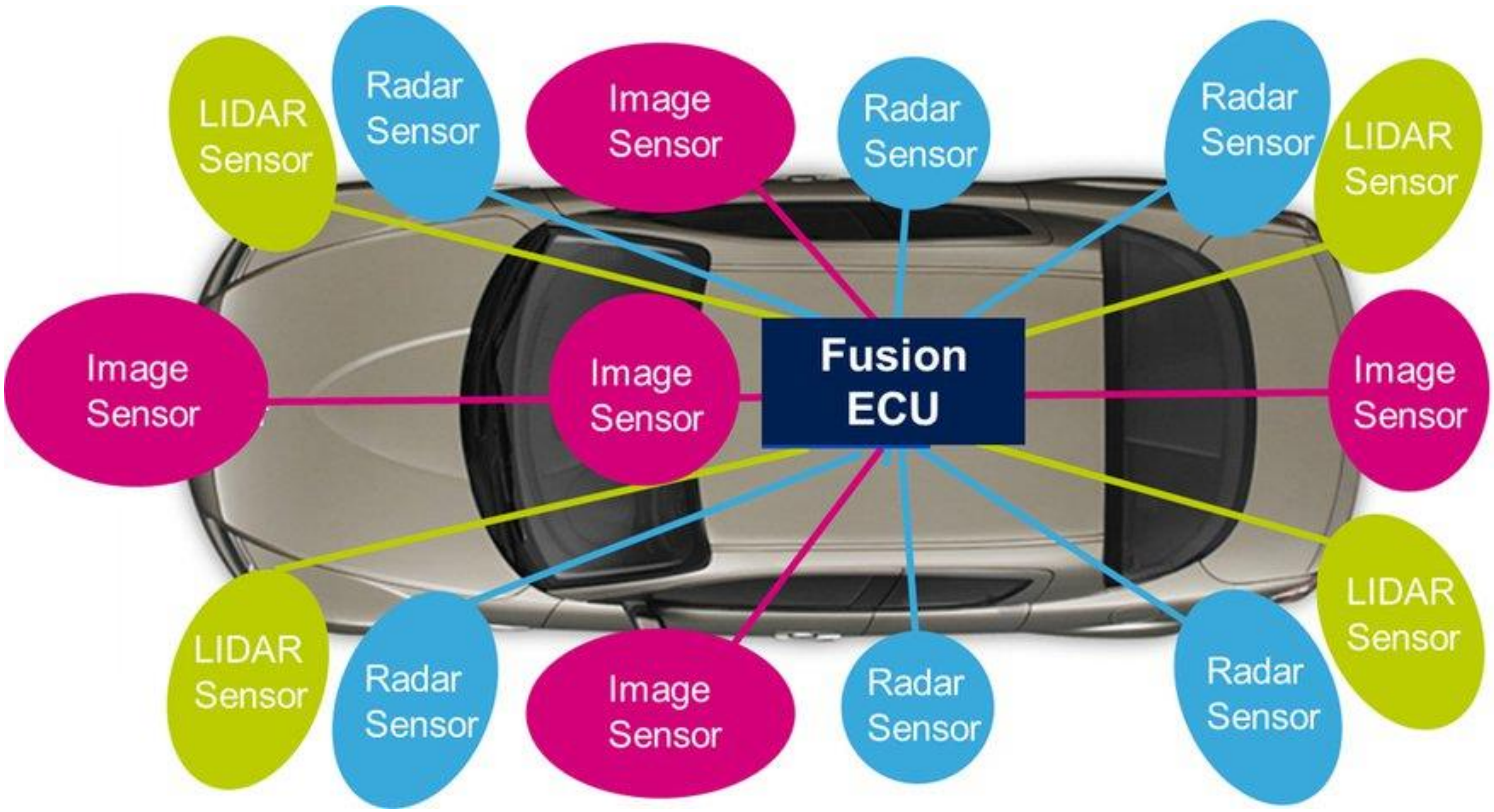
The Perception Subsystems:

- **The Traffic Signalization Detector subsystem (i.e. TSD)** is responsible for the **detection and recognition of traffic signalization**;
  - ❖ **Receives the Sensors' data and the car's State**, and **detects the position of traffic signalizations and recognizes their class or status**

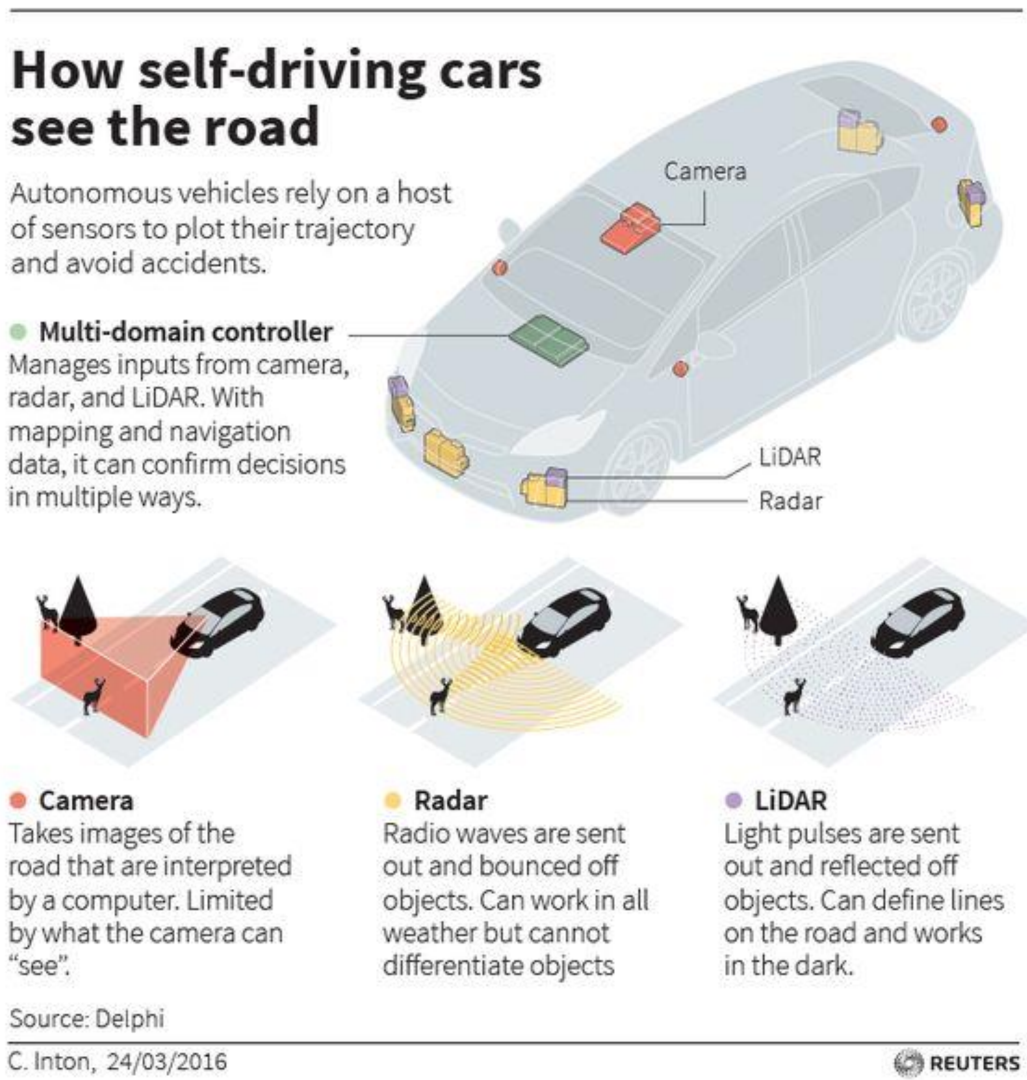
Fusion of Various Sensory Systems

Source: https://www.youtube.com/watch?v=gEy91PGGLR0

Fusion of Various Sensory Systems



Fusion of Various Sensory Systems



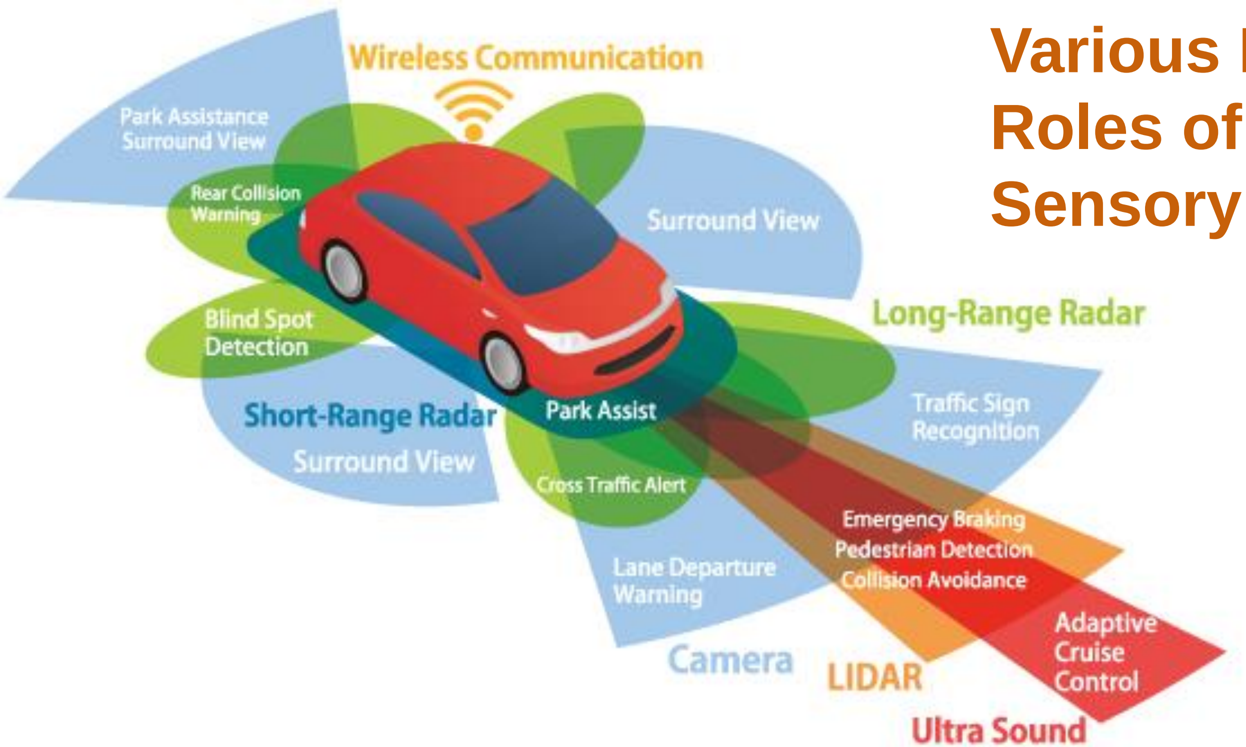
Fusion of Various Sensory Systems

Advantages and disadvantages of each vision sensor:

	CAMERA	LIDAR	RADAR
SPATIAL RESOLUTION	★★★	★★☆	★☆☆
NOISE	★★★	★☆☆	★☆☆
VELOCITY	★☆☆	★☆☆	★★★
ALL WEATHER	★☆☆	★☆☆	★★★
SIZE	★★★	★☆☆	★★★

The aim of sensor fusion is to use the advantages of each to precisely understand its environment

Fusion of Various Sensory Systems



Various Possible Roles of each Sensory System

- The **Camera** is a very good tool for **detecting roads, reading signs or recognizing a vehicle**
- The **Lidar** is better at accurately **estimating the position of this vehicle**
- The **Radar** is better at accurately **estimating the speed**

Fusion of Various Sensory Systems

Features	LIDAR	RADAR	Ultrasonic	Passive Visual (Camera Technology)
Primary technology	Light detection and ranging. It is a surveying technology that measures distance by illuminating a target with laser.	Radio detection and ranging. It is an object-detection system that uses radio waves to determine the range, angle, or velocity of objects.	It is an object detection system which emits ultrasonic sound waves and analyzes their return to determine distance.	It uses sophisticated object detection algorithms like DNN[1] to understand images visible from cameras.
Proximity detection	Poor	Good	Very good	Poor
Range	~100 m	~0.15-250 m	~5 m	250 m
Resolution	Good	Average	Poor	Very Good
Works in dark	Very good	Very good	Very good	Very poor (Almost non-existent)
Works in very bright light	Very Good	Very Good	Very Good	Good
Works in snow/ fog/ rain	Average	Very Good	Very Good	Poor
Provides colour/ contrast	Very Poor	Very Poor	Very Poor	Very good
Detects speed	Good	Very Good	Very Poor	Poor
Sensor cost	Costs around $70,000 (eg: Velodyne HD64)	It is cost effective at an average price of ~ $50-$200	Costs less than $50	~$100-200
Sensor size	Very bulky. It weighs an approx. 15 kg for the most popular Velodyne HD64.	Llight & compact at ~200 grams	Very Good	Very Good
Top 5 players	Robert Bosch, General Motors, Daimler, Ford Global Technologies & Bayerische Motoren Werke	Toyota, Robert Bosch, Denso, Nissan & Fujitsu TEN	Denso, Honda, Nissan, Mitsubishi Electric & Robert Bosch	Denso, Panasonic Electric Works, Robert Bosh, Valeo Systemes Electriques & Hyundai Mobis

Fusion of Various Sensory Systems

Next Generations of Sensor Fusion

Highly automated and fully autonomous cars to have up to six radars and nine camera modules.

AD Market Outlook: Transformation in Industry Approach in Hardware and Software, Global, 2012–2025 and beyond

Level 1 (2012) — Lateral or Longitudinal Assistance

Sensor – L1	#
Ultrasound	4
RADAR (LRR)	1
SRR	2
Camera/ Short- range LiDAR	1
Total	**~6-8**

Level 2 (2016) — Observation of Environment

Sensor -L2	#
Ultrasound	8
RADAR (LRR)	1
RADAR (SRR)	2-4
Camera	2-4
Total	**~17**

Level 3 (2018) — Awareness for Take Over

Sensor – L3	#
Ultrasound	8
RADAR (LRR)	2
RADAR (SRR)	4
Camera (LR)	2
Camera (Sur)	4
Camera (Stereo)	1
µbolo	1
LIDAR	1
Dead reckoning	1
Total	**~24-26**

Level 4 (2020) — No Driver Interaction

Sensor – L4	#
Ultrasound	8
RADAR (LRR)	2
RADAR (SRR)	4
Camera (Stereo/Trifocal)	2/3
Camera (Sur)	4
Camera (Stereo)	1
µbolo	1
LIDAR	2/4
Dead reckoning	1
Total	**~25-28**

Level 5 (>2025) — No Driver

Sensor – L5	#
Ultrasound	8-10
RADAR (LRR)	2
RADAR (SRR)	4
Camera (LR)	2/3
Camera (Sur)	4
Camera (Stereo)	2
µbolo	1/2
LIDAR	4
Dead reckoning	1
Total	**~28-32**

#: Average sensor count in respective levels; µbolo: Thermal camera/IR Sensor for Night Vision Approximation based on number of features supported

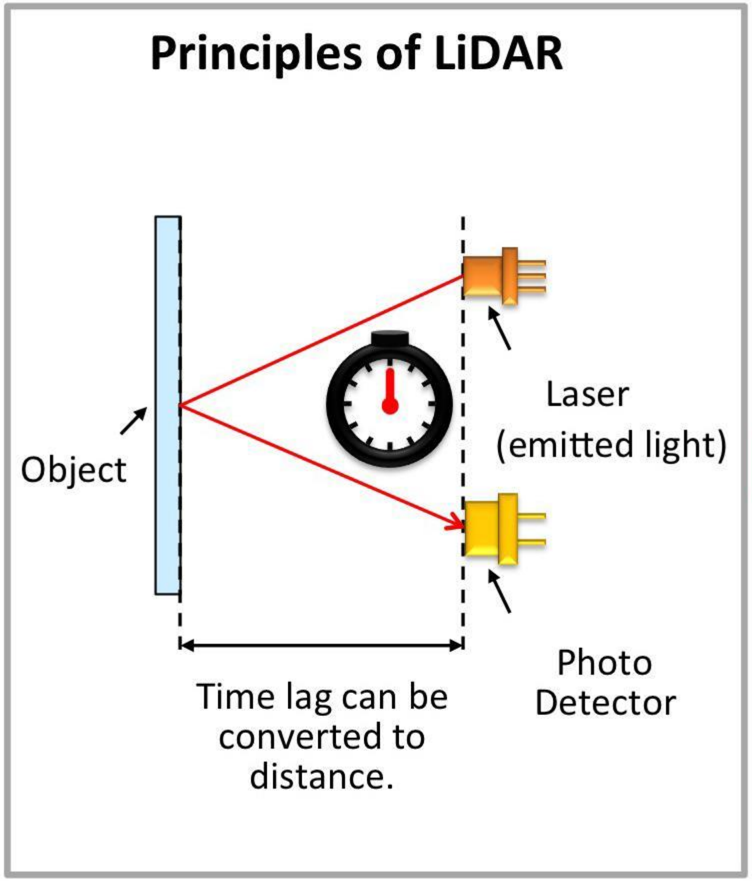
Source: Frost & Sullivan

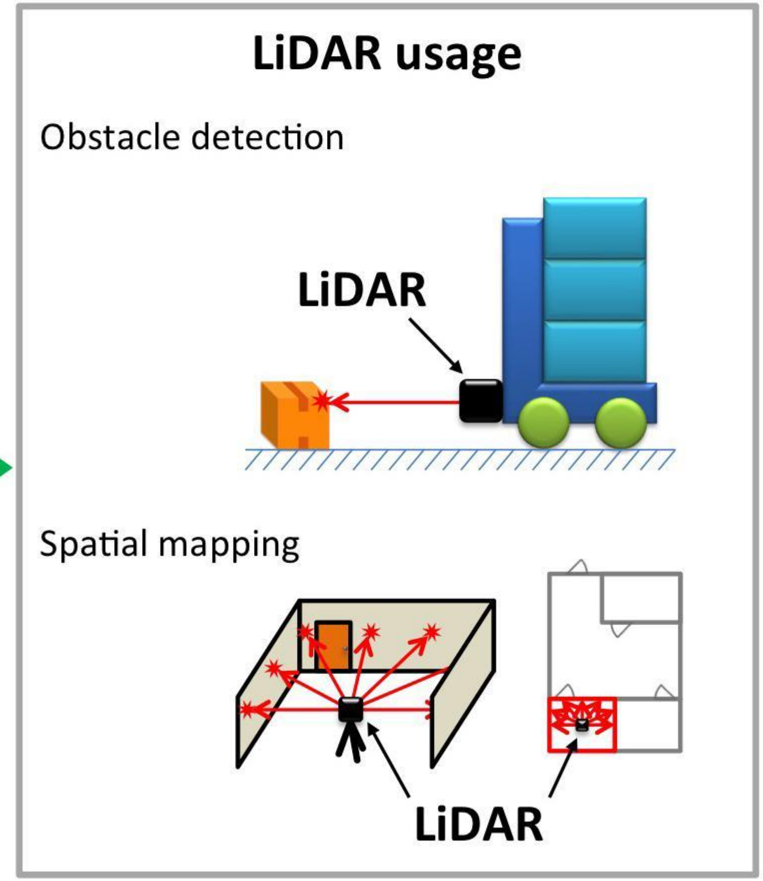
Basics of LiDAR

Source: https://www.youtube.com/watch?v=JC94Y063x58

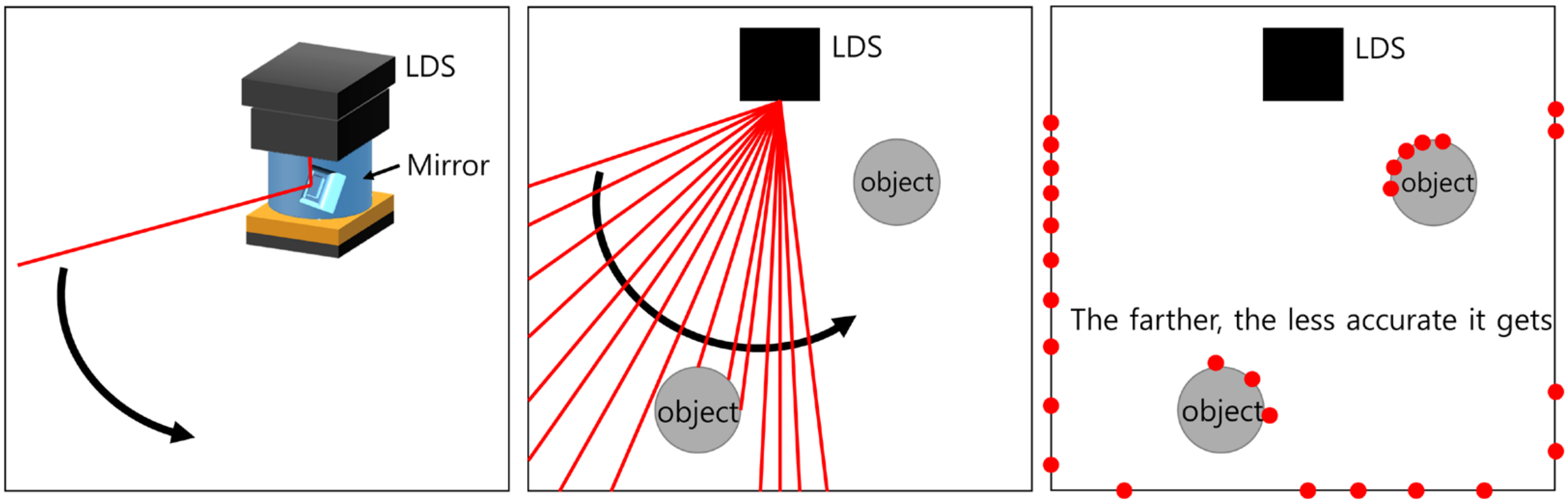
Basics of LiDAR

Laser-based distance measurement sensor

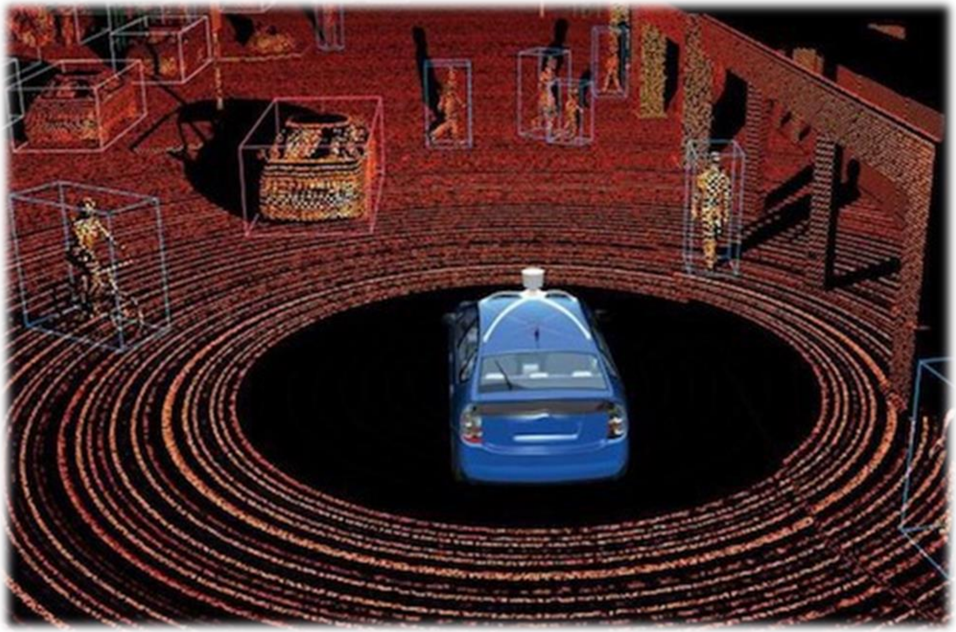




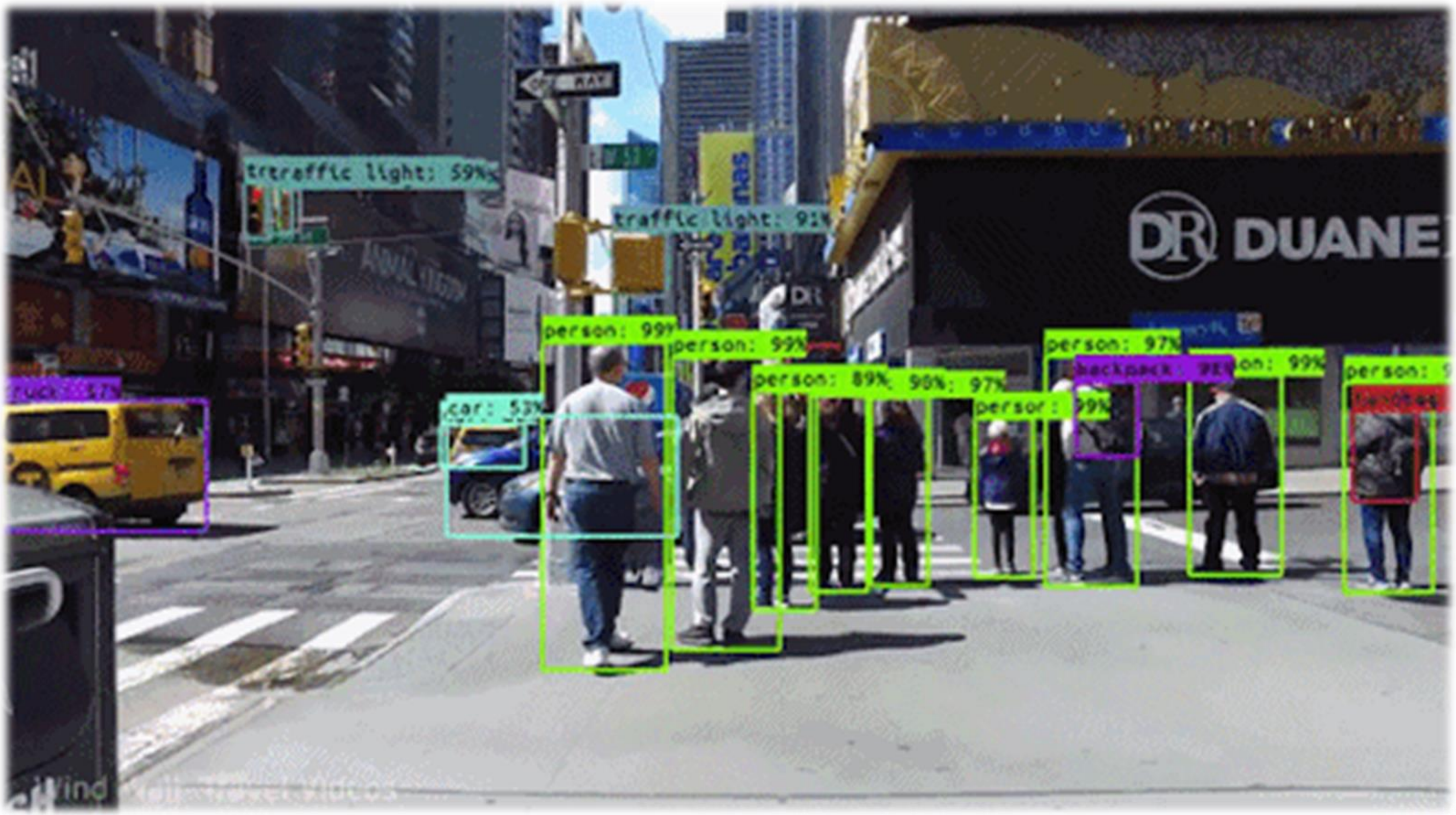
Basics of LiDAR



Basics of LiDAR



Machine Vision in AV: Camera with AI



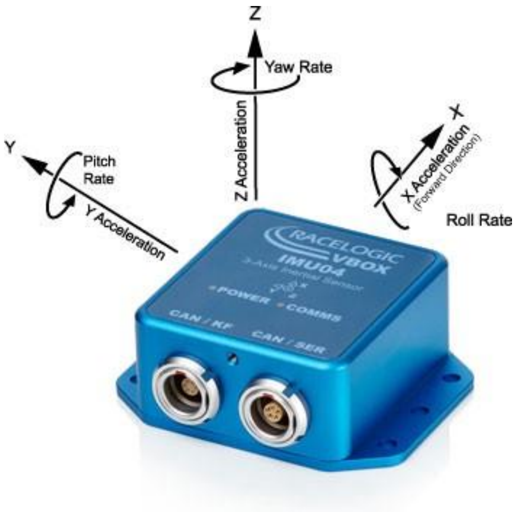
Identifies Objects

(i.e. roads, traffic lights, other vehicles, pedestrians, signs)

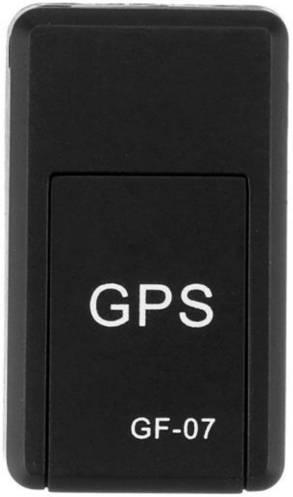
Localization for AV Bosch E.g.

