1804336

maxim shinskiy  University of Essex

CE315 mobile robotics – Assignment 2

Table of Contents

[Introduction 2](#_Toc67474081)

[Navigation 2](#_Toc67474082)

[Path Planners 3](#_Toc67474083)

[Software Design 4](#_Toc67474084)

[Results 5](#_Toc67474085)

[Velocity-Time Graph 5](#_Toc67474086)

[Robot Trajectory 5](#_Toc67474087)

[Laser Data Based Map 6](#_Toc67474088)

[Appendix 6](#_Toc67474089)

[Code 6](#_Toc67474090)

[References 11](#_Toc67474091)

# Introduction

## Navigation

Navigation is a process of controlling the motion of the mobile robot to transport it from point A to B. This includes different methods that vary depending on the situation and a task of a specific robot. The simplest of all the methods that are going to be discussed would be odometry or also known as dead reckoning.

Odometry is based on the idea of using internal motor encoder, which is a electro-mechanical device to convert angular position or motion into an electrical signal[1]. Each time the process starts off from reading encoders, then going though several steps of calculating difference of current position in relation to previous. This is done with a use of trigonometric and/or kinematic equations afterwards, knowing the previous position and current it is possible to calculate next position i.e. . This method however has a number of possible errors both systematic and non-systematic. Systematic errors would be ones that happen regularly and are introduced due to inaccuracy. In mobile robotics these happen as a result of any physical imperfection in a construction of a robot. These can be a misalignment of wheelbase or backlash in motors and gearboxes. This error propagates together with the trajectory and increases uncertainty with time. Others are non-systematic and cannot be predicted or controlled much, they happen more random. For example, it would be difficult to predict heterogeneity of the floor surface.

Due to these errors, dead reckoning becomes very unreliable when is implemented in precision tasks. Some planned trajectories (e.g. 90-degree turn) are not going to be accurate and small errors will accumulate into a large one. Different methods may be used in order to increase accuracy and precision.

One of such methods would be the gyrodometry[2]. With this functionality the gyroscopic sensor and its data are used in the instances when predicted position of a robot (from odometry) differs from gyros data substantially. This also eliminates the gyro drift, which is also one of non-systematic error in gyroscopic sensor. Gyrodometry approach in this[2] study does not make use of Kalman filtering because it cannot predict non-systematic errors produced from motion surface (e.g. floor) and instead is “based on hypothesis that the discrepancy between odometry curve and the gyro curve persists only over a very short amount of time”, - study says.

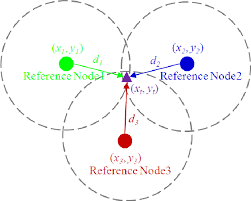
Nevertheless, Extended Kalman Filter (EKF) is a power algorithm to maintain precision in a navigation system. It consists of a three-step recursive operation. It starts from **prediction**, which is the first step in the algorithm. EKF predicts next position of the robot using odometry.

Where , is the Jacobean matrix of the transition function. Step 2 is **observation**, which is obtaining measurements from external sensors such as (Gyro, LIDAR, other proximity sensors etc.). This is done so it can be compared with predictions in step 3, which is **matching and correction**.After finding the difference in the observations and predictions, EKF calculates the innovation and its covariance is:

Where , is the Jacobean matrix of the measurement function. However, every measurement passes through a validation gate which is represented by an inequality:

and is only accepted if the inequality is true, otherwise disregarded. The filter gain, robot state and covariance are then updated and EKF is back at step 1.

Other navigation methods are using Kalman filter to track features in robot environment and locate the robot. One of such is SLAM for Simultaneous Localisation and Mapping. It combines odometry with EKF to reduce uncertainty in the position estimation to dynamically build the map of environment. After the map is built, the robot can travel around it with a knowledge of obstacles. Or else, the map can be pre-loaded and therefore there is going to be no need to explore the area. The robot can use sensors to collect data from environment and compare it to either pre-loaded or generated map.

 Another way of minimising error from odometry is beacons navigation. This method uses at least three beacons to send the signal and one beacon on the robot to receive it. Afterwards with the use of trilateration the robot is able to position itself relative to the beacons. Knowing the distance to every beacon (node) and the trajectory within beacons signal, robot can calculate the difference between its position and the trajectory to correct the path.

In terms of this assignment odometry was only used to plot the trajectory of the bot, but not to navigate it through the environment. Instead LIDAR sensors where used to detect the environment around and use the surrounding as such “beacon”. This means that knowing the distance to an edge or obstacle the robot can understand its location in relation to this object and define its behaviour based on it.

