Mobile Robot Manipulation

Project Report

Submitted by

Rohan Mehra
BS (Electrical Engineering and Computer Science)
Indian Institute of Science Education and Research Bhopal
E-mail: rohan21@iiserb.ac.in

Under the guidance of

Dr. Deepak Mishra
Professor

Department of Avionics
Indian Institute of Space Science and Technology
Thiruvananthapuram, India

May-July 2023

DECLARATION

I, **Rohan Mehra**, hereby declare that, this report entitled **"Mobile Robot Manipulation"** submitted to Indian Institute of Space Science and Technology Thiruvananthapuram as part of the **Summer Internship Program 2023** in **Department of Avionics, IIST Thiruvananthapuram**, is an original work carried out by me under the supervision of **Dr. Deepak Mishra** and has not formed the basis for any other certificate, in this or any other institution or university. I have sincerely tried to uphold academic ethics and honesty.

Thiruvananthapuram - 695 547 **Rohan Mehra**
November 2023

CERTIFICATE

This is to certify that the work contained in this project report entitled **"Mobile Robot Manipulation"** submitted by **Rohan Mehra** to Indian Institute of Space Science and Technology, Thiruvananthapuram as part of **Summer Internship Program 2023** in **Department of Avionics** has been carried out by him under my supervision and that it has not been submitted elsewhere.

Thiruvananthapuram - 695 547
November 2023

Dr. Deepak Mishra
Project Supervisor

ABSTRACT

This internship report delves into the realm of Mobile Robot Manipulation, aiming to contribute to the ongoing advancements in this field. The research encompasses theoretical foundations and its primary aim is to focus on understanding the basics of Robot Manipulation including learning ROS and Gazebo. This report provides an in-depth grasp of various types of Bayesian Filter (Kalman Filter, EKF, Particle Filter etc) which I explored and various SLAM methods through those filters. Furthermore, basic tasks for laying the foundation in SLAM, Navigation and Robotics in general were performed using turtlebot3 and SUMMIT-XL robot. The videos of the simulations are uploaded on following drive link (https://drive.google.com/drive/folders/1BvY1AvkDySe3IZkmHtzSOy3QuP9lPAJs?usp=sharing)

Contents

Chapter 1

Introduction to Robotics and Mobile Robots

1.1 What is Robotics?

A robot is a mechanical agent that may operate autonomously or partially autonomously by responding to its surroundings or by following pre-programmed instructions. To design, manufacture, program, and manage robots, the field of robotics combines engineering, computer science, and other academic disciplines. Mechanical structures, sensors, actuators, and a control system are some of the parts that make up robotic systems.

The robot's physical framework, as well as its capabilities and range of motion, are determined by its mechanical structure. By detecting and measuring several characteristics like distance, temperature, pressure, or visual information, sensors allow the robot to sense and comprehend its environment. Actuators are in charge of carrying out physical operations like moving a robot's limbs or handling objects. The robot's operations are coordinated by the control system, which is frequently powered by software algorithms, depending on data from the sensors and desired tasks or goals.

Overall, robotics is an interdisciplinary field that aims to create machines capable of performing tasks autonomously, enhancing productivity, improving safety, and enabling new possibilities in various domains of human activity.

1.2 Fundamental Aspects of Robotics

Robotics involves a set of fundamental aspects to get its job done. Some of them are listed below-:

1. **Perception and Sensing:** Robots depend on sensors to recognize and comprehend their surroundings. This includes identifying and deciphering sensory data such as chemical or biological signals, as well as visual information, sound, touch, and other sensory data. Robots can navigate their surroundings, recognize things, detect impediments, and interact with their environment thanks to perception.

2. **Localization and Navigation:** The process of Localization involves figuring out where a mobile robot is in relation to its surroundings and the science of navigating a robot (a mobile robot) through its surroundings is known as Navigation. To accomplish precise navigation and localization, methods like GPS, odometry, or simultaneous localization and mapping (SLAM) are used.

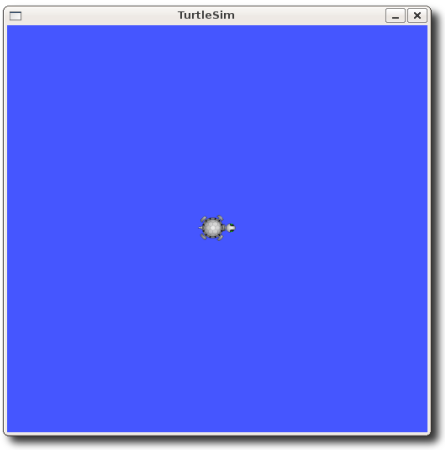
3. **Grasping:** A robot's capacity to operate and grab items with its mechanical end-effectors, such as grippers or robotic hands, is referred to as grasping. The actions of the robot must be coordinated, the object must be perceived, and the right amount of force must be applied to hold the thing firmly. Robotic grasping aims to provide algorithms and methods that let robots grab a variety of things in a variety of situations with adaptability and efficiency.

1.3 Mobile Robots

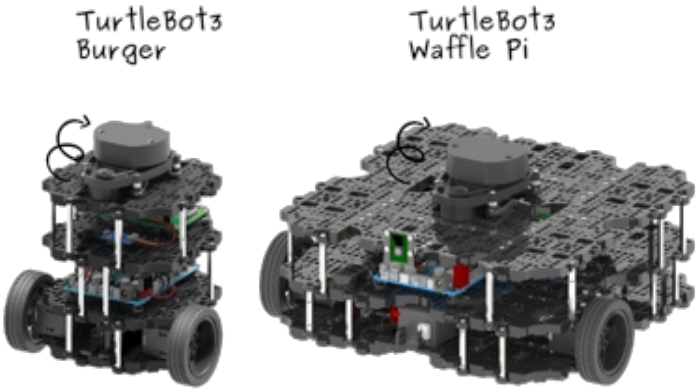
Mobility refers to the ability of the robot to move around in its surrounding. These kinds of robots are also called Autonomous Mobile Robots. They can be fitted with a pre-defined navigating route to move around and work accordingly. In this project, firstly turtlesim has been used for introductory purposes, and then TurtleBot3 and SUMMIT-XL has been used

1.3.1 `turtlesim`

Turtlesim is the most basic type of robot we will be using which is simulation 2-D simulation robot made with the purpose of learning ROS and ROS Packages. For more details of the robot, please visit the the documentation at http://wiki.ros.org/turtlesim



1.3.2 TurtleBot3



TurtleBot3 is an open-source mobile robot platform which comes in various models, including the Burger, Waffle, and Waffle Pi, each offering different features and capabilities. The main components of TurtleBot3 typically include a base with differential drive wheels for mobility, a sensor module for perception, a computing unit (such as a Raspberry Pi or an Intel Joule) for processing, and optional accessories like cameras or LiDAR sensors for enhanced capabilities.

The goal of TurtleBot3 is to dramatically reduce the size of the platform and lower the price without having to sacrifice its functionality and quality, while at the same time offering expandability. The TurtleBot3 can be modified in a number of ways depending on how the mechanical components are put back together and how optional components like the computer and sensor are used. The TurtleBot3

has also evolved with a small-sized, reasonably priced SBC that is appropriate for a reliable embedded system, a 360-degree distance sensor, and 3D printing technology.

TurtleBot3 supports a rich ecosystem of ROS packages and tools, such as the navigation stack for autonomous navigation, the gmapping package for simultaneous localization and mapping (SLAM), and the RViz visualization tool, among others.

1.3.3 SUMMIT-XL

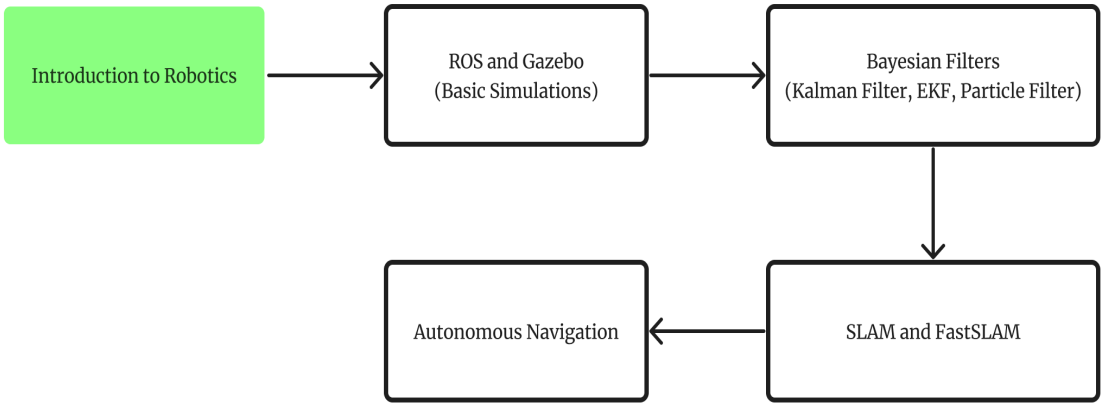
The SUMMIT-XL is a mobile robot developed by Robotnik, and features a robust and rugged construction, making it suitable for demanding tasks and challenging terrains.

It has a wide range of sensors, including laser scanners, cameras, and odometry sensors, allowing for accurate mapping, obstacle recognition, and localisation. The robot is capable of using simultaneous localization and mapping (SLAM) methods, enabling it to automatically map its surroundings and navigate. It has applications in fields where mobility, flexibility, and autonomy are essential, including research and development, logistics, industrial automation, and field robots. Some of the technical specifications of the following are

Dimentions: 978 x 776 x 510 mm
Weight: 105 kg
Payload: Up to 250 Kg

Speed: 3 m/s
Controller: Open architecture ROS Integrated CPU with Linux

1.4 Broad Outline Forward



Chapter 2

ROS and Gazebo

2.1 Robot Operating System

The open-source Robot Operating System, sometimes known as ROS, is a framework designed to facilitate the development of robotic systems. It is not an operating system as the name says but is a collection of software libraries and tools that help developers create and manage complex robotic applications. Willow Garage, a robotics research facility, developed ROS at first, and it has subsequently gained widespread use in both academic and professional contexts.

ROS is primarily supported on Ubuntu Linux, and there are different distributions available (e.g., Melodic, Noetic, etc.), though the newer version ROS 2 is supported by Windows, macOS, and Debian. For this project, we will be using the ROS Noetic distribution which has already been installed and set up on Ubuntu Linux. ROS is primarily composed of the following key components.

1. **Nodes:** Within a robotic system, nodes are separate operations that carry out particular duties. Using the ROS communication framework, they may be created in a variety of programming languages, including Python, C++, and even Rust. Nodes are the fundamental building blocks of a ROS application.

2. **Topics:** The channels of communication that nodes utilize to post and subscribe to messages are referred to as topics. Other nodes interested in the same information can subscribe to the same topic to receive messages published by one node to that subject. Topics allow nodes to communicate asynchronously.