Source: https://youtu.be/KvVv0kooDVU

Sensors that Support Localization Functions



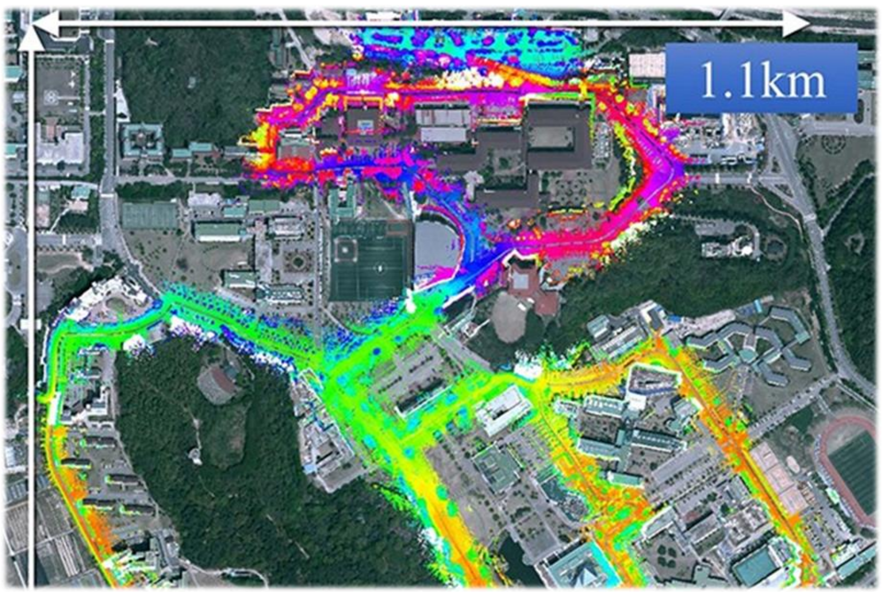
Inertial Sensors (IMU)



Satellite Navigation

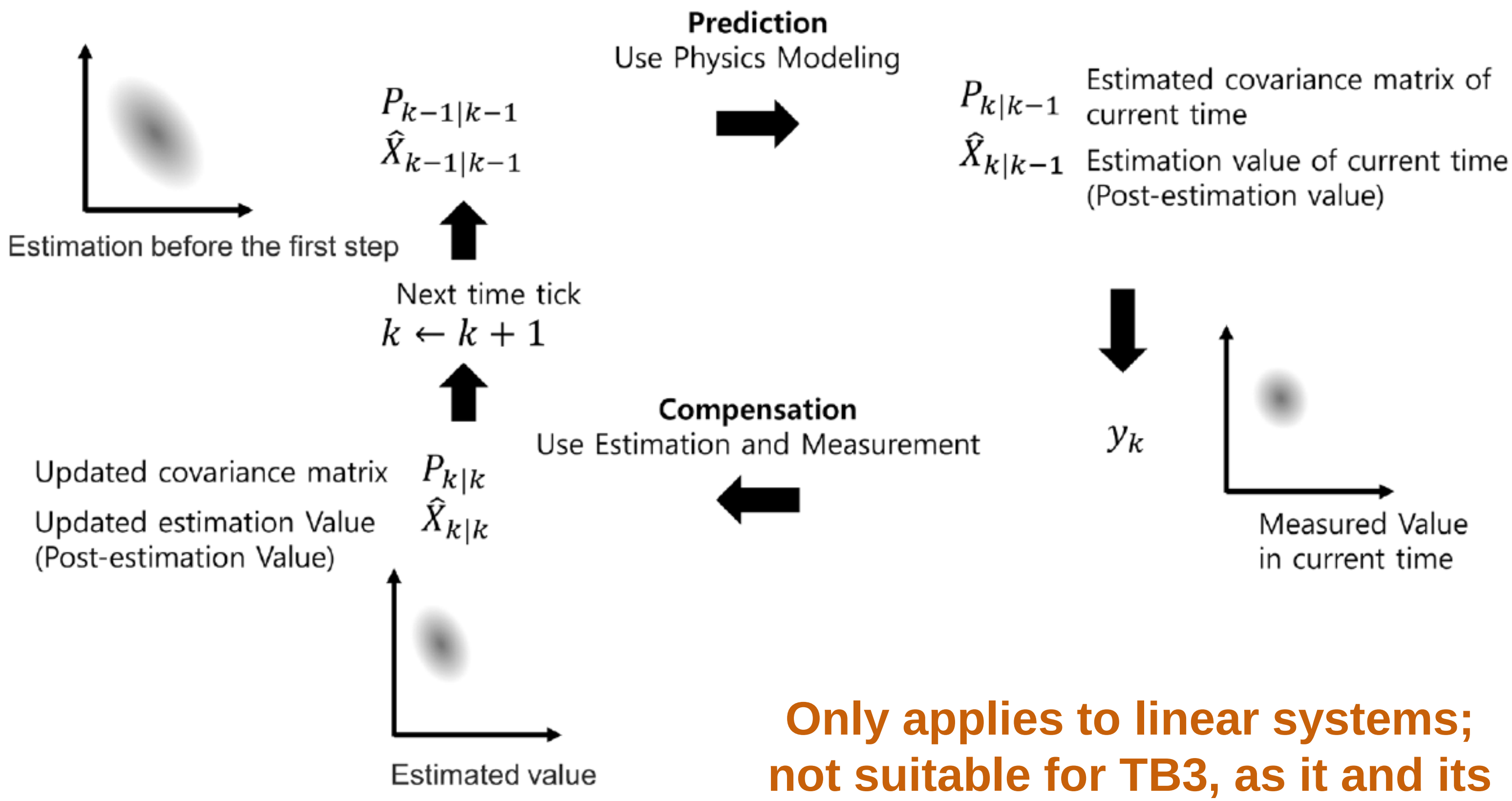
Surround Sensors

SLAM (Simultaneous Localization & Mapping)



- Technique that **estimates the map** (the coordinates of the landmarks) in addition to **estimating the coordinates of our vehicle**
- Can **utilize Lidar** to find walls, sidewalks and thus build a map
- **Utilize IMU data** to track robot's movements
- **Very popular for outdoor and indoor navigation where GPS is not very effective**
- SLAM's algorithms need to know how to recognize landmarks, then position them and add elements to the map

Localization Method 1: Kalman Filter



Only applies to linear systems; not suitable for TB3, as it and its sensors are non-linear systems

Localization Method 2: Particle Filter

- Also a **Bayes Filter Implementation**; Used in TB3 (specifically AMCL; Adaptive Monte Carlo Localization)
- A technique to predict through simulation based on **try-and-error** method
- Estimated value generated by the probability distribution in the system is represented as **particles**
- Estimates the pose of the object **assuming that the error is included** in the incoming information
- The particles are first moved to a new estimated position and orientation, based on the robot’s motion model and probabilities
- The weight of each particle is then measured according to the actual measurement value, and the **noise is gradually reduced to estimate a precise pose**

Localization Method 2: Particle Filter

- Particle filter are initialized by a very high number of particles spanning the entire state space
- As you get additional measurements, you predict and update your measurements which makes your robot have a multi-modal posterior distribution
- This is a big difference from a Kalman Filter which approximates your posterior distribution to be a Gaussian
- Over multiple iterations, the particles converge to a unique value in state space

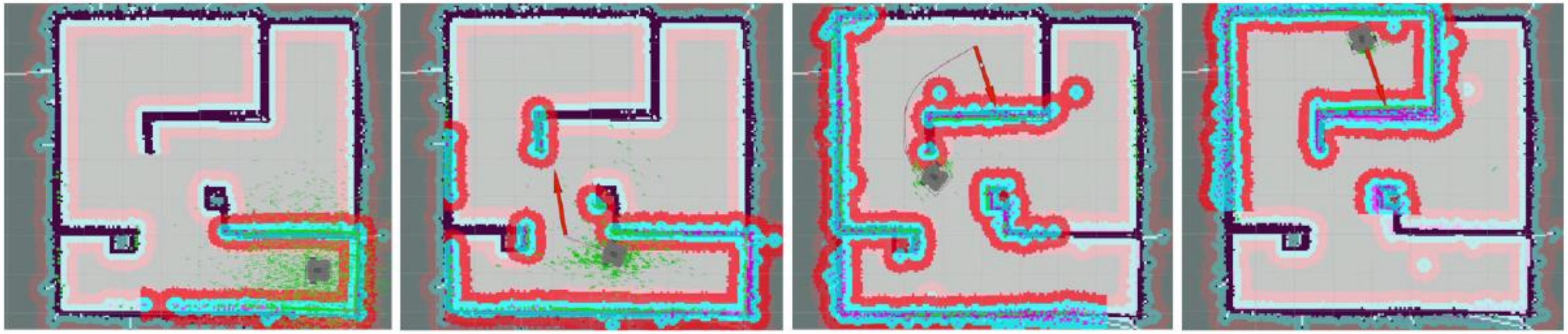
Localization Method 2: Particle Filter

5 procedures (steps 2~5 repeated):

1. **Initialization** – Particles are randomly arranged within the range where the pose can be obtained with N particles
2. **Prediction** – Based on the system model describing the motion of the robot, it moves each particle as the amount of observed movement with odometry information and noise
3. **Update** – Based on the measured sensor information, the probability of each particle is calculated and the weight value of each particle is updated based on the calculated probability
4. **Pose Estimation** – The position, orientation, and weight of all particles are used to calculate the average weight, median value, and the maximum weight value for estimating pose of the robot
5. **Resampling** – The less weighed particles are removed, and new particles that inherit the pose information of the weighted particles will replace the removed particles

Localization Method 2: Particle Filter

AMCL Process for TB3 Pose Estimation:



(t1) (t2) (t3) (t4)

AMCL is an **improved version of Monte Carlo pose estimation**, which **improves real-time performance by reducing the execution time with less number of samples** in the Monte Carlo pose estimation algorithm

AMCL

Adaptive particle filter:

- Better to use an adaptive particle filter which converges much faster and is computationally much more efficient than a basic particle filter
- The key idea is to bound the error introduced by the sample-based representation of the particle filter
- To use adaptive particle filter for localization, we start with a map of our environment and we can either set robot to some position, in which case we are manually localizing it or we could very well make the robot start from no initial estimate of its position
- Now as the robot moves forward, we generate new samples that predict the robot's position after the motion command
- Sensor readings are incorporated by re-weighting these samples and normalizing the weights
- The reason why it takes the filter multiple sensor readings to converge is that within a map, we might have dis-ambiguities due to symmetry in the map, which is what gives us a multi-modal posterior belief

AMCL

- AMCL = An adaptive particle filter method
- A probabilistic localization system for a robot moving in 2D
- Currently AMCL works only with laser scans; however, it can be extended to work with other sensors
- AMCL takes in a laser-based map (generated from SLAM), laser scans, and transform messages, and outputs pose estimates

AMCL

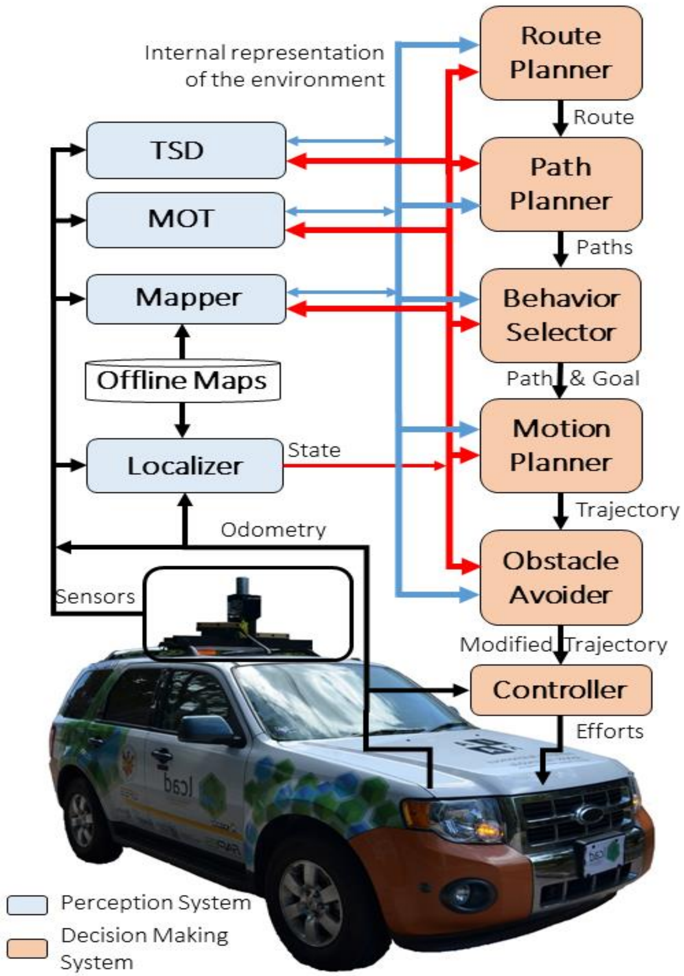
- **Subscribed topics:**
  - scan – Laser scans
  - tf – Transforms
  - initialpose – Mean and covariance with which to (re-) initialize the particle filter
  - map – the map used for laser-based localization
- **Published topics:**
  - amcl_pose – Robot's estimated pose in the map, with covariance.
  - particlecloud – The set of pose estimates being maintained by the filter
  - tf – Publishes the transform from odom (which can be remapped via the ~odom_frame_id parameter) to map.

AMCL

Parameter	Description	Default
min_particles	Minimum allowed number of particles	100
max_particles	Maximum allowed number of particles	5000
laser_model_type	Which model to use, either beam or likelihood_field	likelihood_field
laser_likelihood_max_dist	Maximum distance to do obstacle inflation on map, for use in likelihood_field model	2.0
initial_pose_x	Initial pose mean (x), used to initialize filter with Gaussian distribution	0.0
initial_pose_y	Initial pose mean (y), used to initialize filter with Gaussian distribution	0.0
initial_pose_a	Initial pose mean (yaw), used to initialize filter with Gaussian distribution	0.0

2. AV DECISION-MAKING (DM)

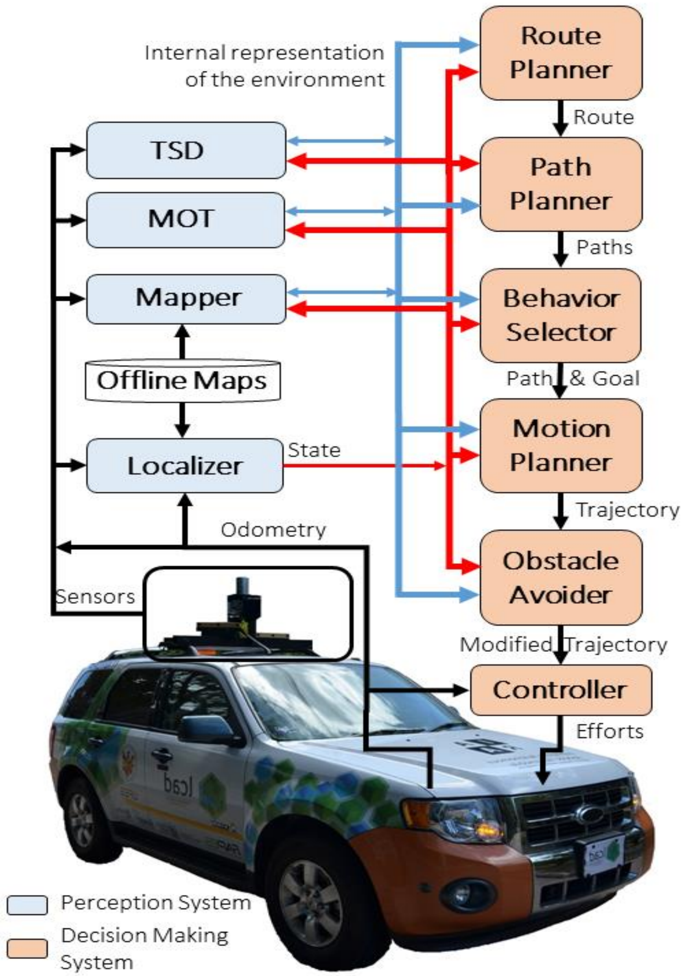
Typical Architecture of Self-Driving Cars



The DM Subsystems:

- **The Route Planner subsystem** is responsible for **computing a Route** (i.e. a sequence of way points) **in the Online Maps, from the current self-driving car's State to the Final Goal**
- **Given the computed Route, the Path Planner subsystem computes a set of Paths** (a Path is a sequence of poses) after **considering the current self-driving car's State** and the **internal representation of the environment** as well as **traffic rules**

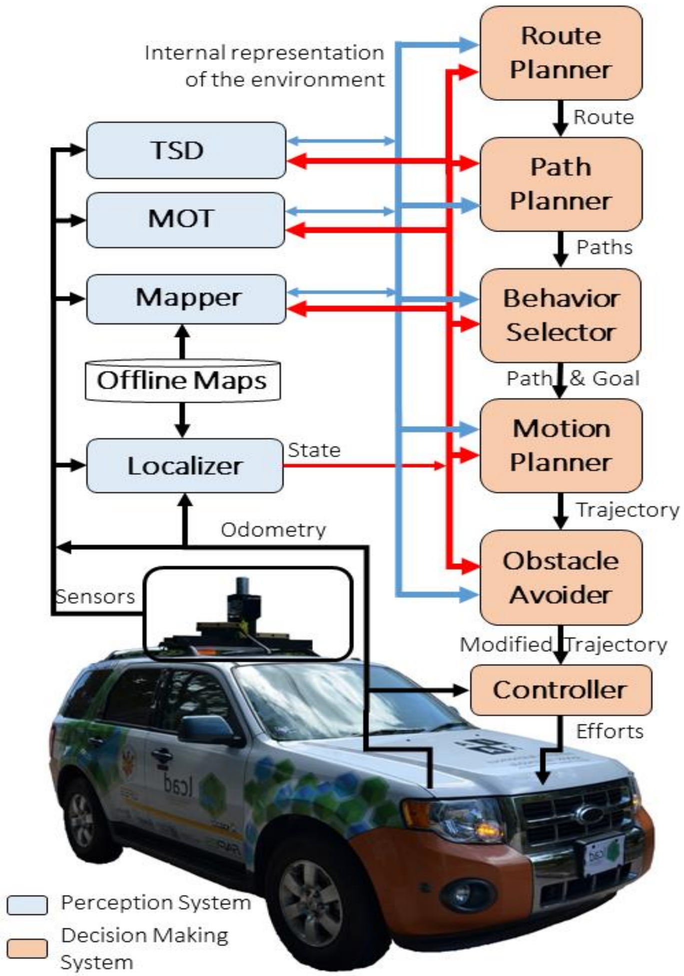
Typical Architecture of Self-Driving Cars



The DM Subsystems:

- **The Behavior Selector subsystem** is responsible for **choosing the current driving behavior** (e.g. lane keeping, intersection handling, traffic light handling)
  - does so by **first selecting the most suitable Path**
  - **From this Path, the anticipated Goal is constantly selected in real-time;** consists of two components: **(a) the anticipated pose, and (b) the anticipated desired velocity**

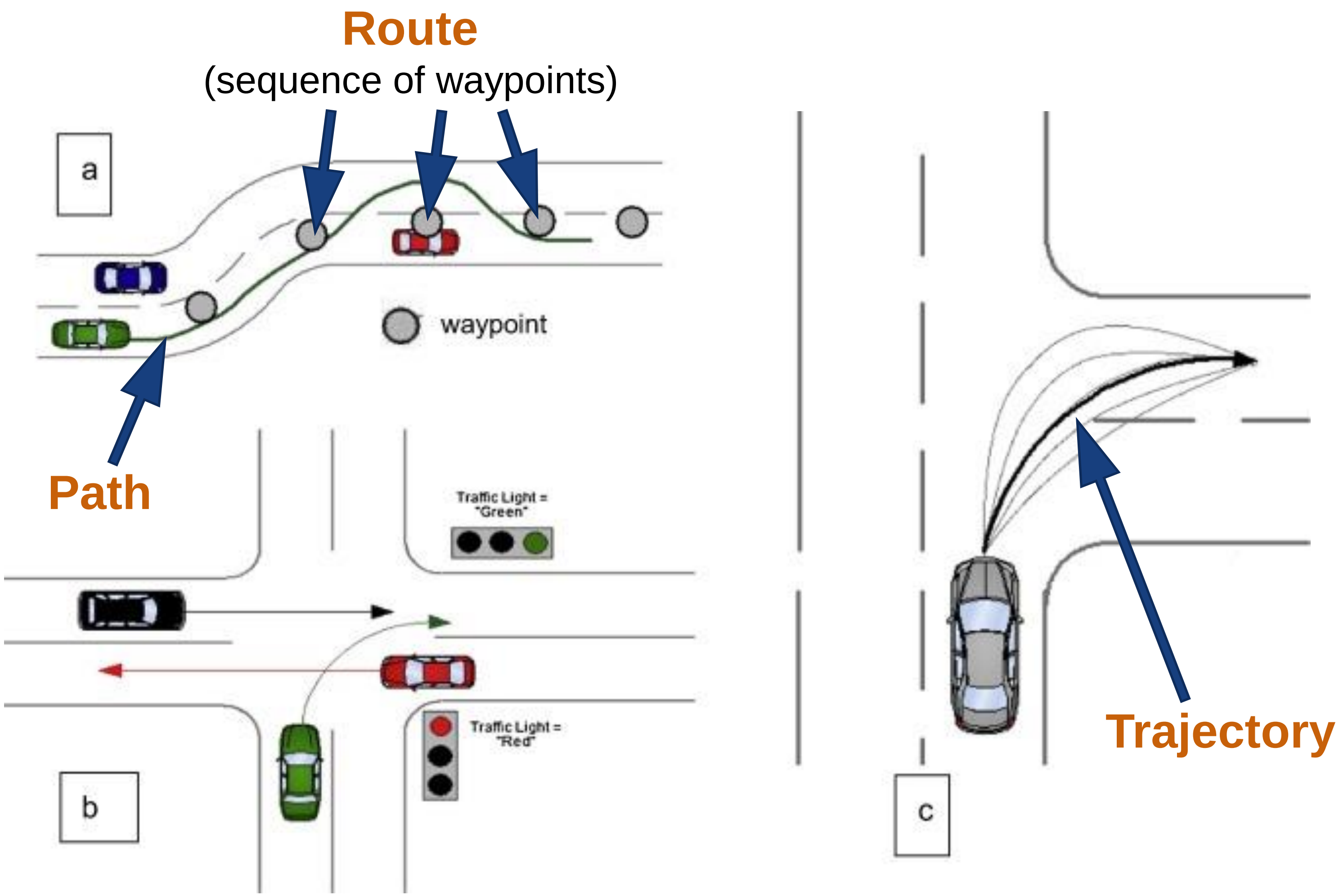
Typical Architecture of Self-Driving Cars



The DM Subsystems:

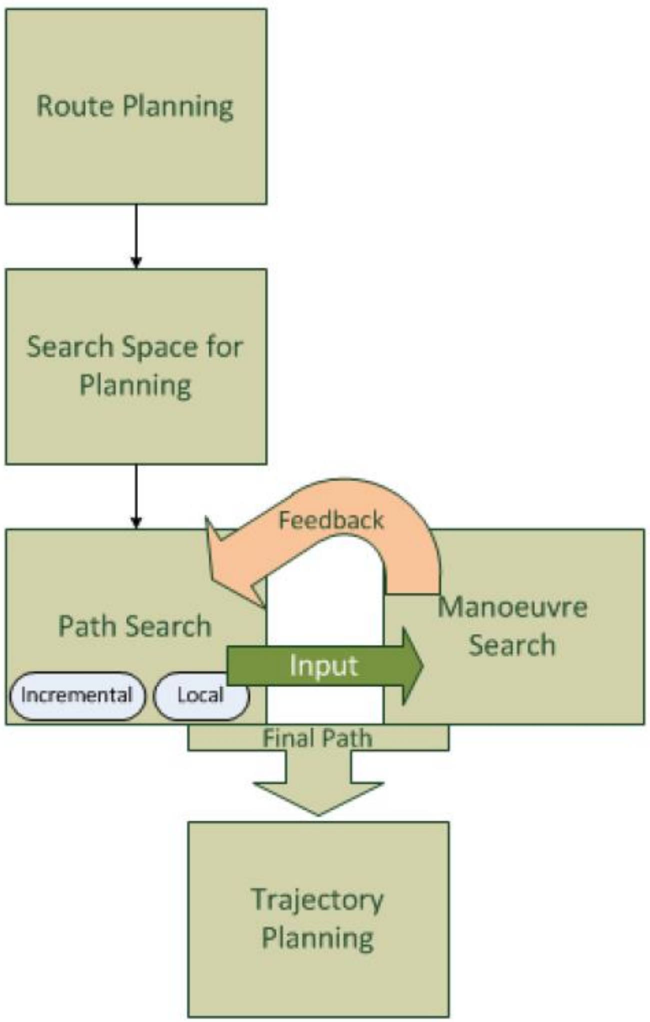
- **The Motion Planner subsystem** is responsible for **computing a Trajectory** (i.e. a sequence of commands);
- The Trajectory
  - takes the car **from its current State to the current Goal smoothly and safely**
  - **follows the Path defined by the Behavior Selector**
  - must **satisfy the car's kinematic and dynamic constraints**, and **provides comfort** to the passengers

Routes vs Paths vs Trajectories



(a) Path planning, (b) Manoeuvre planning and (c) Trajectory planning

Flow Chart of Planning Modules



1. **A route is chosen from the route planner**
2. **Path search is initiated**
3. Path search **acts as input to the search for the best manoeuvre** (i.e. the manoeuvre which places the car with the most correct and safe behaviour)
4. **The final path may change, based on the best manoeuvre**, as shown with a feedback loop between these two search modules
5. **Once the path is finalised, the final trajectory planning is generated**

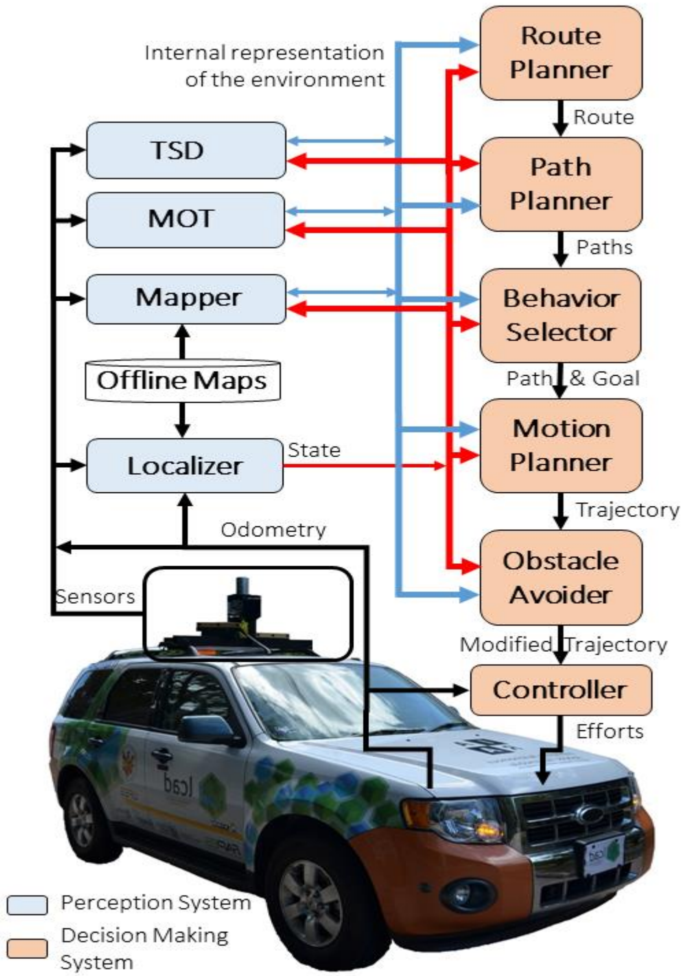
Path Calculation and Driving

Many Algorithms that perform path search and planning (i.e. calculating optimal routes):

1. Dijkstra's algorithm
2. A* algorithm
3. Potential field
4. Particle filter
5. RRT (Rapidly-exploring Random Tree)

Some of which are covered in Robotic Systems Course (i.e. Module 5 Robotic Pathway Planning); See Appendix at bottom of the slides for refresher

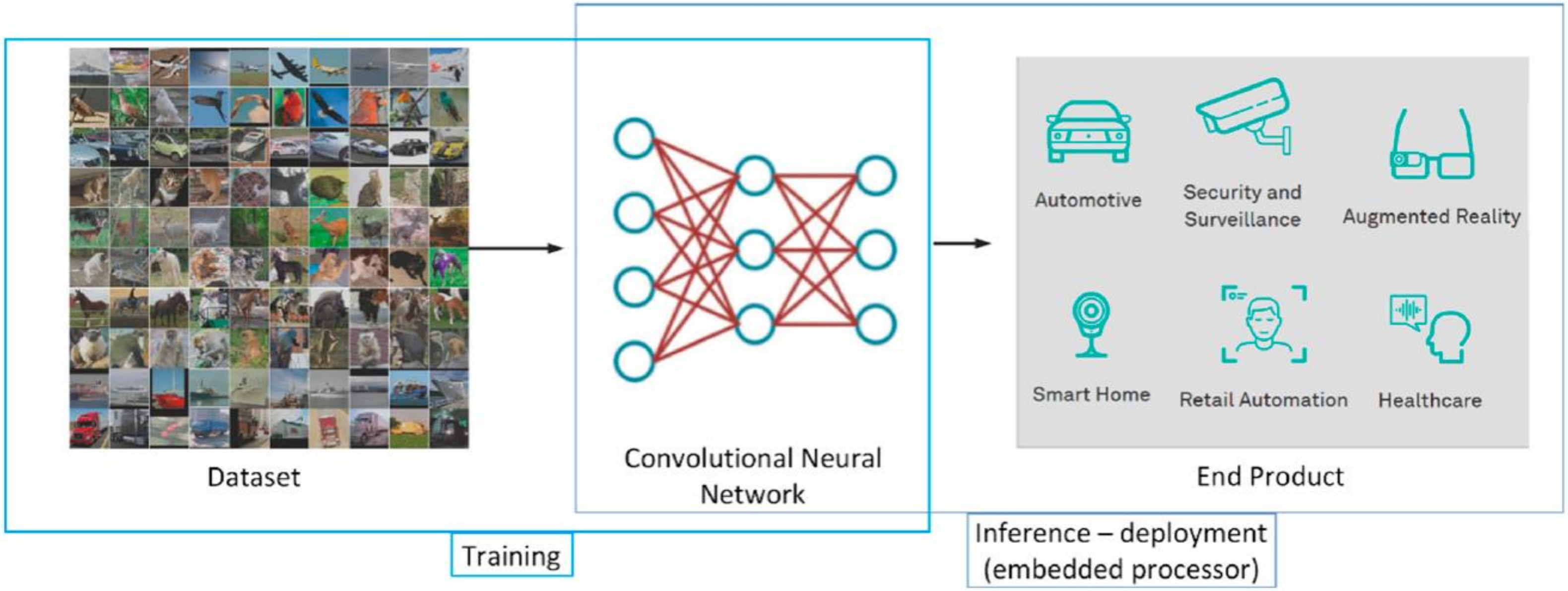
Typical Architecture of Self-Driving Cars



The DM Subsystems:

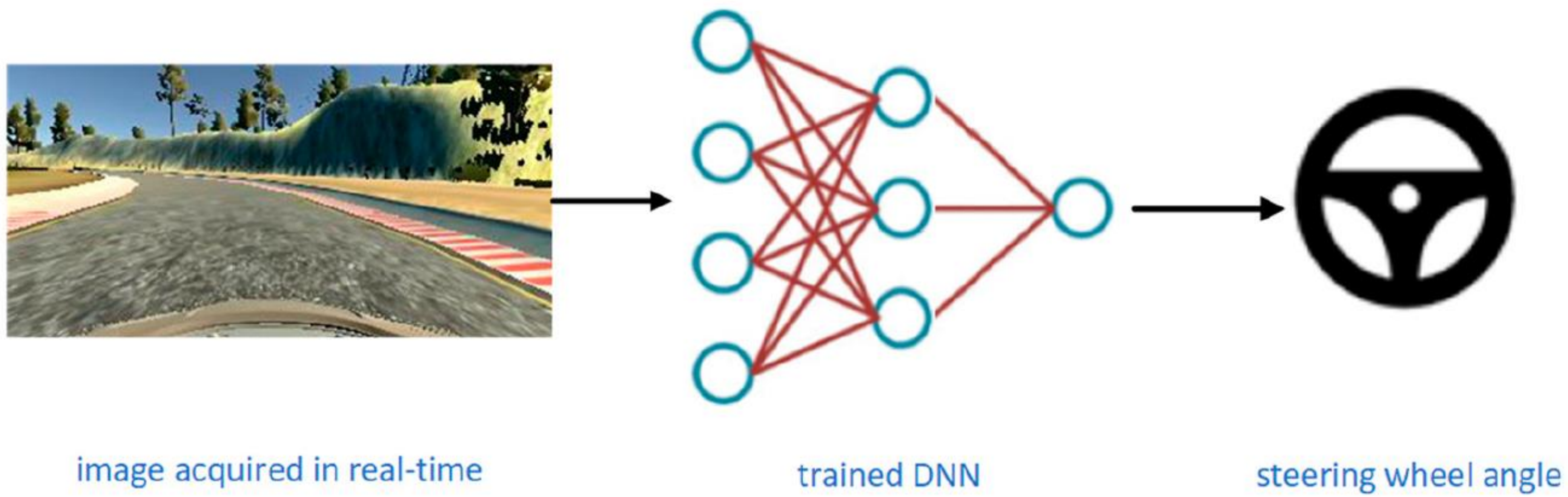
- **The Obstacle Avoider subsystem** is responsible for **modifying the computed Trajectory** (e.g. reducing the velocity, changing directions)**, if necessary, to avoid collisions**
- **The Controller subsystem** is responsible for **computing and sending Effort commands to the actuators of the steering wheel, throttle and brakes** based on the Modified Trajectory

General Deep Learning Development Flow



- **Training** — once the CNN model is developed, it is **trained with the appropriate dataset**
- **Inference** — the **trained model is deployed** at the end product and **used for inference with real-time input data**

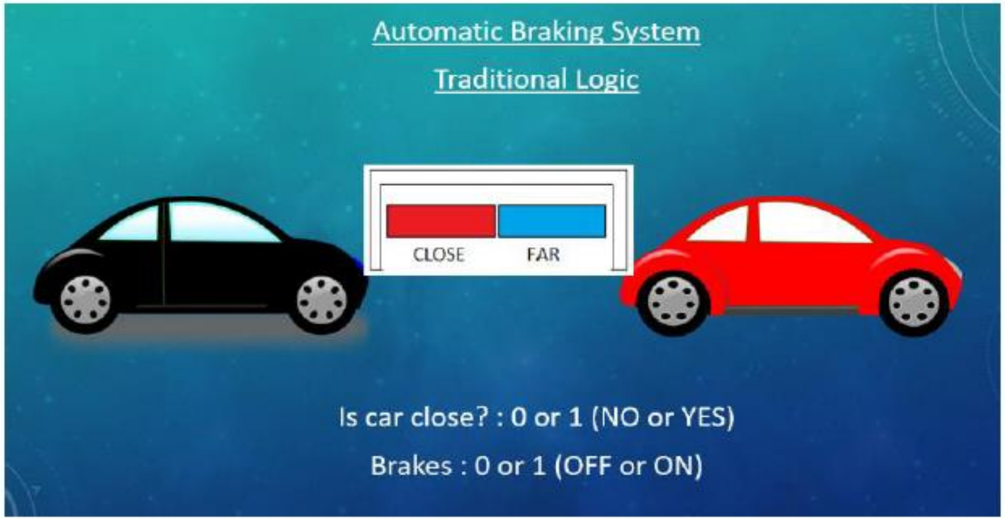
Application of Deep Learning to AVs (E.g.)



