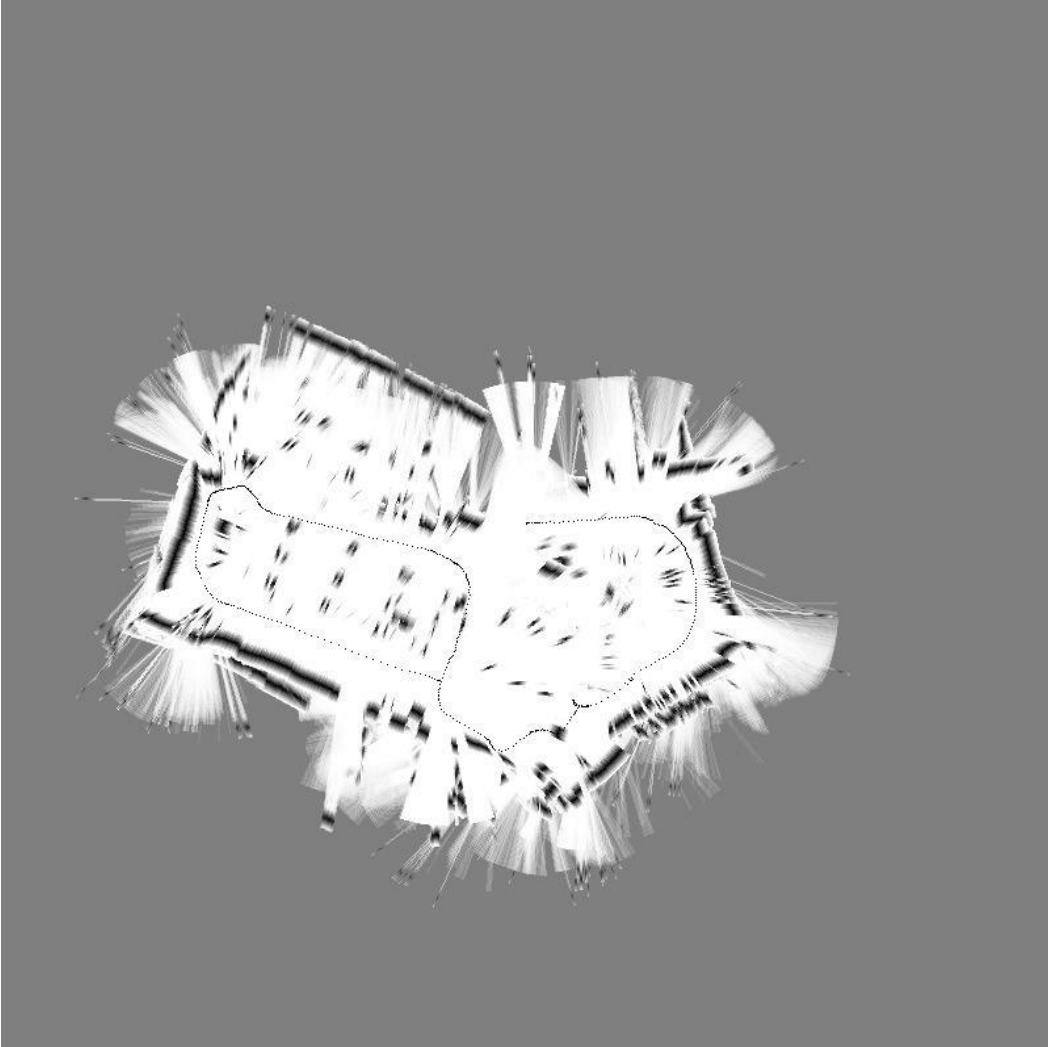


CS 421

Introduction to Robotics



Süleyman Onur Doğan

Serkan KAŞ

Table of Contents

0.Abstract	3
1.The first goal for the course	3
2.Hardware	4
3.Software	7
4.SLAM	11
5.Filters	13
6.BreezySlam	20
7.Problems and overcomes	26
8.Current Status	27
9.References	29

0.Abstract

This is the final report related to this course. So it will be a massive summer-up for the whole semester. We will cover every step that we took in this course, specifically about this project. What we learned, what we applied, how we made the plan, how much we progress etc. It's going to follow the context stated below. For more questions, you can reach any of the authors or the supervisor of this project. We plan our final report as follows. First, we will talk about what was our initial goal. Then what we learned along the way. After that, how we implement things each other. Next, what kind of problems we faced during the project session. Lastly, the current situation about the project. It has been a joyful journey that where we implement our theoretical information into real-world problems. With that being said, let's begin.

1.The First Goal for the Project

For this semester, our group decided to do a hands-on project. After seeking knowledge, the project will be based on ROS (Robotic Operating System). We will explain ROS more in the coming section. Let's talk about our project. Onur and I knew it's a good idea to learn things and implement them based on simulation. But however, it's also important to challenge ourselves for the real-world situation. That's why we willing to make a project that is not only based on simulation but also hands-on/DIY. So, as it follows, we need other components than ROS. ROS is just software where you can create your environment virtually. We also need some hardware information that we can actually build our robots. So let's talk about Hardware.

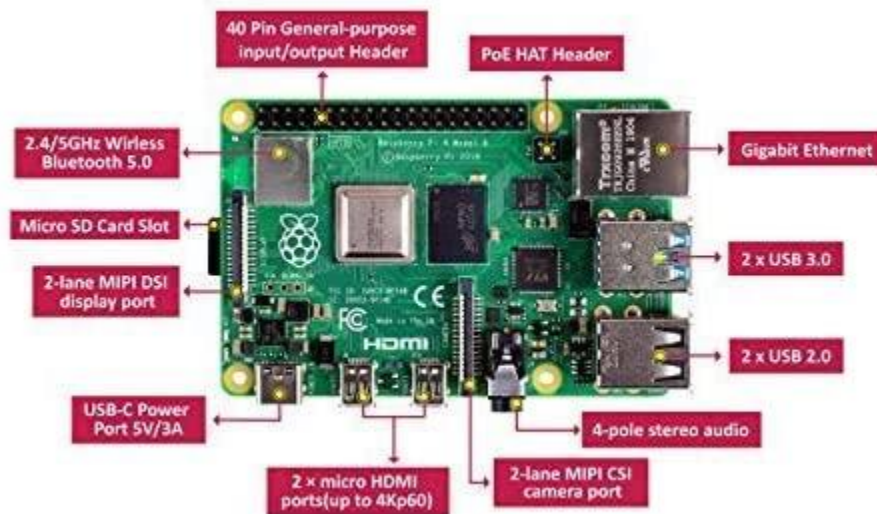
2. Hardware

Since our initial goal making a robot based on computations, we need some sort of computation environment. It can be anything from a Single board computer to Cloud environments (via Ethernet). For this purpose, we decide to work with Raspberry Pi. It is a very famous single-board computer that is usually used for this type of beginner projects. The pros of using RPi are the small form factor and easy accessibility. Also, worth mention that, RPi community is huge. So if you faced any problem, you likely to find a solution via google searching. After we decide on the computation environment, it's time to talk about, what other components we will use. We do not want to overcomplicate the project so we decided to use, DC-Motors, Lidar for sensing. Before explaining the components one by one, what is the project? The project we decide on is, making a robot that will create a map of unknown rooms automatically. So based on this decision, DC motor and Lidar will be enough for this project. And also raspberry pi. Because still, we need some sort of computation power.