3. **Messages:** When exchanging data, nodes in ROS send and receive messages. The data structure and format for communication between nodes are defined by messages. Numerous basic message kinds are provided by ROS, and we can also design your own unique message types to meet your unique requirements.

4. **Services:** In addition to publish-subscribe communication, ROS also supports request-response communication using services. Other nodes can ask for a node to deliver a service with a particular feature. The requester waits for the service provider to respond in this synchronous type of communication.

5. **Actions:** In ROS, actions are a more advanced way of communication that is mainly utilized for lengthy processes that call for feedback and preemption. Nodes can send goals, get feedback, and cancel tasks using actions in a systematic manner.

6. **Launch Files:** Launch files are used by ROS to simultaneously start and configure a number of nodes and parameters. Launch files make it easier to start up complex robotic systems and define their interconnection.

7. **ROS Bags:** Bags are essentially a format for storing time-stamped data streams, including sensor data, images, and messages exchanged between nodes in a ROS system.

8. **Packages:** Packages are used to organize the code and resources in ROS. The components of a package are code, configuration files, documentation, and dependencies. The ROS community can maintain and exchange code with ease thanks to ROS packages.

2.2 Basic Operations with ROS

After the installation and setup of ROS on our system, we will start working on it by starting the ROS Master with the following command

```
roscore
```

By this, we will essentially establish the core infrastructure that enables ROS nodes to communicate, share information, and work together within your robotic system. It acts as the central hub that connects all the parts of our ROS application. It's

important to note that roscore typically needs to be running for the duration of our ROS application's execution. We may may run it in a terminal, and it will continue running until we manually stop it with Ctrl+C or close the terminal.

We can start our first nodes which is the talker and the listener node which are a part of ROS tutorials and are provided by ROS community as an initial touch. We can use them by using the following commands (both in different terminals and we can see the outputs attached.

```
rosrun rospy_tutorials talker
rosrun rospy_tutorials listener
```

The `rosrun` command is used to run individual ROS nodes by specifying the package and node name. Here, the package is `rospy_tutorials` and the specific nodes are `talker` and `listener`. We get the following output with this-:

```
[INFO] [1641196090.574959]: hello world 1641196090.5741975
[INFO] [1641196090.663551]: hello world 1641196090.6634207
[INFO] [1641196090.763800]: hello world 1641196090.7636702
[INFO] [1641196090.863030]: hello world 1641196090.8626063
[INFO] [1641196090.963386]: hello world 1641196090.9632573
[INFO] [1641196091.063094]: hello world 1641196091.0629294
[INFO] [1641196091.163045]: hello world 1641196091.162916
```

Figure 2.1: talker node

```
[INFO] [1641196091.465104]: /listener_13844_1641196086378I heard hello
 world 1641196091.4632475
[INFO] [1641196091.570317]: /listener_13844_1641196086378I heard hello
 world 1641196091.56839
[INFO] [1641196091.664690]: /listener_13844_1641196086378I heard hello
 world 1641196091.6627762
[INFO] [1641196091.764989]: /listener_13844_1641196086378I heard hello
 world 1641196091.7632623
```

Figure 2.2: listener node

We can go ahead with writing our own publisher and subscriber node in Python. Firstly, we will create a node that will publish into the `/turtle1/cmd_vel` topic at a rate of 2 Hz of our turtlebot. We will create a message object and will give some linear and angular velocity. Then we will create a node that subscribes to the `/turtle1/pose` topic and takes the pose of our turtlebot constantly. After executing of both the nodes, we can get a better idea of what is happening internally by using the command `rqt_graph` to visualize through a graph.

my_robot_controller > scripts > my_first_node.py > ...

```python
#!/usr/bin/python3
import rospy
from geometry_msgs.msg import Twist

if __name__=='__main__':
    rospy.init_node("draw_circle")
    rospy.loginfo("Node started")
    pub= rospy.Publisher("/turtle1/cmd_vel",Twist,queue_size=10)
    rate= rospy.Rate(2)

    while not rospy.is_shutdown():
        msg = Twist()
        msg.linear.x = 2.0
        msg.angular.z = 1.0
        pub.publish(msg)
        rate.sleep()

    rospy.loginfo("End of Program")
```

Figure 2.3: Publisher Node

my_robot_controller > scripts > pose_subscriber.py > ...

```python
#!/usr/bin/env python3
import rospy
from turtlesim.msg import Pose

def pose_callback(msg: Pose):
    rospy.loginfo("("+ str(msg.x)+","+str(msg.y)+")")

if __name__=='__main__':
    rospy.init_node("turtle_pose_subscriber")
    sub=rospy.Subscriber("/turtle1/pose", Pose, callback=pose_callback)

    rospy.loginfo("Node has been started")
    rospy.spin()
```

Figure 2.4: Subscriber Node

Please note that in the Subscriber Node, the message has been processed in the callback, and only x and y coordinates will be printed. Now, we will combine Publisher and Subscriber in a Closed Loop System which is done in the next code. In it, our turtlebot will loop back on the edges and will continue straight otherwise according to the pose. It will also change the pen colour according to the x coordinate The reader can view the implementation of the code on the drive link (https://drive.google.com/drive/folders/1BvY1AvkDySe3IZkmHtzSOy3QuP9lPAJs?usp=sharing)

my_robot_controller > scripts > turtle_controller.py > pose_callback

```
8
9   def call_set_pen_service(r, g, b, width, off):
10      try:
11          set_pen = rospy.ServiceProxy("turtle1/set_pen", SetPen)
12          response = set_pen(r, g, b, width, off)
13          rospy.loginfo(response)
14      except rospy.ServiceException as e:
15          rospy.logwarn(e)
16
17  def pose_callback(pose: Pose):
18      cmd=Twist()
19      if pose.x > 9.0 or pose.x < 2.0 or pose.y > 9.0 or pose.y < 2.0:
20          cmd.linear.x = 3
21          cmd.angular.z = 1.2
22      else:
23          cmd.linear.x = 5.0
24          cmd.angular.z = 0.0
25      pub.publish(cmd)
26
27      global previous_x
28      if pose.x >= 5.5 and previous_x < 5.5:
29          call_set_pen_service(255, 0, 0, 3, 0)
30      elif pose.x < 5.5 and previous_x >= 5.5:
31          call_set_pen_service(0, 255, 0, 3, 0)
32      previous_x = pose.x
33
34  if __name__=='__main__':
35      rospy.init_node("turtle_controller")
36      rospy.wait_for_service("turtle1/set_pen")
37      pub=rospy.Publisher("/turtle1/cmd_vel", Twist, queue_size=10)
38      sub = rospy.Subscriber("/turtle1/pose", Pose, callback=pose_callback)
39      rospy.loginfo("Node has been started")
40
41      rospy.spin()
```

Figure 2.5: Publisher and Subscriber in a Closed Loop System

Apart from the Publisher-Subscriber Architecture in ROS, there is also Client-Server Architecture which is demonstrated below. The benefit of this architecture is here the client sends requests directly to the server and receives responses. The Client-Server interactions are inherently synchronous, meaning the client waits for a response from the server.

server.py client.py

server.py > ...

```python
#!/usr/bin/env/python3

import rospy
from std_srvs.srv import SetBool, SetBoolResponse

def callback(req):
    response = SetBoolResponse()
    if req.data == True:
        response.success = True
        response.message = "The device was enabled"
    else:
        response.success = True
        response.message = "The device was disabled"

    return response

rospy.init_node("server")
rospy.Service("test_service", SetBool, callback)
rospy.spin()
```

Figure 2.6: Server

server.py client.py

client.py > ...

```python
#!/usr/bin/env python3

import rospy
from std_srvs.srv import SetBool, SetBoolRequest

rospy.init_node("client")
client = rospy.ServiceProxy("test_service",SetBool)
client.wait_for_service()

request = SetBoolRequest()
request.data = True

response = client(request)

print(response)
```

Figure 2.7: Client

2.3 Introduction to Gazebo

Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.

Typical uses of Gazebo are:

- Testing robotics algorithms.
- Designing Robots.
- Performing regression testing with realistic scenarios.

Gazebo can be installed and run using the tutorials on the Gazebo website. After the gazebo is installed and running, it is good to get familiar with the GUI of Gazebo such as the scene, left and right panel, upper and bottom toolbar, the menu and the mouse control. After this is done, custom creation of a basic model from a model editor can be done.

2.4 Creating Models with Gazebo

For basic understanding, a model of a vehicle can be made with a depth camera sensor to make the vehicle move slowly toward a box. The steps to do for the same are mentioned below. The detailed tutorial for the same can be found in the Gazebo website Tutorial section.

1. Creating the Chassis using a box and adjusting it by resizing and flattening it and bringing it near to the ground.
2. Making the front wheel using cylinders and aliging them and create a joint between them.
3. Making the caster wheel using a sphere by resizing it, creating a joint between the wheel and the main body, and aligning it.
4. Adding a sensor by downloading the file from the online database and adding it on the top of the chassis and joining it.
5. Adding a Plugin in the `Model Plugin` section. Enter `follower` in the `Plugin Name` field

6. Saving the Model and Adding a box in front of the vehicle for the vehicle to follow the box

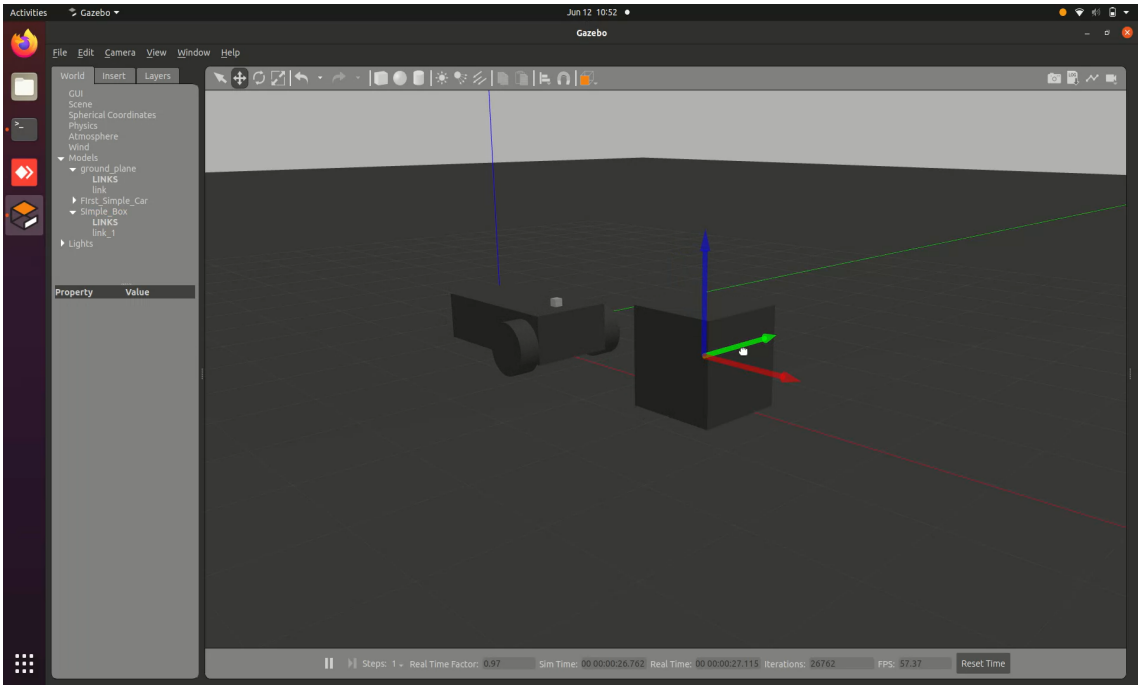


Figure 2.8: The Constructed Vehicle

More complex models can also be made by writing custom SDF files and creating an SDF Model. The next model is of the Velodyne HDL-32 LiDAR which has been created using in which some physical constraints such as inertia have been added. The screenshots of the SDF Files are shown below. The video of the simulation of the LiDAR applying some force has been added to the drive link mentioned in this report. The step-by-step process to create the model is -:

1. A basic SDF model is created
2. Inertia is added to it
3. A Joint is added to it
4. A Sensor is added on top of it

```xml
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">

    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>

    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <model name="velodyne_hdl-32">
  <!-- Give the base link a unique name -->
  <link name="base">

    <!-- Offset the base by half the lenght of the cylinder -->
    <pose>0 0 0.029335 0 0 0</pose>
    <inertial>
      <mass>1.2</mass>
      <inertia>
        <ixx>0.001087473</ixx>
        <iyy>0.001087473</iyy>
        <izz>0.001092437</izz>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyz>0</iyz>
      </inertia>
    </inertial>
    <collision name="base_collision">
      <geometry>
        <cylinder>
          <!-- Radius and length provided by Velodyne -->
          <radius>.04267</radius>
          <length>.05867</length>
        </cylinder>
      </geometry>
    </collision>

    <!-- The visual is mostly a copy of the collision -->
    <visual name="base_visual">
      <geometry>
        <cylinder>
          <radius>.04267</radius>
          <length>.05867</length>
        </cylinder>
      </geometry>
    </visual>
  </link>

  <!-- Give the base link a unique name -->
  <link name="top">

```

Figure 2.9

```
45          <radius>.04267</radius>
46          <length>.05867</length>
47        </cylinder>
48      </geometry>
49    </visual>
50  </link>
51
52  <!-- Give the base link a unique name -->
53  <link name="top">
54
55
56 <!-- Add a ray sensor, and give it a name -->
57 <sensor type="ray" name="sensor">
58
59   <!-- Position the ray sensor based on the specification. Also rotate
60        it by 90 degrees around the X-axis so that the <horizontal> rays
61        become vertical -->
62   <pose>0 0 -0.004645 1.5707 0 0</pose>
63
64   <!-- Enable visualization to see the rays in the GUI -->
65   <visualize>true</visualize>
66
67   <!-- Set the update rate of the sensor -->
68   <update_rate>30</update_rate>
69   <ray>
70
71   <!-- The scan element contains the horizontal and vertical beams.
72        We are leaving out the vertical beams for this tutorial. -->
73   <scan>
74
75     <!-- The horizontal beams -->
76     <horizontal>
77       <!-- The velodyne has 32 beams(samples) -->
78       <samples>32</samples>
79
80       <!-- Resolution is multiplied by samples to determine number of
81            simulated beams vs interpolated beams. See:
82            http://sdformat.org/spec?ver=1.6&elem=sensor#horizontal_resolution
83            -->
84       <resolution>1</resolution>
85
86       <!-- Minimum angle in radians -->
87       <min_angle>-0.53529248</min_angle>
88
89       <!-- Maximum angle in radians -->
90       <max_angle>0.18622663</max_angle>
91     </horizontal>
92   </scan>
93
94   <!-- Range defines characteristics of an individual beam -->
95   <range>
96
97     <!-- Minimum distance of the beam -->
98     <min>0.05</min>
```

Figure 2.10

```
    <min>0.05</min>

    <!-- Maximum distance of the beam -->
    <max>70</max>

    <!-- Linear resolution of the beam -->
    <resolution>0.02</resolution>
  </range>
</ray>
</sensor>

    <!-- Vertically offset the top cylinder by the length of the bottom
        cylinder and half the length of this cylinder. -->
    <pose>0 0 0.095455 0 0 0</pose>
   <inertial>
     <mass>0.1</mass>
     <inertia>
       <ixx>0.000090623</ixx>
       <iyy>0.000090623</iyy>
       <izz>0.000091036</izz>
       <ixy>0</ixy>
       <ixz>0</ixz>
       <iyz>0</iyz>
     </inertia>
   </inertial>
    <collision name="top_collision">
      <geometry>
        <cylinder>
          <!-- Radius and length provided by Velodyne -->
          <radius>0.04267</radius>
          <length>0.07357</length>
        </cylinder>
      </geometry>
    </collision>

    <!-- The visual is mostly a copy of the collision -->
    <visual name="top_visual">
      <geometry>
        <cylinder>
          <radius>0.04267</radius>
          <length>0.07357</length>
        </cylinder>
      </geometry>
    </visual>
  </link>
  <!-- Each joint must have a unique name -->
<joint type="revolute" name="joint">

  <!-- Position the joint at the bottom of the top link -->
  <pose>0 0 -0.036785 0 0 0</pose>

  <!-- Use the base link as the parent of the joint -->
  <parent>base</parent>
```

Figure 2.11

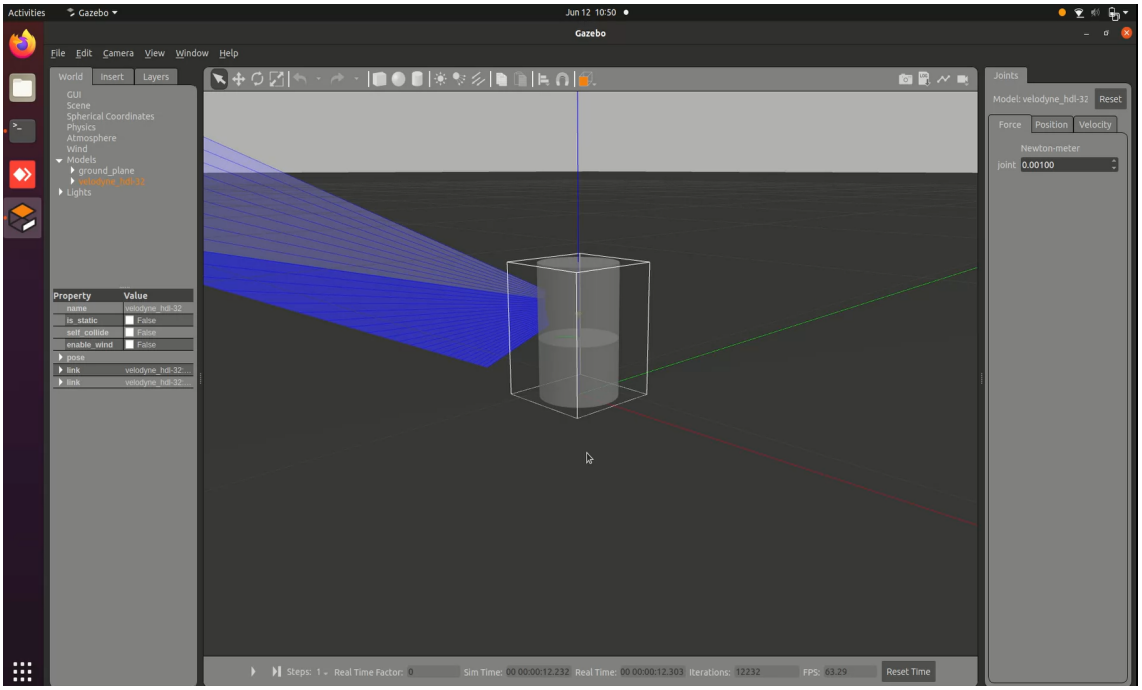
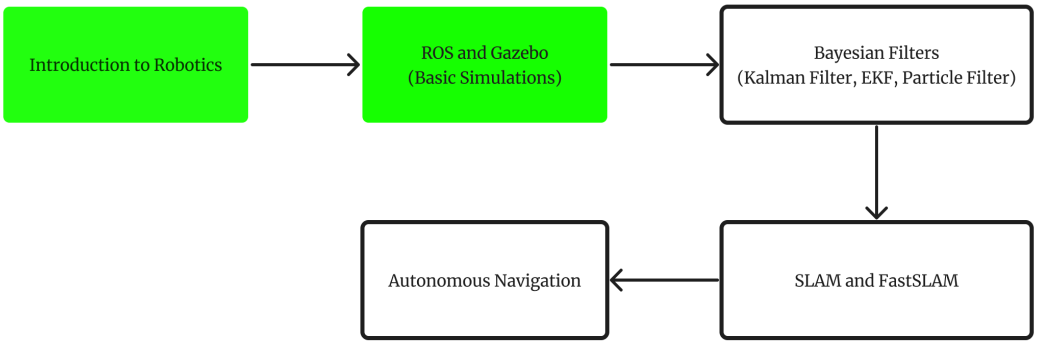


Figure 2.12: The model of Velodyne HDL-32 LiDAR created

2.5 Broad Outline Forward

The next thing studied will be different types of Bayesian Filters while finally focusing on SLAM through it



Chapter 3

Bayesian Filters

3.1 Introduction to Bayes Filter

Bayesian Filtering is a general approach for addressing the problem of estimating an unknown state based on a series of observations or measurements that are subject to uncertainty. It is a mathematical framework that is applied to probabilistic inference in the fields of robotics and artificial intelligence, particularly in situations combining state estimation and sensor fusion. It is named after the Reverend Thomas Bayes, a statistician and theologian who lived in the 18th century.

State Estimation is is the process of estimating the current state of a system (x) based on observations (z) and controls (u). Mathematically, it is defined as

$$p(x|z,u)$$

Through the Bayes filter, we will be majorly interested in doing the state estimation.