Real-time Autonomous Driving

- **Image acquired** from the central camera is **fed to the trained deep neural network (DNN) model**
- **Output of this model is the steering angle prediction** that controls the vehicle

Application of Fuzzy Logic to AVs (E.g.)



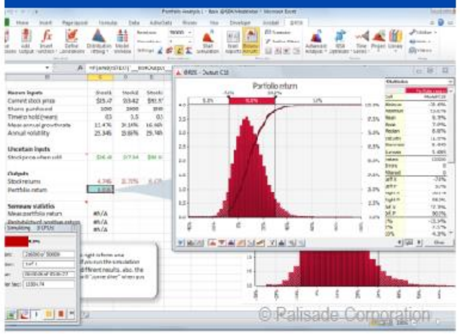
Automatic Braking System

- Using **fuzzy logic to determine the distance of the obstacle** in front of our vehicle and **apply the appropriate amount of brake required to stop the car;**
- Will slowly apply the brakes considering the distance of the vehicle around it and finally stop; will not apply complete brake immediately
- **Avoid unnecessary accident risk**

CHAPTER 2:
BASICS OF SIMULATION & ANALYSIS

Various Testing Avenues for AVs

ISO 29990:2010

Virtual tests	Proving ground tests	Field tests
		
☑ Analysis of a huge number of scenarios, environments, system configurations and driver characteristics	☑ Reproducibility by use of driving robots, self driving cars and targets; critical manoeuvres are possible	☑ Investigation of real driving situations and comparison with system specifications

Effort for coverage of all relevant scenarios & environments

Uncertainties & simplifications

Source: U. Steininger, H.P. Schöner, M. Schiementz: Requirements on tools for assessment and validation of assisted and automated driving systems, 7. Tagung Fahrerassistenz, München, Nov. 2015

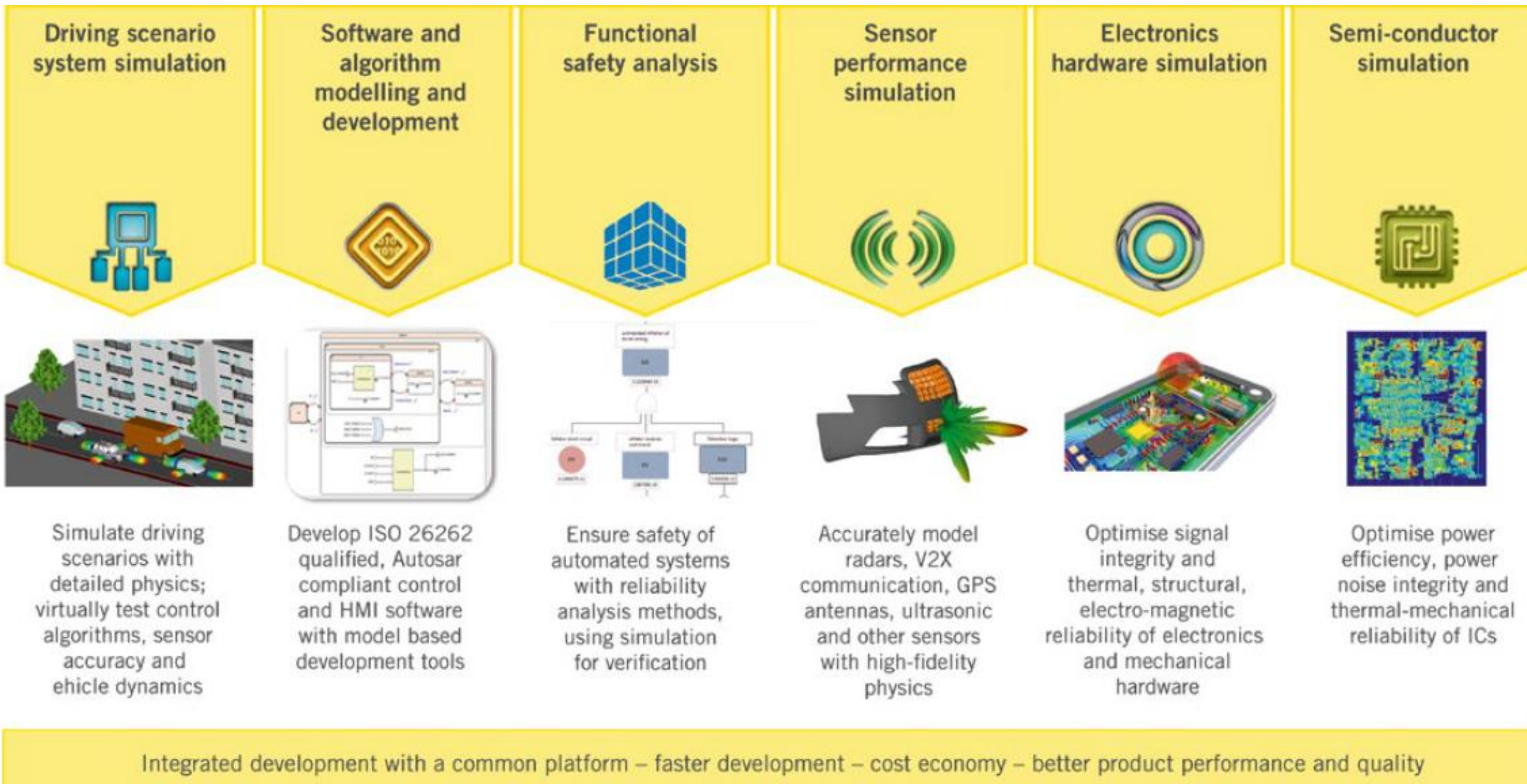
Importance of Simulation & Analysis

Source: https://www.youtube.com/watch?v=vj-YJ4lDS3I

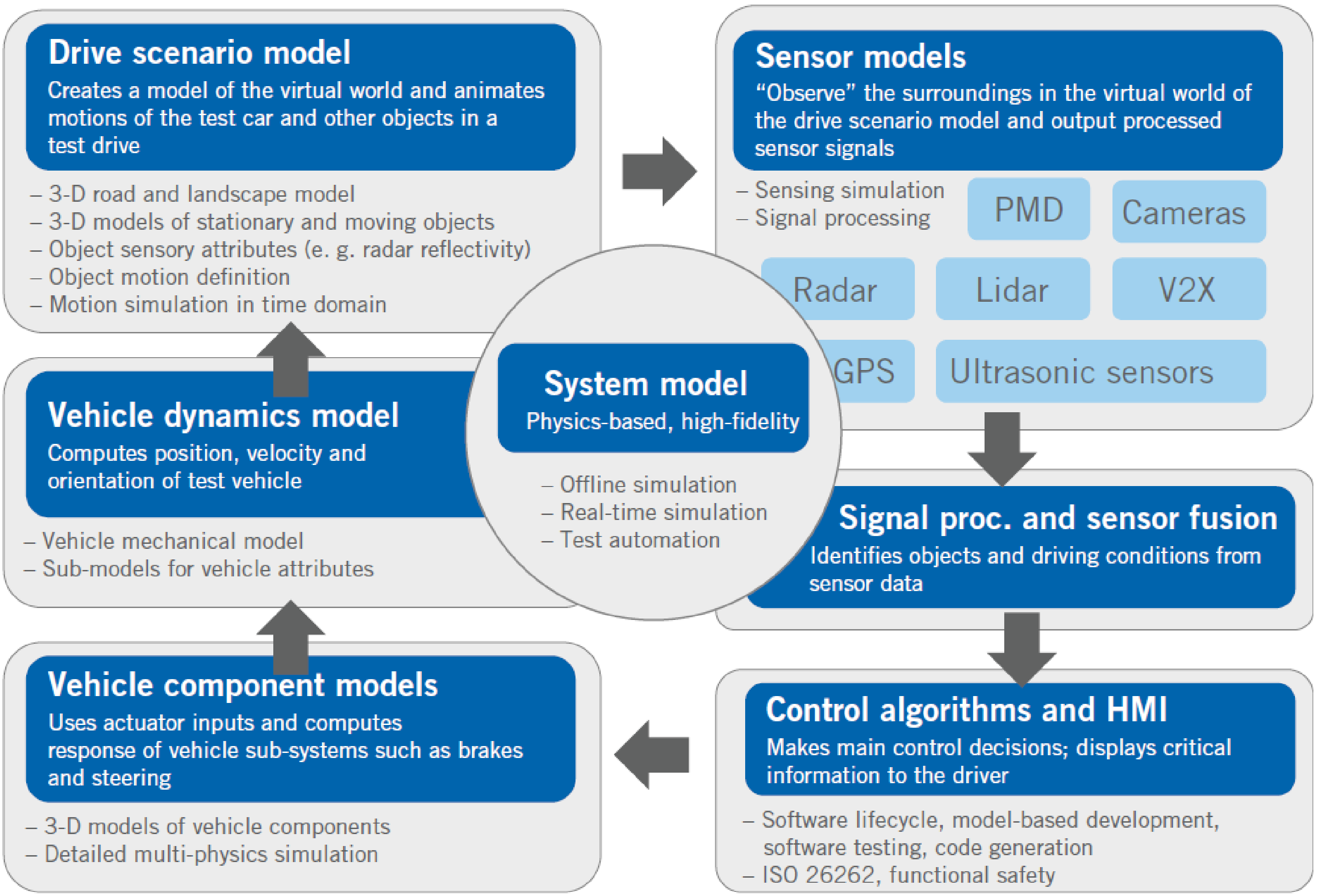
Importance of Simulation & Analysis

- **Validate safety**
- **Shorten analysis times**; efficiently verify functional requirements
- **No real damages/injuries** inflicted in simulations
- **Test and re-test variations** of specific scenarios, environments, system configurations
- **Able to scale testing massively**
- **Channel to train machine learning agents**; quote from Microsoft's AirSim team:
  - *"TRAINING MODELS ON AIRSIM FIRST AND THEN FINE TUNING THEM WITH REAL-WORLD DATA GREATLY REDUCES THE AMOUNT OF REAL-WORLD DATA YOU WILL NEED FOR A FULLY TRAINED MODEL."*

Various Types of Simulation & Analysis Methods for AVs



Simulation of the Control Loop for AVs

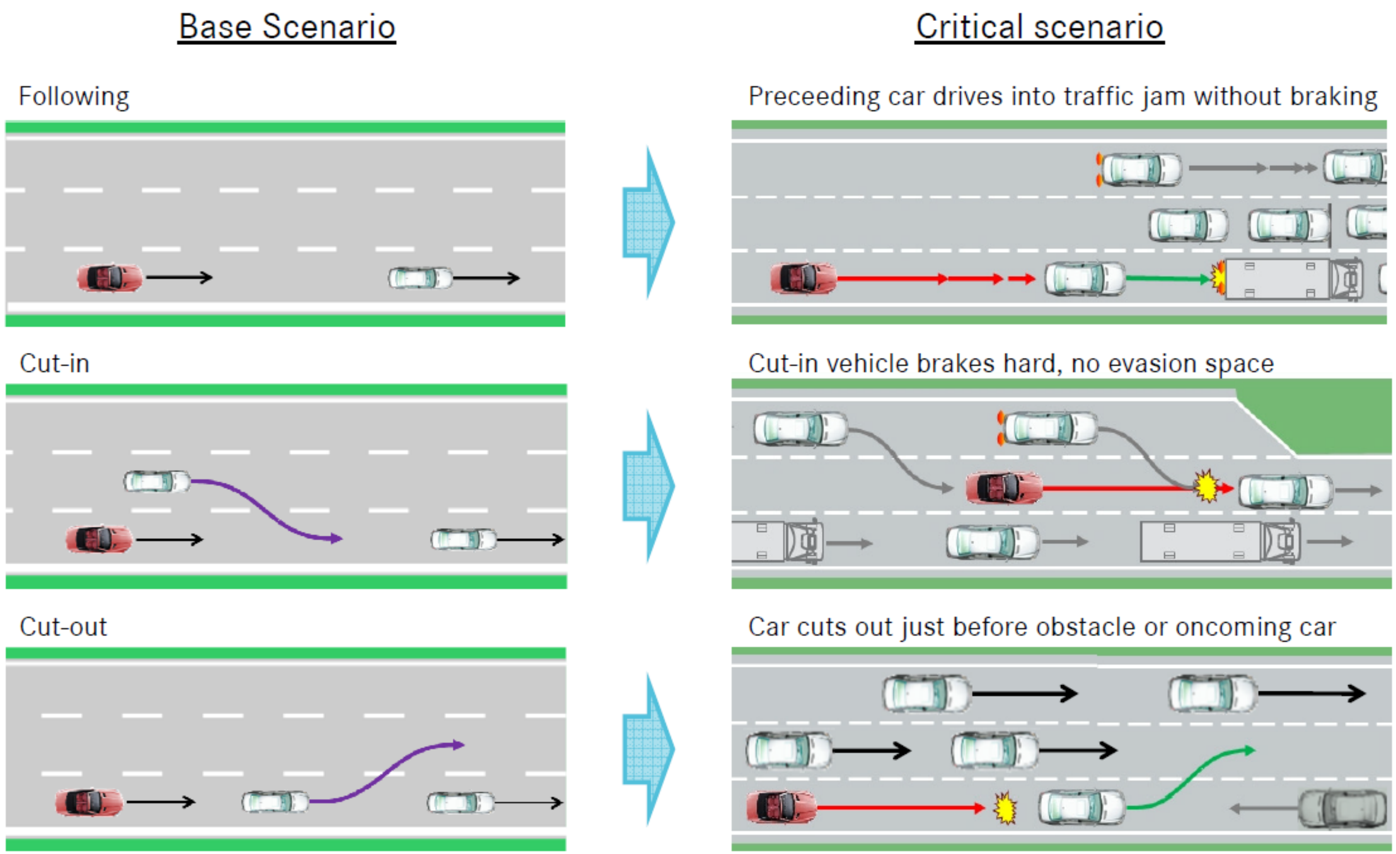


Virtual Driving with Model Based Simulation (Simplified)

Use Case Example: Function Development for Complex Traffic Scenarios

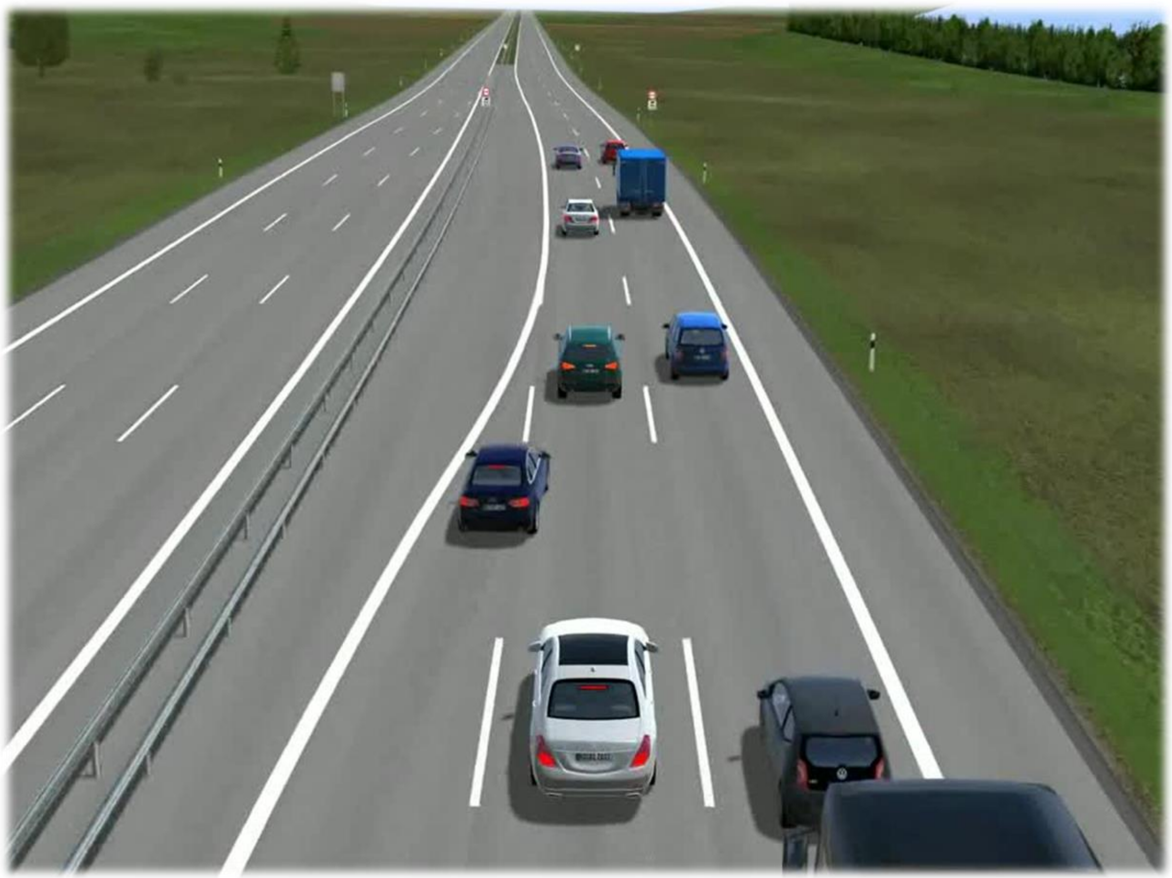
Use Case Example: Function Development for Complex Traffic Scenarios

Challenging Traffic Situations !



Use Case Example: Function Development for Complex Traffic Scenarios

Simulation Example: Cut-in Scenario



WORKSHOP: INTRODUCTION TO SOFTWARE PERTAINING TO TOPICS ON SIMULATION & ANALYSIS FOR AUTONOMOUS VEHICLES

Various Software

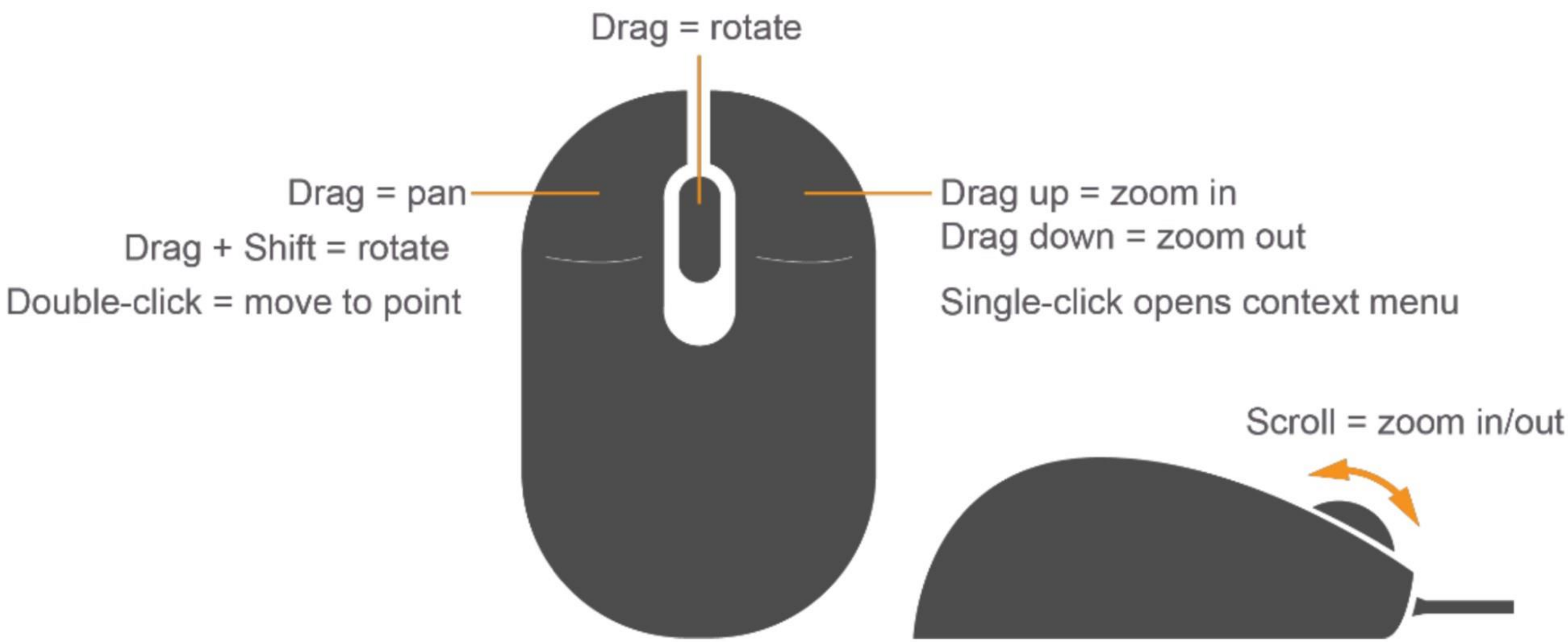
1. **ROS Gazebo (Open Source)**
2. BlenSor (Open Source)
3. CARLA (Open Source)
4. Webots (Open Source)
5. Windows AirSim (Open Source)

Introducing Gazebo

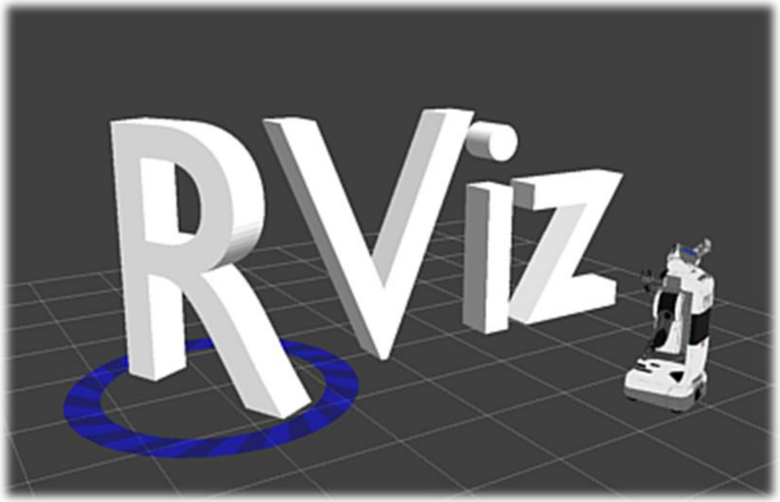
- **3D robot simulator** that provides robots, sensors, environment models for 3D simulation required for robot development
- Offers **realistic simulation** with its physics engine
- **High performance** even though open source
- **Very compatible with ROS**

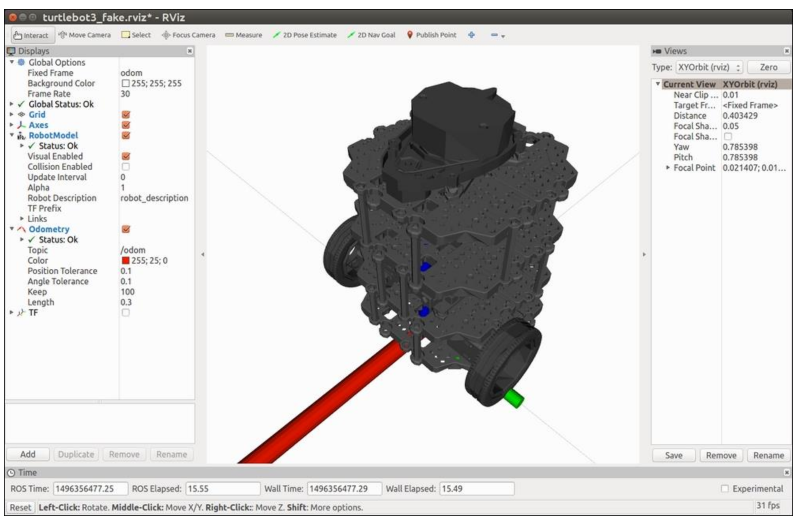
Introducing Gazebo

Gazebo mouse controls



Introducing RViz





- **3D visualization tool**
- **Display sensor data and state information** from ROS
- A very useful tool to **control TurtleBot3** and **test SLAM and Navigation**

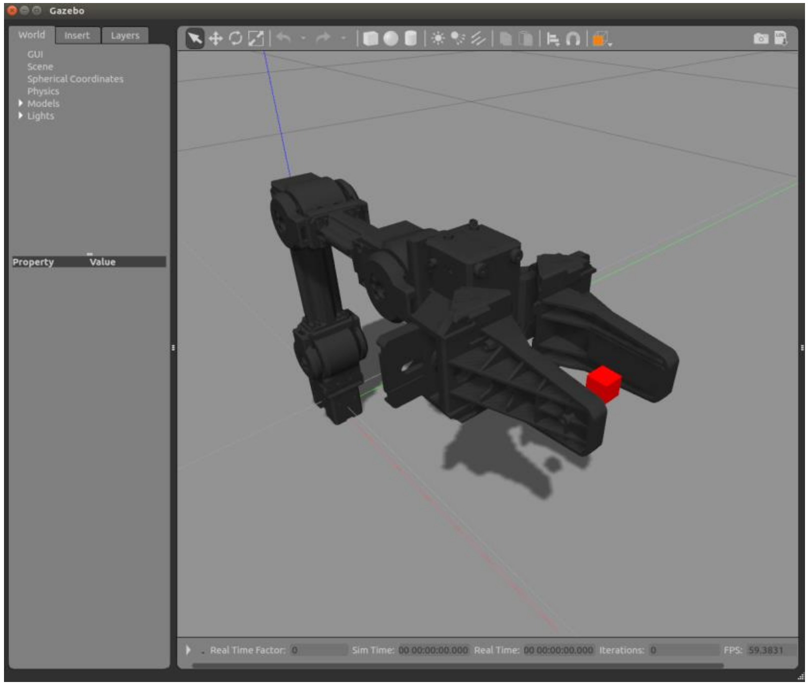
Workshop Stages

Simulation of:

1. **OpenManipulator-X**
2. **TurtleBot3 Waffle Pi**
   a) Manual Movements
   b) Random Autonomous Navigation and Obstacle Avoidance
   c) SLAM and Controlled Navigation
3. **TurtleBot3 Waffle Pi with OpenManipulator-X**

1. SIMULATION OF OPENMANIPULATOR-X

OpenManipulator-X



Basically, the control process is the same as the previous workshop session, except that the physical robot is replaced by the virtual one

Type in Terminal:

- `$ roslaunch open_manipulator_gazebo open_manipulator_gazebo.launch`
- This will load OpenManipulator-X on Gazebo simulator
- Click “Play” when Gazebo is launched

OpenManipulator-X

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/set_link_state
/gazebo/set_model_state
/open_manipulator/gripper/kinematics_pose
/open_manipulator/gripper_position/command
/open_manipulator/gripper_sub_position/command
/open_manipulator/joint1_position/command
/open_manipulator/joint2_position/command
/open_manipulator/joint3_position/command
/open_manipulator/joint4_position/command
/open_manipulator/joint_states
/open_manipulator/option
/open_manipulator/states
/rosout
/rosout_agg
```

Type (in New Terminal):

- `$ rostopic list`
- This will list up the activated topics (examples on left)

Type (in New Terminal):

- `$ rosservice list`
- This will list up the active services

OpenManipulator-X

Activate the Controller

```
SUMMARY
========

PARAMETERS
 * /open_manipulator/control_period: 0.01
 * /open_manipulator/moveit_sample_duration: 0.05
 * /open_manipulator/planning_group_name: arm
 * /open_manipulator/using_moveit: False
 * /open_manipulator/using_platform: False
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES
  /
    open_manipulator (open_manipulator_controller/open_manipulator_controller)

ROS_MASTER_URI=http://localhost:11311

process[open_manipulator-1]: started with pid [9820]
[ INFO] [1544506914.862653563]: Ready to simulate /open_manipulator on Gazebo
```

*****You have to activate the controller using this code before you can control the virtual robot via the following various methods**

Next, type (in New Terminal):

- `$ roslaunch open_manipulator_controller open_manipulator_controller.launch use_platform:=false`
- If launched successfully, the terminal will appear as follows (left)
- You are ready to control your virtual robotic arm

OpenManipulator-X

(a) Manual Control

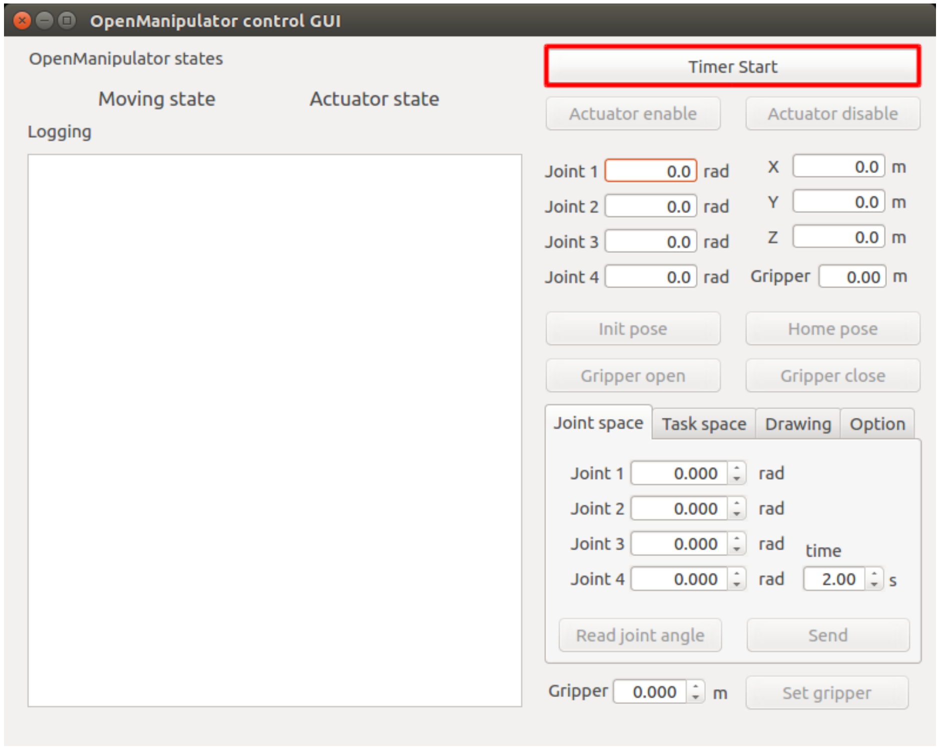
```
---------------------------
Control Your OpenMANIPULATOR-X!
---------------------------
w : increase x axis in task space
s : decrease x axis in task space
a : increase y axis in task space
d : decrease y axis in task space
z : increase z axis in task space
x : decrease z axis in task space

y : increase joint 1 angle
h : decrease joint 1 angle
u : increase joint 2 angle
j : decrease joint 2 angle
i : increase joint 3 angle
k : decrease joint 3 angle
o : increase joint 4 angle
l : decrease joint 4 angle