## Path Planners

The navigation part tells robot where it is, but path planning tells the robot how to get to certain point. Path planning algorithm is responsible for computing a way to the goal avoiding the obstacles and building a trajectory. However, not always path planning requires computing the trajectory, in the same way that was done with maps in navigation, the same can happen here.

The simplest way however very task and environment specific is a fixed path. It requires pre-loading the possible landmarks that are connected into a chain. The robot will follow the chain and the rest may be manual or autonomous. In this assignment, this approach was chosen to reach the goal. The robot was instructed on how to react to several changes in the sensor activity. Hence, the robot is following the desired path.

In other situations, the robot needs to find the path to follow autonomously. In this case, the graph with nodes and edges with weights can be plotted to represent the path. Edges and weight represent the part of path and a weight (distance or similar) of such path. To find the most optimal path out of all, different algorithms can be implemented. Dijkstra’s algorithm is one of the best single-destination methods to estimate the minimum path to reach destination point on the road network[3].

Similar approach is A\* search algorithm. It is a refinement of the edges and nodes search. To find the branch with lowest cost (distance) it uses principles of dynamic programming, by which it deletes the paths with greater cost. It also uses heuristics to guide its search.

# Software Design

Diagram

Description automatically generatedThe flow chart shows the logic of the code. It starts from initializing constants like speed values. Then it opens files to write the laser map data, odometry and velocity data. After a while loop checks for an external interrupt, if such does not happen, the program receives data from LIDAR and dead reckoning data. Laser data is used for navigation, it is a condition in a robot path planning. Odometry is used for plotting a trajectory of a robot and it’s velocity after the end of a run. There is a switch statement inside the while loop, it defines what stage the robot is in currently. Each stage is a part of planned trajectory, that the robot follows during the simulation. If current laser range data obeys the if condition for current stage, this means that the robot is still in the same stage and it has to stay in it, otherwise, change to next state. In the end of each iteration the program writes the data to files. After the program is interrupted, the stream buffers are closed meaning that the program is terminated.

# Results

## Velocity-Time Graph

From the graph it is clear to say that there were 2 speed constants, 0.8 cm/s for high and 0.2 cm/s for low. The robot moves on high speed at the parts when the trajectory is in the straight line and slows down when higher accuracy is required (e.g. turns).

## Robot Trajectory

Trajectory graph, obtained by odometry, show the path that the robot followed. Spacing of the the reading show the velocity at which the robot was going at that part of a path. In here it shows that the maximum speed was at the regions with a straight-line path. It also shows that the robot reached the charger (goal position).

## Laser Data Based Map

Plotting the environment map was not the priority of the robot’s task and it was collecting data in addition to the main goal, reaching the charger. The gap at the top left occurred because that part of the wall was blocked by an obstacle and therefore the LIDAR couldn’t reach there. In case of that obstacle, the same happened, LIDAR couldn’t reach the part of the cylinder that was not facing the robot. A few points in the wall at the bottom mean low sampling rate because the robot was moving at high speed. Two obstacles at the bottom right were also not facing the robot. To increase the accuracy of the map, the robot could move at lower speed and change the trajectory, so it catches the parts that are missing on the map at the moment.

# Appendix

## Code

#include "ros/ros.h"

#include "geometry\_msgs/Twist.h"

#include "nav\_msgs/Odometry.h"

#include <fstream>

#include <time.h>

#include <iomanip>

#include "sensor\_msgs/LaserScan.h"

**using** **namespace** std**;**

ofstream odomVelFile**;**

struct EulerAngles**{**

double roll**,** pitch**,** yaw**;**

**};**

struct Quaternion**{**

double w**,** x**,** y**,** z**;**

**};**

EulerAngles ToEulerAngles**(**Quaternion q**){**

EulerAngles angles**;**

//roll (x-asis rotation)

double sinr\_cosp **=** **+**2.0 **\*** **(**q**.**w **\*** q**.**x **+** q**.**y **\*** q**.**z**);**

double cosr\_cosp **=** **+**1.0 **-** 2.0 **\*** **(**q**.**x **\*** q**.**x **+** q**.**y **\*** q**.**y**);**

angles**.**roll **=** atan2**(**sinr\_cosp**,** cosr\_cosp**);**

//pitch (y-axis rotation)

double sinp **=** **+**2.0 **\*** **(**q**.**w **\*** q**.**y **-** q**.**z **\*** q**.**x**);**

**if(**fabs**(**sinp**)** **>=** 1**)**

angles**.**pitch **=** copysign**(**M\_PI**/**2**,** sinp**);** //use 90 degrees if out of range