3.2 Recusive Bayes Filter

The Recursive Bayesian Filter is a fundamental concept in estimation theory. It is used for tracking dynamic systems and estimating their states over time, even in the presence of noisy sensor measurements and process dynamics. Firstly , something called belief (denoted as $bel(x_t)$) will be defined as

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t})$$

Expanding and Modifying it, we get

$$
\begin{aligned}
bel(x_t) =& p(x_t|z_{1:t}, u_{1:t}) \\
&= \eta p(z_t|x_t, z_{1:t-1}, u_{1:t}) p(x_t|z_{1:t-1}, u_{1:t}) && [Bayes' \ Rules] \\
&= \eta p(z_t|x_t) p(x_t|z_{1:t-1}, u_{1:t}) && [Markov \ Assumption] \\
&= \eta p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, z_{1:t-1}, u_{1:t}) p(x_{t-1}|z_{1:t-1}, u_{1:t}) \ dx_{t-1} \\
&= \eta p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) p(x_{t-1}|z_{1:t-1}, u_{1:t}) \ dx_{t-1} && [Markov \ Assumption] \\
&= \eta p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) p(x_{t-1}|z_{1:t-1}, u_{1:t-1}) \ dx_{t-1} && [Markov \ Assumption] \\
&= \eta p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) bel(x_{t-1}) \ dx_{t-1}
\end{aligned}
$$

Therefore, the Bayes filter can be written as a two-step process

- **Prediction Step**

$$\bar{bel}(x_t) = \int p(x_t|x_{t-1}, u_t) bel(x_{t-1}) \ dx_{t-1}$$

  where $p(x_t|x_{t-1}, u_t)$ is called the motion model

- **Correction Step**

$$bel(x_t) = \eta p(z_t|x_t) \bar{bel}(x_t)$$

  where $p(z_t|x_t)$ is called the sensor or observation model

Based on the different properties of Models (such as Linear or non-linear models for motion and observation models, parametric or non-parametric filters etc), two types of Bayesian Filters will be studied

1. **Kalman Filter and Family**
   - Only Gaussian Distributions
   - Linear or Linearized motion and observation model
2. **Particle Filter**
   - For Non-parametric Distributions
   - For Arbitrary motion and observation model

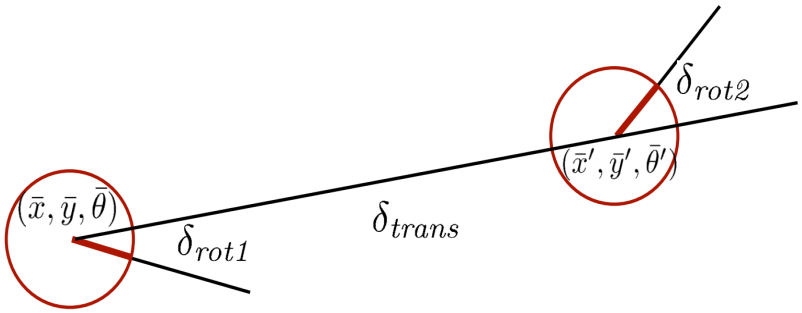
3.3 Motion Model

The Motion model specifies the posterior probability that an action u carrier the robot from x to x'. It can be written as:

$$p(x_t|x_{t-1}, u_t)$$

In practice, two types of models can be found, odometry-based models and velocity-based models.

- **Odometry Model**
  These are models for systems that are equipped with wheel encoders to get odometry information.

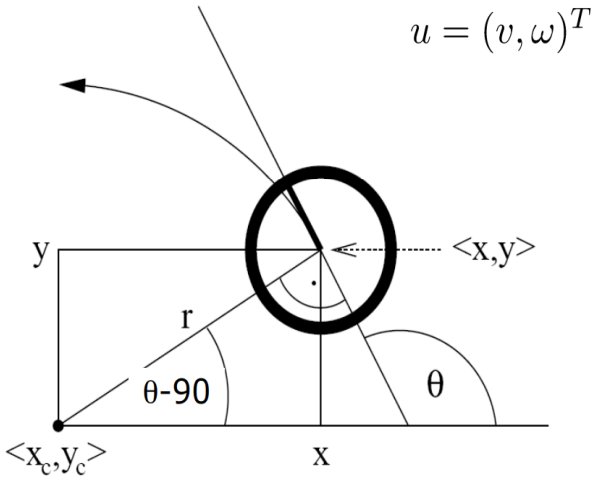


  The robot moves from $(\bar{x}, \bar{y}, \bar{\theta})$ to $(\bar{x}', \bar{y}', \bar{\theta}')$

  The odometry information that we get is $u = (\delta_{rot1}, \delta_{trans}, \delta_{rot2})$

$$\begin{aligned} \delta_{trans} &= \sqrt{(\bar{x}' - \bar{x})^2 + (\bar{y}' - \bar{y})^2} \\ \delta_{rot1} &= atan2(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta} \\ \delta_{rot2} &= \bar{\theta}' - \bar{\theta} - \delta_{rot1} \end{aligned}$$

- **Velocity Models**
  Velocity-based motion models are a key component of Bayesian Filters, such as the Kalman Filter or the Extended Kalman Filter (EKF), used in the context of state estimation, tracking, and control in robotics and autonomous systems. These models describe how a system's state, often including position and velocity, evolves over time due to control inputs. This is for systems where no wheel encoders are present, such as aerial vehicles etc where we just give a linear and angular velocity.



The robot moves from (x, y, θ) to (x', y', θ')

The Velocity: $u = (v, \omega)$

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v}{\omega}\sin\theta + \frac{v}{\omega}\sin(\theta + \omega\Delta t) \\ \frac{v}{\omega}\cos\theta - \frac{v}{\omega}\cos(\theta + \omega\Delta t) \\ \omega\Delta t + \gamma\Delta t \end{pmatrix}$$

The additional $\gamma\Delta t$ term is to account for the pose after the robot has reached its final position

3.4 Sensor Model

The Sensor model specifies the probability of sensor measurement z_t given the postion x_t of the robot. It can be written as $p(z_t|x_t)$
Say, a particular scan z consists of K measurements

$$z_t = \{{z_t}^1, ..., {z_t}^k\}$$

Then, the individual measurements are independent given the robot's position

$$p(z_t|x_t, m) = \prod_{i=1}^{k} p({z_t}^i|x_t, m)$$

Various examples of Sensor Model include Beam-Endpoint Model and Ray-cast Model, etc.

3.5 Kalman Filter

The Kalman Filter was developed by Rudolf E. Kalman in the 1960s which is an optimal estimator for linear models and Gaussian distributions. The Gaussian Distributions can be written in general as

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\Big(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\Big)$$

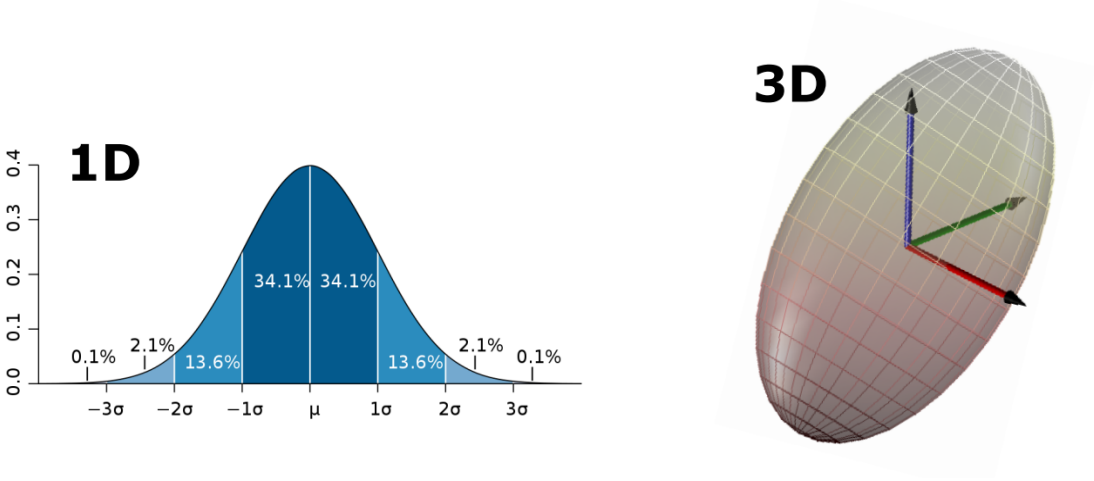


Figure 3.1: Gaussian Distributions

The Kalman filter assumes a linear transition and observation model with zero mean Gaussian Noise.

$$\begin{aligned} x_t &= A_t x_{t-1} + B_t u_t + \epsilon_t \\ z_t &= C_t x_t + \delta_t \end{aligned}$$

where

A_t : *Matrix* $(n \times n)$ *that describes how the state evolves from* $t-1$ *to* t *without controls or noise.*

B_t : *Matrix* $(n \times l)$ *that describes how the control* u_t *changes the state from* $t-1$ *to* t.

C_t : *Matrix* $(k \times n)$ *that describes how to map the state* x_t *to an observation* z_t

ϵ_t *and* δ_t :*Random variables representing the process and measurement noise that are assumed to be independent and normally distributed with covariance* R_t *and* Q_t *respectively.*

- **Linear Motion Model**

$$p(x_t \mid u_t, x_{t-1}) = \det(2\pi R_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x_t - A_t x_{t-1} - B_t u_t)^T R_t^{-1}(x_t - A_t x_{t-1} - B_t u_t)\right)$$

- **Linear Observation Model**

$$p(z_t \mid x_t) = \det(2\pi Q_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(z_t - C_t x_t)^T Q_t^{-1}(z_t - C_t x_t)\right)$$

- **Algorithm of Kalman Filter**

Kalman_filter$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$**:**

$\bar{\mu}_t = A_t\ \mu_{t-1} + B_t\ u_t$

$\bar{\Sigma}_t = A_t\ \Sigma_{t-1}\ A_t^T + R_t$

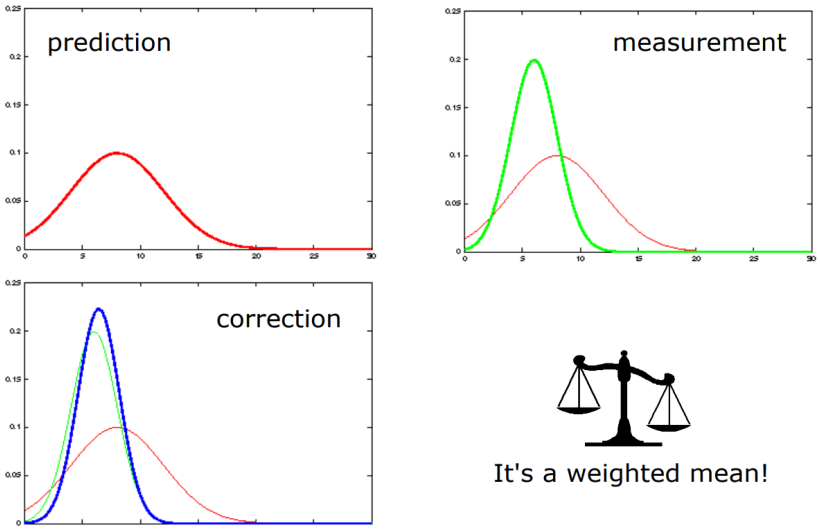
$K_t = \bar{\Sigma}_t\ C_t^T (C_t\ \bar{\Sigma}_t\ C_t^T + Q_t)^{-1}$

$\mu_t = \bar{\mu}_t + K_t(z_t - C_t\ \bar{\mu}_t)$

$\Sigma_t = (I - K_t\ C_t)\ \bar{\Sigma}_t$

return μ_t, Σ_t

Lines 2 and 3 are concerned with the prediction step and lines 4, 5, 7 are concerned with the correction step. K_t (called as the Kalman Gain) is a parameter that determines how much weight should be given to the sensor measurements versus the prior estimate when updating the state estimate in the Kalman Filter.