2.1 Raspberry Pi

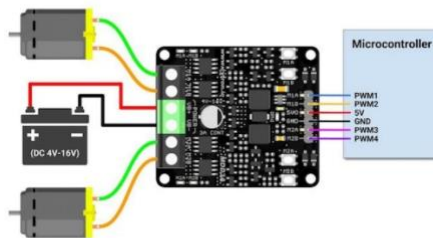
We should start describing our component with the heart element of the project. It is the one-board computer that will be used as the brain of our project. Raspberry Pi is a platform that can be connected to our external components (Lidar, motor driver, etc.). Additional to the connection advantage for external components, we will get benefit from the processing power of raspberry pi. The ROS and actual RTOS (in this case, Ubuntu OS) will proceed/work on the raspberry pi.

<https://www.raspberrypi.org/help/what-is-a-raspberry-pi/>



2.2 DC Motor and Driver

Since our aim is to make a robot that will need to move, we need some sort of motor to make this movement happened. Motors are basic devices, that convert power to mechanical movements. In our cases, the dc motor will convert the dc (electrical) power to the movement (either forward or backward). Depends on which dc motor we will use, we need a driver for that motor. What the driver does is, communicating between raspberry pi and the motor itself. So that, whatever movement wishes to be happened by raspberry pi is understood clearly by dc motor itself.



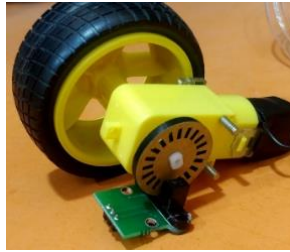
2.3 Lidar

Since one of our aims is to create a map of the environment, we will be using lidar(Light Detection and Ranging) for this purpose. For this goal, different devices may be using such as cameras as Tesla Company uses. However, we decided to use lidar. Basically, Lidar sends a laser signal 360 degrees to the environment, calculates the time of laser going from target to object. Thanks to this data, Lidar detects objects with variable distances. Lidar has 3-d representations but, we will use 2-d representations. We use RPLidar for our robot.

<http://wiki.ros.org/rplidar>



2.4 Odometry



Odometry is a basic sensor that gives the output speed of a vehicle. Based on that information, you can implement your data to find better localization. How it's going to work:

1. We will read the RPM(Revolutions per minute) of the motor
2. Then, we will convert that RPM into our sensor reading speed.
3. Then, we will multiply that spinning number with the area of the circle, $2*\pi*r$
4. Then we have almost the exact distance that one wheel made. We will apply the same for the other wheel as well.
5. After we had two-wheel velocity and displacement for each corresponded, we can make the mathematical calculations
6. Finally we can visualize where our final location is and what is our current frame.

- Kinematics

$$\begin{aligned}\Delta x &= \Delta s \cos(\theta + \Delta\theta/2) \\ \Delta y &= \Delta s \sin(\theta + \Delta\theta/2) \\ \Delta\theta &= \frac{\Delta s_r - \Delta s_l}{b} \\ \Delta s &= \frac{\Delta s_r + \Delta s_l}{2}\end{aligned}$$
$$p = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \quad p' = p + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix}$$

This is the math behind how you can use your odometer data to apply localization.

3. Software

3.1 ROS

ROS, Robot Operating System. It sounds like a usual OS, but unlike the RTOSes (Real-Time Operating System), the ROS is not an actual operating system. It's more of a framework, a package collection that makes the operations easy. After a couple of searching completed, we saw that Ubuntu OS is one of the popular environments. None of the team members has Ubuntu OS on computers, so instead of changing what we use already, we decide to install and use a virtual machine for the learning phase.

<https://www.virtualbox.org/>



We should find some examples/tutorials that can guide and understand which version of ROS and Ubuntu OS we need to install. We have a couple of examples links that we thought will be very useful for guidance.

https://www.youtube.com/watch?v=asrw-iZdgz8&list=PL8k221NjWwDe_aorjpZY7liBykHziSJ--&index=1

<https://www.youtube.com/watch?v=N6K2LWG2kRI&list=PLuteWQUgtU9BU0sQIVqRQa24p-pSBCYNv&index=1>

<https://www.youtube.com/c/JustinHuang101/videos>

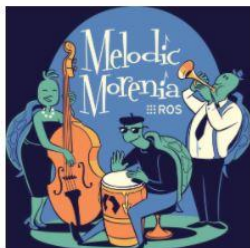
<https://learning.edx.org/course/course-v1:DelftX+ROS1x+1T2020/home>

Based on our search, we decide to install and use ROS Melodic Morenia, which is supported on Ubuntu OS 18.04 (Bionic Beaver). So we install what we need and then follow the first official tutorial(<http://wiki.ros.org/melodic/Installation/Ubuntu>) on ROS which is installing and configuring on to Ubuntu OS.

<http://wiki.ros.org/melodic/Installation>

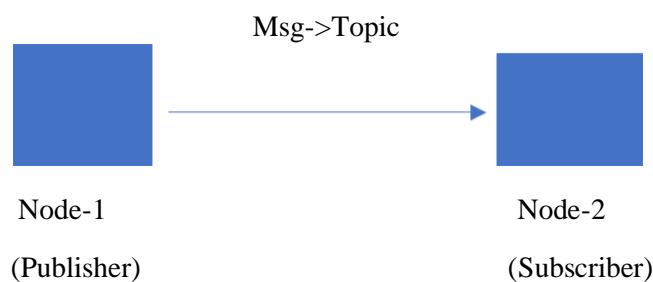
https://releases.ubuntu.com/18.04.5/?_ga=2.180484962.433194806.1616620064-1768194010.1616496774

ROS Melodic Morenia
Released May, 2018
LTS, supported until May, 2023
Recommended for Ubuntu 18.04



In robotics, people have many options for the software. People may use directly python scripts but, people also may use ROS(Robotic Operating System) for making things easier. ROS is not an actual operating system. It's more of a framework, a package collection that makes the operations easy. As we know, In the robot there are many units such as lidars, odometry, cameras, and these units have to be in communication because they need to transfer data and use these data for different applications such as SLAM, autonomous movement.

In the ROS we are defining our units as nodes. The main aim of ROS is that supply communication between nodes. Nodes are basically executable files within a ROS package. Nodes can publish or subscribe to a Topic. For instance, Lidar will be providing, publishing data, and another unit of the robot will be getting and using this data. In our ROS system, lidar is the publisher and another unit of the robot is a subscriber.



We did an example for Publisher and subscriber node and we have sent a message to the publisher to the subscriber. We used string as a message type but, in the next step of the project, we will create different message types or use different message types.

```

import rospy
from std_msgs.msg import String

def simplePublisher():

    simple_publisher =
    rospy.Publisher('topic_1', String,
    queue_size = 10)

    rospy.init_node('node_1')

    rate = rospy.Rate(10) #hz

    message = "my first ROS topic"

    while not rospy.is_shutdown():
        simple_publisher.publish(message)
        rate.sleep()

if __name__ == '__main__':
    simplePublisher()
  
```

EXPLANATION OF PUBLISHER CODE:

In the simplePublisher function we created our node and message. Also, we published that message.

simple_publisher variable contains our topic name, type of message(String), queue size.

Queue_size: Limits the amount of queued messages if any subscriber is not receiving them fast enough.

rospy.init_node('node_1'): we clarified our Publisher node.

rate = rospy.Rate(10) : With the help of its method sleep(), it offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second.

message variable contains our message.