g : gripper open
f : gripper close

1 : init pose
2 : home pose

q to quit
---------------------------
```

Type (in New Terminal):

- `$ roslaunch open_manipulator_teleop open_manipulator_teleop_keyboard.launch`
- If the node is successfully launched, the following instruction will appear in the terminal window (left)
- Use your keyboard keys to control the arm; response will reflect in the Gazebo
- Close terminal when done

OpenManipulator-X

(b) Control via GUI



*****Ensure Gazebo and the Controller are activated first**

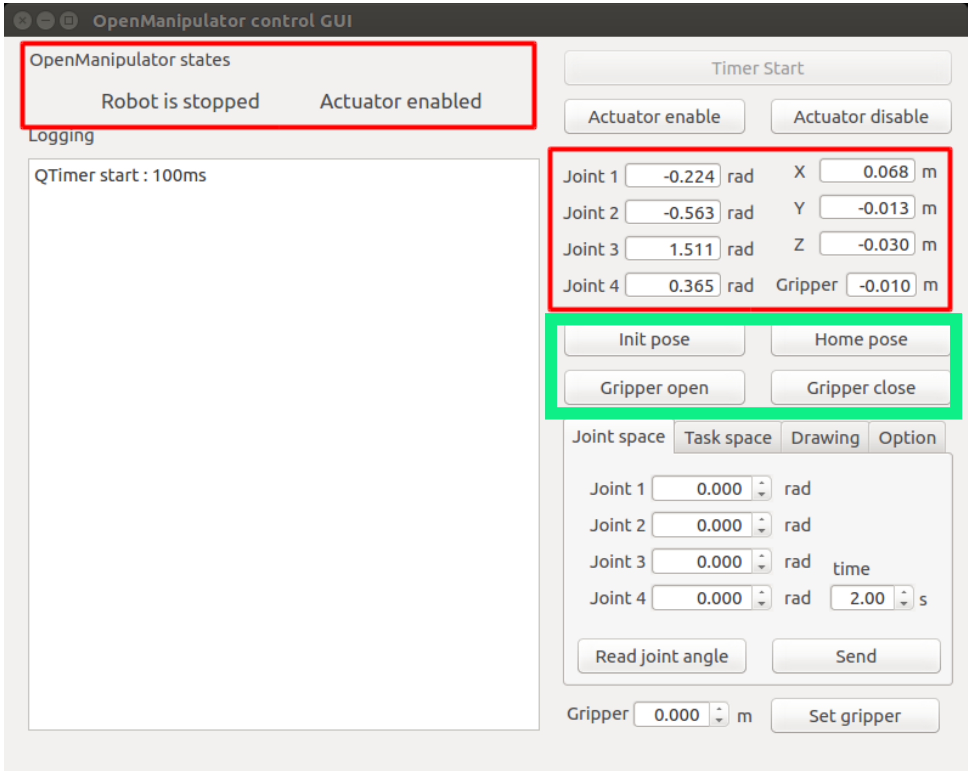
Type (in New Terminal):

- `$ roslaunch open_manipulator_control_gui open_manipulator_control_gui.launch`
- This will open the GUI for the arm control
- Click “Timer Start” & then “Actuator enable” and you can start controlling the arm; see Gazebo window for response

OpenManipulator-X

(b) Control via GUI

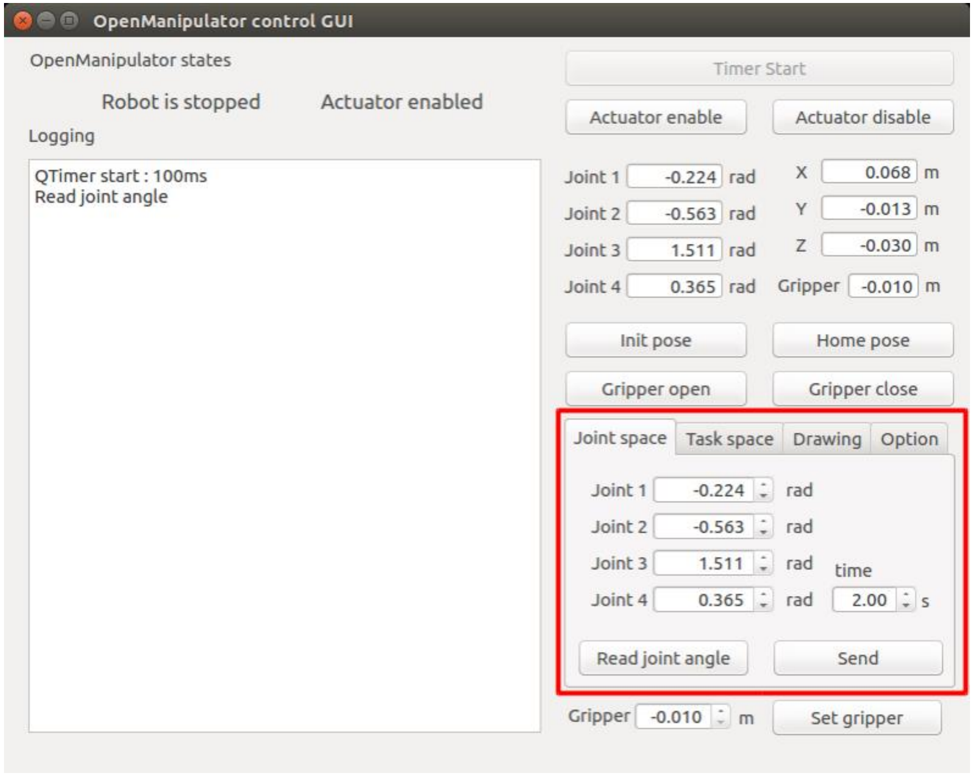
- You can **check the status of the arm** (i.e. joint states, kinematics pose, end effector position on X-Y-Z)
- To manipulate the arm to get it into **simple poses** (i.e. Init pose, Home pose, gripper open/close), click on the various buttons (in green box) one-by-one



OpenManipulator-X

(b) Control via GUI

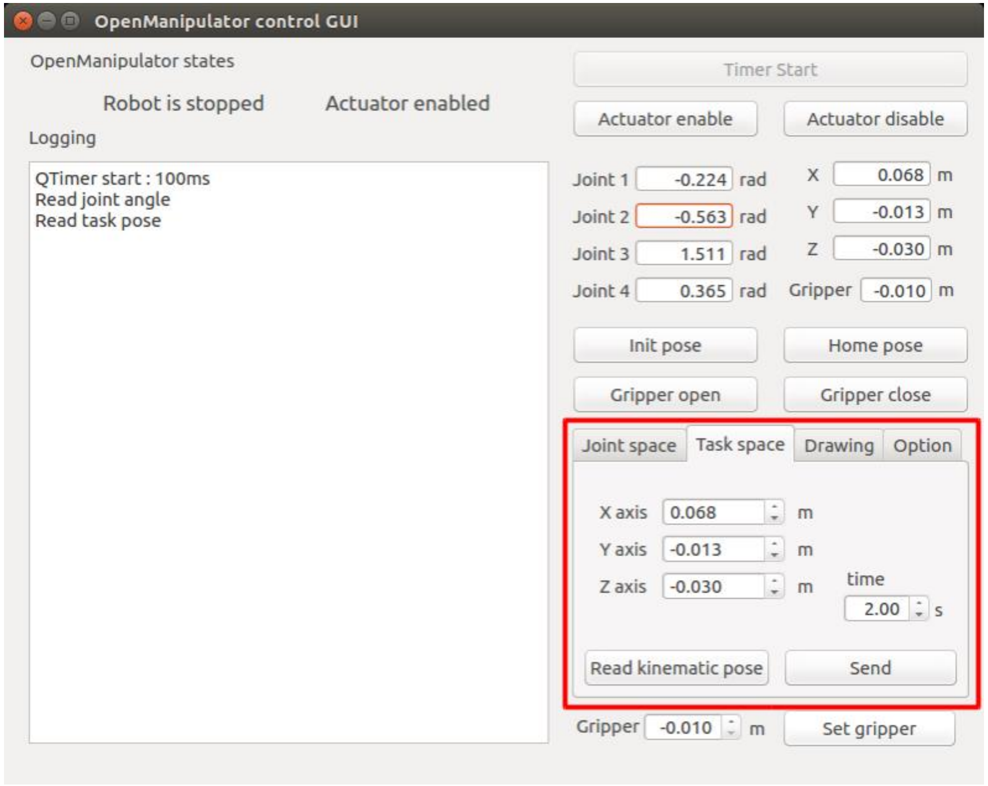
To manipulate the arm in the joint space (i.e. **Forward Kinematics**), enter the joint angles and trajectory time; click “Send” when done:



OpenManipulator-X

(b) Control via GUI

To manipulate the arm in the task space (i.e. **Inverse Kinematics**), enter the X-Y-Z values for your end effector's desired position; click "Send" when done:



OpenManipulator-X

(c) Control via Python (rospy)

*****Ensure Gazebo and the Controller are activated first**

From terminal, type: `$ rosservice list`

```
/goal_joint_space_path
/goal_joint_space_path_from_present
/goal_joint_space_path_to_kinematics_orientation
/goal_joint_space_path_to_kinematics_pose
/goal_joint_space_path_to_kinematics_position
/goal_task_space_path
/goal_task_space_path_from_present
/goal_task_space_path_from_present_orientation_only
/goal_task_space_path_from_present_position_only
/goal_task_space_path_orientation_only
/goal_task_space_path_position_only
/goal_tool_control
/gripper_position/pid/set_parameters
/gripper_sub_position/pid/set_parameters
/gripper_sub_publisher/get_loggers
/gripper_sub_publisher/set_logger_level
/open_manipulator_controller/get_loggers
/open_manipulator_controller/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
/set_actuator_state
```

To move OM in the Gazebo is the same as in actual situation

For actual/Virtual OM, you have to service messages to move the arm (covered in M3 and in RBS Course)

OpenManipulator-X

(c) Control via Python (rospy)

- Open terminal and type:

```
$ rosrun autonomous control_om.py
```

- The OM will move to a point determined by the joint angles. To alter these joint angles, open the python file, find the following line and change the angles in it; these angles are in radians:

```
joint_position.position =  [-0.5, 0, 0.5, -0.5]
```

- To alter the gripper angles, alter the following line (-0.01 for fully close and 0.01 for fully open):

```
gripper_position.position =  [0.01]
```

(c) Control via Python (rospy)

Note if you cant execute the python file, you have to give execution permissions to it by typing:

```
$ chmod +x name_of_the_file.py
```

control_real_om.py

```
#!/usr/bin/env python
# works for actual OM ONLY!
# does not work for actual OM_with_TB3

import rospy                                                #import the python library for ROS
from open_manipulator_msgs.msg import JointPosition        #import JointPosition message from the open_manipulato
from open_manipulator_msgs.srv import SetJointPosition
from sensor_msgs.msg import JointState
import math
import time

def callback(msg):                                          #define a function called 'callback' that receives a parameter
        print msg.name
        print msg.position

def talker():
        rospy.init_node('OM_publisher') #Initiate a Node c
        set_joint_position = rospy.ServiceProxy('/open_manipulator/goal_joint_space_path', SetJointPosition)
        set_gripper_position = rospy.ServiceProxy('/open_manipulator/goal_tool_control', SetJointPosition)

        while not rospy.is_shutdown():
                joint_position = JointPosition()
                joint_position.joint_name = ['joint1','joint2','joint3','joint4']
                joint_position.position =  [-0.5, 0, 0.5, -0.5]          # in radians
                resp1 = set_joint_position('planning_group',joint_position, 3)
                gripper_position = JointPosition()
                gripper_position.joint_name = ['gripper']
                gripper_position.position =  [0.01]      # -0.01 for fully close and 0.01 for fully open
                respg2 = set_gripper_position('planning_group',gripper_position, 3)

                sub_joint_state = rospy.Subscriber('/open_manipulator/joint_states', JointState, callback)

if __name__== '__main__':
        try:
                talker()
        except rospy.ROSInterruptException:
                pass
```

Service Server

Subscriber

OpenManipulator-X

(d) Hand Guiding via Python

Instead of using the “*Processing*” app as done in M3, we will use *rospy* to execute the Hand Guiding process for the simulated OM

Using Given Package,

- Change to the source space directory of the catkin workspace:
  `$ cd ~/catkin_ws/src`
- Git Clone the Robotics package:
  `$ git clone https://github.com/nicholashojunhui/open_manipulator_save_and_load.git`
- Build the packages in the catkin workspace:
  `$ cd ~/catkin_ws && catkin_make`
- Go to *catkin_ws/src/open_manipulator_save_and_load/nodes* and make python file executable
- **Run Gazebo and the OM controller (refer to README file for commands)**
- **Run `roslaunch` on a new terminal to begin Hand Guiding (refer to README file for commands)**

(d) Hand Guiding via Python (Cont)

- Since you cant use your hand to guide the OM in Gazebo, use teleop manual control instead. In new terminal, type command:

```
$ roslaunch open_manipulator_teleop
open_manipulator_teleop_keyboard.launch
```

- Go back to terminal that you ran your save_and_load launch command (make sure you click on it first). When ready, type “1” and the programme starts recording the movements of the OM
- Go back to the “teleop” terminal, and move the OM in Gazebo using your keyboard. When done …..

(d) Hand Guiding via Python (Cont)

- Go back to the “save_and_load” terminal, and type “2” to stop the recording and to save the records into a file in the cfg folder
- To command the OM to perform the recorded tasks, at the “save_and_load” terminal, type “3”. Observe that the OM will move by itself by loading the recorded file (i.e. the trajectories/actions that you teach the OM)

2. SIMULATION OF TURTLEBOT3 WAFFLE PI

(a) Manual Movements (RViz)

Using fake node in RViz; type in Terminal:

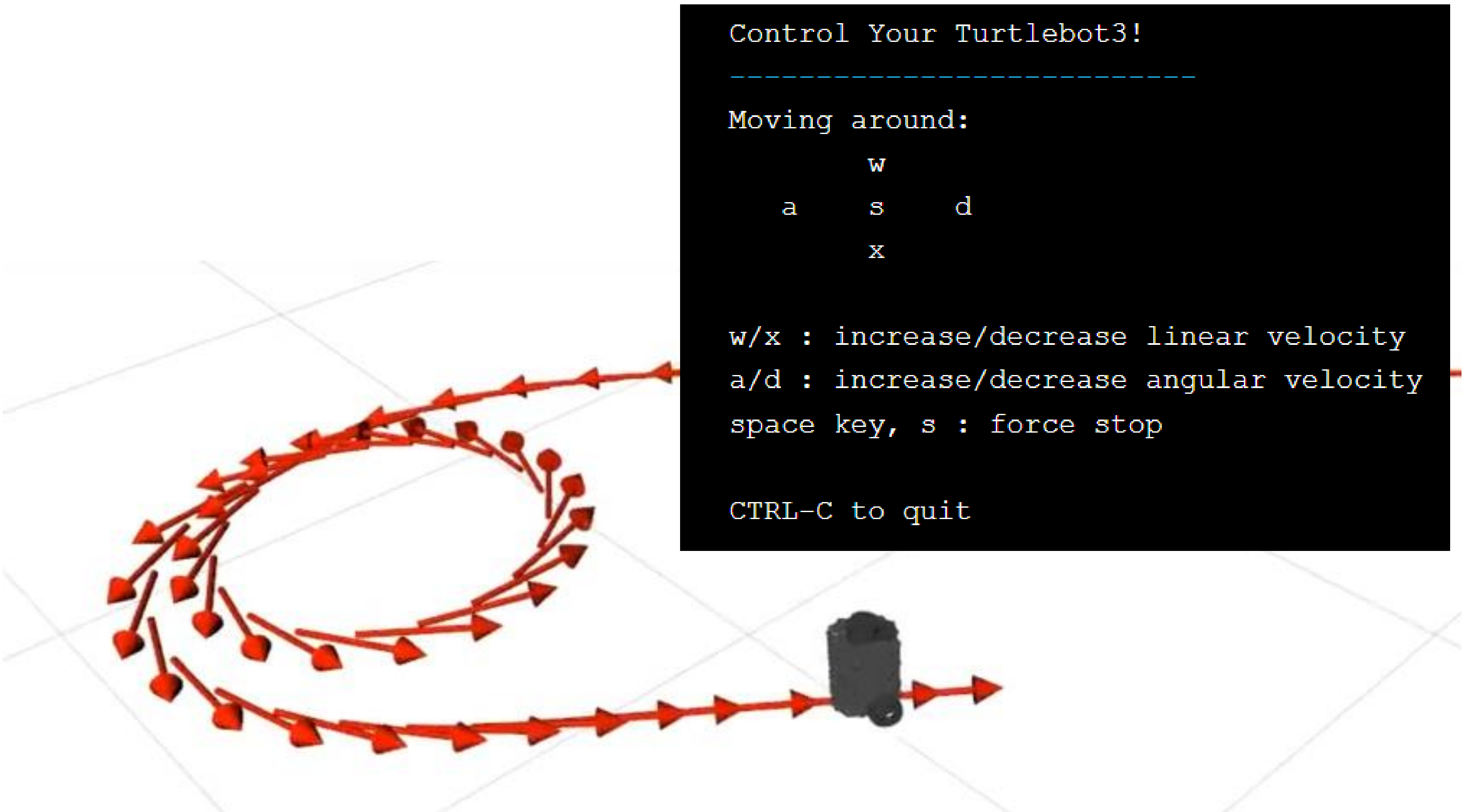
- `$ roslaunch turtlebot3_fake turtlebot3_fake.launch`

To move the TB3 around using keyboard, type in a new terminal (`Ctrl-Alt-T`):

- `$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`

(a) Manual Movements (RViz)

Using fake node in Rviz (cont):

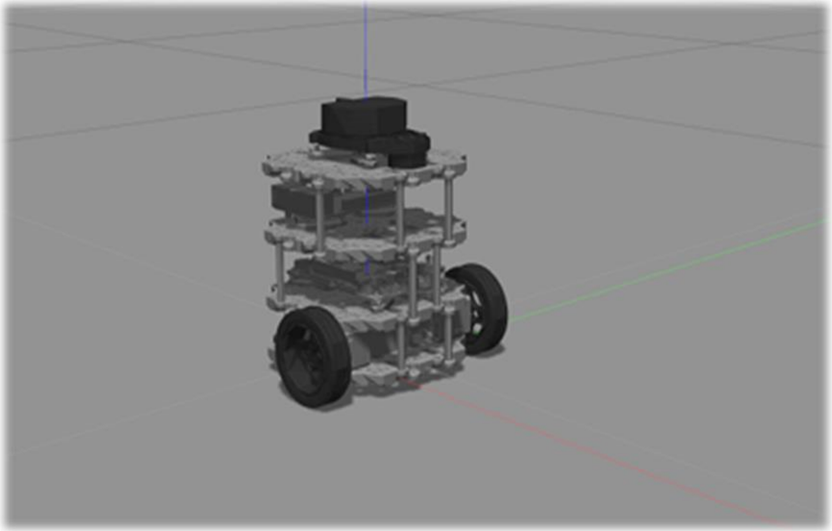


Introduction of Various Worlds

Using Gazebo:

Empty World

- $ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch

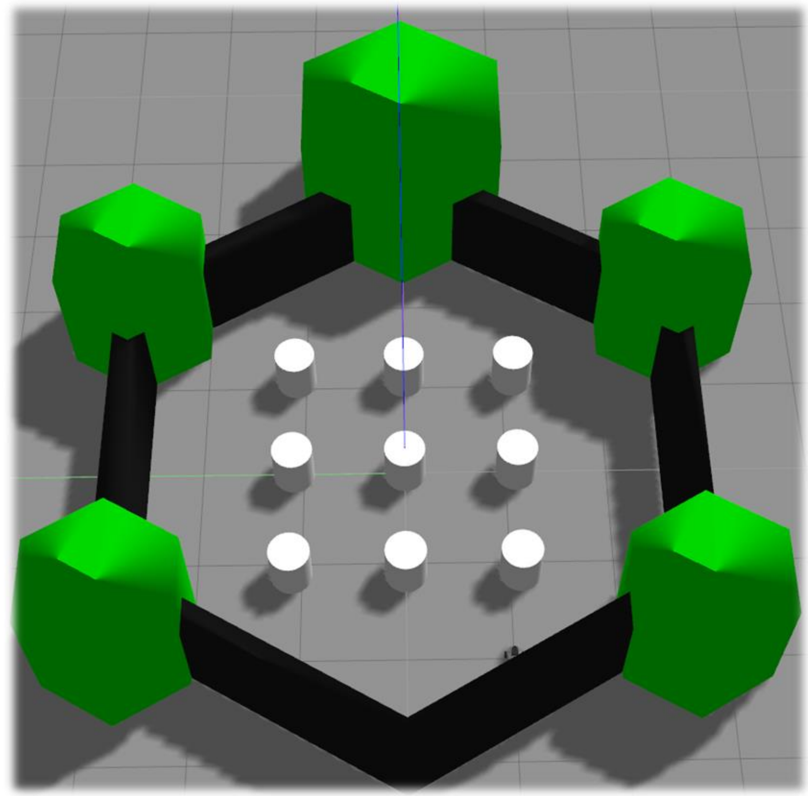


Introduction of Various Worlds

Using Gazebo:

TurtleBot3 World (Map to be mostly used for today)

- `$ roslaunch turtlebot3_gazebo turtlebot3_world.launch`



ROS Basics (RECAP)

After launching the following command:

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

We will open new terminal and try the following (one by one):

1. `$ rostopic list`
2. `$ rosservice list`
3. `$ rostopic echo /odom -n1`
4. `$ rostopic echo /imu -n1`
5. `$ rqt`
6. `$ rqt_image_view`

(a) Manual Movements (Gazebo)

Using Gazebo; while any of the worlds above is launched, type in new terminal:

- `$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`
- **You are now able to move the TB3 around using the keyboard (i.e. w, s, x, a, d)**

(b) Random Autonomous Navigation and Obstacle Avoidance

Using Gazebo, type in new terminal to launch the TurtleBot3 World:

- `$ roslaunch turtlebot3_gazebo turtlebot3_world.launch`

In new terminal, enter below command to activate autonomous navigation and obstacle avoidance:

- `$ roslaunch turtlebot3_gazebo turtlebot3_simulation.launch`
- Do not end the programme yet; see next slide

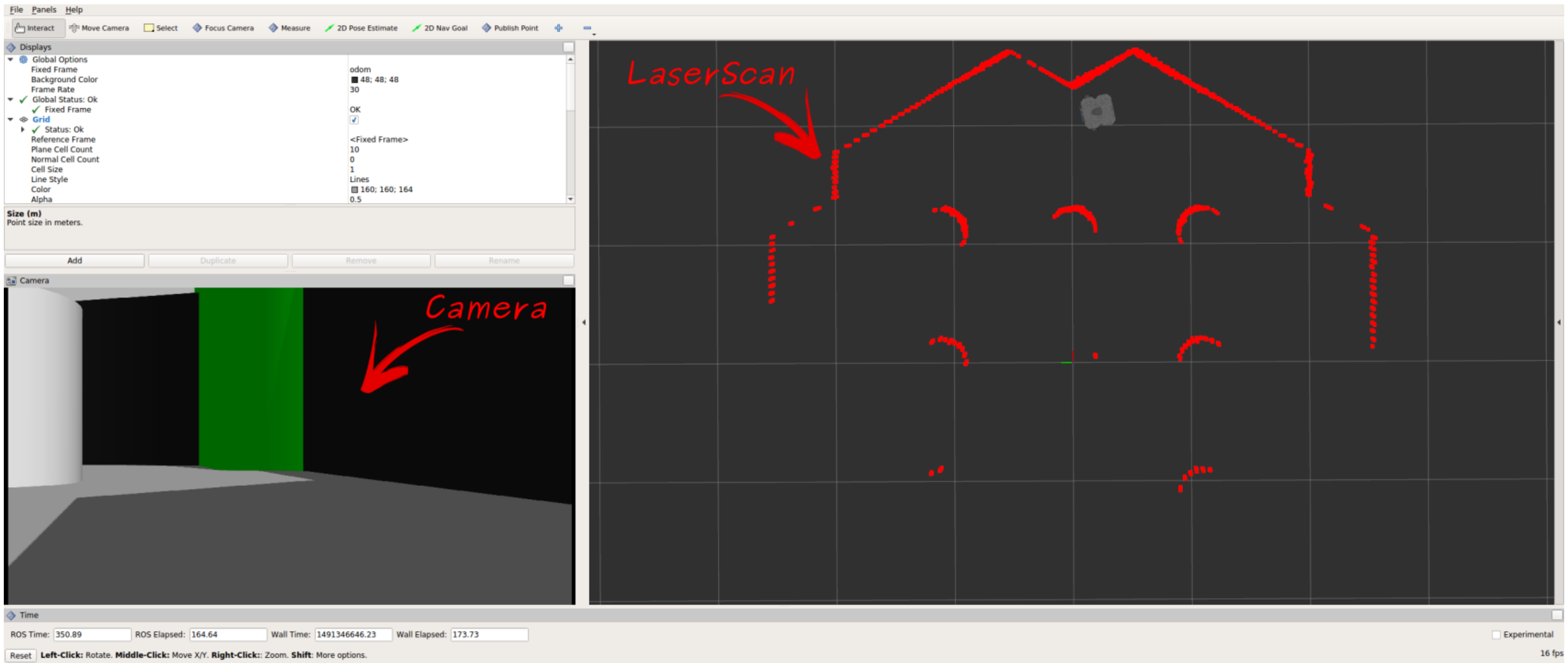
(b) Random Autonomous Navigation and Obstacle Avoidance (cont)

While the TB3 is moving autonomously around the virtual world, we can open RViz to visualize the LaserScan topic; in a new terminal, type:

- `$ roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch`

(b) Random Autonomous Navigation and Obstacle Avoidance (cont)

Your output in RViz will be as follows:



(c) SLAM and Controlled Navigation

For Virtual SLAM (one TB3), 4 Steps:

Step 1 - Launch Gazebo in TurtleBot3 World

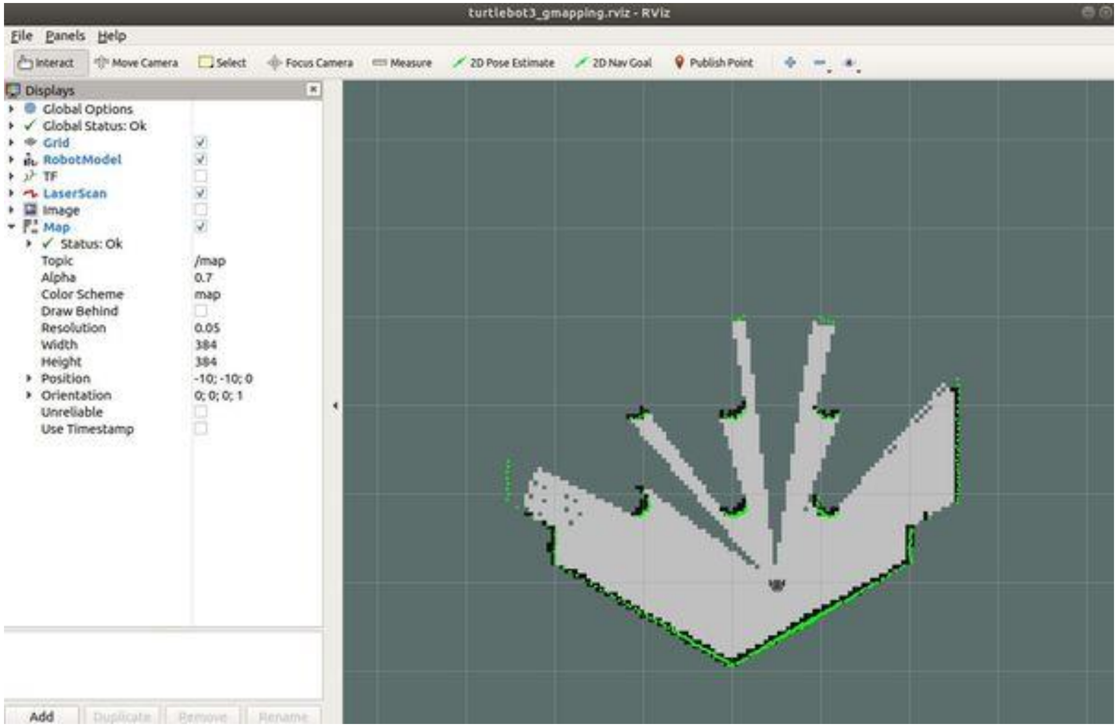
- `$ roslaunch turtlebot3_gazebo turtlebot3_world.launch`

Step 2a - Launch SLAM in a new terminal

- `$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping`
- Note that if your SLAM module don’t work, close all terminals, install it in a new terminal:
  - `$ sudo apt install ros-kinetic-slam-gmapping`
  - Restart from Step 1

(c) SLAM and Controlled Navigation

Step 2a - Launch SLAM in a new terminal (cont); the initial state will be like below:



(c) SLAM and Controlled Navigation

<u>Step 3a</u> - Activate Autonomous Navigation in a new terminal (only works for simulation!):

- `$ roslaunch turtlebot3_gazebo turtlebot3_simulation.launch`

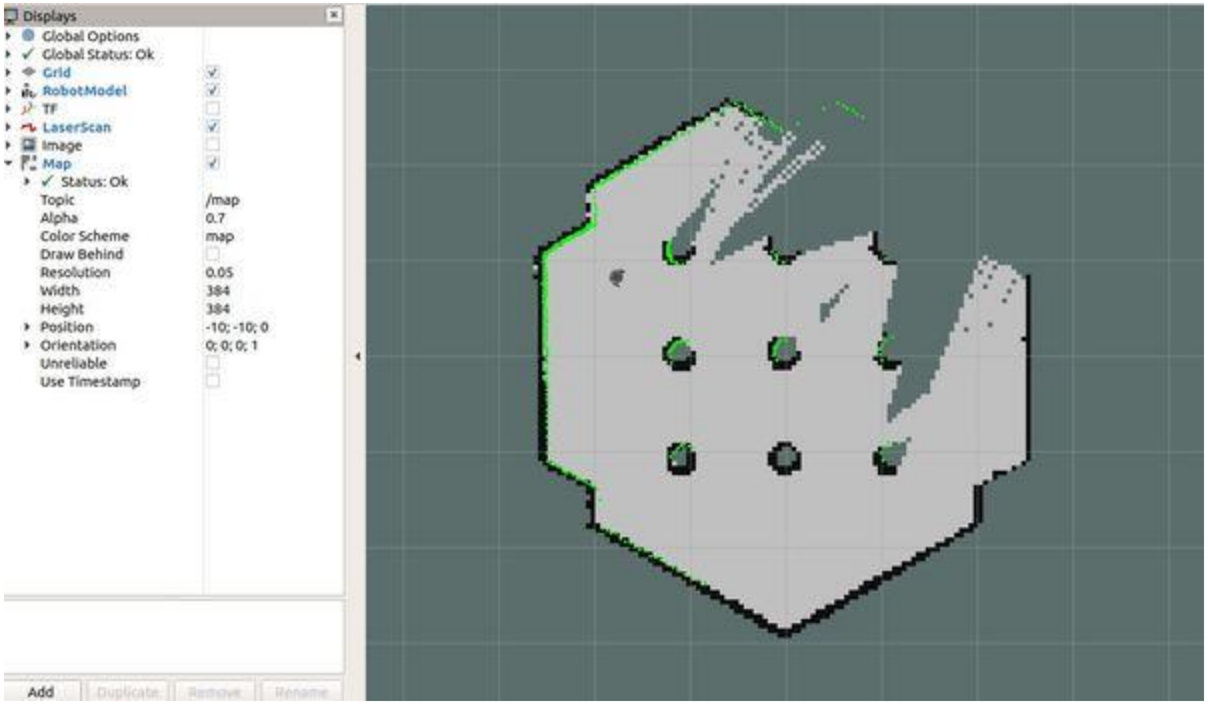
Note that you can choose to do this manually (i.e. Teleop key node as covered earlier)

<u>Step 4</u> - Save the Map once Completed in a new terminal:

- `$ rosrun map_server map_saver -f ~/TB3_WORLD`
- **You can name your map (in red)**

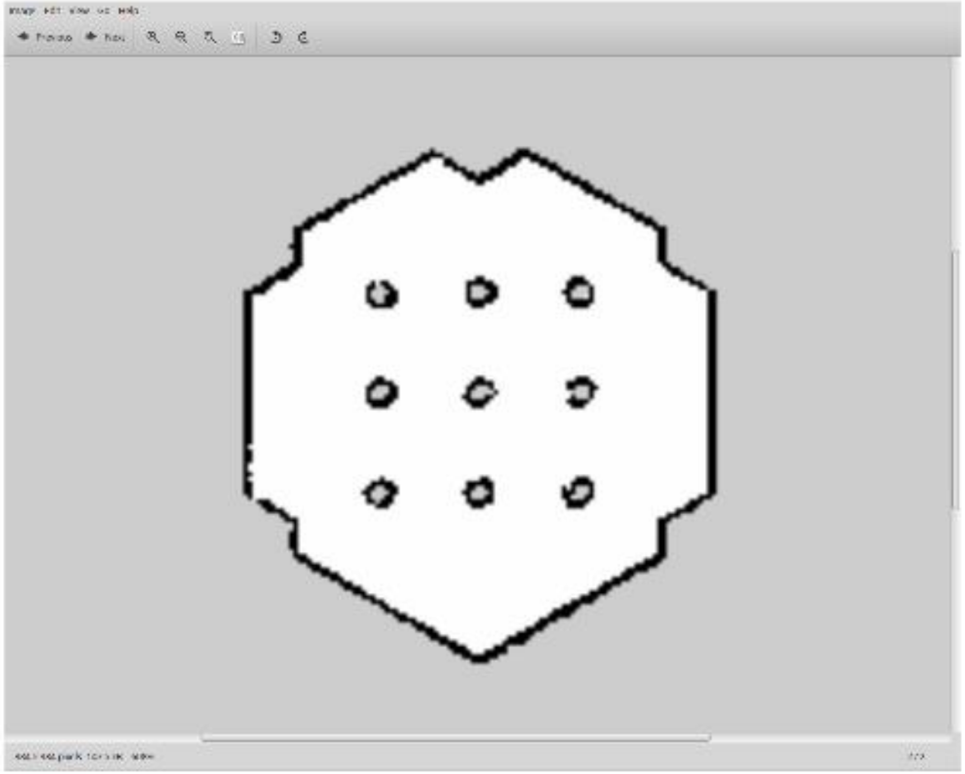
(c) SLAM and Controlled Navigation

Step 3a - Activate Autonomous Navigation OR Teleop Node in a new terminal (cont); the map will be created as the TB3 moves around:



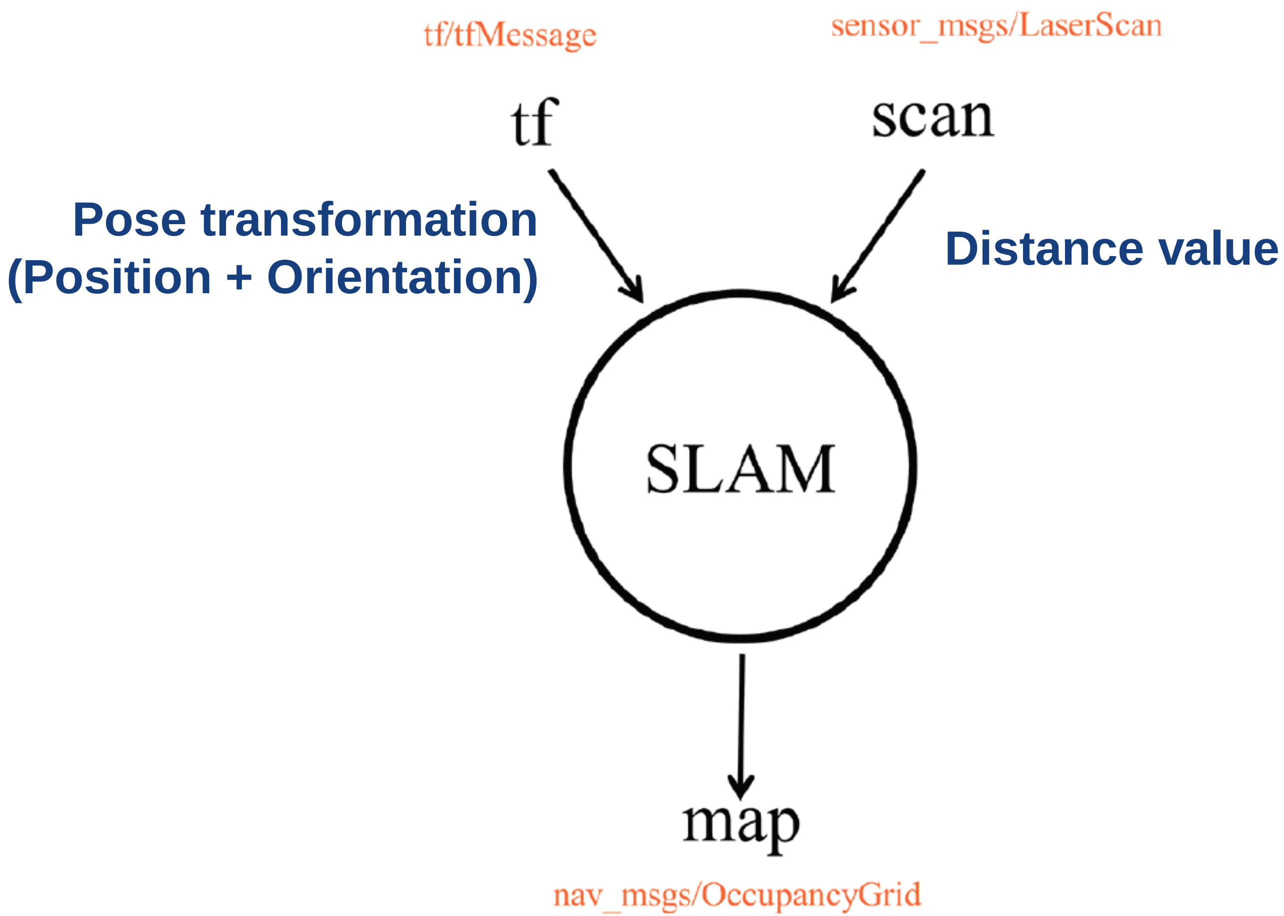
(c) SLAM and Controlled Navigation

Step 4 - Save the Map once Completed in a new terminal (cont):



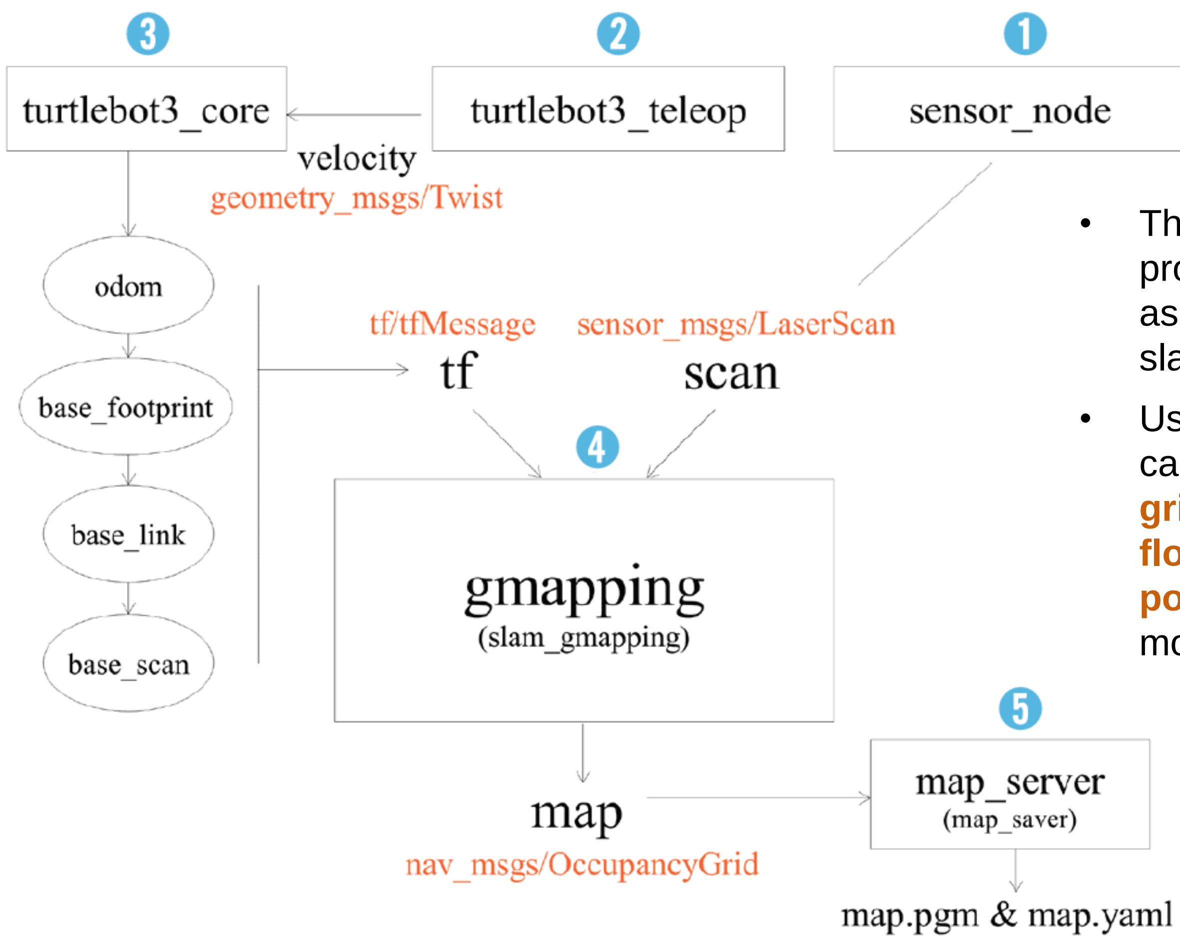
SLAM Application

Information Required in SLAM:



SLAM Application

SLAM Process for TB3:



- The gmapping package provides **laser-based SLAM**, as a ROS node called slam_gmapping
- Using slam_gmapping, you can **create a 2-D occupancy grid map (like a building floorplan) from laser and pose data** collected by a mobile robot

(c) SLAM and Controlled Navigation

For Step 2 & 3, try another SLAM function named as **Frontier Exploration** (previous function is Gmapping); you have to shut down (i.e. `clt c`) or close all terminals and restart from Step 1 first

Step 2b - Launch SLAM in a new terminal

- `$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=frontier_exploration`

Step 3b - Activate Autonomous SLAM

- Click on “Publish Point” button and click the points (you have to click the button for each point) on the map (illustrated on next slide) to indicate what is the area of exploration you want the robot to explore
- After closing the boundary box, click on “Publish Point” button and indicate anywhere within the boundary box (preferably in front of the robot) for the robot to start exploring at

Step 4 - Save the Map once Completed in a new terminal

Frontier Exploration

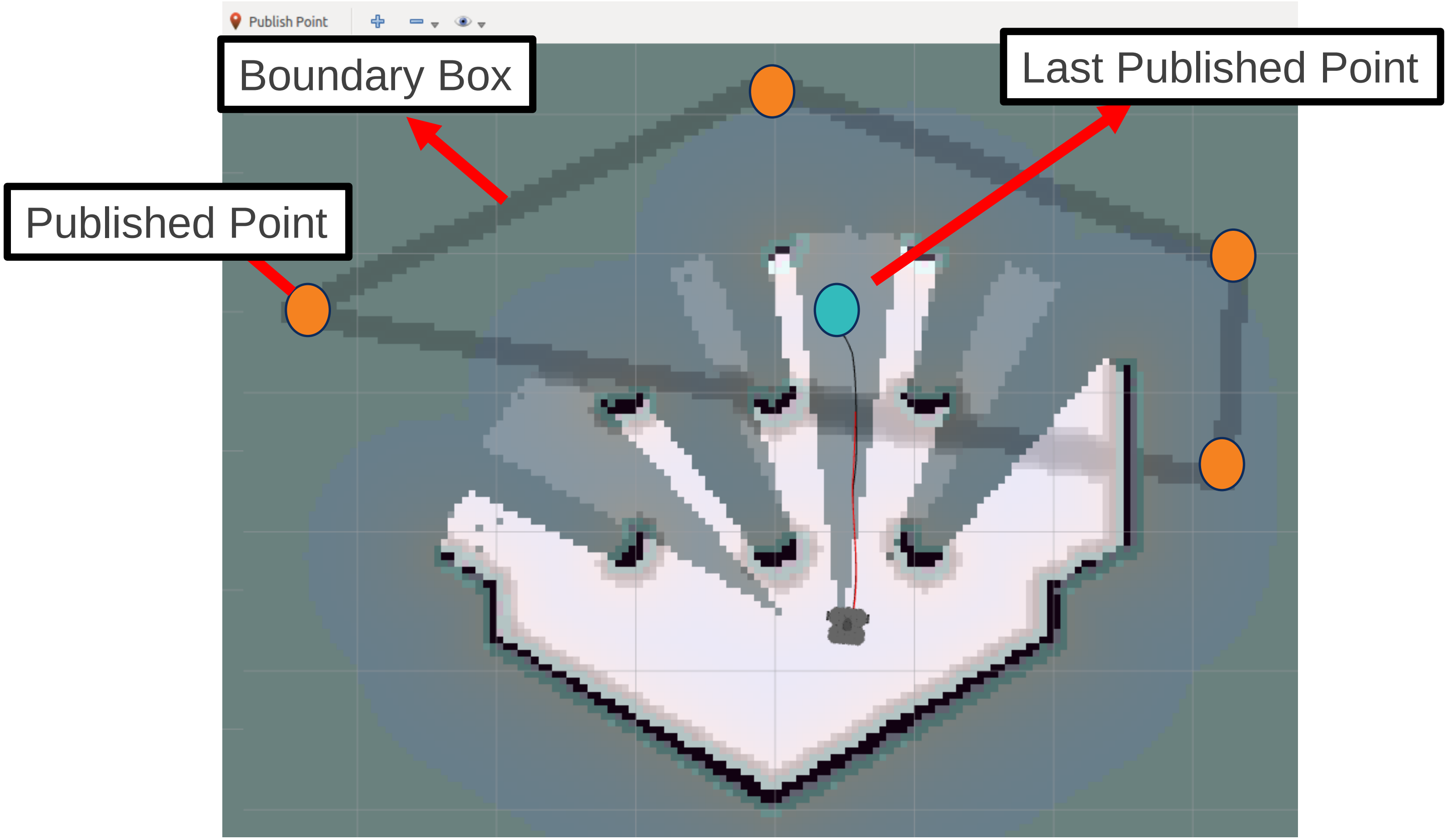
Note that if your frontier exploration cannot run, please help to install it:

- `$ sudo apt-get update`
- `$ sudo apt-get install ros-kinetic-frontier-exploration ros-kinetic-navigation-stage`

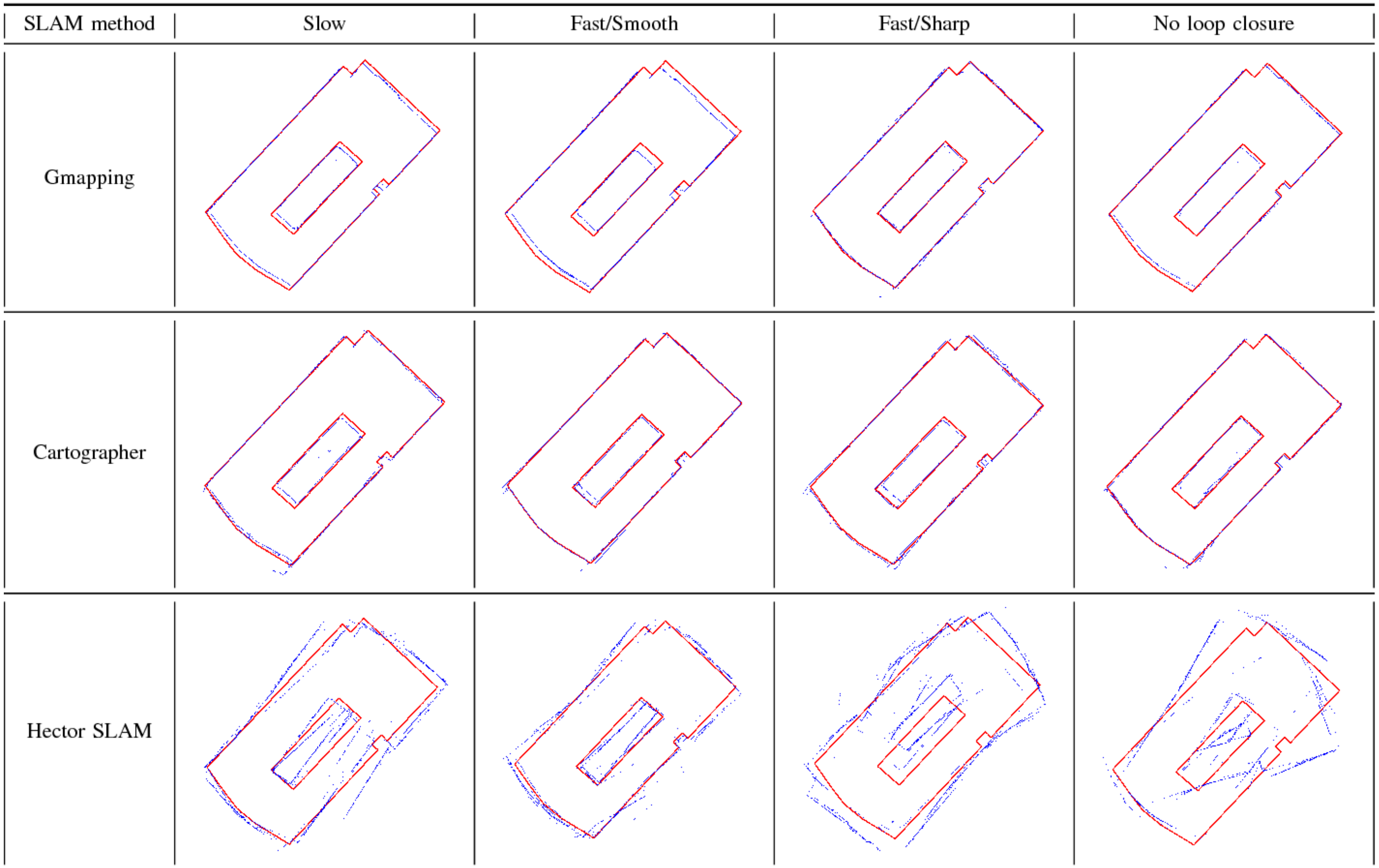
Note that **Frontier Exploration** also uses Gmapping but in a different manner

(c) SLAM and Controlled Navigation

Step 3b - Activate Autonomous SLAM (cont):



Extra: Comparisons among SLAM Methods



Map comparison constructed from SLAM Algorithms (blue colour) and Ground Truth (red colour)

Extra: Comparisons among SLAM Methods

Condition	SLAM algorithm		
	Gmapping	Cartographer	Hector SLAM
Slow ride, smooth rotations, loop closure	8.05	7.41	27.95
Fast ride with smooth rotations, loop closure	11.92	5.35	19.36
Fast ride with sharp rotations, loop closure	3.21	7.37	44.03
Without loop closure	6.11	4.97	51.67

Map Comparison (Cont): Error Calculated with ADNN (Average Distance to the Nearest Neighbor) Metrics for SLAM Methods Relative to the ground truth, in cm

ADNN = Sum of all distances from each point of the SLAM map to the nearest neighbor point on the ground truth map divided by number of occupied cells

$$\mathbf{ADNN} = \frac{\sum_{i=1}^{N} Nearest\_Neighbour(occupied\_grid\_cell(i))}{N}$$

Extra: Comparisons among SLAM Methods

TABLE 1 Popular ROS-compatible lidar and visual SLAM approaches with their supported inputs and online outputs

	Inputs							Online outputs			
	Camera				Lidar				Occupancy		Point
	Stereo	RGB-D	Multi	IMU	2D	3D	Odom	Pose	2D	3D	Cloud
GMapping					✓		✓	✓	✓		
TinySLAM					✓		✓	✓	✓		
Hector SLAM					✓			✓	✓		
ETHZASL-ICP					✓	✓	✓	✓	✓		Dense
Karto SLAM					✓		✓	✓	✓		
Lago SLAM					✓		✓	✓	✓		
Cartographer					✓	✓	✓	✓	✓		Dense
BLAM						✓		✓			Dense
SegMatch						✓					Dense
VINS-Mono				✓				✓			
ORB-SLAM2	✓	✓									
S-PTAM	✓							✓			Sparse
DVO-SLAM		✓						✓			
RGBiD-SLAM		✓									
MCPTAM	✓		✓					✓			Sparse
RGBDSLAMv2		✓					✓	✓		✓	Dense
RTAB-Map	✓	✓	✓		✓	✓	✓	✓	✓	✓	Dense

Note. ICP: iterative closest point; IMU: inertial measurement unit; RTAB-Map: real-time appearance-based mapping.

(c) SLAM and Controlled Navigation

For Virtual Controlled Navigation, 2 Steps:

Step 1 - Launch Gazebo in TurtleBot3 World

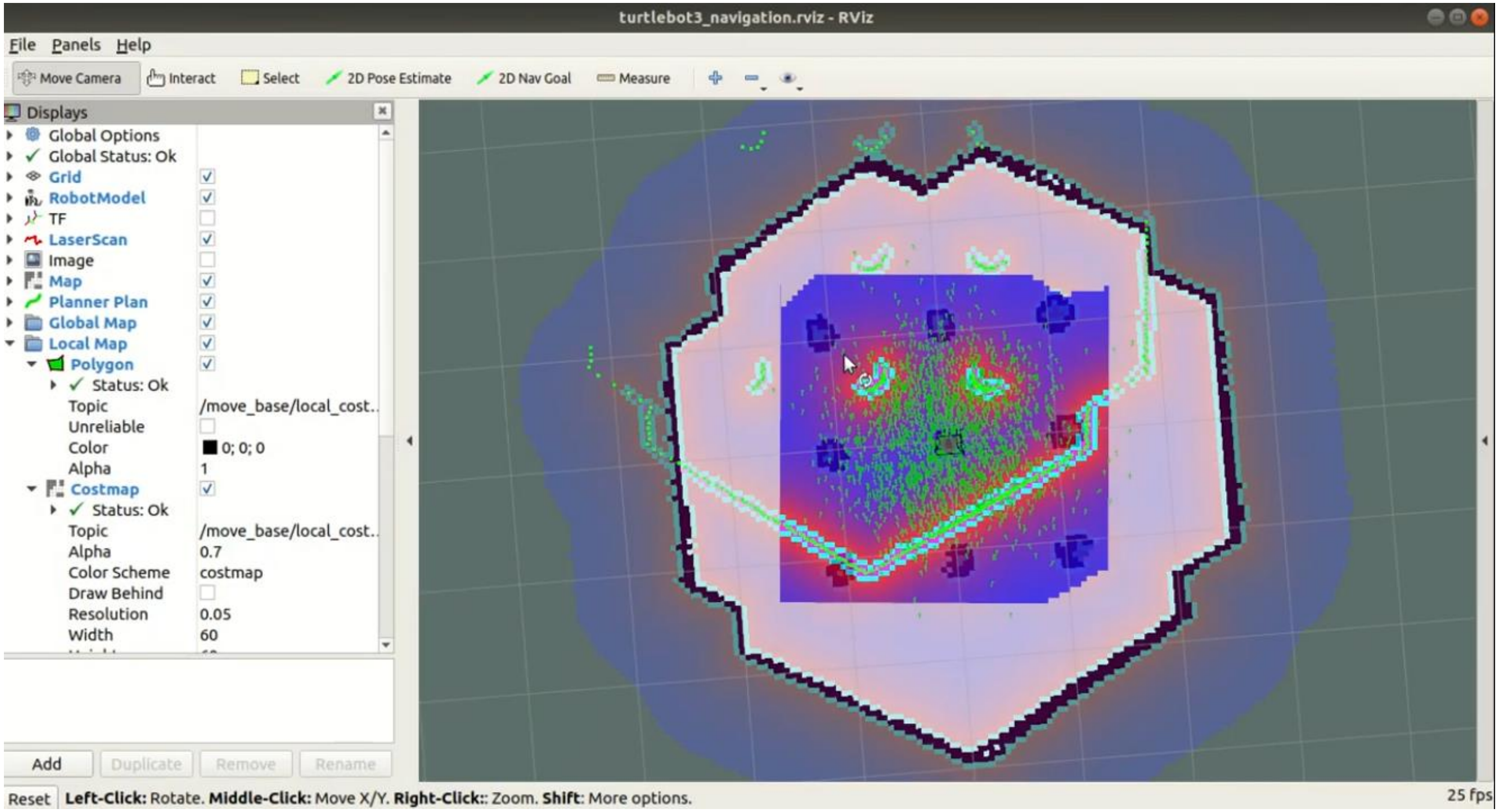
- `$ roslaunch turtlebot3_gazebo turtlebot3_world.launch`

Step 2 - Execute Controlled Navigation in a new terminal

- `$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/`**`TB3_WORLD`**`.yaml`
- Remember to change the name of your map to the correct one

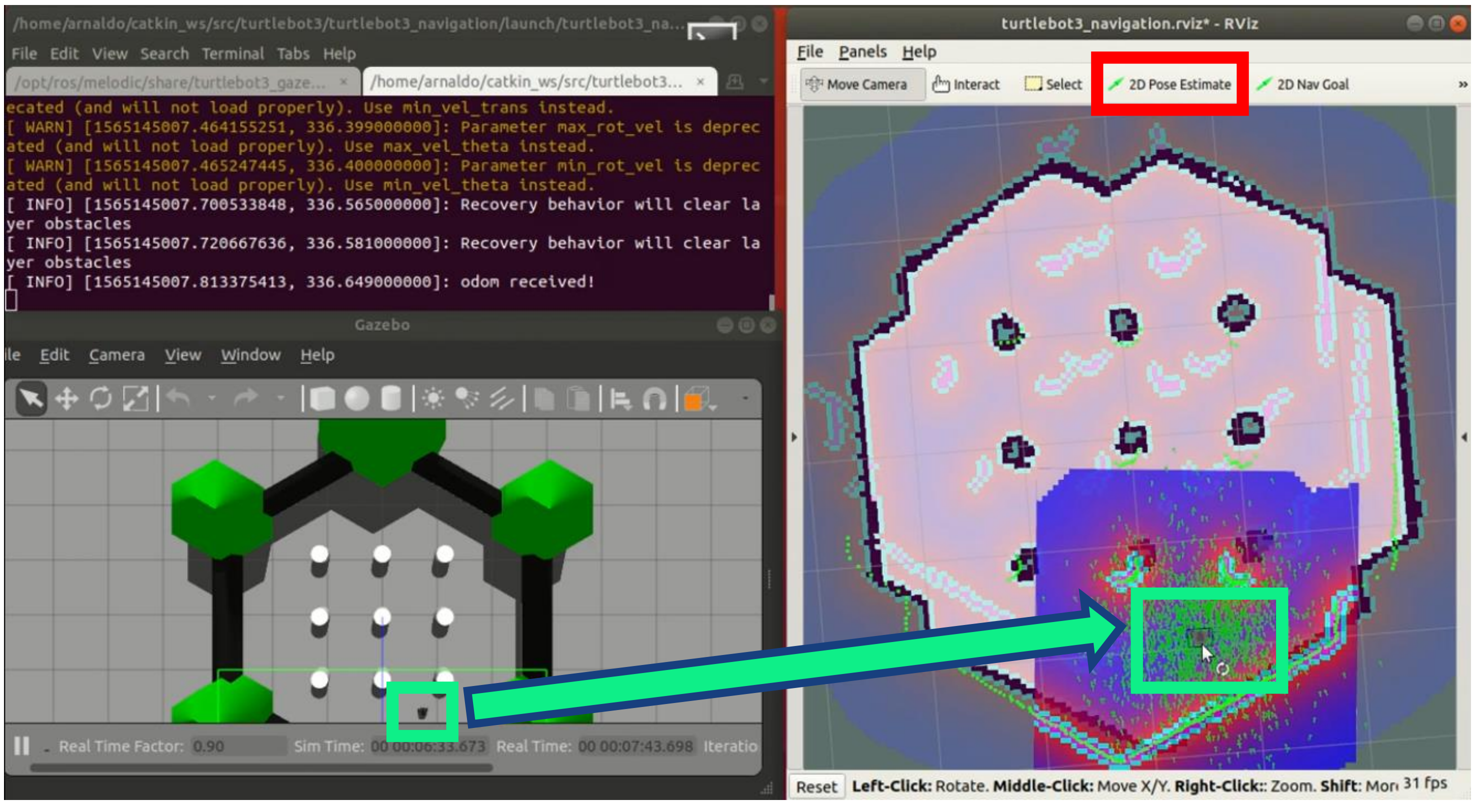
(c) SLAM and Controlled Navigation

<u>Step 2</u> - Execute Controlled Navigation in a new terminal (cont); **the RViz programme will open as follows:**

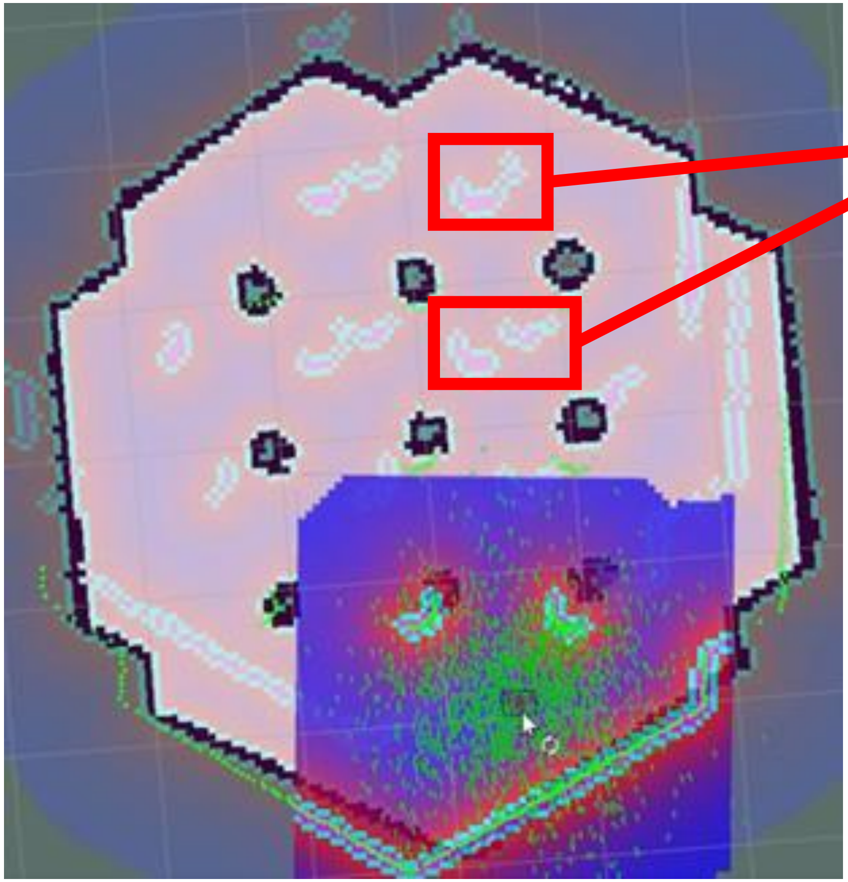


(c) SLAM and Controlled Navigation

Step 2 (cont) - **Click on "*2D Pose Estimate*" (red box) and set the correct initial position for the TB3 (right green box); refer to the left green box in Gazebo:**

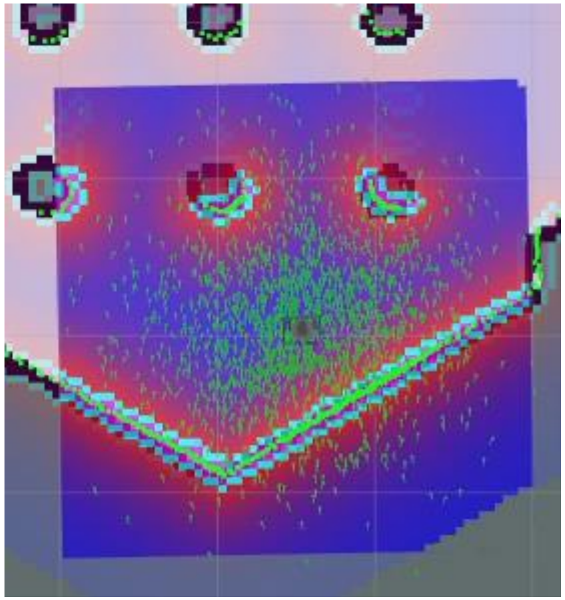


Removing Unwanted Cost Maps

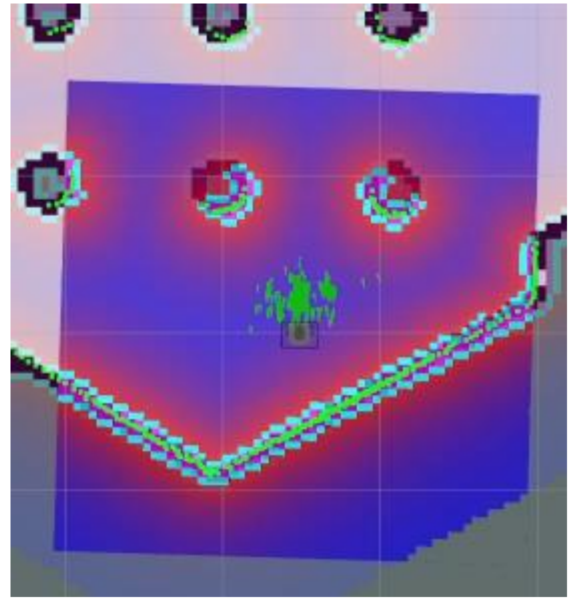


- Sometimes after you have performed the initial pose, there will be **unwanted cost maps that wrongly represent obstacles in the navigation system**
- The cause of this is usually **due to symmetrical effect** within the environment
- We **have to get rid of these unwanted cost maps** as much as possible before performing the navigation

Removing Unwanted Cost Maps



Before Action



After Action

- To remove the unwanted cost maps, **simply rotate the TB3 and/or move it back and forth a bit** to collect the surrounding environment information; to do this, **launch the teleop node to precisely locate it on the map:**

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

- This will narrow down the estimated location of the TurtleBot3 on the map which is displayed with tiny green arrows

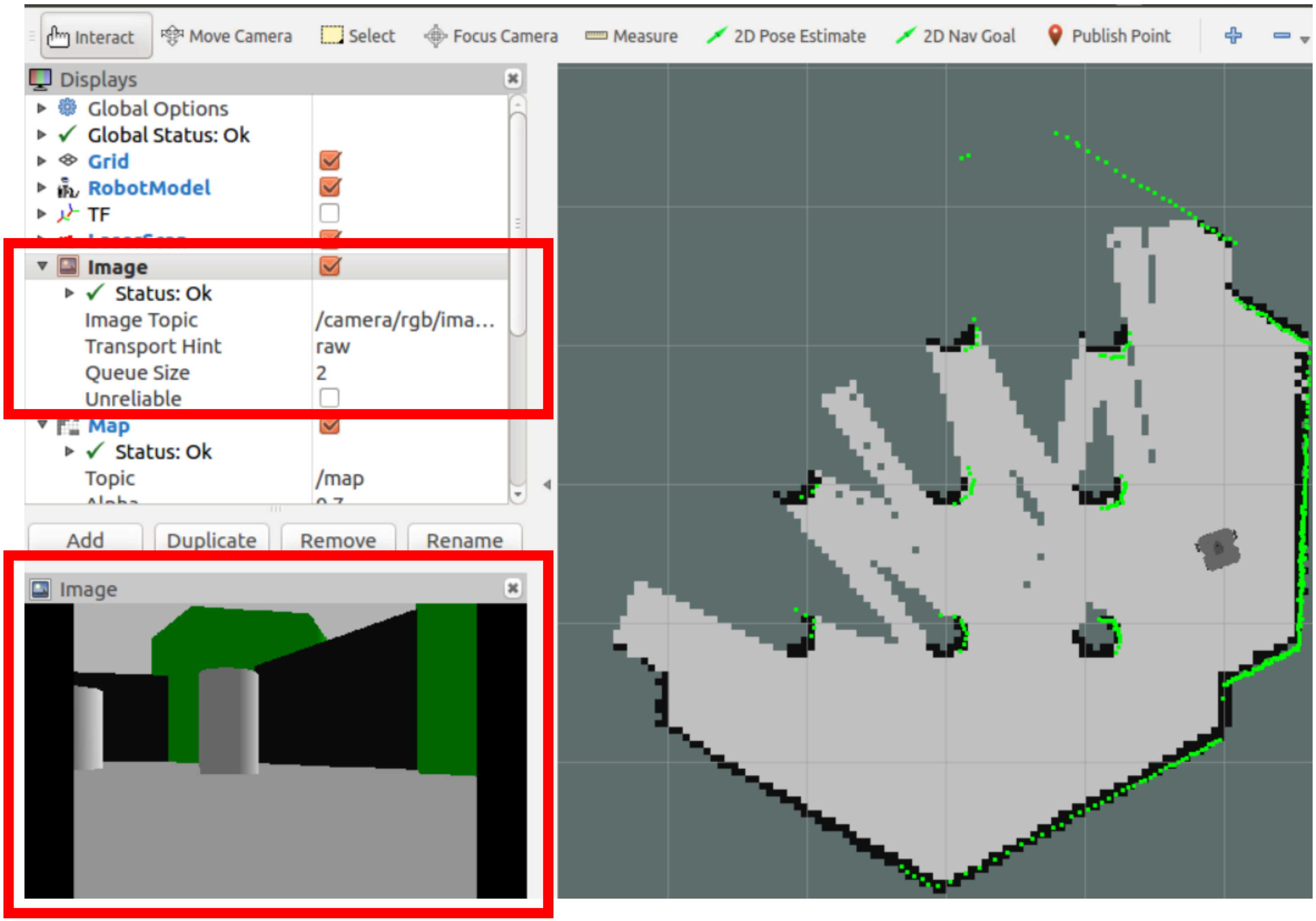
Displaying Camera in RViz

Note that **you can display the TB3's camera while RViz is launched during SLAM or NAVIGATION:**

1. While RViz is launched, go to "Displays" (left panel)
2. Expand "Image" topic at the left panel
3. Under "Image Topic", change topic to "/camera/rgb/image_raw"; you can just click on it and scroll down to find this topic and click on this option
4. Under "Transport Hint", change type to "raw"

***If you had explored enough, you will realize that your topic monitor will be able to do the same thing as well

Displaying Camera in RViz



Displaying Camera in RViz

Note that your current Gazebo version (i.e. 7.0.0) might not be able to support the image topic well. Hence, you **may need to update your Gazebo** to version 7.16.0 to enable the image topic; follow the following steps to update your Gazebo to version 7.16.0:

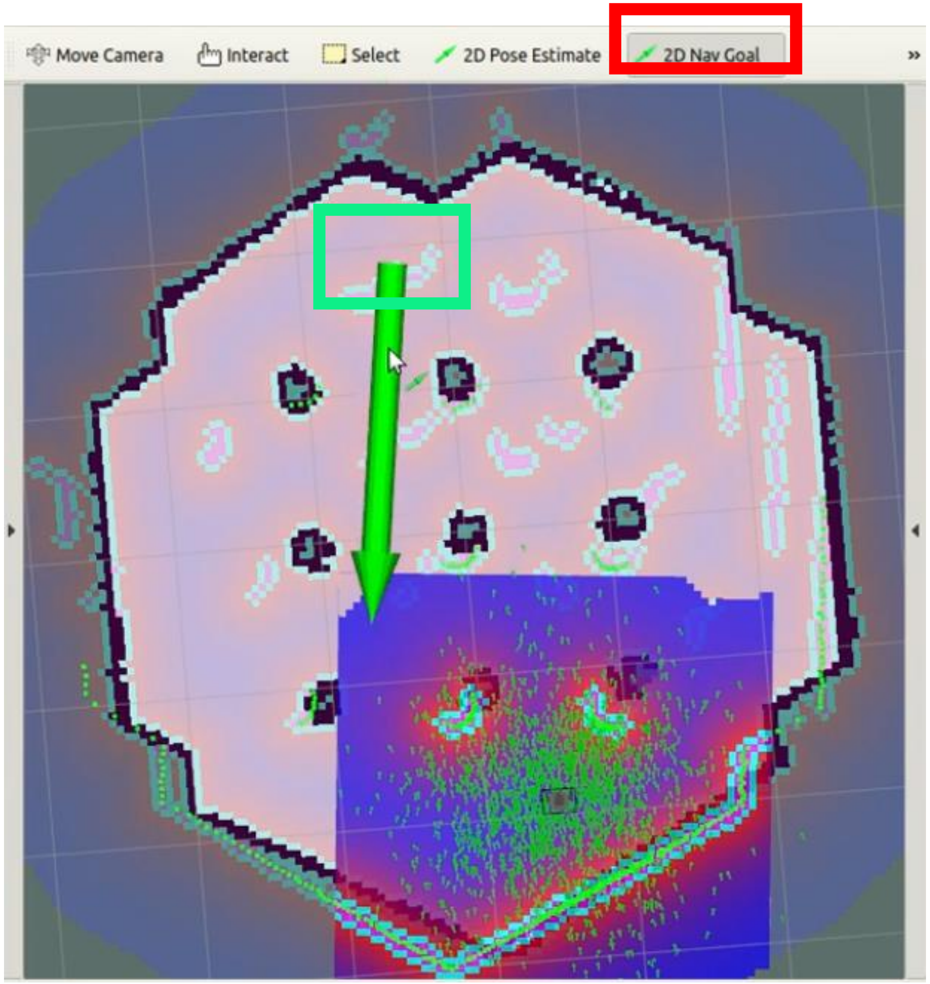
- `$ sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'`
- `$ wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -`
- `$ sudo apt-get update`
- `$ sudo apt-get install gazebo7 -y`

(c) SLAM and Controlled Navigation

Rmb to terminate any teleop node first!

Step 2 (cont)

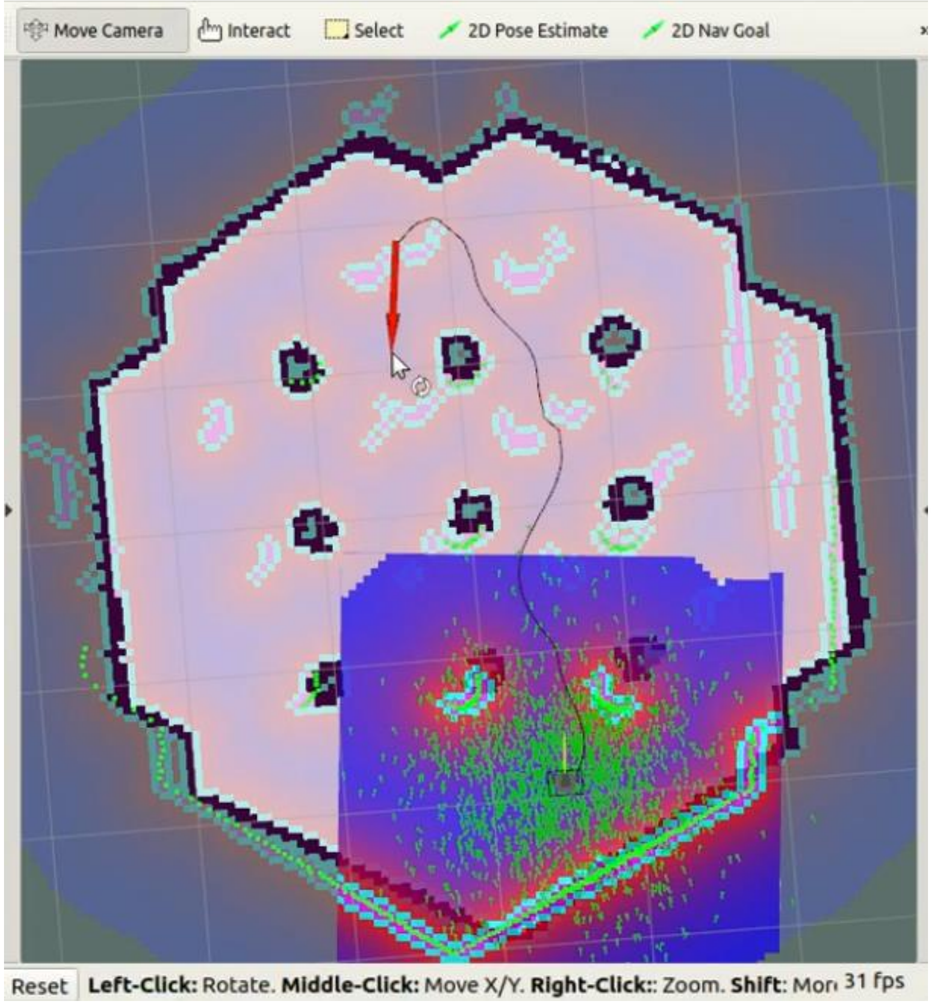
- **Click on "*2D NAV Goal*" (red box) and set the target destination for the TB3 (green box)**
- **The green arrow represent the direction of TB3 at its end state**



(c) SLAM and Controlled Navigation

Step 2 (cont)

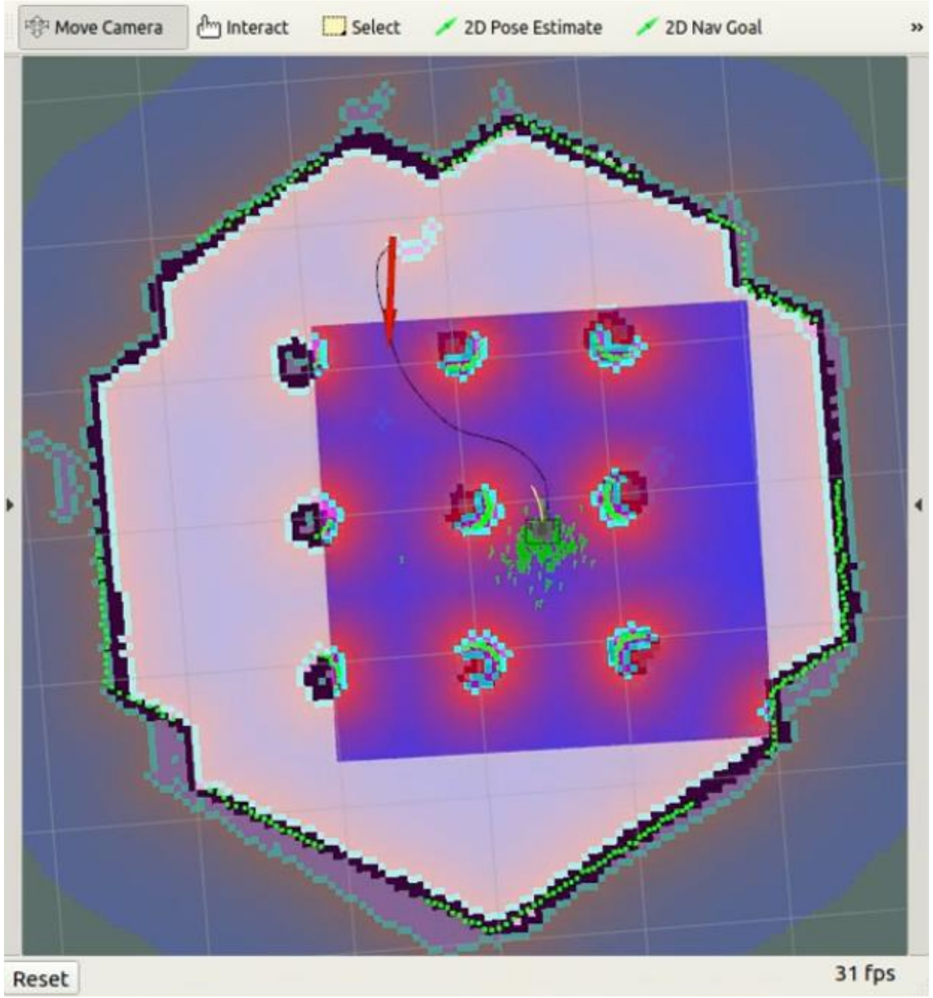
- **The tool will create a path for the TB3 to follow to reach to the end goal**
- **The bot will follow the created path to its destination**



(c) SLAM and Controlled Navigation

Step 2 (cont)

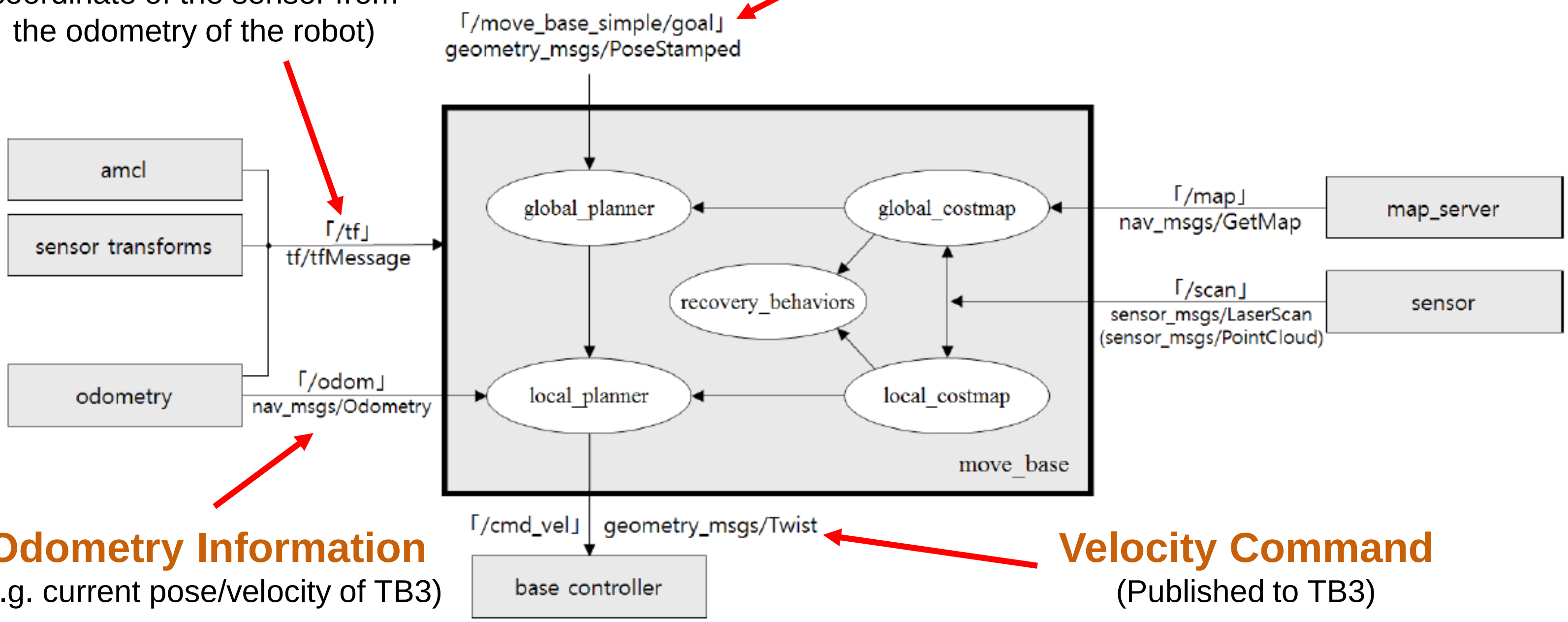
- **The path may be adjusted automatically to its optimum during the TB3's journey**



Navigation Application

Relative Coordinate Transformation
(Describes the relative x, y, z coordinate of the sensor from the odometry of the robot)

Target Coordinates
(Determined by user; two-dimensional coordinates and orientations)



Odometry Information
(e.g. current pose/velocity of TB3)

Velocity Command
(Published to TB3)

Using Python to Control TB3

1. Moving TB3 in circles
2. Moving TB3 according to a fixed distance using position feedback
3. Getting laser data
4. Avoiding obstacle in front of TB3
5. Initial pose estimate for navigation
6. Path planning for navigation

Moving TB3 in circles

- Open terminal and type:

```
$ roslaunch turtlebot3_gazebo
turtlebot3_empty_world.launch
```

- Open new terminal and type:

```
$ rosrun autonomous trajectory.py
```

- Observe in Gazebo that the TB3 will continuously move in circles
- To change the destination point along the x-axis, change the value for the following variable:

```
move.linear.x
```

- To change the direction of movement, change the value for the following variable (in radians):

```
move.angular.z
```

Moving TB3 in circles

trajectory.py

```
#!/usr/bin/env python

import rospy                                    #import the python library for ROS
from geometry_msgs.msg import Twist             #import the twist message from the std_msgs package

def talker():
        rospy.init_node('vel_publisher')        #Initiate a Node called 'vel_publisher'
        pub = rospy.Publisher('cmd_vel', Twist, queue_size=10)  #Create a Publisher object
        move = Twist()                          #Create a var named move of type Twist
        rate = rospy.Rate(1)                    #Set a publish rate of 0.5 Hz
        while not rospy.is_shutdown():
                move.linear.x = 1
                move.angular.z = 1
                pub.publish(move)
                rate.sleep()

if __name__== '__main__':
        try:
                talker()
        except rospy.ROSInterruptException:
                pass
```

Moving TB3 according to a fixed distance using position feedback

- Open terminal and type:

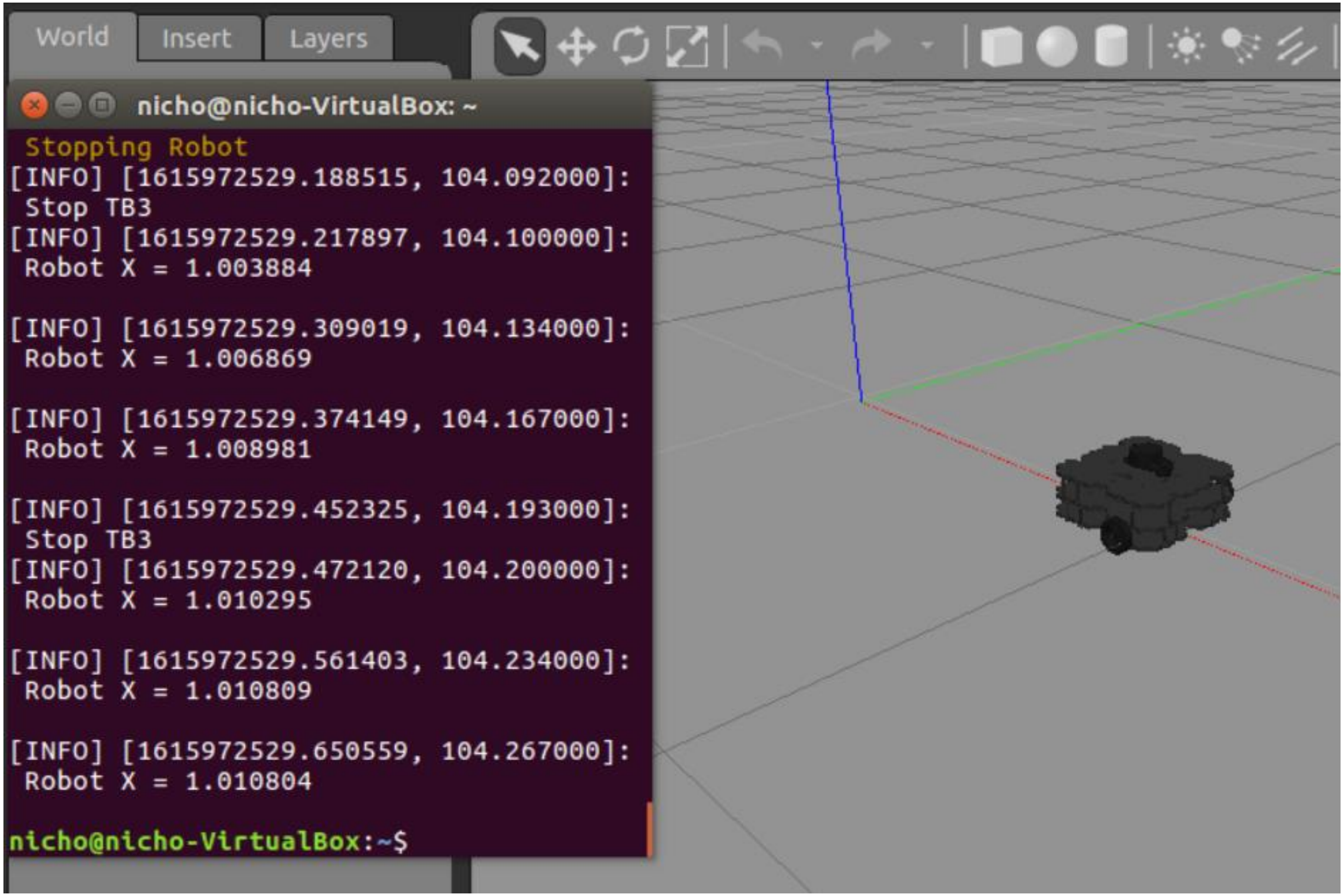
```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

- Open new terminal and type:

```
$ rosrun autonomous move_turtlebot_distance.py 0.1 0.0 1.0
```

- Inputs in the following order: [linear.x, angular.z, distance]; you can choose the inputs in your command line
- Observe in Gazebo that the TB3 will move along the x-axis at linear velocity 0.1m/s (i.e. `lin_vel = 0.1`) to the position when x = 1m (i.e. `distance = 1.0`)

Moving TB3 according to a fixed distance using position feedback



Moving TB3 according to a fixed distance using position feedback

move_turtlebot_distance.py

```
# Function to move turtle: Linear and angular velocities, and distance are arguments
def move_turtle(lin_vel, ang_vel, distance):
        global pub
        global rate
        global robot_x
        rospy.init_node('move_turtlebot_distance')      #Initiate a Node

        # The /turtle1/cmd_vel is the topic in which we have to send Twist messages
        pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10) #Create a Publisher object
        rospy.Subscriber('/odom', Odometry, pose_callback) #Creating new subscriber

        vel = Twist()                           #Create a var named "vel" of type Twist
        rate = rospy.Rate(10)                   #Set a publish rate of 10 Hz

        rospy.on_shutdown(shutdown)

        while not rospy.is_shutdown():

                # Adding linear and angular velocity to the message
                vel.linear.x = lin_vel
                vel.linear.y = 0
                vel.linear.z = 0

                vel.angular.x = 0
                vel.angular.y = 0
                vel.angular.z = ang_vel

                #rospy.loginfo("Linear Vel = %f: Angular Vel = %f", lin_vel, ang_vel)

                # Checking the robot distance is greater than the desired distance
                # If it is greater, stop the node
                if(robot_x >= distance):
                        rospy.loginfo("Robot Reached Destination")
                        rospy.logwarn("Stopping Robot")
                        shutdown()
                        break
```

Getting laser data

- Open terminal and type:

```
$ roslaunch turtlebot3_gazebo
turtlebot3_world.launch
```

- Open new terminal and type:

```
$ rosrun autonomous laser_data.py
```

- Observe in the terminal of the received laser data in 5 directions
- You may activate the teleop node to move the TB3 around and observe the change in collected laser data:

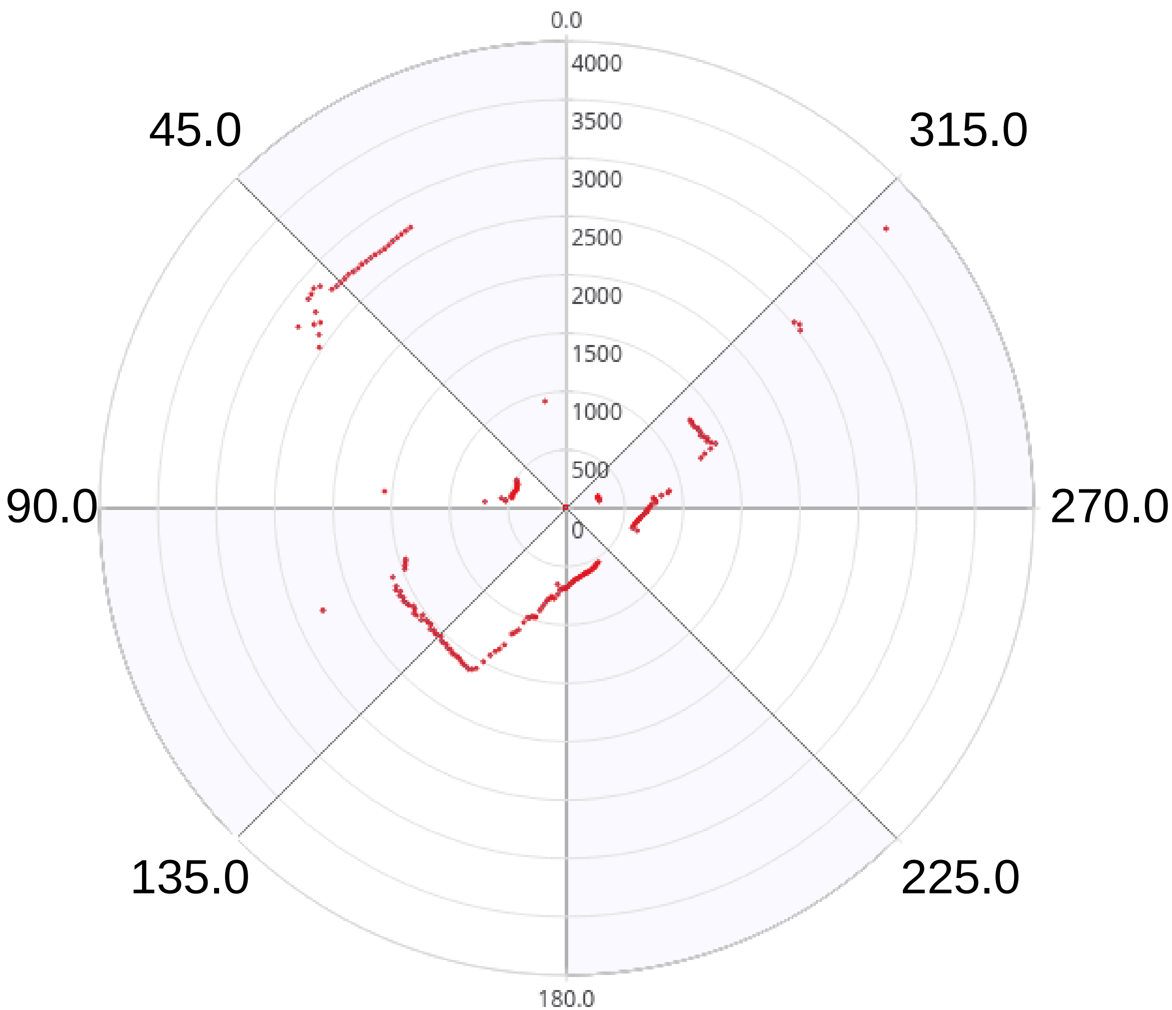
```
$ roslaunch turtlebot3_teleop
turtlebot3_teleop_key.launch
```

- To change the direction of collection for the laser data, change the value for the following variable:

```
msg.ranges[X]
```

Getting laser data

Directions Map of LDS:



Getting laser data

laser_data.py

```
#!/usr/bin/env python

import rospy                                      #import the python library for ROS
from sensor_msgs.msg import LaserScan             #import the laserscan message from the std_msgs package

def callback(msg):                                #define a function called 'callback' that receives a parameter named 'msg'
        print('=========================================')
        print('s1 [0]')                           #value for front-direction laser beam
        print msg.ranges[0]                       #print the distance to any obstacle in front of the robot. The sensor
returns a vector of 359 values, with 0 being in front of the TB3

        print('s2 [90]')
        print msg.ranges[90]

        print('s3 [180]')
        print msg.ranges[180]

        print('s4 [270]')
        print msg.ranges[270]

        print('s5 [359]')
        print msg.ranges[359]

rospy.init_node('laser_data')                     # Initiate a Node called 'laser_data'
sub = rospy.Subscriber('scan', LaserScan, callback) #Create a Subsriber to the laser/scan topic

rospy.spin()
```

Avoiding obstacle

- Open terminal and type:

```
$ roslaunch turtlebot3_gazebo
turtlebot3_world.launch
```

- Open new terminal and type:

```
$ rosrun autonomous avoid_obstacle.py
```

- Observe in Gazebo that the TB3 will move in a straight line until it reaches an obstacle that is less than 0.5m in front of it
- To change the destination point along the x-axis, change the value for the following variable:

```
move.linear.x
```

- To change the direction of movement, change the value for the following variable (in radians):

```
move.angular.z
```

Avoiding obstacle

avoid_obstacle.py

```
#!/usr/bin/env python

import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

def callback(msg):
        print('=========================================')
        print('s1 [0]')
        print msg.ranges[0]

        print('s2 [90]')
        print msg.ranges[90]

        print('s3 [180]')
        print msg.ranges[180]

        print('s4 [270]')
        print msg.ranges[270]

        print('s5 [359]')
        print msg.ranges[359]

        #If obstacle is at least 0.5m in front of the TB3, the TB3 will move forward
        if msg.ranges[0] > 0.5:
                move.linear.x = 0.5
                move.angular.z = 0.0
        else:
                move.linear.x = 0.0
                move.angular.z = 0.0

        pub.publish(move)

rospy.init_node('obstacle_avoidance')                  # Initiate a Node called 'obstacle_avoidance'
sub = rospy.Subscriber('/scan', LaserScan, callback)
pub = rospy.Publisher('/cmd_vel', Twist)
move = Twist()

rospy.spin()
```

Initial pose estimate (using only rosrun)

- Open terminal and type:

```
$ roslaunch turtlebot3_gazebo
turtlebot3_world.launch
```

- Open new terminal and type:

```
$ roslaunch turtlebot3_navigation
turtlebot3_navigation.launch
map_file:=$HOME/TB3_WORLD.yaml
```

Ensure you still have the map that you have saved previously in the HOME directory

- After RViz is launched, open new terminal and type:

```
$ rosrun autonomous initial_pose.py
```

- Observe in the RViz application that the TB3's pose estimate will be automatically executed

Initial pose estimate

initial_pose.py

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import PoseWithCovarianceStamped
from tf.transformations import quaternion_from_euler

rospy.init_node('init_pos')
pub = rospy.Publisher('/initialpose', PoseWithCovarianceStamped, queue_size = 10)
rospy.sleep(3)
checkpoint = PoseWithCovarianceStamped()

# Use $ rostopic echo /odom -n1 to obtain poses (both position and orientation)

checkpoint.pose.pose.position.x = -1.99968900545
checkpoint.pose.pose.position.y = -0.499122759107
checkpoint.pose.pose.position.z = -0.0010073990697