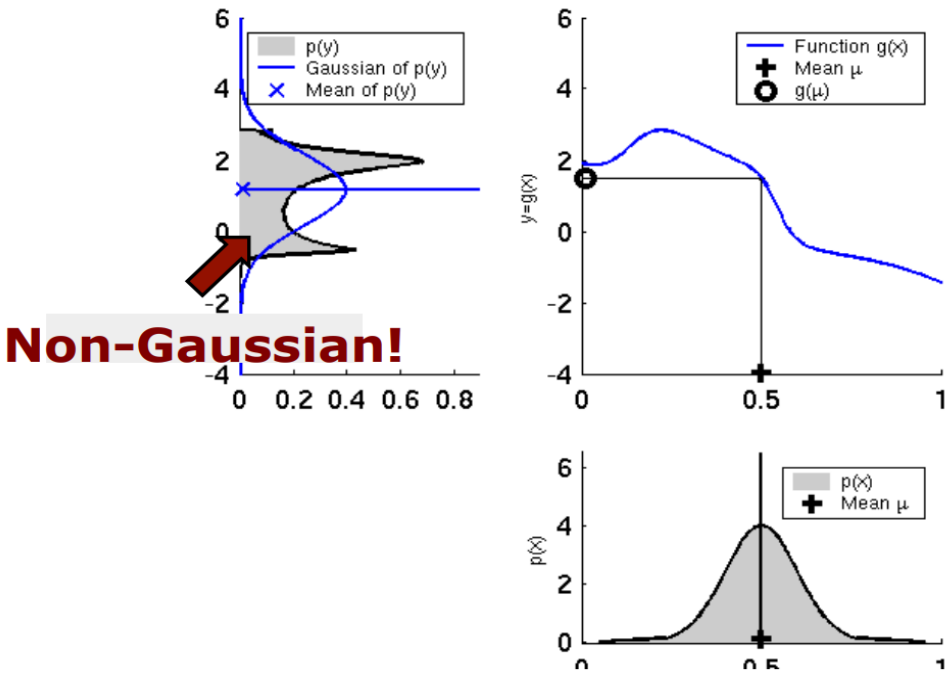


3.6 Extended Kalman Filter

The need to bring Extended Kalman Filter arose as most of the real-world problems in robotics involve nonlinear functions, whereas Kalman Filter assumes linear motion and observation models. More than that, Gaussian functions while passing through linear function gives Gaussian distribution only but in non-linear functions, Gaussian distributions does not give Gaussian distributions. Therefore Kalman Filter is not applicable anymore.

$$\begin{aligned} x_t &= \cancel{A_t x_{t-1} + B_t u_t + \epsilon_t} \rightarrow x_t = g(u_t, x_{t-1}) + \epsilon_t \\ z_t &= \cancel{C_t x_t + \delta_t} \rightarrow z_t = h(x_t) + \delta_t \end{aligned}$$

For the problem where due to non-linear motion and observation model, we loose Gaussian function, we do local linearization. We do the same by First Order Taylor Expansion by involving Jacobian matrices. By this, we can retain the gaussian nature of the distribution.



Linearization through First Order Taylor Expansion

- **Prediction Step**

$$g(u_t, x_{t-1}) \approx g(u_t, \mu_{t-1}) + \underbrace{\frac{\partial g(u_t, \mu_{t-1})}{\partial x_{t-1}}}_{=: G_t}(x_{t-1} - \mu_{t-1})$$

- **Correction Step**

$$h(x_t) \approx h(\bar{\mu}_t) + \underbrace{\frac{\partial h(\mu_t)}{\partial x_t}}_{=: H_t}(x_t - \bar{\mu}_t)$$

where G_t and H_t are Jacobian Matrices which are linear functions

1. **Linearized Motion Model**

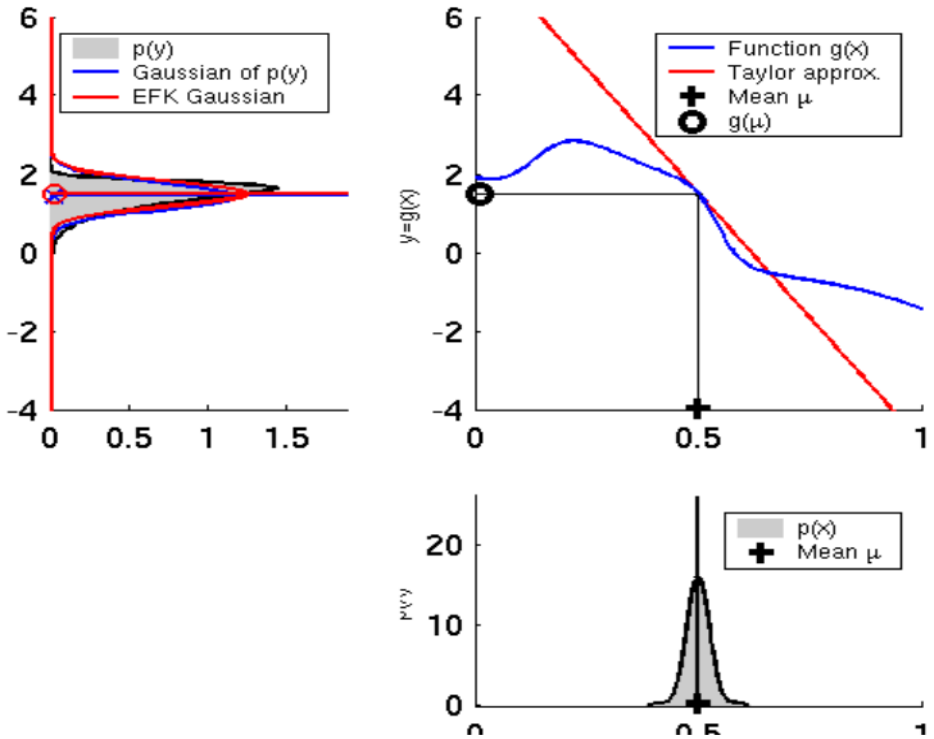
$$\begin{aligned} p(x_t|u_t, x_{t-1}) \approx\ & det(2\pi R_t)^{-\frac{1}{2}} \\ & exp(-\frac{1}{2}(x_t - g(u_t, \mu_{t-1})) - G_t(x_{t-1} - \mu_{t-1})^T \\ & R_t{}^{-1}(x_t - \underbrace{g(u_t, \mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))}_{linearized\ model}) \end{aligned}$$

where R_t describes the measurement noise

2. **Linearized Observation Model**

$$p(z_t \mid x_t) = \det(2\pi Q_t)^{-\frac{1}{2}} \exp\Big(-\frac{1}{2}(z_t - h(\bar{\mu}_t) - H_t(x_t - \bar{\mu}_t))^T Q_t^{-1}(z_t - \underbrace{h(\bar{\mu}_t) - H_t(x_t - \bar{\mu}_t)}_{\text{linearized model}})\Big)$$

where Q_t describes the measurement noise



- **EKF Algorithm**

  In the EKF Algorithm, again lines 2 and 3 are concerned with Prediction Step and lines 4, 5, 6 are concerned with the Correction Step. Here, G_t is used instead of A_t and H_t is used instead of C_t which are respective jacobian matrices.

1: **Extended_Kalman_filter**$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$**:**

2: $\bar{\mu}_t = g(u_t, \mu_{t-1})$

3: $\bar{\Sigma}_t = G_t \, \Sigma_{t-1} \, G_t^T + R_t$

4: $K_t = \bar{\Sigma}_t \, H_t^T (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$

5: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$

6: $\Sigma_t = (I - K_t \, H_t) \, \bar{\Sigma}_t$

7: *return* μ_t, Σ_t

$A_t \leftrightarrow G_t$

$C_t \leftrightarrow H_t$

KF vs. EKF

3.7 Particle Filter

Particle filters are non-parametric, recursive Bayes filters. The problem with the Kalman Filter and its variants is that it can only model Gaussian distributions while the particle fiter can deal with any arbitrary distributions. It uses multiple samples to represent distributions.The more particles fall into a region, the higher the probability of the region.

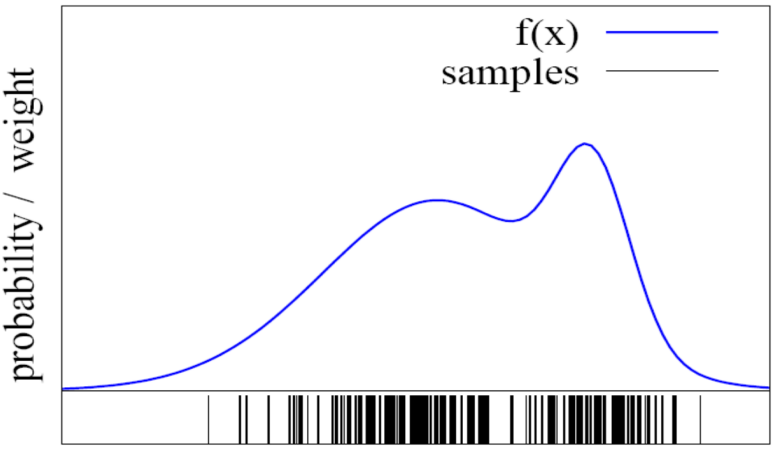


Figure 3.2: Particle filter uses samples to represent distributions

With every state hypothesis, there is an importance weight associated with it.

$$\chi = \{\langle \underbrace{x^{[j]}}_{state\ hypothesis} , \underbrace{w^{[j]}}_{importance\ weight} \rangle\}_{j=1,....,J}$$

but, the problem remains that how is any arbitrary distribution sampled? For that, we bring the Importance Sampling Principle.