After we defined necessary statements we created a while loop and send a message to the subscriber

Code for creating Subscriber:

```
import rospy
from std_msgs.msg import String

def stringListenerCallback(data):
    rospy.loginfo(' The contents of topic1:',
    data)

def stringListener():
    rospy.init_node('node_2' )

    rospy.Subscriber('topic_1' ,
String, stringListenerCallback)

    rospy.spin()
```

```
if __name__ == '__main__':
    stringListener()
```

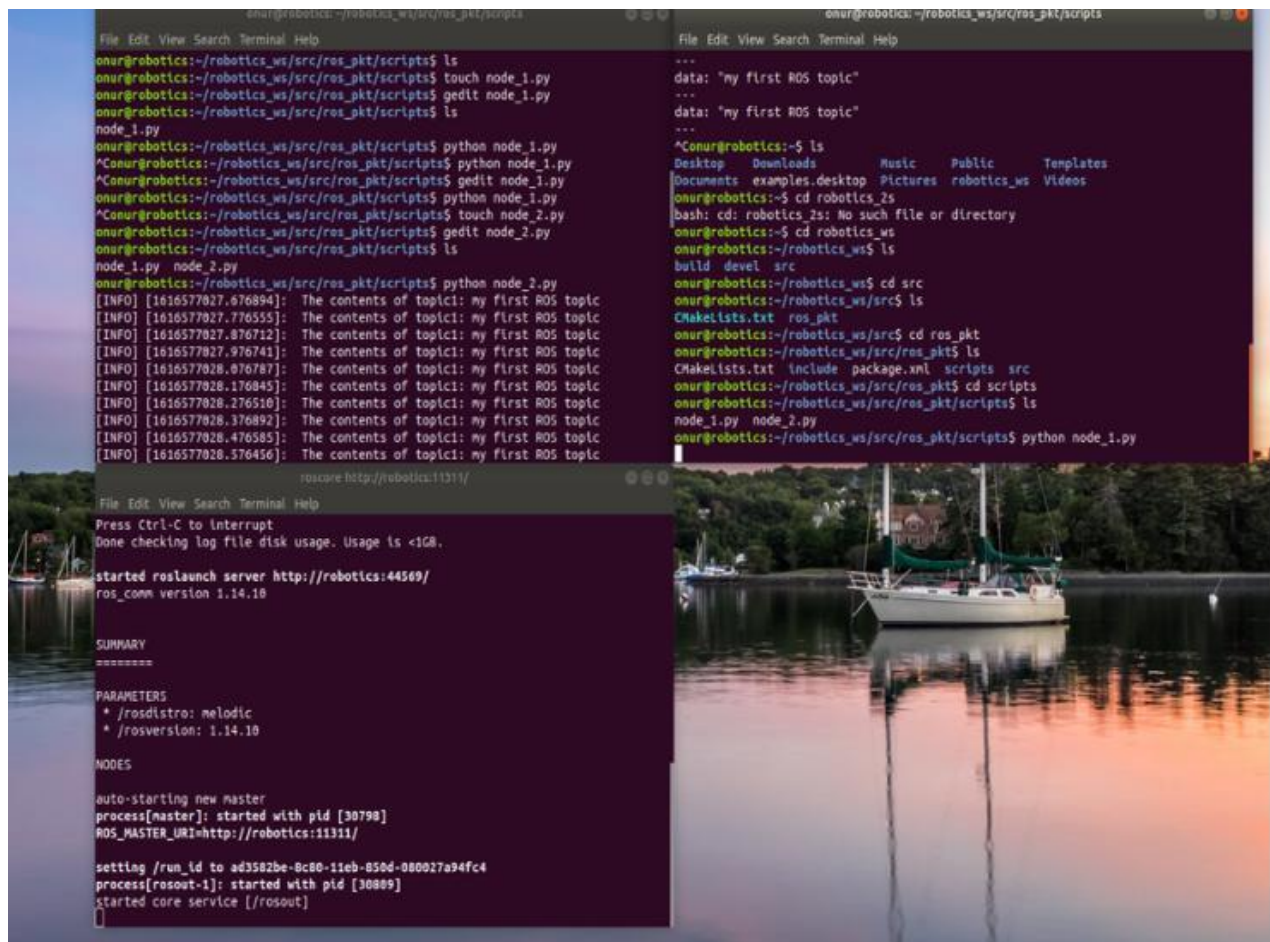
EXPLANATION OF SUBSCRIBER CODE:

In the stringListener function we defined our node as a subscriber and the topic which subscriber gets.

rospy.Subscriber('topic_1',String, stringListenerCallback): Defined topic,type of topic, and output.

rospy.spin(): keeps python from exiting until this node is stopped

With **stringListenerCallback** function we defined feedback output of subscriber



```
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ ls
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ touch node_1.py
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ gedit node_1.py
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ ls
node_1.py
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ python node_1.py
^Conur@robotics:~/robotics_ws/src/ros_pkt/scripts$ python node_1.py
^Conur@robotics:~/robotics_ws/src/ros_pkt/scripts$ gedit node_1.py
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ python node_1.py
^Conur@robotics:~/robotics_ws/src/ros_pkt/scripts$ touch node_2.py
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ gedit node_2.py
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ ls
node_1.py node_2.py
onur@robotics:~/robotics_ws/src/ros_pkt/scripts$ python node_2.py
[INFO] [1616577027.676894]: The contents of topic1: my first ROS topic
[INFO] [1616577027.776555]: The contents of topic1: my first ROS topic
[INFO] [1616577027.876712]: The contents of topic1: my first ROS topic
[INFO] [1616577027.976741]: The contents of topic1: my first ROS topic
[INFO] [1616577028.076787]: The contents of topic1: my first ROS topic
[INFO] [1616577028.176845]: The contents of topic1: my first ROS topic
[INFO] [1616577028.276510]: The contents of topic1: my first ROS topic
[INFO] [1616577028.376892]: The contents of topic1: my first ROS topic
[INFO] [1616577028.476585]: The contents of topic1: my first ROS topic
[INFO] [1616577028.576456]: The contents of topic1: my first ROS topic

roscore http://robotics:11311/
File Edit View Search Terminal Help
Press Ctrl-C to Interrupt
Done checking log file disk usage. Usage is <1GB.
started roslaunch server http://robotics:44569/
ros_comm version 1.14.10

SUMMARY
=====
PARAMETERS
 * /rostdistro: melodic
 * /rosversion: 1.14.10

NODES
auto-starting new master
process[master]: started with pid [30798]
ROS_MASTER_URI=http://robotics:11311/

setting /run_id to ad3582be-8c80-11eb-850d-080027a94fc4
process[roscout-1]: started with pid [30809]
started core service [/roscout]
```

On the right top, we can see that our Publisher node is running, in the left top, our subscriber node is getting the message. For this process to happen, the roscore has to be running, we can see roscore in the bottom left.

At the beginning of our project, We were considering using ROS but, we used python directly.

4. SLAM (Simultaneous Localization And Mapping)

In the project, we aimed to create a robot that can do move on an unknown map. For this application, the robot will move on an unknown map so, the robot has to know the location but, for location, the robot needs to know the map. At that point we need to understand mapping, localization, and simultaneous localization and mapping (SLAM). So,

Mapping: estimating a map of the environment given location.