**else**

angles**.**pitch **=** asin**(**sinp**);**

//yaw (z-axis rotation)

double siny\_cosp **=** **+**2.0 **\*** **(**q**.**w **\*** q**.**z **+** q**.**x **\*** q**.**y**);**

double cosy\_cosp **=** **+**1.0 **-** 2.0 **\*** **(**q**.**y **\*** q**.**y **+** q**.**z **\*** q**.**z**);**

angles**.**yaw **=** atan2**(**siny\_cosp**,** cosy\_cosp**);**

**return** angles**;**

**}**

class Stopper **{**

public**:**

// Tunable parameters

constexpr const static double FORWARD\_SPEED\_LOW **=** 0.1**;**

constexpr const static double FORWARD\_SPEED\_HIGH **=** 0.2**;**

constexpr const static double FORWARD\_SPEED\_SHIGH **=** 0.8**;**

constexpr const static double FORWARD\_SPEED\_STOP **=** 0**;**

constexpr const static double TURN\_LEFT\_SPEED\_HIGH **=** 1.0**;**

constexpr const static double TURN\_LEFT\_SPEED\_LOW **=** 0.3**;**

constexpr const static double TURN\_RIGHT\_SPEED\_HIGH **=** **-**2.4**;**

constexpr const static double TURN\_RIGHT\_SPEED\_LOW **=** **-**0.3**;**

constexpr const static double TURN\_RIGHT\_SPEED\_MIDDLE **=** **-**0.6**;**

Stopper**();**

void startMoving**();**

void moveForward**(**double forwardSpeed**);**

void moveStop**();**

void moveRight**(**double turn\_right\_speed **=** TURN\_RIGHT\_SPEED\_HIGH**);**

void moveForwardRight**(**double forwardSpeed**,** double turn\_right\_speed**);**

void odomCallback**(**const nav\_msgs**::**Odometry**::**ConstPtr**&** odomMsg**);**

void scanCallback**(**const sensor\_msgs**::**LaserScan**::**ConstPtr**&** scan**);**

int stage**;**

double PositionX**,** PositionY**;**

double robVelocity**;**

double startTime**;** //start time

double frontRange**,** mleftRange**,** leftRange**,** rightRange**,** mrightRange**,** backRange**,** leftbackRange**,** rightbackRange**;**

Quaternion robotQuat**;**

EulerAngles robotAngles**;**

double robotHeadAngle**;**

void transformMapPoint**(**ofstream **&** fp**,** double laserRange**,** double laserTh**,** double robotTh**,** double robotX**,** double robotY**);**

private**:**

ros**::**NodeHandle node**;**

ros**::**Publisher commandPub**;** // Publisher to the robot's velocity command topic

ros**::**Subscriber odomSub**;** //Subscriber to robot's odometry topic

ros**::**Subscriber laserSub**;** //Subscriber to robot's laser topic

**};**

Stopper**::**Stopper**(){**

//Advertise a new publisher for the simulated robot's velocity command topic at 10Hz

commandPub **=** node**.**advertise**<**geometry\_msgs**::**Twist**>(**"cmd\_vel"**,** 10**);**

// subscribe to the odom topic

odomSub **=** node**.**subscribe**(**"odom"**,** 20**,** **&**Stopper**::**odomCallback**,** **this);**

laserSub **=** node**.**subscribe**(**"scan"**,** 1**,** **&**Stopper**::**scanCallback**,** **this);**

**}**

//send a velocity command

void Stopper**::**moveForward**(**double forwardSpeed**){**

geometry\_msgs**::**Twist msg**;**//The default constructor to set all commands to 0

msg**.**linear**.**x **=** forwardSpeed**;** //Drive forward at a given speed along the x-axis.

commandPub**.**publish**(**msg**);**

**}**

void Stopper**::**moveStop**(){**

geometry\_msgs**::**Twist msg**;**

msg**.**linear**.**x **=** FORWARD\_SPEED\_STOP**;**

commandPub**.**publish**(**msg**);**

**}**

void Stopper**::**moveRight**(**double turn\_right\_speed**){**

geometry\_msgs**::**Twist msg**;**

msg**.**angular**.**z **=** turn\_right\_speed**;**

commandPub**.**publish**(**msg**);**

**}**

void Stopper**::**moveForwardRight**(**double forwardSpeed**,** double turn\_right\_speed**){**

//move forward and right at the same time

geometry\_msgs**::**Twist msg**;**

msg**.**linear**.**x **=** forwardSpeed**;**

msg**.**angular**.**z **=** turn\_right\_speed**;**

commandPub**.**publish**(**msg**);**

**}**

//add callback funciton to determine the robot position.