- **Importance Sampling Principle**
  It says that in order to generate samples from f, a different distribution g can be used. After, the differences between g and f can be accounted for by using a weight w = f/g. Therefore, f will be called the target distribution and g will be called the proposal distribution. Note that there is also a pre-condition associated which says $\forall f(x) > 0 \ \rightarrow g(x) > 0$

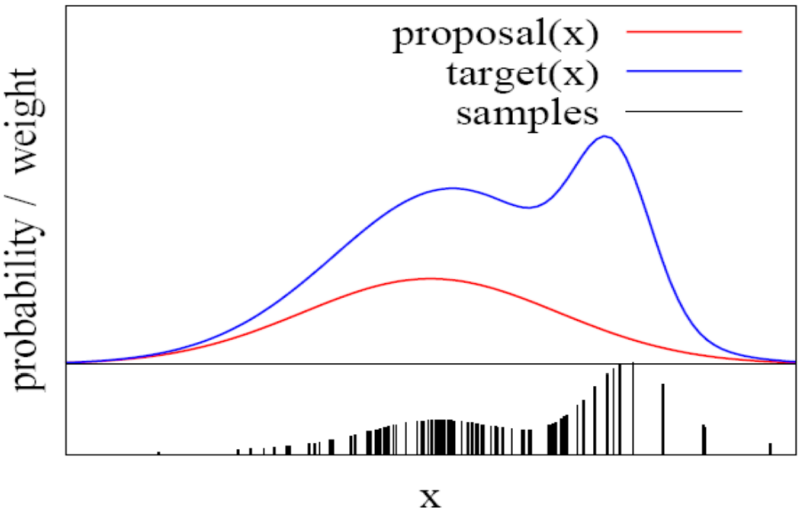


Figure 3.3: Target and Proposal distributions

The prediction step is drawing samples from the proposal and and correction step is weighting by the ratio of target and proposal.

Particle Filter Algorithm

1. Sample the particles using the proposal distribution

$$x_t{}^{[j]} \sim \pi(x_t|...)$$

2. Compute the importance weights

$$w_t{}^{[j]} = \frac{target(x_t{}^{[j]})}{proposal(x_t{}^{[j]})}$$

3. Draw sample i with probability ${w_t}^{[i]}$ and repeat J times

Particle_filter$(\mathcal{X}_{t-1}, u_t, z_t)$**:**

1: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
2: *for* $j = 1$ *to* J *do*
3: *sample* $x_t^{[j]} \sim \pi(x_t)$
4: $w_t^{[j]} = \frac{p(x_t^{[j]})}{\pi(x_t^{[j]})}$
5: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[j]}, w_t^{[j]} \rangle$
6: *endfor*
7: *for* $j = 1$ *to* J *do*
8: *draw* $i \in 1, \ldots, J$ *with probability* $\propto w_t^{[i]}$
9: *add* $x_t^{[i]}$ *to* $\mathcal{X}_t$
10: *endfor*
11: *return* $\mathcal{X}_t$

Monte Carlo Localization

Monte Carlo Localization is a very special case of Particle Filter where each particle is a pose hypothesis, the proposal distribution is the motion model and we do the correction via the observation model

$$ {x_t}^{[j]} \sim p(x_t | x_{t-1}, u_t) \ , \ {w_t}^{[j]} \propto p(z_t | x_t, m) $$

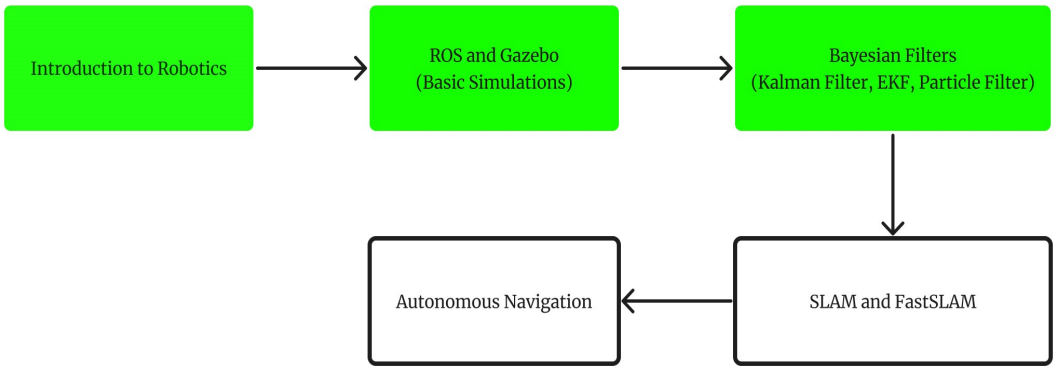
Particle_filter$(\mathcal{X}_{t-1}, u_t, z_t)$**:**

1: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
2: *for* $j = 1$ *to* J *do*
3: *sample* $x_t^{[j]} \sim p(x_t \mid u_t, x_{t-1}^{[j]})$
4: $w_t^{[j]} = p(z_t \mid x_t^{[j]})$
5: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[j]}, w_t^{[j]} \rangle$
6: *endfor*
7: *for* $j = 1$ *to* J *do*
8: *draw* $i \in 1, \ldots, J$ *with probability* $\propto w_t^{[i]}$
9: *add* $x_t^{[i]}$ *to* $\mathcal{X}_t$
10: *endfor*
11: *return* $\mathcal{X}_t$

To do resampling, there also also different ways such as Roulette wheel, Stochastic universal sampling, etc

3.8 Broad Outline Forward

After the various types of Bayesian Filters are studied, the implementation part of it will be studied which includes its use in Simultaneous Localization and Mapping (SLAM). Finally, Navigation will be studied after that.



Chapter 4

SLAM

4.1 Introduction

Simultaneous Localization and Mapping, commonly referred to as SLAM, is a fundamental and challenging problem in the field of robotics and autonomous navigation. As the word says, it can be broken down into two components, Localization and Mapping and both are done simultaneously.

1. **Localization:** Localization in robotics refers to the process of determining and continuously updating a robot's position within its environment.

2. **Mapping:** Mapping refers to the process of creating a representation of the environment, using sensor data and other information. The primary goal of mapping is to build an accurate and detailed model of the robot's surroundings, which is essential for various robotic applications, including navigation, exploration, and interaction with the environment.

SLAM is a technique that enables a robot or a device to create a map of an unknown environment while simultaneously determining its own position within that environment. SLAM is a "chicken and egg" problem: a robot needs a map to determine its position accurately, but it also needs to know its position to create the map. It is a complex task involving filtering, optimization, and data fusion techniques to estimate over time both the robot's trajectory and the environment's structure. SLAM is the basis for most navigation systems. SLAM is central to a range of indoor, outdoor, air and underwater applications for both manned and autonomous vehicles. Examples include vacuum cleaner, lawn mover at home, surveillance with unmanned air vehicles in air, reef monitoring underwater, exploration of mines underground, and terrain mapping for localization in space,

etc. SLAM is a hard problem as both robot path and map are unknown, the mapping between observations and the map is unknown, and picking wrong data associations can have catastrophic consequences

4.2 Mathematical Definition of SLAM Problem

Given:

- The robot's controls
  $u_{1:T} = \{u_1, u_2, u_3..., u_T\}$
- Observations
  $z_{1:T} = \{z_1, z_2, z_3..., z_T\}$

Wanted:

- Map of the environment
  m
- Path of the robot
  $x_{0:T} = \{x_0, x_1, x_2..., x_T\}$

As there are uncertainties in the robot's motions and observations, we will use the probability theory to explicitly represent the uncertainty. Therefore, to estimate the robot's path and the map, we can define the mathematical notation as mentioned below. This is also called as "Full SLAM"

$$p(x_{0:T}, m|z_{1:T}, u_{1:T}) \tag{4.1}$$

There is another SLAM approach called the "Online Slam" or "Incremental SLAM" which focuses on estimating the robot's current pose and updating the map in real-time as new sensor data becomes available. It is concerned with finding the robot's pose at each time step and gradually improving the map as the robot explores its environment. Online SLAM often use filtering techniques such as Extended Kalman Filter, Particle Filter, or Unscented Kalman Filter to maintain and update the state estimate. It is defined mathematically as:

$$p(x_t, m|z_{1:t}, u_{1:t}) \tag{4.2}$$

Online SLAM means marginalizing out the previous poses

$$p(x_t, m|z_{1:t}, u_{1:t}) = \int_{x_0} .... \int_{x_{t-1}} p(x_{0:t}, m|z_{1:t}, u_{1:t})\ dx_{t-1}...dx_0$$

The Integrals are typically solved recursively, one at at time.

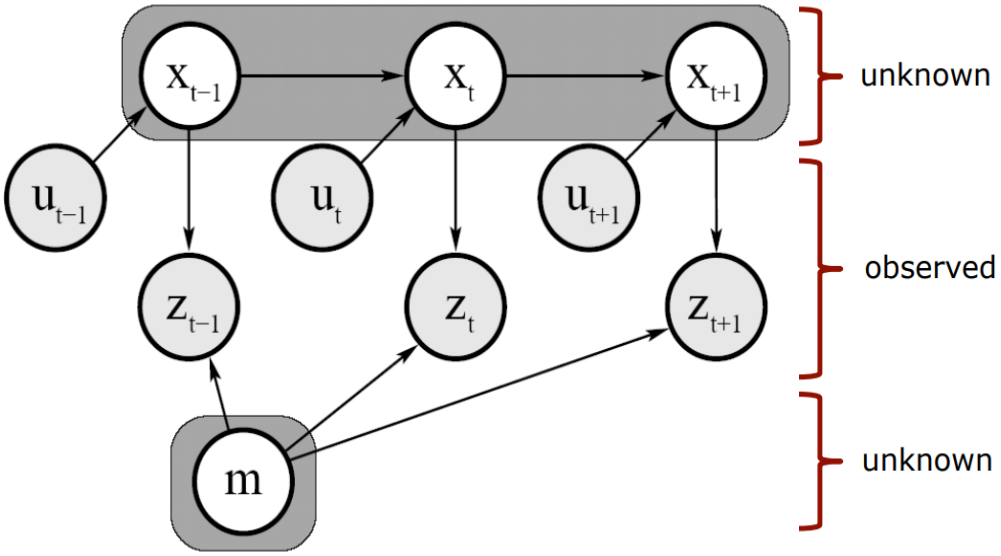


Figure 4.1: Graphical Model of SLAM

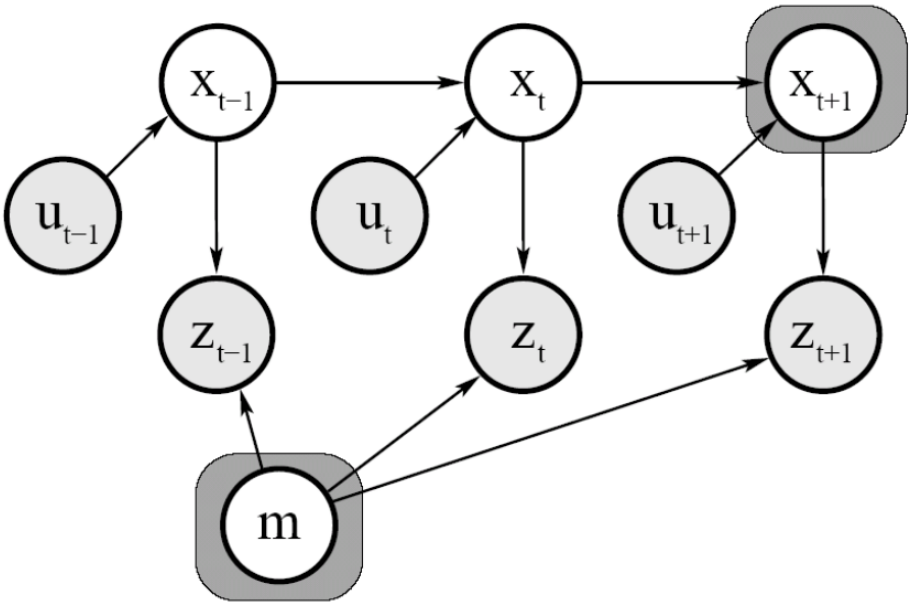


Figure 4.2: Graphical Model of Online SLAM

4.3 EKF SLAM

Historically the earliest, and perhaps the most influential SLAM algorithm is based on the extended Kalman filter, or EKF. In a nutshell, the EKF SLAM algorithm applies the EKF to online SLAM using maximum likelihood data association. In doing so, it is approximated that there are only known correspondences. The approximations exits to reduce the computational complexity.