Localization: estimating the position of robot given map

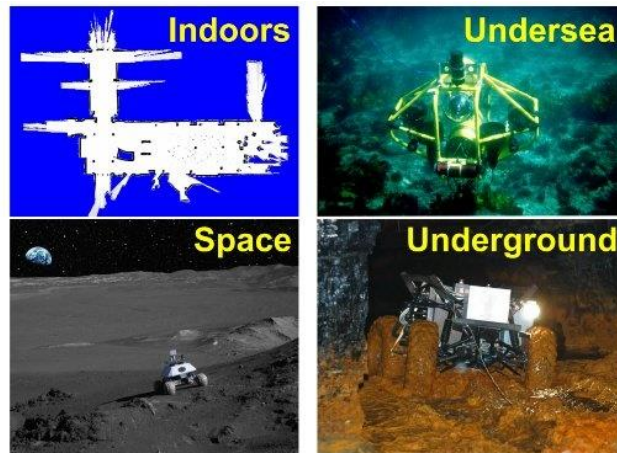
What if a robot tries to estimate position and estimating a map of the environment at the same time in the unknown map?

Let me give you an example:

We are standing middle of class and we have a laser, we know where is a laser, it is in the middle of calls with us and When we use a laser to measure where is Wall? It is mapping, because we know the position of the laser and estimating where is Wall. What if we don't know where the laser is? It is a combination of mapping and localization because localizations says us where is a laser, and mapping says us where is Wall according to the position.

In this case, we can use simultaneous localization and mapping (SLAM). Basically, Slam is the Estimate of the pose of a robot and the map of the environment at the same time. However, SLAM is a hard problem because we need a map for localization and we need a position estimate for mapping. It is like a chicken-or-egg problem.

SLAM has different applications:



There is 3 main solution for SLAM:

- 1-Kalman Filter Based SLAM
- 2-Particle Filter Based SLAM
- 3-Graph Based SLAM

In the Kalman Filter Based SLAM, Kalman filter using to estimate robot position.

In the Particle Filter Based SLAM, the Particle filter using to estimate robot position.

In the Graph-Based SLAM, Positions of the robot are shown as nodes and creating links between nodes then estimates robot position.

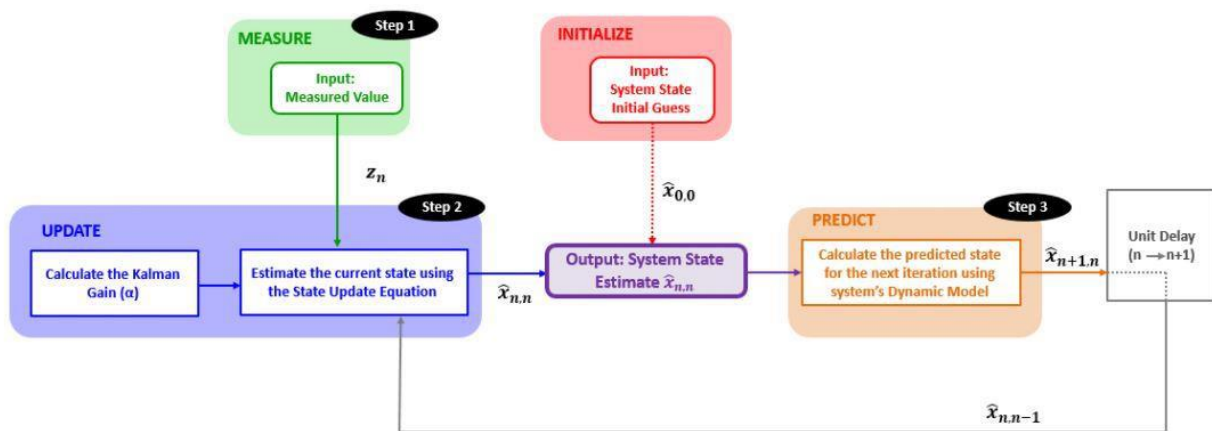
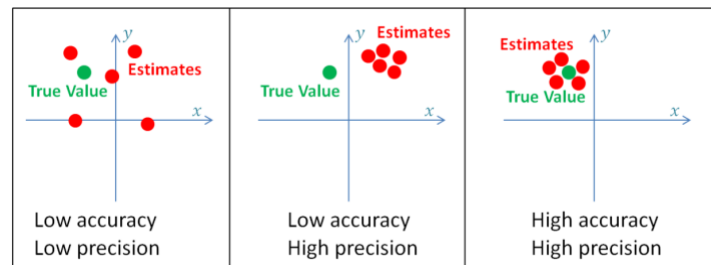
Every SLAM algorithm has different advantages and disadvantages. For example, some of the SLAM algorithm based on camera some of them based on Lidar and some of them uses particle filter or Kalman filter. This kind of change makes differences.

5. Filters

5.1 the Kalman Filter

Before we dive into the Kalman filter, we need to make sure that we know some basic concepts:

- 1) Mean: It means average. It is useful if we know the values if they are not hidden. μ
- 2) Expected Value: It's also the average value, but in this case, the values are hidden. Mean by, maybe it measured a couple of times with some errors. So we don't know the exact measurement(value) but with all measurements, we can expect something from it. E
- 3) Variance: It's a measure of spreading of the data set from its mean. σ^2
- 4) Standard Deviation: Square root of variance. σ
- 5) Distribution: Like Gaussian, PDF (Probability Density Function), CDF (Cumulative Distribution Function)
- 6) Random Variable: The variable values are based on experiment outcomes, but don't know exactly what is it.
- 7) Estimate: Prediction of true values, Accuracy: Indicates how successful your estimation. Precision: The distribution.



From here I can continue with this website:

<https://www.kalmanfilter.net/kalman1d.html>

<https://www.kalmanfilter.net/kalmanmulti.html>

<https://towardsdatascience.com/kalman-filter-an-algorithm-for-making-sense-from-the-insights-of-various-sensors-fused-together-ddf67597f35e>

We created a demo for the Kalman, both 1d and 2d.

EXAMPLE FOR 1-D:

In the demo, We have an object that has acceleration and moves at the x-axis. We will measure the position of the object and track an object with Kalman Filter. We are giving these parameters:

DT:time for 1 cycle

std_acc: standard deviation of the acceleration

std_meas: standard deviation of the measurement

U: The control input.

Position:

$$x_k = x_{k-1} + \dot{x}_{k-1}\Delta t + \frac{1}{2}\ddot{x}_{k-1}(\Delta t)^2$$

Velocity:

$$\dot{x}_k = \dot{x}_{k-1} + \ddot{x}_{k-1}\Delta t$$

State Equation:

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ \dot{x}_{k-1} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \\ \Delta t \end{bmatrix} \ddot{x}_{k-1}$$

Measurement Equation:

$$\mathbf{z}_k = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} + \mathbf{v}_k$$

As we can see, we defined our A, B, and F in the Kalman filter state-space model.