void Stopper**::**odomCallback**(**const nav\_msgs**::**Odometry**::**ConstPtr**&** odomMsg**){**

PositionX **=** odomMsg**->**pose**.**pose**.**position**.**x**;**

PositionY **=** odomMsg**->**pose**.**pose**.**position**.**y**;**

robVelocity **=** odomMsg**->**twist**.**twist**.**linear**.**x**;**

robotQuat**.**x **=** odomMsg**->**pose**.**pose**.**orientation**.**x**;**

robotQuat**.**y **=** odomMsg**->**pose**.**pose**.**orientation**.**y**;**

robotQuat**.**z **=** odomMsg**->**pose**.**pose**.**orientation**.**z**;**

robotQuat**.**w **=** odomMsg**->**pose**.**pose**.**orientation**.**w**;**

robotAngles **=** ToEulerAngles**(**robotQuat**);**

robotHeadAngle **=** robotAngles**.**yaw**;**

double currentSeconds **=** ros**::**Time**::**now**().**toSec**();**

double elapsedTime **=** currentSeconds **-** startTime**;**

odomVelFile **<<** elapsedTime **<<** " " **<<** robVelocity **<<** endl**;**

**}**

void Stopper**::**scanCallback**(**const sensor\_msgs**::**LaserScan**::**ConstPtr**&** scan**){**

frontRange **=** scan**->**ranges**[**0**];** //get at 0

mleftRange **=** scan**->**ranges**[**89**];** //get at -pi/4

leftRange **=** scan**->**ranges**[**179**];** //get at -pi/2

leftbackRange **=** scan**->**ranges**[**269**];** //get at 3pi/4

backRange **=** scan**->**ranges**[**359**];** //get at pi

rightbackRange **=** scan**->**ranges**[**449**];** //get at -3pi/4

rightRange **=** scan**->**ranges**[**539**];** //get at pi/2

mrightRange **=** scan**->**ranges**[**629**];**//get at pi/4

**}**

void Stopper**::**transformMapPoint**(**ofstream **&** fp**,** double laserRange**,** double laserTh**,** double robotTh**,** double robotX**,** double robotY**){**

double transX**,** transY**,** homeX **=** 0.3**,** homeY **=** 0.3**;**

transX **=** laserRange **\*** cos**(**robotTh **+** laserTh**)** **+** robotX**;**

transY **=** laserRange **\*** sin**(**robotTh **+** laserTh**)** **+** robotY**;**

**if(**transX **<** 0**)** transX **=** homeX**;**

**else** transX **+=** homeX**;**

**if(**transY **<** 0**)** transY **=** homeX**;**

**else** transY **+=** homeY**;**

fp **<<** transX **<<** " " **<<** transY **<<** endl**;**

**}**

void Stopper**::**startMoving**(){**

ofstream odomTrajFile**;**

odomTrajFile**.**open**(**"/home/local/CAMPUS/ms18975/M-Drive/ros\_workspace/src/tutorial\_pkg/odomTrajData.csv"**,** ios**::**trunc**);**

odomVelFile**.**open**(**"/home/local/CAMPUS/ms18975/M-Drive/ros\_workspace/src/tutorial\_pkg/odomVelData.csv"**,** ios**::**trunc**);**

startTime **=** ros**::**Time**::**now**().**toSec**();** //obtain the start time

ofstream laserFile**;**

laserFile**.**open**(**"/home/local/CAMPUS/ms18975/M-Drive/ros\_workspace/src/tutorial\_pkg/laserData.csv"**,** ios**::**trunc**);**

int i **=** 0**;** //the index to record laser scan data

double frontAngle **=** 0**,** mleftAngle **=** 0.785**,** leftAngle **=** 1.57**;**

double rightAngle **=** **-**1.57**,** mrightAngle **=** **-**0.785**;**

ofstream laserMapFile**;**

laserMapFile**.**open**(**"/home/local/CAMPUS/ms18975/M-Drive/ros\_workspace/src/tutorial\_pkg/laserMapData.csv"**,** ios**::**trunc**);**

stage **=** 1**;**

ros**::**Rate rate**(**20**);** //Define rate for repeatable operations.