In 2-D plane, the state space is represented by

$$x_t = (\underbrace{x, y, \theta}_{\text{robot's pose}}, \underbrace{m_{1,x}, m_{1,y}}_{\text{landmark 1}}, ..., \underbrace{m_{n,x}, m_{n,y}}_{\text{landmark } n})^T$$

The Belief can be represented by

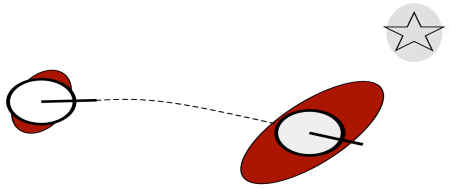
$$\underbrace{\begin{pmatrix} x \\ y \\ \theta \\ m_{1,x} \\ m_{1,y} \\ \vdots \\ m_{n,x} \\ m_{n,y} \end{pmatrix}}_{\mu} \quad \underbrace{\begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xm_{1,x}} & \sigma_{xm_{1,y}} & \cdots & \sigma_{xm_{n,x}} & \sigma_{xm_{n,y}} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{ym_{1,x}} & \sigma_{ym_{1,y}} & \cdots & \sigma_{m_{n,x}} & \sigma_{m_{n,y}} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} & \sigma_{\theta m_{1,x}} & \sigma_{\theta m_{1,y}} & \cdots & \sigma_{\theta m_{n,x}} & \sigma_{\theta m_{n,y}} \\ \sigma_{m_{1,x}x} & \sigma_{m_{1,x}y} & \sigma_{\theta} & \sigma_{m_{1,x}m_{1,x}} & \sigma_{m_{1,x}m_{1,y}} & \cdots & \sigma_{m_{1,x}m_{n,x}} & \sigma_{m_{1,x}m_{n,y}} \\ \sigma_{m_{1,y}x} & \sigma_{m_{1,y}y} & \sigma_{\theta} & \sigma_{m_{1,y}m_{1,x}} & \sigma_{m_{1,y}m_{1,y}} & \cdots & \sigma_{m_{1,y}m_{n,x}} & \sigma_{m_{1,y}m_{n,y}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_{m_{n,x}x} & \sigma_{m_{n,x}y} & \sigma_{\theta} & \sigma_{m_{n,x}m_{1,x}} & \sigma_{m_{n,x}m_{1,y}} & \cdots & \sigma_{m_{n,x}m_{n,x}} & \sigma_{m_{n,x}m_{n,y}} \\ \sigma_{m_{n,y}x} & \sigma_{m_{n,y}y} & \sigma_{\theta} & \sigma_{m_{n,y}m_{1,x}} & \sigma_{m_{n,y}m_{1,y}} & \cdots & \sigma_{m_{n,y}m_{n,x}} & \sigma_{m_{n,y}m_{n,y}} \end{pmatrix}}_{\Sigma}$$

It can written more compactly (given $x_R \to x$)

$$\begin{pmatrix} x \\ m \end{pmatrix} \begin{pmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{pmatrix}$$

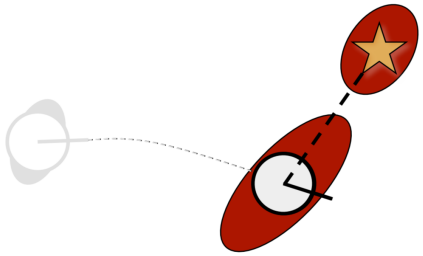
where the first matrix is the representing the mean (μ) and the second matrix is the covariance matrix (Σ). EKF SLAM will be studied in broadly whose various steps in the algorithm are

1. **State Prediction**



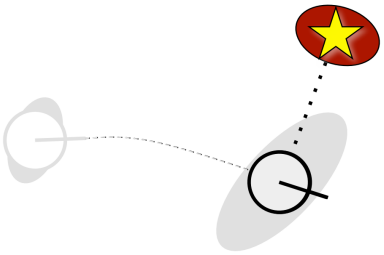
In this step, initial state prediction based on the control and previous state is being done in context to the robot.

2. **Measurement Prediction**



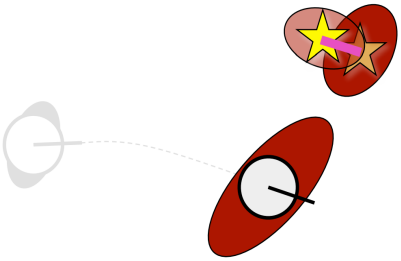
After the prediction of the state, measurement of the landmark from that state is predicted.

3. **Obtained Measurement**



The actual measurement of where the landmark is with context to the robot is done in this step.

4. **Data Association**

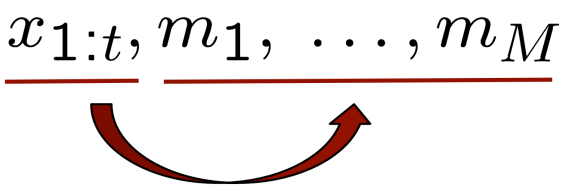


In this step, data association between the predicted landmark location and the real landmark location is done.

5. **Update Step**
   After the data association is done, the mean and covariance parameters are updated.

4.4 FastSLAM

Particle filters are effective in low dimensional spaces as the likely regions of the state space need to be covered with samples. If we use the particle set only to model the robot's path, each sample is a path hypothesis. For each sample, we can compute an individual map of landmarks.



Before that, something called Rao-Blackwellization will be studies

- **Rao-Blackwellization** This is done in order to exploit the dependencies between variables. It is a method that takes advantage of conditional expectations to create more accurate and lower-variance estimators of a parameter. We have:

  $$p(a, b) = p(b|a)p(a)$$

  If $p(a|b)$ can be computed efficiently, $p(a)$ can be represented only with samples and $p(b|a)$ can be computer for every sample. In the robot's perpective,

  $$\begin{aligned} p(x_{0:T}, m_{1:M}|z_{1:t}, u_{1:t}) &= p(x_{0:t}|z_{1:T}, u_{1:t})p(m_{1:M}|x_{0:t}, z_{1:t}) \\ &= \underbrace{p(x_{0:t}|z_{1:T}, u_{1:t})}_{Particle\ Filter} \prod_{i=1}^{M} \underbrace{p(m_i|x_{0:t}, z_{1:t})}_{2\text{-}dimentional\ EKF} \end{aligned}$$

  in FastSLAM, Each sample is a path hypothesis and older poses of the sample are not revisited again. Moreover, each landmark is represented by a 2x2 EKF and each particle therefore has to maintain M individual EKFs

The key steps and algorithm of FastSLAM can be looked upon now

Key Steps and Algorithm of FastSLAM

1. Extend the path posterior by sampling a new pose for each sample

$$x_t{}^{[k]} \sim p(x_t{}^{[k]} | x_{t-1}, u_t)$$

2. Compute the particle weight

$$w^{[k]} = |2\pi Q|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(z_t - \hat{z}^{[k]})^T Q^{-1} (z_t - \hat{z}^{[k]})\right\}$$

measurement covariance

3. Update belief of observed landmarks by the EKF update rule
4. Resample

FastSLAM1.0_known_correspondence$(z_t, c_t, u_t, \mathcal{X}_{t-1})$**:**

for $k = 1$ *to* N *do* // *loop over all particles*

  Let $\left\langle x_{t-1}^{[k]}, \left\langle \mu_{1,t-1}^{[k]}, \Sigma_{1,t-1}^{[k]} \right\rangle, \ldots \right\rangle$ *be particle* k *in* $\mathcal{X}_{t-1}$

  $x_t^{[k]} \sim p(x_t \mid x_{t-1}^{[k]}, u_t)$ // *sample pose*

  $j = c_t$ // *observed feature*

  if feature j *never seen before*

    $\mu_{j,t}^{[k]} = h^{-1}(z_t, x_t^{[k]})$ // *initialize mean*

    $H = h'(\mu_{j,t}^{[k]}, x_t^{[k]})$ // *calculate Jacobian*

    $\Sigma_{j,t}^{[k]} = H^{-1} Q_t (H^{-1})^T$ // *initialize covariance*

    $w^{[k]} = p_0$ // *default importance weight*

  else

11: *else*

12: $\hat{z}^{[k]} = h(\mu_{j,t-1}^{[k]}, x_t^{[k]})$ // *measurement prediction*

13: $H = h'(\mu_{j,t-1}^{[k]}, x_t^{[k]})$ // *calculate Jacobian*

14: EKF update $Q = H\,\Sigma_{j,t-1}^{[k]}\,H^T + Q_t$ // *measurement covariance*

15: $K = \Sigma_{j,t-1}^{[k]}\,H^T\,Q^{-1}$ // *calculate Kalman gain*

16: $\mu_{j,t}^{[k]} = \mu_{j,t-1}^{[k]} + K(z_t - \hat{z}^{[k]})$ // *update mean*

17: $\Sigma_{j,t}^{[k]} = (I - K\,H)\Sigma_{j,t-1}^{[k]}$ // *update covariance*

18: $w^{[k]} = |2\pi Q|^{-\frac{1}{2}}\ \exp\left\{-\frac{1}{2}(z_t - \hat{z}^{[k]})^T\right.$
$\left.Q^{-1}\,(z_t - \hat{z}^{[k]})\right\}$ // *importance factor*

19: *endif*

20: *for all unobserved features* j' *do*

21: $\langle \mu_{j',t}^{[k]}, \Sigma_{j',t}^{[k]} \rangle = \langle \mu_{j',t-1}^{[k]}, \Sigma_{j',t-1}^{[k]} \rangle$ // *leave unchanged*

23: *endfor*

24: *endfor*

25: $\mathcal{X}_t = \text{resample}\left(\left\langle x_t^{[k]}, \left\langle \mu_{1,t}^{[k]}, \Sigma_{1,t}^{[k]} \right\rangle, \ldots, w^{[k]} \right\rangle_{k=1,\ldots,N}\right)$

26: *return* $\mathcal{X}_t$

There is another FastSLAM technique that also considers the measurements during sampling and models from the following distribution:

$$x_t{}^{[k]} \sim p(x_t | x_{t-1}{}^{[k]}, u_{1:t}, z_{1:t})$$

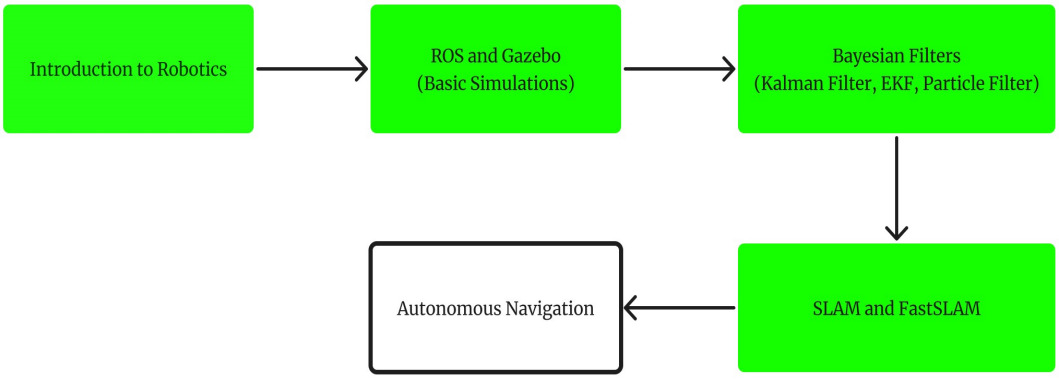
This results in a more peaked proposal distribution and less particles are required. This version of SLAM is more accurate but is more complex.