With deviation we can get variance:

For state equation:

$$\mathbf{Q} = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix} \sigma_a^2$$

For measurement equation:

$$R = \sigma_z^2$$

We have defined this information in the code part. After that, predict and update part of the Kalman filter defined in code.

```
@author: onurdogan
"""

import numpy as np
import matplotlib.pyplot as plt

class KalmanFilter(object):
    def __init__(self, dt, u, std_acc, std_meas):

        self.dt = dt
        self.u = u
        self.std_acc = std_acc

        self.A = np.matrix([[1, self.dt],
                             [0, 1]])
        self.B = np.matrix([[self.dt**2]/2, [self.dt]])
        self.H = np.matrix([[1, 0]])
        self.Q = np.matrix([[self.dt**4)/4, (self.dt**3)/2],
                             [(self.dt**3)/2, self.dt**2]) * self.std_acc**2

        self.R = std_meas**2

        self.P = np.eye(self.A.shape[1])

        self.x = np.matrix([[0], [0]])

    def predict(self):

        self.x = np.dot(self.A, self.x) + np.dot(self.B, self.u)

        self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q
        return self.x

    def update(self, z):

        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R

        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))

        self.x = np.round(self.x + np.dot(K, (z - np.dot(self.H, self.x))))

        I = np.eye(self.H.shape[1])
        self.P = (I - (K * self.H)) * self.P

dt = 0.1
t = np.arange(0, 35, dt)

real_track = 0.1*((t**2) - t)

u= 2
std_acc = 0.8
std_meas=1.8

kf = KalmanFilter(dt, u, std_acc, 1.8)

predictions = []
```

```

self.x = np.matrix([[0], [0]])

def predict(self):

    self.x = np.dot(self.A, self.x) + np.dot(self.B, self.u)

    self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q
    return self.x

def update(self, z):

    S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R

    K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))

    self.x = np.round(self.x + np.dot(K, (z - np.dot(self.H, self.x))))

    I = np.eye(self.H.shape[1])
    self.P = (I - (K * self.H)) * self.P

dt = 0.1
t = np.arange(0, 35, dt)

real_track = 0.1*((t**2) - t)

u= 2
std_acc = 0.8
std_meas=1.8

kf = KalmanFilter(dt, u, std_acc, 1.8)

predictions = []
measurements = []
for x in real_track:
    z = kf.H * x + np.random.normal(0, std_meas)

    measurements.append(z.item(0))
    predictions.append(kf.predict()[0])
    kf.update(z.item(0))

fig = plt.figure()

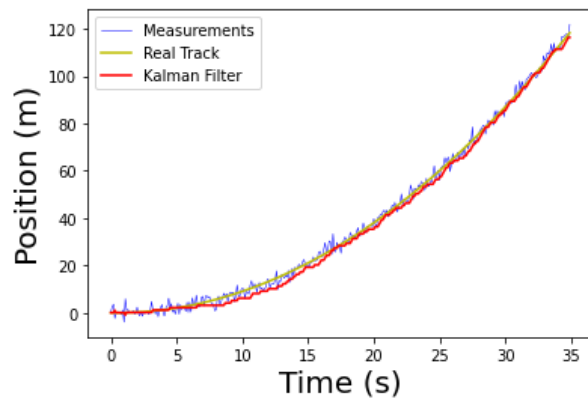
plt.plot(t, measurements, label='Measurements', color='b', linewidth=0.5)

plt.plot(t, np.array(real_track), label='Real Track', color='y', linewidth=1.5)
plt.plot(t, np.squeeze(predictions), label='Kalman Filter ', color='r', linewidth=1.5)

plt.xlabel('Time (s)', fontsize=20)
plt.ylabel('Position (m)', fontsize=20)
plt.legend()
plt.show()

```

Output:



$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2.$$

Also, we calculate Mean Square error end of the code.

```
print("mse: ", (mean_squared_error(np.array(real_track), np.squeeze(predictions))))
```

PS: we import sklearn.metrics to use mean_squared_error function.

As we can see in the output, There is noise in measurements and the Kalman filter works very well. We predict position via acceleration for this demo, we created another demo to predict position via velocity. For this demo, we just changed our state equation and other parts of the code are the same.

$$\begin{bmatrix} x_n \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ \dot{x}_{n-1} \end{bmatrix} + \begin{bmatrix} v_{n,1} \\ v_{n,2} \end{bmatrix}.$$

$$\mathbf{Q}_n = \sigma_x^2 \begin{bmatrix} \frac{(\Delta t)^3}{3} & \frac{(\Delta t)^2}{2} \\ \frac{(\Delta t)^2}{2} & \Delta t \end{bmatrix}.$$

$$\mathbf{F}_n = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix},$$

2-D KALMAN FILTER:

If our object moves x and y axis so, we will use 2-D Kalman Filter. There is two extra input.

1. std_meas_y: standard deviation of the measurement for y axis.
2. U_y : The control input

Also, A, B, H matrixes changes for multi dimension and Q, R calculation changes.

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ \dot{x}_k \\ \dot{y}_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \dot{x}_{k-1} \\ \dot{y}_{k-1} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}(\Delta t)^2 & 0 \\ 0 & \frac{1}{2}(\Delta t)^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} \ddot{x}_{k-1} \\ \ddot{y}_{k-1} \end{bmatrix}$$

$$\mathbf{z}_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ \dot{x}_k \\ \dot{y}_k \end{bmatrix} + \mathbf{v}_k$$

$$\mathbf{Q} = \begin{bmatrix} \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} & 0 \\ 0 & \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & 0 & \Delta t^2 & 0 \\ 0 & \frac{\Delta t^3}{2} & 0 & \Delta t^2 \end{bmatrix} \sigma_a^2$$

$$\mathbf{R} = \begin{matrix} & \begin{matrix} x & y \end{matrix} \\ \begin{matrix} x \\ y \end{matrix} & \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \end{matrix}$$

```
import numpy as np
import matplotlib.pyplot as plt

class KalmanFilter(object):
    def __init__(self, dt, u_x, u_y, std_acc, x_std_meas, y_std_meas):
        self.dt = dt

        self.u = np.matrix([[u_x], [u_y]])

        self.x = np.matrix([[0], [0], [0], [0]])

        self.A = np.matrix([[1, 0, self.dt, 0],
                             [0, 1, 0, self.dt],
                             [0, 0, 1, 0],
                             [0, 0, 0, 1]])

        self.B = np.matrix([[((self.dt**2)/2), 0],
                             [0, ((self.dt**2)/2)],
                             [self.dt, 0],
                             [0, self.dt]])

        self.H = np.matrix([[1, 0, 0, 0],
                             [0, 1, 0, 0]])

        self.Q = np.matrix([[((self.dt**4)/4), 0, (self.dt**3)/2, 0],
                             [0, ((self.dt**4)/4), 0, (self.dt**3)/2],
                             [(self.dt**3)/2, 0, self.dt**2, 0],
                             [0, (self.dt**3)/2, 0, self.dt**2]]) * std_acc**2

        self.R = np.matrix([[x_std_meas**2, 0],
                             [0, y_std_meas**2]])

        self.P = np.eye(self.A.shape[1])

    def predict(self):
        self.x = np.dot(self.A, self.x) + np.dot(self.B, self.u)

        self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q
        return self.x[0:2]

    def update(self, z):
        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))

        self.x = np.round(self.x + np.dot(K, (z - np.dot(self.H, self.x)))) #
        I = np.eye(self.H.shape[1])

        self.P = (I - (K * self.H)) * self.P #
        return self.x[0:2]
```

For our project, we may not need to use a 2D filter because our robot will not move the x and y plane at the same time. That's why we didn't prepare any demo for the 2D filter.