ROS\_INFO**(**"Start moving"**);**

**while(**ros**::**ok**()){**

//There is some external bug with a few first steps

**if(**i **>** 6**)** **{**

**switch(**stage**){**

//Start off from moving fast in a straight line

**case** 1**:**

**if(**frontRange **>** 1.2**)**

moveForward**(**FORWARD\_SPEED\_SHIGH**);**

**else**

stage **=** 2**;**

**break;**

//Turn towards right obstacle in first gap

**case** 2**:**

**if(**mrightRange **>** 0.5**)**

moveForwardRight**(**FORWARD\_SPEED\_HIGH**,** TURN\_RIGHT\_SPEED\_MIDDLE**);**

**else**

stage **=** 3**;**

**break;**

//Move to the gap

**case** 3**:**

**if(**leftRange **<** 0.2 **||** mleftRange **<** 0.2**)**

moveForward**(**FORWARD\_SPEED\_SHIGH**);**

**else**

stage **=** 4**;**

**break;**

//Turn towards left (from robot) obstacle of 2nd gap

**case** 4**:**

**if(**frontRange **>** 1.7**)**

moveForwardRight**(**FORWARD\_SPEED\_HIGH**,** TURN\_RIGHT\_SPEED\_LOW**);**

**else**

stage **=** 5**;**

**break;**

//Approach 2nd gap

**case** 5**:**

**if(**frontRange **>** 0.65**)**

moveForward**(**FORWARD\_SPEED\_SHIGH**);**

**else**

stage **=** 6**;**

**break;**

//Turn towards inside of the 2nd gap

**case** 6**:**

**if(**mleftRange **>** 0.3**)**

moveForwardRight**(**FORWARD\_SPEED\_HIGH**,** TURN\_RIGHT\_SPEED\_MIDDLE**);**

**else**

stage **=** 7**;**

**break;**

//Go through 2nd gap

**case** 7**:**

**if(**frontRange **>** 0.55**)**

moveForward**(**FORWARD\_SPEED\_HIGH**);**

**else**

stage **=** 8**;**

**break;**

//Move in arc trajectory towards charger

**case** 8**:**

**if(**frontRange **>** 0.25 **&&** leftRange **>** 0.20**)**

moveForwardRight**(**FORWARD\_SPEED\_HIGH**,** TURN\_RIGHT\_SPEED\_LOW**);**

**else**

moveStop**();**

**break;**

**}**

**}**

transformMapPoint**(**laserMapFile**,** frontRange**,** frontAngle**,** robotHeadAngle**,** PositionX**,** PositionY**);**

transformMapPoint**(**laserMapFile**,** mleftRange**,** mleftAngle**,** robotHeadAngle**,** PositionX**,** PositionY**);**

transformMapPoint**(**laserMapFile**,** leftRange**,** leftAngle**,** robotHeadAngle**,** PositionX**,** PositionY**);**

transformMapPoint**(**laserMapFile**,** rightRange**,** rightAngle**,** robotHeadAngle**,** PositionX**,** PositionY**);**

transformMapPoint**(**laserMapFile**,** mrightRange**,** mrightAngle**,** robotHeadAngle**,** PositionX**,** PositionY**);**

odomTrajFile **<<** PositionX **<<** " " **<<** PositionY **<<** endl**;**

laserFile **<<** i**++** **<<** " " **<<** frontRange **<<** " " **<<** mleftRange **<<** " " **<<** leftRange **<<** " " **<<** rightRange **<<** " " **<<** mrightRange **<<** endl**;**

ros**::**spinOnce**();** // Allow ROS to process incoming messages

rate**.**sleep**();** // Wait until defined time passes.

**}**

odomTrajFile**.**close**();**

odomVelFile**.**close**();**

laserFile**.**close**();**

laserMapFile**.**close**();**

**}**

int main**(**int argc**,** char **\*\***argv**)** **{**

ros**::**init**(**argc**,** argv**,** "stopper"**);** // Initiate new ROS node named "stopper"

Stopper stopper**;** // Create new stopper object

stopper**.**startMoving**();** // Start the movement

**return** 0**;**

**}**

## References

[1]: Motor encoder - <https://en.wikipedia.org/wiki/Rotary_encoder>

[2]: J. Borenstein and L. Feng, "Gyrodometry: a new method for combining data from gyros and odometry in mobile robots," Proceedings of IEEE International Conference on Robotics and Automation, Minneapolis, MN, USA, 1996, pp. 423-428 vol.1, doi: 10.1109/ROBOT.1996.503813.

[3]: A. Alyasin, E. I. Abbas and S. D. Hasan, "An Efficient Optimal Path Finding for Mobile Robot Based on Dijkstra Method," 2019 4th Scientific International Conference Najaf (SICN), Al-Najef, Iraq, 2019, pp. 11-14, doi: 10.1109/SICN47020.2019.9019345.