4.5 Implementation in TurtleBot3

SLAM enables TurtleBot3 to autonomously explore and build maps of its surroundings, which can be utilized for navigation or other robotic tasks. One commonly used SLAM package in ROS is the gmapping package which we will be using. It utilizes laser range finders, such as the one mounted on TurtleBot3, to perform laser-based SLAM. Gmapping combines sensor measurements and robot odometry to estimate the robot's pose and generate a 2D map of the environment.
The following are the steps that are followed

1. Simulation World is launched (TurtleBot3 World or TurtleBot3 House)
2. SLAM Node is launched and for now, gmapping is taken as SLAM method.
3. Teleoperation Node is run to move the robot. (See video in the drive link)
4. Finally, save the map as a pgm file.

4.6 Broad Outline Forward



Chapter 5

Navigation

5.1 What is Navigation?

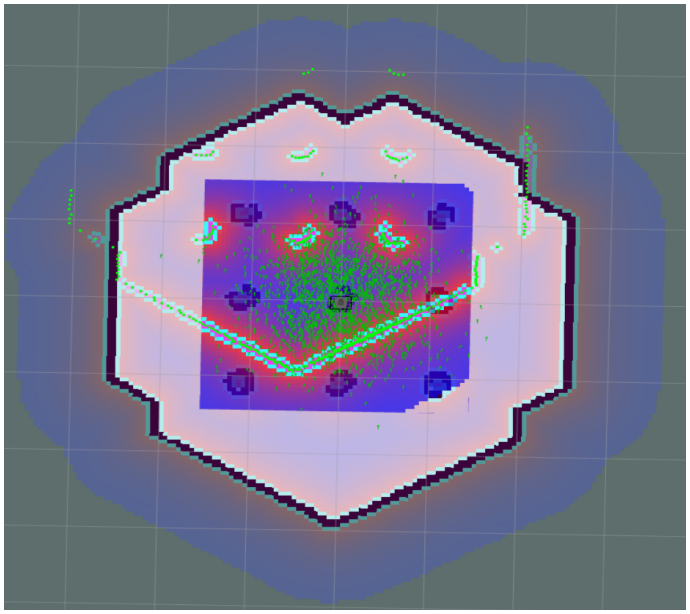
Navigation, in a general sense, refers to the process of determining and controlling the movement of a vehicle or object from one place to another. In terms of robotics, it refers to the ability of a robot to plan and execute its own movement in an environment to achieve a specific goal or task. It allows the robots to perform tasks such as exploration, mapping, and path planning without human intervention.

Navigation uses path planning algorithms such as A* (A-star), Dijkstra's algorithm, RRT (Rapidly Exploring Random Trees). More than that, various machine learning techniques, including deep learning, can be applied to robot navigation. This could involve using neural networks for perception tasks, learning-based mapping, or reinforcement learning for decision-making. For example, A* is designed to find the shortest path from a starting point (node) to a goal point (node) in a graph or grid which uses a heuristic function ($h(n)$) to estimate the cost of reaching the goal from any given node. It also uses a cost function ($g(n)$) to determine the cost of getting from the start node to a particular node and the total cost is denoted as $f(n) = g(n) + h(n)$. A* maintains a priority queue (often implemented as a min-heap) to prioritize nodes based on their total cost. Nodes with lower total costs are explored first.

Now, simple navigation for our turtlebot3 will be done on the map which was saved though SLAM. Firstly, the simulation world was launched and the Navigation node was run using the map saved before. After, 2D Pose Estimation feature of RViz to estimate the initial pose and with the teleoperation keys, we will move

to precisely locate the robot on the map. Finally, we use the 2D Nav Goal button in the RViz menu to set navigation goal and navigate the robot

.



The videos of the simulation can be found at the drive link (https://drive.google.com/drive/folders/1BvY1AvkDySe3IZkmHtzSOy3QuP9lPAJs?usp=sharing)

5.2 Autonomous Navigation

With pre-saved map

2D Pose Estimation and Navigation on RViz will now be tried to be done manually by writing a Python code for it.

- For automated 2D Pose Estimation, pose from odom topic will be taken and our turtlebot3 will be put to that position.
- After that, move_base service will be used to move our robot using the client-server architecture.

The videos of the simulation can be found at the drive link shared along with this report

init_node.py goal_pose.py

init_node.py > ...

```python
#!/usr/bin/env python3

import rospy
from geometry_msgs.msg import PoseWithCovarianceStamped
from nav_msgs.msg import Odometry

#Node Initialization
rospy.init_node('init_pose')
pub = rospy.Publisher('/initialpose', PoseWithCovarianceStamped, queue_size=1)

#Construct Message
init_msg= PoseWithCovarianceStamped()
init_msg.header.frame_id="map"

#Get Initial Pose from Gazebo
odom_msg = rospy.wait_for_message('/odom', Odometry)
init_msg.pose.pose.position.x = odom_msg.pose.pose.position.x
init_msg.pose.pose.position.y = odom_msg.pose.pose.position.y
init_msg.pose.pose.orientation.x = odom_msg.pose.pose.orientation.x
init_msg.pose.pose.orientation.y = odom_msg.pose.pose.orientation.y
init_msg.pose.pose.orientation.z = odom_msg.pose.pose.orientation.z
init_msg.pose.pose.orientation.w = odom_msg.pose.pose.orientation.w

#Delay
rospy.sleep(1)

#Publish Message
rospy.loginfo("Setting Initial Pose")
pub.publish(init_msg)
rospy.loginfo("Initial Pose is Set")
```

Figure 5.1: Code for setting initial code

init_node.py ● goal_pose.py ×

goal_pose.py > ...

```python
#!/usr/bin/env python3

import rospy
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

# Callbacks definition

def active_cb(extra):
    rospy.loginfo("Goal pose being processed")

def feedback_cb(feedback):
    rospy.loginfo("Current location: "+str(feedback))

def done_cb(status, result):
    if status == 3:
        rospy.loginfo("Goal reached")
    if status == 2 or status == 8:
        rospy.loginfo("Goal cancelled")
    if status == 4:
        rospy.loginfo("Goal aborted")


rospy.init_node('goal_pose')

navclient = actionlib.SimpleActionClient('move_base',MoveBaseAction)
navclient.wait_for_server()

# Example of navigation goal
goal = MoveBaseGoal()
goal.target_pose.header.frame_id = "map"
goal.target_pose.header.stamp = rospy.Time.now()

goal.target_pose.pose.position.x = -2.16
goal.target_pose.pose.position.y = 0.764
goal.target_pose.pose.position.z = 0.0
goal.target_pose.pose.orientation.x = 0.0
goal.target_pose.pose.orientation.y = 0.0
goal.target_pose.pose.orientation.z = 0.662
goal.target_pose.pose.orientation.w = 0.750

navclient.send_goal(goal, done_cb, active_cb, feedback_cb)
finished = navclient.wait_for_result()

if not finished:
    rospy.logerr("Action server not available!")
else:
    rospy.loginfo ( navclient.get_result())
```

Figure 5.2: Code to move the robot

Without pre-saved map

For this, a launch files is created which launches our turtlebot3, SLAM package, move_base node together. For, the video simulation, please refer to the drive link.

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="burger" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="slam_methods" default="gmapping" doc="slam type [gmapping, cartographer, hector, karto, frontier_exploration]"/>
  <arg name="configuration_basename" default="turtlebot3_lds_2d.lua"/>
  <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>

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

  <!-- SLAM -->
  <include file="$(find turtlebot3_slam)/launch/turtlebot3_$(arg slam_methods).launch">
   <arg name="model" value="$(arg model)"/>
   <arg name="configuration_basename" value="$(arg configuration_basename)"/>
  </include>

  <!-- move_base -->
  <include file="$(find auto_nav)/launch/move_base.launch">
    <arg name="model" value="$(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 auto_nav)/rviz/turtlebot3_auto_nav.rviz"/>
  </group>
</launch>
```

For the SUMMIT-XL robot, after downloading all the packages and dependencies, the whole simulation was run. It was moved using the teleop keys and the it scanned and the whole map of the enviroment was created using the gmapping package. Finally, 2D Pose Estimation and 2D Navigation Goal was done through RViz.

- **How does the 2D Nav Goal Function? (For Reference)**

  1. RViz publishes a goal message
  2. Goal message publication: publishes on topic "/move_base_simple/goal"
  3. Navigation system receives the goal message
  4. Path planning and calculation: Using algorithms such as A* (A-star), Dijkstra's algorithm, or a variant of the RRT (Rapidly Exploring Random Tree)
  5. Execution of control commands
  6. Navigation feedback

Bibliography

[1] Ashutosh Saxena, Justin Driemeyer, Justin Kearns, Andrew Y. Ng. Robotic Grasping of Novel Objects

[2] Kenechi F. Dukor, Tejumade Afonja. A Survey: Robot Grasping

[3] T. Naseer, M. Ruhnke, L. Spinello, C. Stachniss, and W. Burgard. Robust Visual SLAM Across Seasons

[4] Introduction to Robot Mapping (2013/14; Cyrill Stachniss) [Lectures]

[5] Robot Operating System (ROS) Course by Mecharithm - Robotics and Mechatronics [Lectures]

[6] ROS Tutorials - ROS Noetic For Beginners by Robotics Back-End [Lectures]

[7] https://wiki.ros.org/ROS/Tutorials

[8] https://classic.gazebosim.org/tutorials