The Kalman Filter Usage:

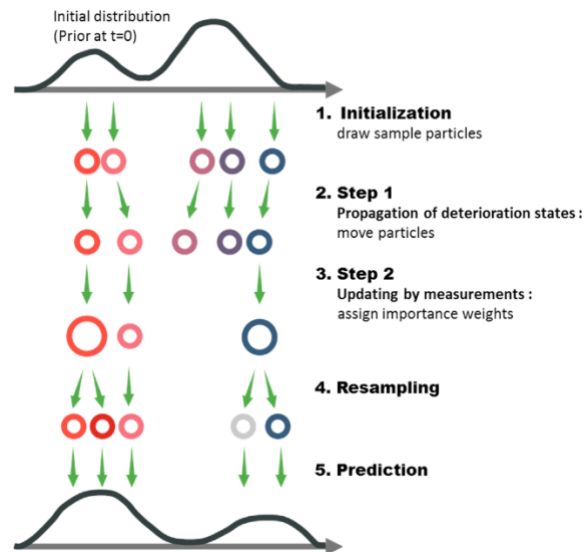
- 1) Position Estimation: With all the movement estimation and correction in a specific time, we can visualize where our robot could be according to the first position.
- 2) Motion Estimation: With noisy odometry sensing, we can predict, and almost visualize the full motion of our robots
- 3) Localization: Since we now bot position and motion at the same time, we can guess where our localization is with (X and Y coordinates) work frame (the Angle relation between base frame)

5.2 Particle Filter

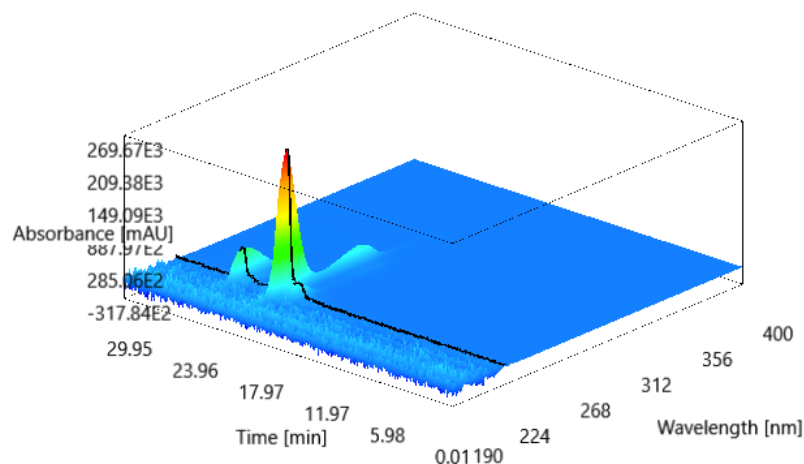
So how does the Particle filter works?

Well it is complicated in terms of word, but I suggested watching people video which is made by MATLAB(https://www.youtube.com/watch?v=NrzmH_verBU)

For a simple term, it works as shown in the picture.



Initially, the nodes were distributed without any weights. Then, it crossed with a signal. From that, some nodes gain weights and some nodes lose weights. Before going further to the next step, it reduces the nodes. Less weighted nodes are deleted, and weighted nodes are left as they were. Then, nodes are distributed based on their weights. The purpose of this application, make nodes even (without any weight distribution). You left with the same amount of nodes as the start. But now, it is not distributed evenly as the first step. It's distributed at more on point which has more potential to be likely. For 2 dimensional expect, you can visualize as in the picture.



6. BreezySlam

In our Project, We wanted to create a robot that constructs a map of the room while simultaneously keeping track of the position of a moving robot. For this goal, We will be using SLAM(2.3). However, There are SLAM algorithms based on computer vision by the camera, and algorithms that use a laser sensor or sonars. As we mentioned in 3.1, There are SLAM algorithms based on Kalman Filter, Particle Filter or Rao-Blackwellized particle filters, Graph based.

In choosing step of the SLAM algorithm, we have decided to have SLAM algorithm which is :

- Lidar Based
- Particle filter Based
- In Python

While we are choosing our SLAM algorithm we reviewed some Lidar Based algorithms[1],[2] and checked resources. We found very different algorithms such as GMapping, DP-SLAM. but, we wanted to use BreezySLAM. BreezySLAM is a Lidar-Based SLAM in Python. Also, In our Project, we wanted to use python so, BreezySLAM is one of the best options. BreezySLAM uses C extension for optimization.

Basically, BreezySLAM uses CoreSLAM[3] with the RMHC algorithm. Before getting into any detail, we need to understand CoreSLAM for better understanding of BreezySLAM.

CoreSLAM: CoreSLAM developed by Bruno Steux, Oussama El Hamzaoui. CoreSLAM is a laser-based approach created to be simple. In the Particle filter, the CoreSLAM algorithm depends on DP-SLAM [4]. However, In contrary to DP-SLAM, CoreSLAM decided to use only one map so, CoreSLAM tracks a single hypothesis about the world state. CoreSLAM algorithm[3] can work by using odometry and Lidar or just Lidar. However, the stand-alone version of CoreSLAM uses odometry and Lidar so, in our simulations, we have focused on odometry and Lidar.

The algorithm is divided into two different steps: distance calculation and update of the map[2]. In the first step, for each incoming scan, it calculates

the distance-based Particle Filter. Thus, for odometer changes of x , y , and q , a particle filter applies the error model and obtains for particle i ,[2]

$$\begin{aligned}x_i &= a_x * x + b_x + \mathcal{N}(0, \sigma_x) \\y_i &= a_y * y + b_y + \mathcal{N}(0, \sigma_y) \\\theta_i &= a_\theta * \theta + b_\theta + \mathcal{N}(0, \sigma_\theta)\end{aligned}$$

The method changes it by the addition of independent normally distributed random values to each of the robot state's components (x , y , θ). Then it makes a hypothesis about the best robot

position. The hypothesized position is checked by matching the current laser scan data in it against the map state.

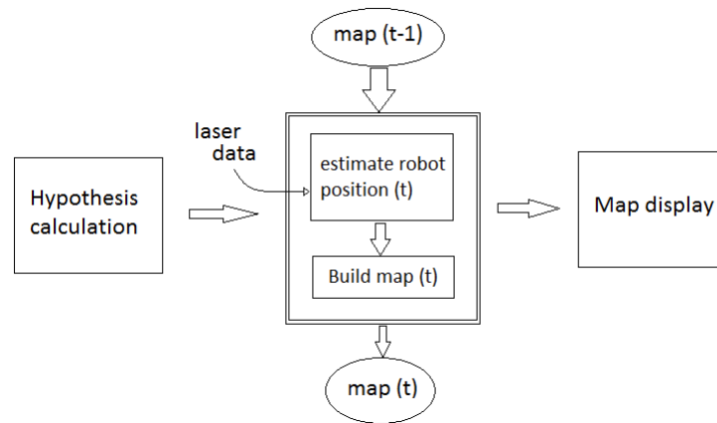
Distance reported by lidar is d and reported distance is d' so the probability density of observing discrepancy is normally distributed.[2]