[x,y,z,w]=quaternion_from_euler(0.0,0.0,0.0)
checkpoint.pose.pose.orientation.x = -6.03836998093e-06
checkpoint.pose.pose.orientation.y = 0.00158963960308
checkpoint.pose.pose.orientation.z = 0.00275331003065
checkpoint.pose.pose.orientation.w = 0.999994946134

print checkpoint
pub.publish(checkpoint)
```

Initial pose estimate (using roslaunch No. 1)

Using roslaunch to reduce steps:

- Open terminal and type:

```
$ roslaunch turtlebot3_gazebo
turtlebot3_world.launch
```

- **Copy and paste given launch files in your “autonomous” package (i.e. initial_pose.launch, initial_pose2.launch)**; you may create a launch folder to place all your launch files
- Open new terminal and type:

```
$ roslaunch autonomous initial_pose.launch
map_file:=$HOME/TB3_WORLD.yaml
```

- Observe in the RViz application that the TB3’s pose estimate will be automatically executed

Initial pose estimate

initial_pose.launch

initial_pose.launch (~/Desktop/autonomous/launch) - gedit

Open Save

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="false"/>

  <!-- Turtlebot3 -->
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch">
    <arg name="model" value="$(arg model)" />
  </include>

  <!-- Map server -->
  <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)"/>

  <!-- AMCL -->
  <include file="$(find turtlebot3_navigation)/launch/amcl.launch"/>

  <!-- move_base -->
  <include file="$(find turtlebot3_navigation)/launch/move_base.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation.rviz"/>
  </group>

  <!-- Initial Pose Node -->
  <node pkg="autonomous" name="init_pos" type="initial_pose.py" args="$(arg map_file)"/>

</launch>
```

Initial pose estimate (using roslaunch No. 2)

Subscribing `/odom` data for initial pose positions

- Open terminal and type:

```
$ roslaunch turtlebot3_gazebo
turtlebot3_world.launch
```

- Open new terminal and type:

```
$ roslaunch autonomous initial_pose2.launch
map_file:=$HOME/TB3_WORLD.yaml
```

- This launch file involves the following node: `initial_pose2.py`
- Observe in the RViz application that the TB3's pose estimate will be automatically executed

Initial pose estimate

initial_pose2.launch

initial_pose2.launch (~/Desktop/autonomous/launch) - gedit

Open | Save

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="false"/>

  <!-- Turtlebot3 -->
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch">
    <arg name="model" value="$(arg model)" />
  </include>

  <!-- Map server -->
  <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)"/>

  <!-- AMCL -->
  <include file="$(find turtlebot3_navigation)/launch/amcl.launch"/>

  <!-- move_base -->
  <include file="$(find turtlebot3_navigation)/launch/move_base.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation.rviz"/>
  </group>

  <!-- Initial Pose Node with Automatic Odom Data Collection -->
  <node pkg="autonomous" name="init_pos" type="initial_pose2.py" args="$(arg map_file)"/>

</launch>
```

Initial pose estimate

initial_pose2.py

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import PoseWithCovarianceStamped
from tf.transformations import quaternion_from_euler
from nav_msgs.msg import Odometry

def callback(msg):                                    #define a function called 'callback'
parameter named 'msg'
        print msg.pose.pose

        global px, py, pz, ox, oy, oz, ow
        px = msg.pose.pose.position.x
        py = msg.pose.pose.position.y
        pz = msg.pose.pose.position.z

        ox = msg.pose.pose.orientation.x
        oy = msg.pose.pose.orientation.y
        oz = msg.pose.pose.orientation.z
        ow = msg.pose.pose.orientation.w

rospy.init_node('init_pos')
odom_sub = rospy.Subscriber("/odom", Odometry, callback)
pub = rospy.Publisher('/initialpose', PoseWithCovarianceStamped, queue_size = 10)
rospy.sleep(3)
checkpoint = PoseWithCovarianceStamped()

checkpoint.pose.pose.position.x = px
checkpoint.pose.pose.position.y = py
checkpoint.pose.pose.position.z = pz

[x,y,z,w]=quaternion_from_euler(0.0,0.0,0.0)
checkpoint.pose.pose.orientation.x = ox
checkpoint.pose.pose.orientation.y = oy
checkpoint.pose.pose.orientation.z = oz
checkpoint.pose.pose.orientation.w = ow

print checkpoint
pub.publish(checkpoint)
```

Path planning

- Open terminal and type:

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

- Open new terminal and type:

```
$ roslaunch turtlebot3_navigation
turtlebot3_navigation.launch
map_file:=$HOME/TB3_WORLD.yaml
```

Ensure you still have the map that you have saved previously in the HOME directory

- After RViz is launched, initiate the pose estimate manually or via python file execution
- Open new terminal and type:

```
$ rosrun autonomous path_planning.py
```

- Observe in the RViz application and/or Gazebo that the TB3 will first move to the 1st waypoint, later move to the 2nd waypoint, and back to the 1st; this behaviour will be looped until the file stops running (`clt c`)

Path planning

path_planning.py

```python
#!/usr/bin/env python

import rospy
import os,sys
import actionlib

# move_base is the package that takes goals for navigation
# there are different implemenetations with a common interface
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

# You need to know the coordinates that the map you are working
# in is. I estimated these numbers using the turtlebot3_world
# map from Turtlebot3. The center of the map is (0.0, 0.0, 0.0); each grid is 1m.

#The first waypoint array is x,y,z location.
#The second one is a "quaternion" defining an orientation.
# Quaternions are a different mathematical represetnation
#for "euler angles", yaw, pitch and roll.

#path planning sequences (loop phase)
waypoints = [
    [ (-0.5, 0.0, 0.0),
      (0.0, 0.0, 0.0, 1.0)],
    [ (0.0, 2.0, 0.0),
      (0.0, 0.0, 0.0, 1.0)]
]

def goal_pose(pose):
        goal_pose = MoveBaseGoal()
        goal_pose.target_pose.header.frame_id = 'map'
        goal_pose.target_pose.pose.position.x = pose[0][0]
        goal_pose.target_pose.pose.position.y = pose[0][1]
        goal_pose.target_pose.pose.position.z = pose[0][2]
        goal_pose.target_pose.pose.orientation.x = pose[1][0]
        goal_pose.target_pose.pose.orientation.y = pose[1][1]
        goal_pose.target_pose.pose.orientation.z = pose[1][2]
        goal_pose.target_pose.pose.orientation.w = pose[1][3]
        return goal_pose

# Main program starts here
if __name__ == '__main__':
        rospy.init_node('Navigation_Node')
        client = actionlib.SimpleActionClient('move_base', MoveBaseAction)

         # wait for action server to be ready
        client.wait_for_server()

        # Loop until ^c; delete this line if you want the TB3 to stop at the last waypoint
        while not rospy.is_shutdown():

                # repeat the waypoints over and over again
                for pose in waypoints:
                        goal = goal_pose(pose)
                        print("Going for goal: ", goal)
                        client.send_goal(goal)
                        client.wait_for_result()
```

GROUP PROJECT 3

APPLICATION OF ROSRUN, ROSLAUNCH, PATH PLANNING, LIDAR WITHIN THE TB3 WORLD

Project 3

For this project, you are **required to write 2 Python files and 1 launch file** with the following details; **use turtlebot3_world map** for both:

a) `autonomous_exploring.py`

- This file when executed will allow the TB3 to autonomously explore the map while avoiding collisions

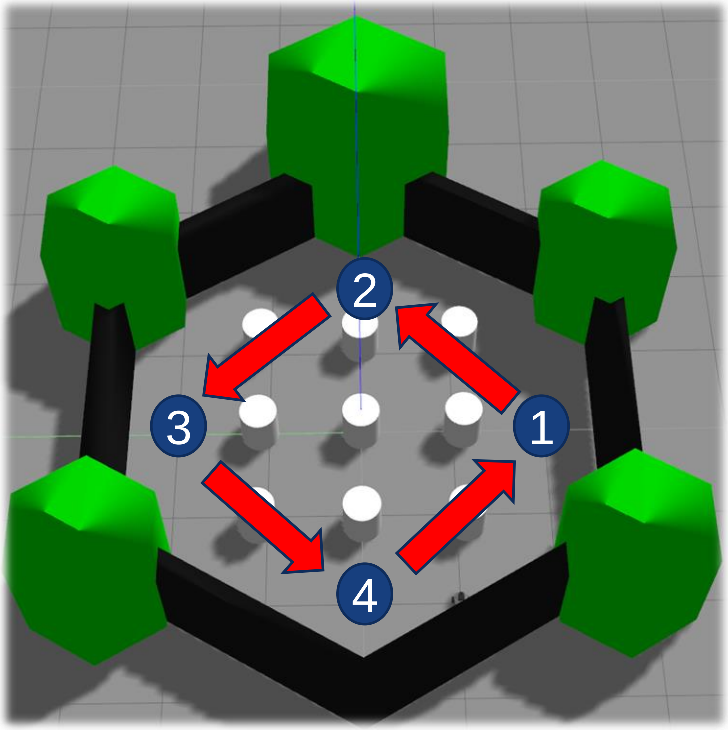
(i.e. similar to `turtlebot3_simulation.launch`)

- Come out with a simple algorithm first that will allow you to achieve this

Project 3

b) `Project3.launch` & `P3_path_planning.py`

The launch file when executed will open the RViz, then run the initial pose node (i.e. `initial_pose.py`), which enables the TB3 to first initiate the pose estimate for you. And lastly, run the path planning node (i.e. `P3_path_planning.py`), which enables the TB3 to follow the desired path; the desired path is illustrated below in the map:



FAQ:

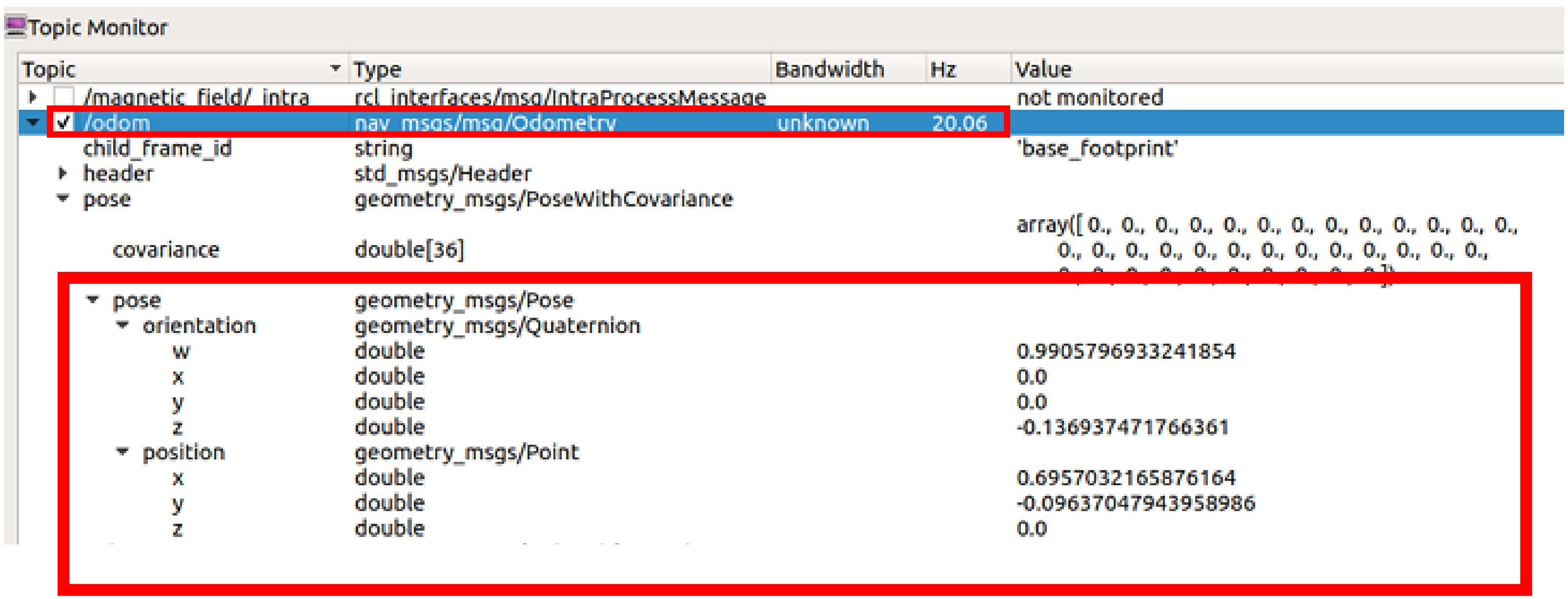
How do I obtain the coordinates of the respective locations???

RECAP

`/odom` → related to current pose of the TB3

As part of the setup, you can first manually adjust the TB3's positions/orientations that you desire and type the command below in a new terminal to obtain the position/orientation values at each location

```
$ rostopic echo /odom -n1
```



Project 3

REMINDER:

You have to launch your Gazebo first (to load TB3 in the TB3 World) first before you launch the launch file, `Project3.launch`

Example:

1. Launch Gazebo (to load TB3 in the TB3 World) :
```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```
2. Execute launch file to execute required tasks:
```
$ roslaunch autonomous Project3.launch
map_file:=$HOME/TB3_WORLD.yaml
```

Meaning you are only required to type 2 command lines on 2 separate terminals to perform the given tasks

Group Project 3

3 Items to Submit

(can zip all in one file):

1. `autonomous_exploring.py`
2. `Project3.launch`
3. `P3_path_planning.py`

Please upload all your codes in Luminus with your names as part of the title:

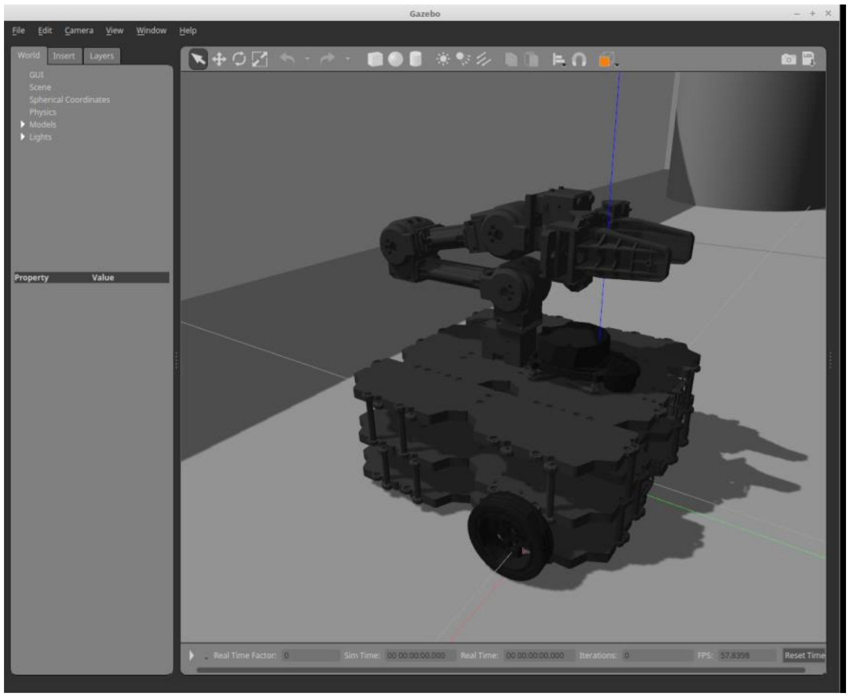
(e.g. DANIEL_KEN_TAN_Group_Project_3)

OR

(e.g. AXX_AXX_AXX_Group_Project_3)

3. SIMULATION OF TURTLEBOT3 WAFFLE PI WITH ARM

Run Gazebo



- Launch TurtleBot3 with arm in Gazebo
- `$ roslaunch turtlebot3_manipulation_gazebo turtlebot3_manipulation_gazebo.launch`
- Click Play button

Run move_group Node

In a new terminal, type the command to use the Moveit feature:

```
$ roslaunch turtlebot3_manipulation_moveit_config
move_group.launch
```

If the node is launched successfully, the following message will be printed in the terminal:

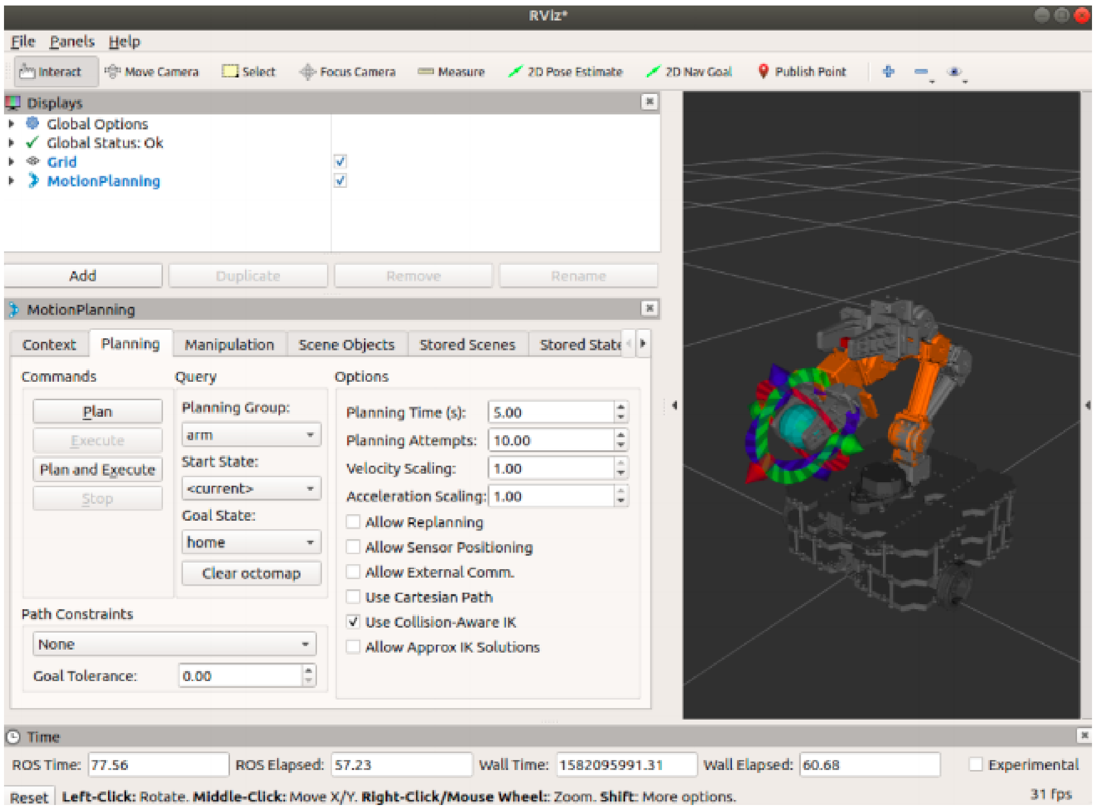
"You can start planning now!"

*****<u>Important:</u> you must run both Gazebo and move_group nodes first before you can control the OM on the TB3**

(a) Control via RViz

In a new terminal, type the command:

```
$ roslaunch turtlebot3_manipulation_moveit_config moveit_rviz.launch
```



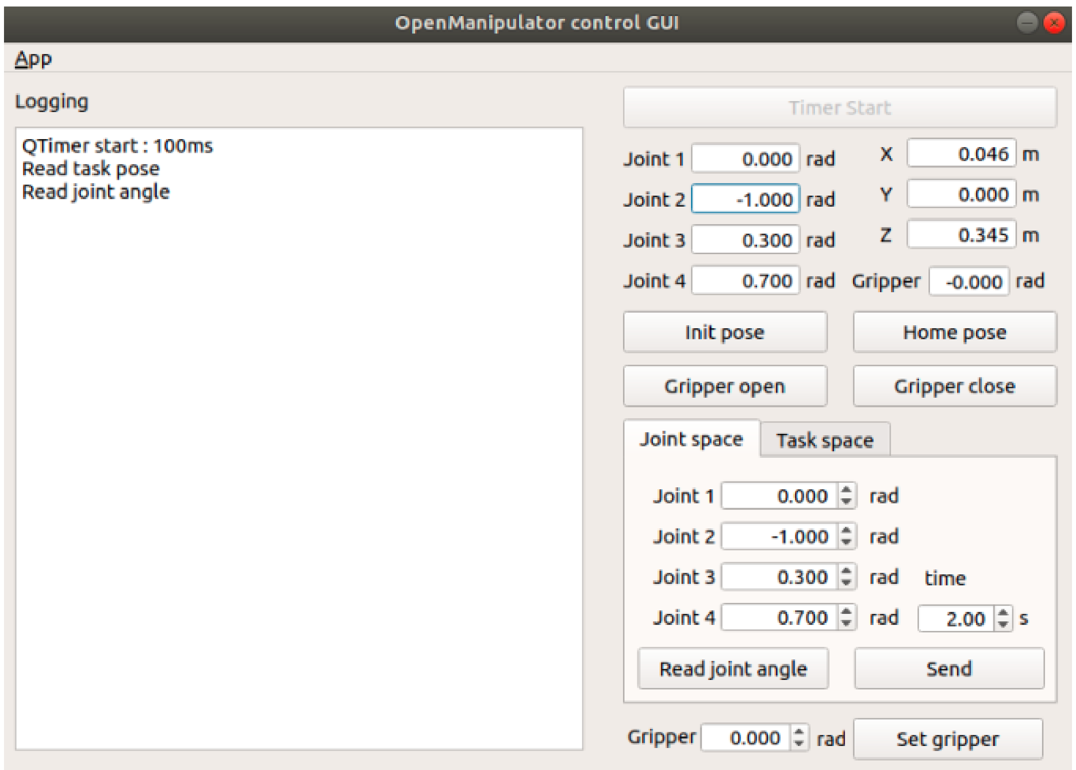
You can control the mounted OM using the Interactive Markers:

- Under MotionPlanning, click on the “Planning” Tab
- Move the Interactive Markers to shift the arm to your desired position
- Click “Plan and Execute” to move the arm

(b) Control via GUI

In a new terminal, type the command:

```
$ roslaunch turtlebot3_manipulation_gui
turtlebot3_manipulation_gui.launch
```



You can control the mounted OM using the GUI:

- **Run Gazebo and the Move Group Node first**
- **Same way of using as the OM only**
- **Task Space Control** and **Joint Space Control**

(c) SLAM and Controlled Navigation

Firstly, navigate to:

/home/catkin_ws/src/turtlebot3_manipulation_simulations/turtlebot3_manipulation_gazebo/launch

Copy and paste the given file in this folder:

`TB3_with_OM.launch`

This edited launch file allows you to use the TB3 world as your map in Gazebo for the TB3 with OM robot

(c) SLAM and Controlled Navigation

For <u>Virtual SLAM (one TB3)</u>, 4 Steps; same as TB3 without arm but commands differ slightly (<u>you may skip this process</u>):

<u>Step 1</u> - Launch the TB3 with arm in the *TB3 World* map in Gazebo

- `$ roslaunch turtlebot3_manipulation_gazebo TB3_with_OM.launch`
- Click Play button

<u>Step 2</u> - Launch RViz SLAM app in new terminal

- `$ roslaunch turtlebot3_manipulation_slam slam.launch`

(c) SLAM and Controlled Navigation

Step 3 – Launch teleop node in a new terminal:

- `$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`
- Use the keys to make the TB3 explore the area to capture data on the map

Step 4 - Save the Map once Completed in a new terminal:

- `$ rosrun map_server map_saver -f ~/map`

(c) SLAM and Controlled Navigation

For <u>Virtual Controlled Navigation</u>, 2 Steps:

<u>Step 1</u> - Launch Gazebo in *TB3* World

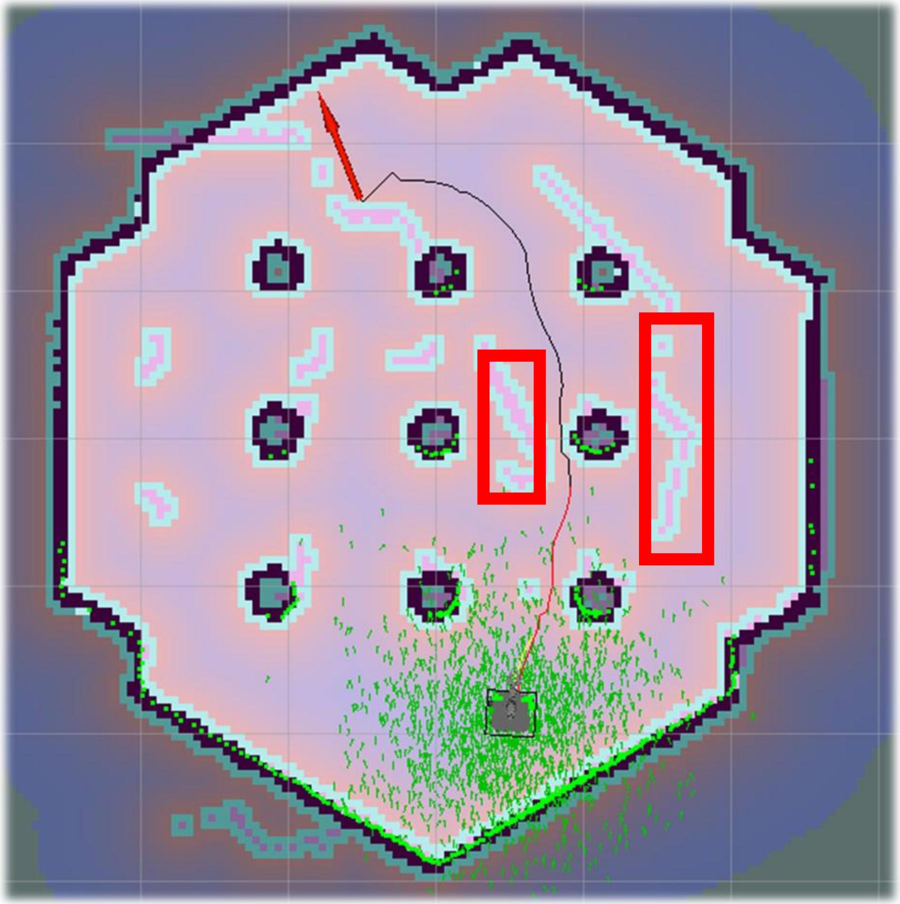
- `$ roslaunch turtlebot3_manipulation_gazebo TB3_with_OM.launch`
- Click Play button

<u>Step 2</u> - Execute Controlled Navigation in a new terminal

- `$ roslaunch turtlebot3_manipulation_navigation navigation.launch`
- Click on “2D Pose Estimate” and set the correct initial position for the TB3 with arm
- Click on “2D NAV Goal” and set the target destination for the TB3 with arm

(c) SLAM and Controlled Navigation

RViz Navigation in the *TB3 World* map:



- **RECAP:** Due to the symmetrical effect within the environment, there will be unwanted cost maps after you have performed the initial pose
- Do remember to get rid of these unwanted cost maps before performing the navigation
- **Rotating and/or Moving the robot back and forth a bit first with teleop node**

(d) Rospy to control OM

For TB3_with_OM robot model, the system is created as such that **you cannot control the OM with messages (i.e. service, publish)**. Instead, you **have to control with Move Group nodes**. You can write a python file to control the OM with **MoveIt interface:**

- Open terminal and type

  ```
  $ rosrun autonomous OM_moveit.py
  ```

- The OM will move to a point determined by the joint angles. To alter these joint angles, open the python file, change the values for the following variables; under the sub-functions *arm_joints()*

  ```
  joint_goal[X]
  ```

- To alter the gripper angle, change the value for the following variable (-0.01 for fully close and 0.01 for fully open); under the sub-functions *gripper_open/close()*

  ```
  joint_grip[0]
  ```

(d) Rospy to control OM

`OM_moveit.py` (imports and Class):

Note if you cant execute the python file, you have to give execution permissions to it by typing:

```
$ chmod +x name_of_the_file.py
```

```python
import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
```

```python
def all_close(goal, actual, tolerance):
  """
  Convenience method for testing if a list of values are within a tolerance
another list
  @param: goal       A list of floats, a Pose or a PoseStamped
  @param: actual     A list of floats, a Pose or a PoseStamped
  @param: tolerance  A float
  @returns: bool
  """
  all_equal = True
  if type(goal) is list:
    for index in range(len(goal)):
      if abs(actual[index] - goal[index]) > tolerance:
        return False

  elif type(goal) is geometry_msgs.msg.PoseStamped:
    return all_close(goal.pose, actual.pose, tolerance)

  elif type(goal) is geometry_msgs.msg.Pose:
    return all_close(pose_to_list(goal), pose_to_list(actual), tolerance)

  return True
```

```python
class MoveGroupPythonIntefaceTutorial(object):
  """MoveGroupPythonIntefaceTutorial"""
  def __init__(self):
    super(MoveGroupPythonIntefaceTutorial, self).__init__()

    ## First initialize `moveit_commander`_ and a `rospy`_ node:
    moveit_commander.roscpp_initialize(sys.argv)
```

(d) Rospy to control OM

`OM_moveit.py` (arm_joints1 sub function):

```
def arm_joints1(self):
  # Copy class variables to local variables to make the web tutorials more clear.
  # In practice, you should use the class variables directly unless you have a good
  # reason not to.
  group = self.group

  ## Planning to a Joint Goal
  # We can get the joint values from the group and adjust some of the values:
  joint_goal = group.get_current_joint_values()
  joint_goal[0] = 0
  joint_goal[1] = 0
  joint_goal[2] = 0
  joint_goal[3] = 0

  # The go command can be called with joint values, poses, or without any
  # parameters if you have already set the pose or joint target for the group
  group.go(joint_goal, wait=True)

  # Calling ``stop()`` ensures that there is no residual movement
  group.stop()

  ## END_SUB_TUTORIAL

  # For testing:
  # Note that since this section of code will not be included in the tutorials
  # we use the class variable rather than the copied state variable
  current_joints = self.group.get_current_joint_values()
  return all_close(joint_goal, current_joints, 0.01)
```

(d) Rospy to control OM

`OM_moveit.py` (gripper_close sub function):

```
def gripper_close(self):
    grip = self.grip
    joint_grip = grip.get_current_joint_values()
    joint_grip[0] = -0.01
    grip.go(joint_grip, wait=True)
    grip.stop()
    current_grip = self.grip.get_current_joint_values()
    return all_close(joint_grip, current_grip, 0.01)
```

(d) Rospy to control OM

`OM_moveit.py` (main function):