$$\delta = d' - d$$

The total posterior for particle i

$$P_i = \prod_k P(\delta_{ik} | s_i, m) \quad [2]$$

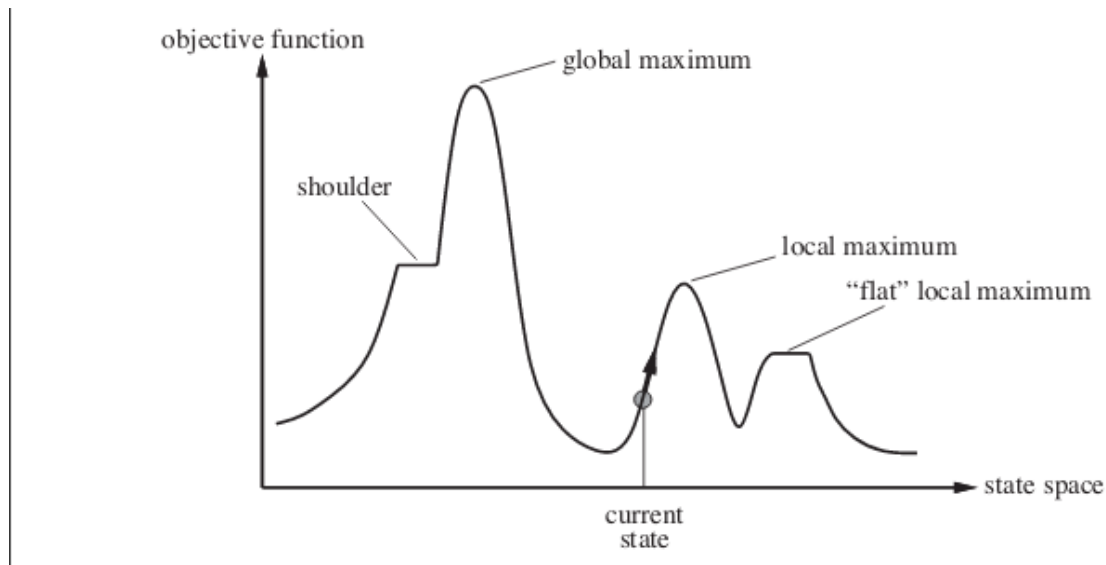
Also, the matching function computes the sum of range measurements that hit occupied map cells. If the hypothesis has a better matching score then it becomes the current robot pose. CoreSLAM uses the Monte-Carlo algorithm[5] to match the current scan with the map and to retrieve the updated position of the robot.



In the update part, the update function builds a map. According to the creator of CoreSLAM, Building a map compatible with a particle filter is not straightforward so, they used a grey-level map. When an obstacle is detected, the algorithm does not draw one single point to show the obstacle, instead, the algorithm draws an adjustable set of points surrounding the obstacle.

BreezySLAM python implementation of CoreSLAM. Also, there is a rmhc slam in BreezySLAM, rmhc slam uses RMHC (Random-Mutation Hill-Climbing) algorithm to CoreSLAM for optimization. We will be using rmhc SLAM but, during the report, we will mention rmhc slam as a BreezySLAM for preventing any confusion.

So, how RMHC works?



From the picture expect, let say we are at the current state. The algorithm work like this:

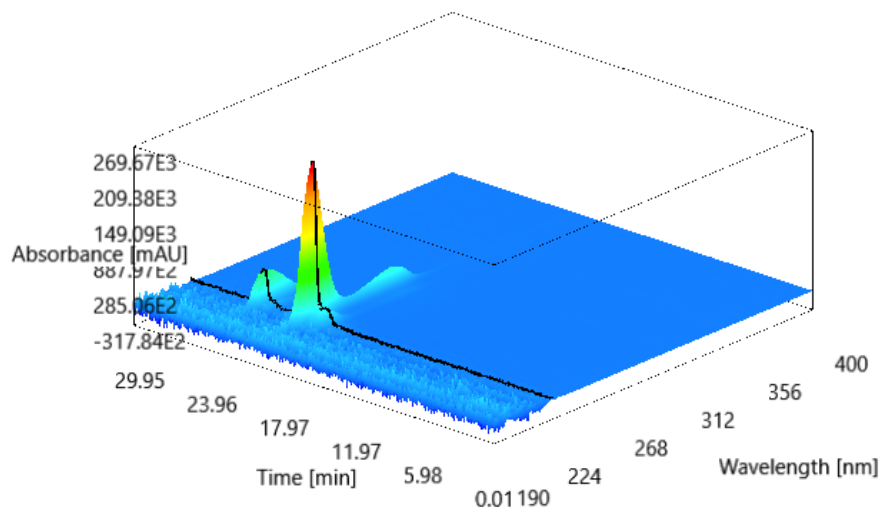
1-) from the current state, take a small step for both sides and check if one of the values is bigger than the current state or not. In this case, the right-hand side is bigger than the current state.

2-) Bigger value state is set as the new current state, and repeat these two steps until reaching the peak of the value.

3-) When we reach the peak and check both side values, none of the values will be bigger than the current state. So we reach the local maximum and this is how Hill Climbing works.

But note that, the local maximum value may not be the answer to the global maximum value.

In our past reports, we described codes which we will use. However, We will describe functions and code examples in different ways because we had some changes in our Project.



How it's going to work with our project?

Well, after particle filter works, with weight distribution, let's say we have this kind of result. The beginning location of the map. As you can see in here there is three option, two of them less likely but one of them is very likely. So how we can select which one is our solution or initial position at the beginning. This decision was made by the Hill Climb algorithm.

```
if odometries is None:
    # Update SLAM with lidar alone
    slam.update(lidars[scanno])
else:
    # Convert odometry to velocities
    velocities = robot.computePoseChange(odometries[scanno])
    # Update SLAM with lidar and velocities
    slam.update(lidars[scanno], velocities)
# Get new position
pose[0],pose[1],pose[2] = slam.getpos()
# Get new map
slam.getmap(mapbytes)
```

This is SLAM related part of the code. We will be explaining this code and the functions. As we mentioned before, CoreSLAM can work with lidar or odometry and lidar. In the If conditions, we are choosing this. We will focus on the model which uses both.

We have 3 function here for slam:

.update()

.getpos()

.getmap()

Update Function:

The main idea of the update function comes from CoreSLAM.

```
def update(self, scans_mm, pose_change, scan_angles_degrees=None, should_update_map=True):
    """
    Updates the scan and odometry, and calls the the implementing class's _updateMapAndPointcloud method with
    the specified pose change.

    scan_mm is a list of Lidar scan values, whose count is specified in the scan_size
    attribute of the Laser object passed to the CoreSlam constructor
    pose_change is a tuple (dxy_mm, dtheta_degrees, dt_seconds) computed from odometry
    scan_angles_degrees is an optional list of angles corresponding to the distances in scans_mm
    should_update_map flags for whether you want to update the map
    """

    # Convert pose change (dxy,dtheta,dt) to velocities (dxy/dt, dtheta/dt) for scan update
    velocity_factor = (1 / pose_change[2]) if (pose_change[2] > 0) else 0 # units => units/sec
    dxy_mm_dt = pose_change[0] * velocity_factor
    dtheta_degrees_dt = pose_change[1] * velocity_factor
    velocities = (dxy_mm_dt, dtheta_degrees_dt)

    # Build a scan for computing distance to map, and one for updating map
    self._scan_update(self.scan_for_mapbuild, scans_mm, velocities, scan_angles_degrees)
    self._scan_update(self.scan_for_distance, scans_mm, velocities, scan_angles_degrees)

    # Implementing class updates map and pointcloud
    self._updateMapAndPointcloud(pose_change[0], pose_change[1], should_update_map)
```

This is an update function, as we can see this function updates scan then gives pose changes to another function `_updateMapAndPointCloud` for estimating new position.

```
def _updateMapAndPointCloud(self, dxy_mm, dtheta_degrees, should_update_map):
    """
    Updates the map and point-cloud (particle cloud). Called automatically by CoreSLAM.update()
    velocities is a tuple of the form (dxy_mm, dtheta_degrees, dt_seconds).
    """

    # Start at current position
    start_pos = self.position.copy()

    # Add effect of velocities
    start_pos.x_mm += dxy_mm * self._costheta()
    start_pos.y_mm += dxy_mm * self._sintheta()
    start_pos.theta_degrees += dtheta_degrees

    # Add offset from laser
    start_pos.x_mm += self.laser.offset_mm * self._costheta()
    start_pos.y_mm += self.laser.offset_mm * self._sintheta()

    # Get new position from implementing class
    new_position = self._getNewPosition(start_pos)

    # Update the current position with this new position, adjusted by laser offset
    self.position = new_position.copy()
    self.position.x_mm -= self.laser.offset_mm * self._costheta()
    self.position.y_mm -= self.laser.offset_mm * self._sintheta()

    # Update the map with this new position if indicated
    if should_update_map:
        self.map.update(self.scan_for_mapbuild, new_position, self.map_quality, self.hole_width_mm)
```


This is updateMapAndPointcloud function. This function gets position changes and estimates the new position with the _getNewPosition function. For estimating a new position, it uses another function which is the _getNewPosition function.

```
def _getNewPosition(self, start_position):
    """
    Implements the _getNewPosition() method of SinglePositionSLAM. Uses Random-Mutation Hill-Climbing
    search to look for a better position based on a starting position.
    """

    # RMHC search is implemented as a C extension for efficiency
    return pybreezyslam.rmhcPositionSearch(
        start_position,
        self.map,
        self.scan_for_distance,
        self.laser,
        self.sigma_xy_mm,
        self.sigma_theta_degrees,
        self.max_search_iter,
        self.randomizer)
```

This is _getNewPosition function. This function is under the rmhc_slam class. This function estimates new positions via the rmhc position search algorithm and particle filter. As we can see, It gets different parameters such as map.

These functions correspond to the distance calculation step of CoreSlam except rmhc position search algorithm. CoreSlam doesn't use the rmhc position algorithm but, BreezySLAM uses it. This point makes difference between CoreSLAM and BreezySLAM.

getpos Function: This function gives us x,y, and theta of position to display the position in the map.

getmap Function: It fills the map via map bytes.

7. Problems and overcomes

We started to build our robot. As a start, we should install OS to our robot brain (Raspberry Pi Model 4B 4GB Ram model).

<https://www.direnc.net/raspberry-pi-4-model-b-4gb-en>

We install Ubuntu (desktop version) from the ubuntu official website.

<https://ubuntu.com/raspberry-pi>

The flashing SD Card is a straightforward process. With the help of 3rd party programs, we easily manage to install Ubuntu OS into our SD Card. Then we plugged our sd card and installed the necessary libraries and set it up for BreezySlam.

Naturally, we were faced with some problems.

1-) Tkinter installing issue.

from the setup.py, it couldn't handle the installing Tkinter itself, so we install Tkinter with "sudo pip3 install tk". we use pip3 instead of pip because our main codes work with python3+ version, since we are not using any virtual environment for python packages, we directly install Tkinter onto our main python3 files. *IF virtual environment is installed, and set it up based on venv, then consider connecting virtual environment first.

2-) Couldn't connect lidar to RPI

On the first day, we were at school, and couldn't connect lidar to raspberry pi no matter what we try. Later on, when we arrive home, we try with another cable (we thought maybe that's the issue).

And finally, Lidar is shown up at "\$ lsusb". But still, we couldn't communicate with lidar with the "/dev/ttyUSB0" path, which is a problem. Somehow, we need to create communication through this path, but the mount function is also not worked. In our case, the "modprobe ftdi_sio" function is worked for us. It may not be installed in your os automatically. So consider install if needs.

3-) BreezySlam crashing

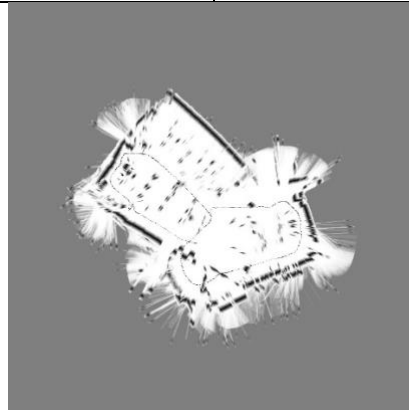
Still, we are working on this issue. Lidar seems to send more data than our RPi handle for unit time. We started to change parameters to set the correct data speed for our RPi, but we couldn't solve it yet. Currently, it seems like the first frame is getting from lidar but then, it crashes.

8 CURRENT STATUS

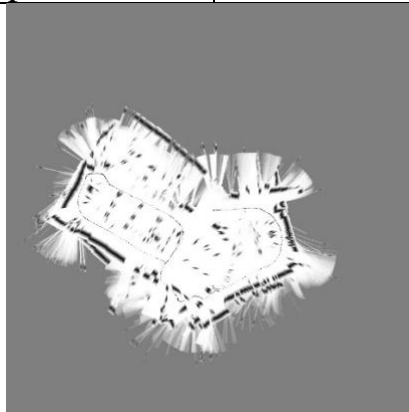
We couldn't find any solution for these errors and we wanted to create a simulation on BreezySLAM and started to work on the simulation. We found odometry and scan data from Paris Mines Tech on the Github page of BreezySLAM. BreezySLAM can test with different parameters such as the amount of particle, the number of max search iteration (about rmhc algorithm). In our demo, we will change the number of particles in the demonstration to see the difference. We gave 9999 for the amount of particle because the amount of particle in the demo which is in breezyslam is 9999 as a default setting.

In total, we have 2 different experiments. The room of experiments is the same but, the path of the robot is different. We decided our main project will work by using odometry and lidar so, we will be giving lidar and odometry data.

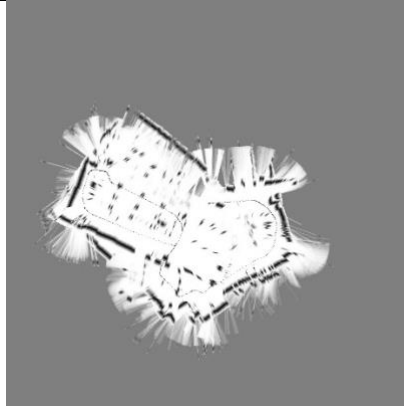
Experiment Number	Amount of particle
Exp1	9999



Experiment Number	Amount of particle
Exp2	9999



Experiment Number	Amount of particle
Exp2	50



As we said, the room is the same but, the path of the robot is different and we can see that.

->We changed the amount of particle for exp2(9999 to 50) and we see some differences. More detailed inferences can be made based on the results.

Also, As you can see in the final map, obstacles are represented by multiple points rather than a single point so, constructed maps are not looking like traditional maps. We mentioned this difference before and we can see the difference in the results of simulations.

To sum up, BreezySlam has different options to make SLAM happen. With that all variation, you can visualize slight differences, which makes SLAM better or worse depending on your expectation. It gives you more options for computation. It's good because if you have a weaker component (for hardware) you can tweak the inputs, and get the best result for yourself. This project can be further via;

- 1-) Solving the hardware problems
- 2-) Add more sensors (camera, etc.) for more complex SLAMs
- 3-) It could complete computation on cloud servers.
- 4-) It could feature more. Like, object tracking(and moving) or serving via robotic arms, and many more.

9.References

- [1]Cyrill Stachniss, Udo Frese, Giorgio Grisetti- OpenSLAM
 - [2] An Evaluation of 2D SLAM Techniques Available in Robot Operating System
 - [3]Bruno Steux, Oussama El Hamzaoui- CoreSLAM : a SLAM Algorithm in less than 200 lines of C code
 - [4]Austin Eliazar and Ronald Parr. Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. Proceedings of the International Joint Conference on Artificial Intelligence, 2003.
 - [5] S. Thrun, D. Fox, W. Burgard, F. Dellaert, Robust Monte Carlo Localization for Mobile Robots, In Artificial Intelligence, 128, 2001.
 - [6] Chandan Hegde, Nirmala S Gupta Implementation of Mapping Algorithm for SLAM Operation
- <https://github.com/simondlevy/BreezySLAM>
- <https://openslam-org.github.io/>