```
def main():
  try:
    print "============ Setting up the moveit_commander ..."
    #raw_input()
    tutorial = MoveGroupPythonIntefaceTutorial()

    print "============ Execute movement using 1st joint state goal ..."
    #raw_input()
    tutorial.arm_joints1()
    time.sleep(7)

    print "============ Execute gripper open ..."
    #raw_input()
    tutorial.gripper_open()
    time.sleep(3)

    print "============ Execute movement using 2nd joint state goal ..."
    #raw_input()
    tutorial.arm_joints2()
    time.sleep(7)

    print "============ Execute gripper close ..."
    #raw_input()
    tutorial.gripper_close()
    time.sleep(3)

    print "============ Python tutorial demo complete!"
  except rospy.ROSInterruptException:
    return
  except KeyboardInterrupt:
    return
```

(e) Pick, Move & Place Example

Procedures:

1. Load TB3 with OM in TB3 World in Gazebo

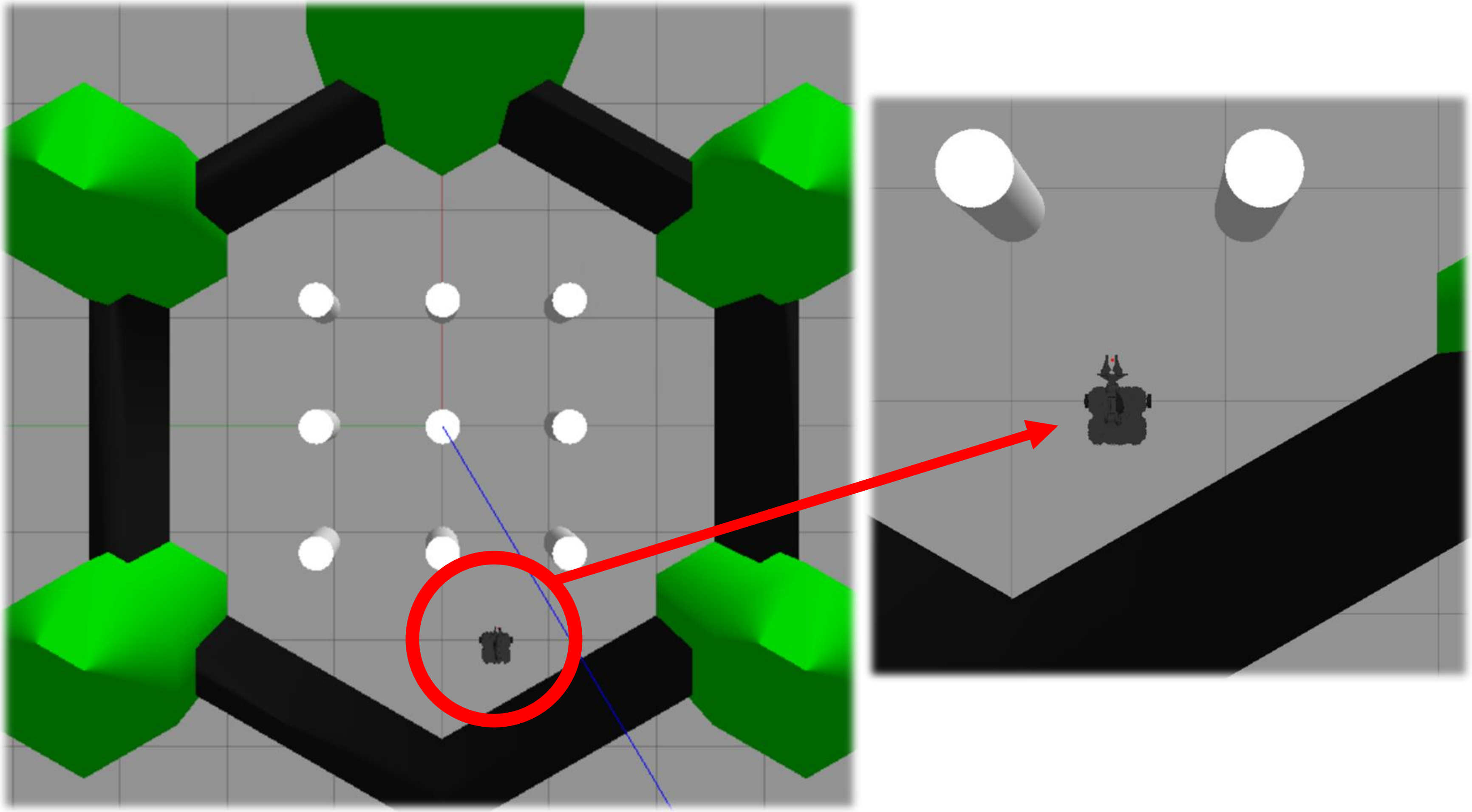
```
$ roslaunch turtlebot3_manipulation_gazebo TB3_with_OM.launch
```

2. Run launch file example to **activate OM control (via move_group)**, to **activate the navigation node**, and to **activate the pick_move_place node**, which make OM "pick" first, after which move TB3 to a certain location, and lastly to make OM "place"

```
$ roslaunch autonomous pick_move_place.launch
```

Pick, Move & Place Example

1. Load TB3 with OM in TB3 World in Gazebo:



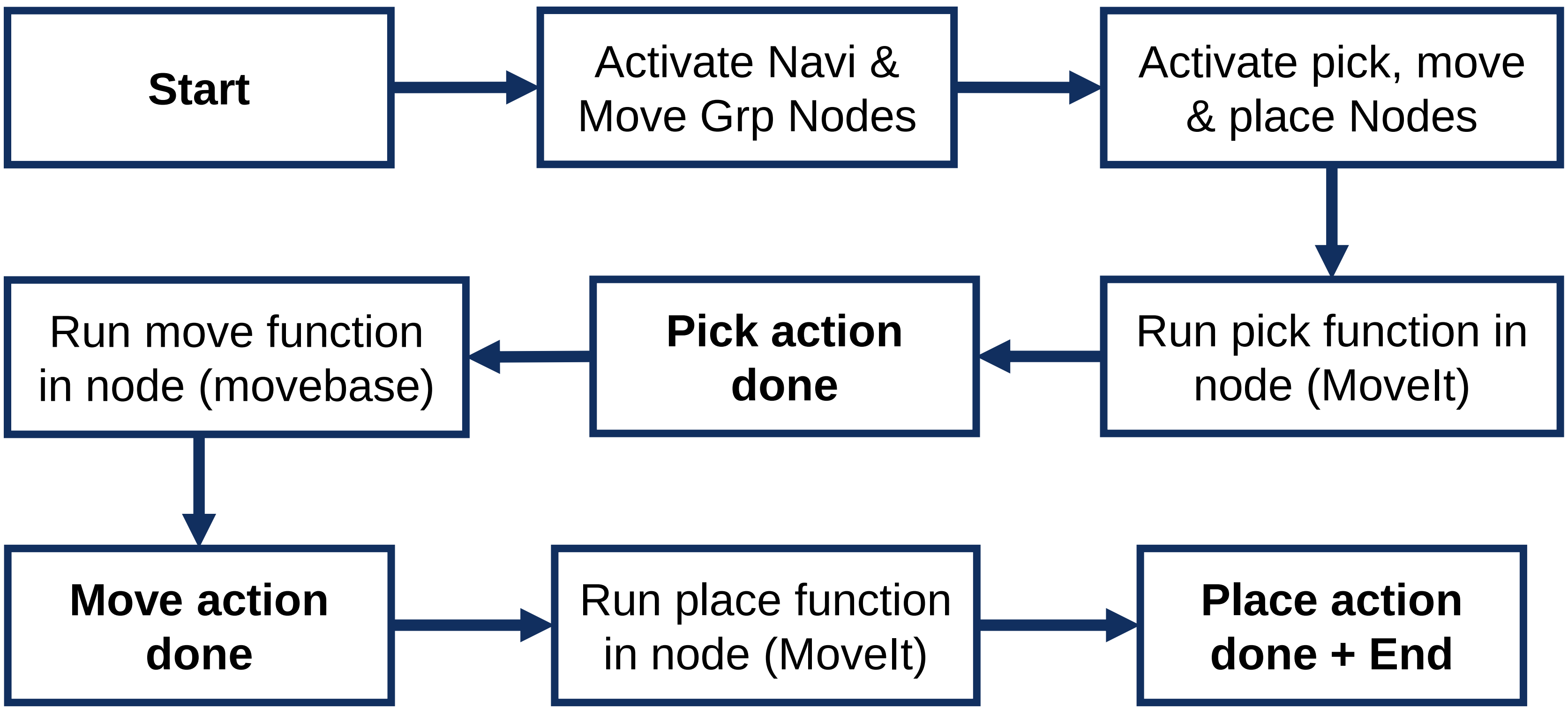
Pick, Move & Place Example

2. Run launch file example to “Pick”, Move and “Place”:

“Pick” **Move** **“Place”**

Pick, Move & Place Example

Block Diagram Representation of Algorithm:



BONUS PROJECT

OPTIONAL; IF YOU HAVE EXTRA TIME

Bonus Project

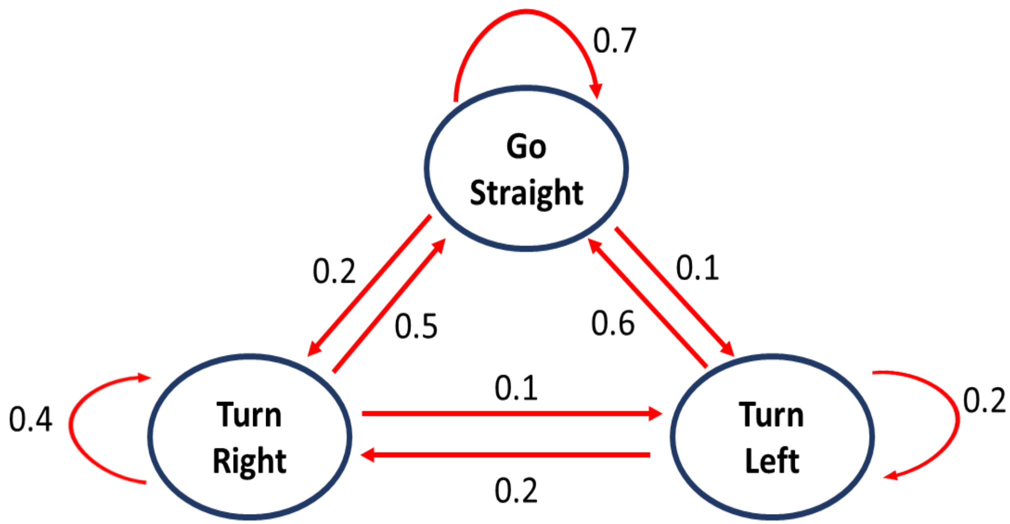
For this project, you are **required to modify the Python file** `autonomous_exploring.py` which enables the TB3 to:

1. **Autonomously explore the space** based on a given Markov Chain algorithm (uncertainty representation)
2. **Avoid obstacles** using a given algorithm that is dependent on the laser data (knowledge representation)

The algorithms for Goal 1 and Goal 2 are depicted in the next few slides. Use TB3 world map for this project. You may use the **given template:** "`bonus_project_template.py`" as a reference

Bonus Project: Markov Chain Algorithm to Decide on the TB3 Exploring Movements

The required Markov Chain for the TB3's exploring movements is summarized below:



Details on various states:

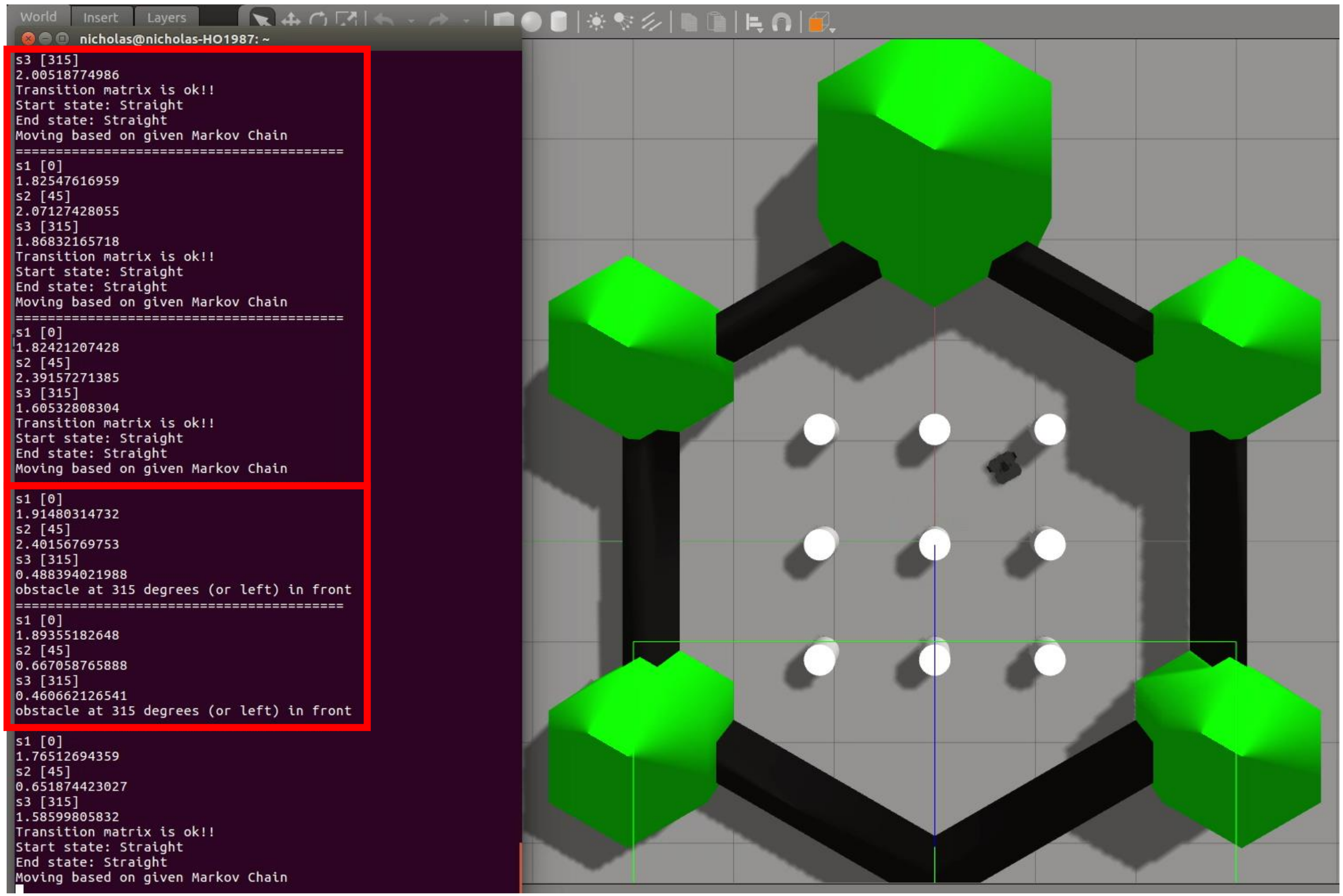
1. **Go Straight**
   (move.linear.x = 0.4)
2. **Turn Right**
   (move.linear.x = 0.2 & move.angular.z = -1)
3. **Turn Left**
   (move.linear.x = 0.2 & move.angular.z = 1)

Bonus Project: Algorithm to Avoid Obstacle based on Laser Data

The algorithm for the TB3 to avoid obstacles is summarized below:

- If the TB3 is all clear of obstacles (i.e. >= 0.5m at the front of the TB3 in directions 0, 45 and 315 degrees with respect to the TB3), it will perform its Markov-Chain-based exploring movements as mentioned in the previous slides
- Else if an obstacle is detected (< 0.5m) in direction 45 degrees with respect to the TB3, the TB3 will stop moving linearly and move 60 degrees to the right until it is cleared of obstacles
- Else if an obstacle is detected (< 0.5m) in direction 315 degrees with respect to the TB3, the TB3 will stop moving linearly and move 60 degrees to the left until it is cleared of obstacles
- Considering all else situations whereby obstacles are detected, the TB3 will stop moving linearly and move 90 degrees to the left until it is cleared of obstacles

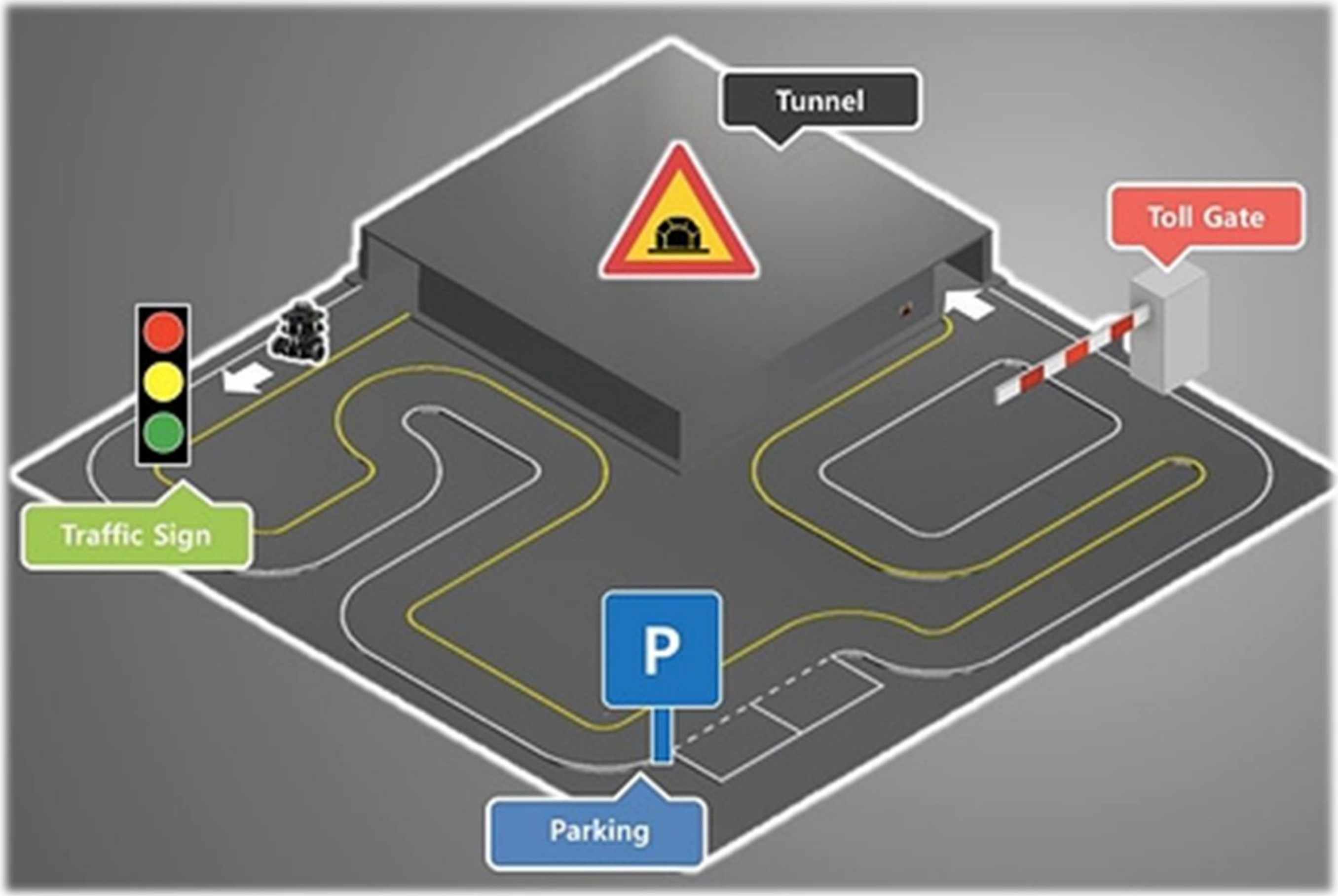
Bonus Project: Ideal Output



AUTONOMOUS VEHICLE SIMULATION

OPTIONAL; IF YOU HAVE EXTRA TIME

AutoRace with Gazebo



4 Steps

Step 1 – Install AutoRace packages (**DONE ALREADY**):

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_autorace.git
$ cd ~/catkin_ws && catkin_make
```

Install Additional Dependent Packages:

```
$ sudo apt-get install ros-kinetic-image-transport ros-kinetic-cv-bridge ros-kinetic-vision-opencv python-opencv libopencv-dev ros-kinetic-image-proc
```

AutoRace with Gazebo

Before you continue, do the following:

Firstly, navigate to
/home/*accountname*/cakin_ws/src/turtlebot3_simulations/turtlebot3_gazebo/launch

- **Copy and paste the given file: "*turtlebot3_autorace.launch*" in this folder**

Secondly, navigate to
/home/*accountname*/cakin_ws/src/turtlebot3_simulations/turtlebot3_gazebo/worlds

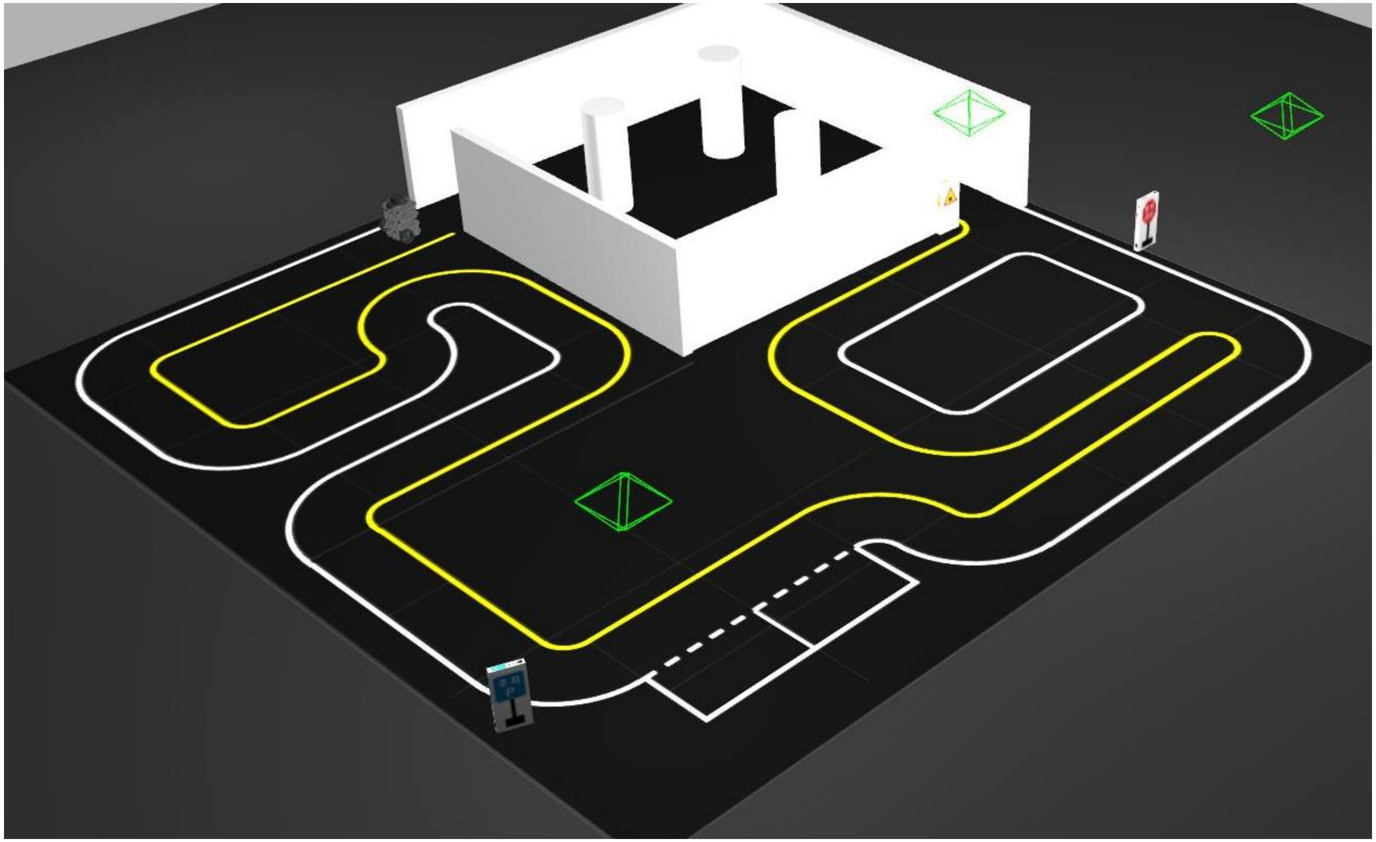
- **Copy and paste the given file: "*turtlebot3_autorace_2.world*" in this folder**

AutoRace with Gazebo

Step 2 - Run AutoRace Gazebo

```
$ roslaunch turtlebot3_gazebo turtlebot3_autorace.launch
```

You can see the AutoRace 2017 map in Gazebo:

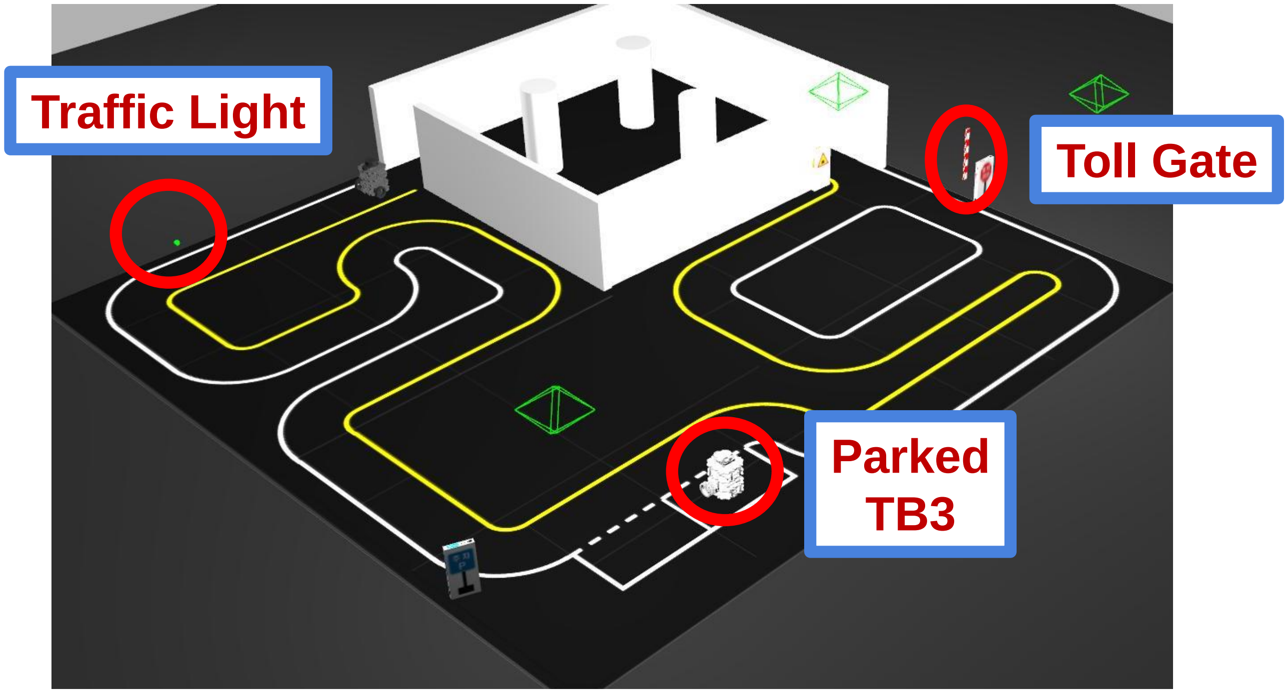


AutoRace with Gazebo

Step 3 - Run Mission Launch

```
$ roslaunch turtlebot3_gazebo
turtlebot3_autorace_mission.launch
```

You can see the various additional components (i.e. traffic light, Parked TurtleBot3, Toll Gate) in Gazebo:



AutoRace with Gazebo

Step 4 - Run AutoRace Launch

(a) Virtual calibration of camera

- `$ export GAZEBO_MODE=true`
- `$ export AUTO_IN_CALIB=action`
- `$ roslaunch turtlebot3_autorace_camera turtlebot3_autorace_intrinsic_camera_calibration.launch`

(b) Type in New Terminal

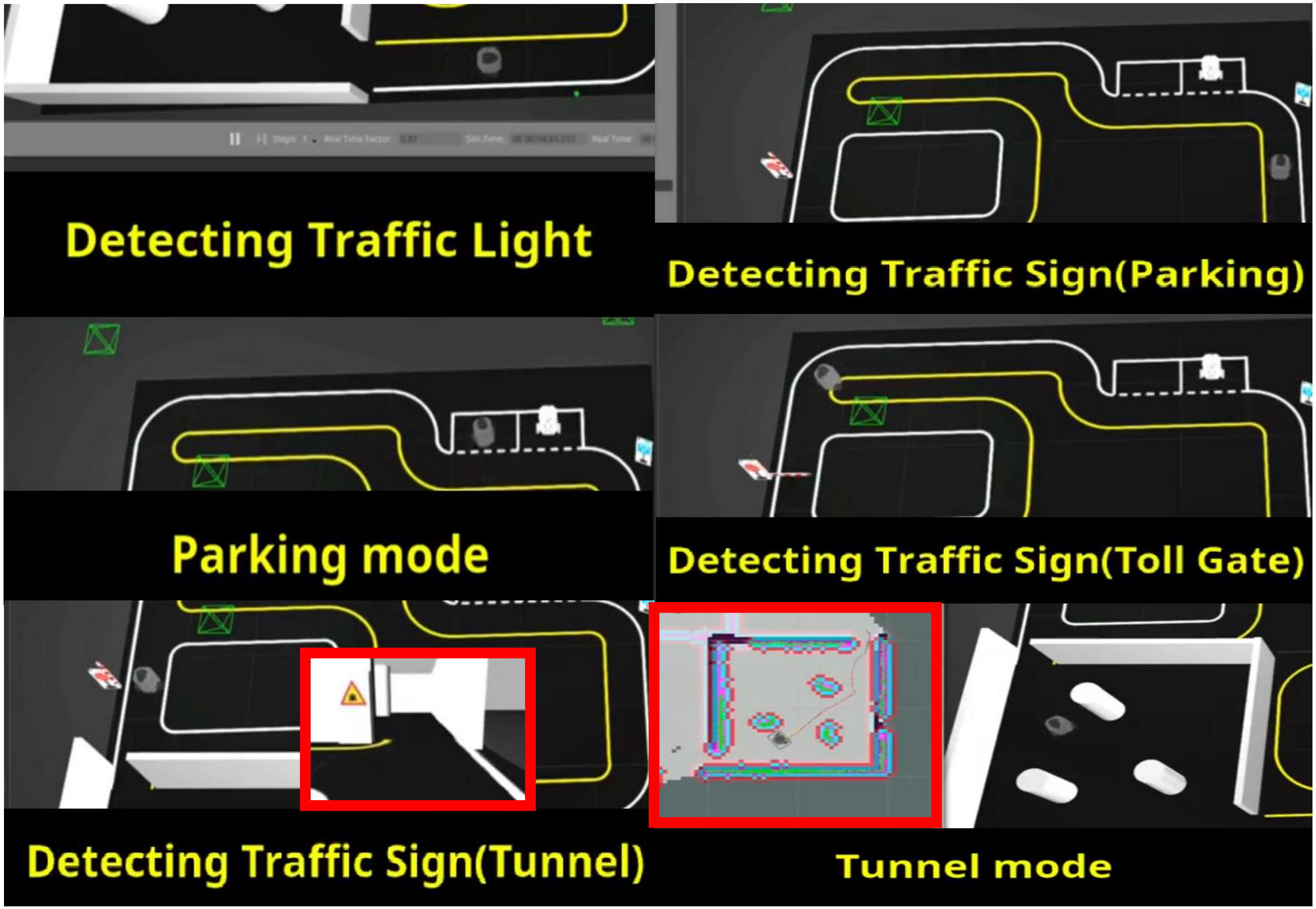
- `$ export AUTO_EX_CALIB=action`
- `$ export AUTO_DT_CALIB=action`
- `$ export TURTLEBOT3_MODEL=burger`
- `$ roslaunch turtlebot3_autorace_core turtlebot3_autorace_core.launch`

(c) Type in New Terminal

- `$ rostopic pub -1 /core/decided_mode std_msgs/UInt8 "data: 2"`

AutoRace with Gazebo

Step 4 - Run AutoRace Launch



AutoRace with Gazebo

Go to the following folder:

/home/*accountname*/cakin_ws/src/turtlebot3_autorace/turtlebot3_autorace_core/nodes

Various nodes involved :

1. Decision-making node:
`core_node_decider.py`
2. Main Controller node:
`core_node_controller.py`
3. Control Mission node:
`core_node_mission.py`

Check out these codes to know more about the nodes that control the autonomous movements for AutoRace

OTHER PROJECTS YOU CAN WORK WITH

(WILL NOT BE COVERED IN THIS WORKSHOP)

Other Projects

1. Machine Learning

- http://emanual.robotis.com/docs/en/platform/turtlebot3/machine_learning/#machine-learning

2. Simulated Prius in Gazebo

- https://github.com/osrf/car_demo
- https://www.osrfoundation.org/simulated-car-demo/

End of Module 5

THANK YOU for your kind attention!

Main References

- Self-Driving Cars: A Survey, Cornell University, by Badue, et al.
- Kocić, J., Jovičić, N., & Drndarević, V. (2019). An End-to-End Deep Neural Network for Autonomous Driving Designed for Embedded Automotive Platforms. Sensors (Basel, Switzerland), 19(9), 2064. doi:10.3390/s19092064
- Shinde, Vipul et al. (2018). Automatic Car Driving System Using Fuzzy Logic. International Journal of Advanced Research in Science, Engineering and Technology, 5(3), 5367-72
- Katrakazas, Christos & Quddus, Mohammed & Chen, Wen-Hua & Deka, Lipika. (2015). Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. Transportation Research Part C: Emerging Technologies. 60. 416-442. 10.1016/j.trc.2015.09.011.
- Schoener, Hans-Peter. (2017). The Role of Simulation in Development and Testing of Autonomous Vehicles. 10.1007/978-3-658-21194-3_82.
- Sovani, Sandeep. (2017). Simulation Accelerates Development of Autonomous Driving. ATZ worldwide. 119. 24-29. 10.1007/s38311-017-0088-y.
- https://towardsdatascience.com/sensor-fusion-90135614fde6
- https://towardsdatascience.com/self-driving-cars-the-guide-f1f427b9656b
- http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/
- http://emanual.robotis.com/docs/en/platform/openmanipulator_main