Extended Kalman-Filter based Localization of a Four-Wheeled Differential Drive Robot in an Uneven Terrain

Autonomous Mobile Robotics - Course Project

Submitted in partial fulfillment of the requirements of
BITS F451 Course Project

By

Aditya Nair
ID No. 2019A4PS0437P
Samaksh Judson
ID No. 2019A4PS0278P

Under the supervision of:

Dr. Avinash Gautam
&
Dr. Sudeept Mohan

December 2021

Abstract

BITS F451 Course Project

Extended Kalman-Filter based Localization of a Four-Wheeled Differential Drive Robot in an Uneven Terrain

by Aditya Nair & Samaksh Judson

The following report sees the implementation of an Extended Kalman Filter on a four wheel differential driven vehicle. The project includes the use of a custom robot and controller. Devices on the robot include optical encoders(as wheel odometers), an inertial measurement unit and a pair of electric motors to ensure independent rear wheel speeds. The Extended Kalman Filter also accounts for sensor noise and the steering algorithm has been derived to account for the lack of viscoelastic properties of wheels in the simulator. Once localization has been achieved, we have tried to implement path planning and navigation using certain the A^* algorithm with a heuristic based on direct distance of way points from the end-point.

Acknowledgements

We'd want to take this time to thank our mentors for this research project, Prof. Avinash Gautam and Prof. Sudeept Mohan, for allowing us to work on such a fantastic project in our field of interest. Because of this opportunity and the professors' instruction, we have learned a lot since the beginning of this project. After every interaction with them, we felt even more motivated. We were given the necessary tools to create an in-depth understanding of the essential ideas required to jumpstart the project through this Autonomous Mobile Robotics course. We'd also like to express our gratitude to our friends and family members for modifying their lifestyles to offer us a productive work environment that ensures uninterrupted focus and is free of any distractions, which was critical throughout the project. Finally, we'd like to extend our gratitude to all of the Open Source contributors without which advanced research in robotics wouldn't be possible.

Contents

Abstract i

Acknowledgements ii

Contents iii

1 Introduction 1

2 Robot Build 2
- 2.1 Model . . . 2
- 2.2 Actuation and Sensors . . . 3
  - 2.2.1 Rotary encoder . . . 3
  - 2.2.2 Accelerometer and IMU . . . 4
  - 2.2.3 GPS . . . 5
- 2.3 Steering . . . 5
- 2.4 Kinematics and Localization in Navigation . . . 6
  - 2.4.1 Navigation and Kinematics . . . 6
  - 2.4.2 Navigation and Localization . . . 7

3 Extended Kalaman Filter 8

4 Path Planner 11

5 Conclusion and Future Work 14

A Code 15
- A.1 Algorithm for Ackermann Steering . . . 15
- A.2 Move to a Waypoint . . . 15
- A.3 Position Estimation with Wheel Odometry . . . 21
- A.4 Velocity Estimation from Accelerometer Runge-Kutta . . . 21
- A.5 Extended Kalman Filter . . . 21
- A.6 A* Path Planning to generate List of wayponints. Forked from here . . . 22

Chapter 1

Introduction

With the commercial availability of mobile robots for personal, industrial, military, hospital and academic requirements, the need for a versatile and economic design has also increased over the past few years. A critical design of these robots is to ensure they are physically stable. While a majority of robots may have 2 wheels, balancing such a robot requires the use of advanced controllers and uses up a good amount of computational resources(the alternative may to use wheels larger than the wheelbase of the platform, creating an impractical design). A way around this can be found in the use of four wheeled robot systems to the centre of gravity of the system remains in equilibrium.

From a theoretical view, the concept of implementing a 4 wheel drive system seems quite feasible but the alignment of the wheels must be carefully considered, keeping in mind that we must ensure pure rolling motion at all times to maintain efficiency and durability of components . A few ways to implement this are by using Ackermann Steering, Cab–Drive Steering and the use of Omnidirectional Wheels. Using our knowledge of Kinematics and Dynamics of Machines taught to us as a Mechanical Engineering course, we have decided to find the inversion points of the wheel axles and implement Ackermann Steering for our robot.

The sensors on the robot to detect its motion consist of optical encoders for the wheel odometer and an IMU to establish a control state for the aforementioned measurement state. Using real time measurements from both these sensors, we compute optimal coordinates of our robot irrespective of any mechanical uncertainty associated with wheeled locomotion.

Chapter 2

Robot Build

2.1 Model

The robot has been modeled in the Webots simulation software. It has four wheels that have a width of 10 mm and a radius of 25 mm. Fig. 2.1 has a CAD layout of the body of the robot. The robot is designed to have four wheels since over the years the four wheeled vehicle design has been proven to be the most versatile for a variety of terrains. The values have been chosen in order to make the robot symmetric, balanced and place the chassis moderately high up.

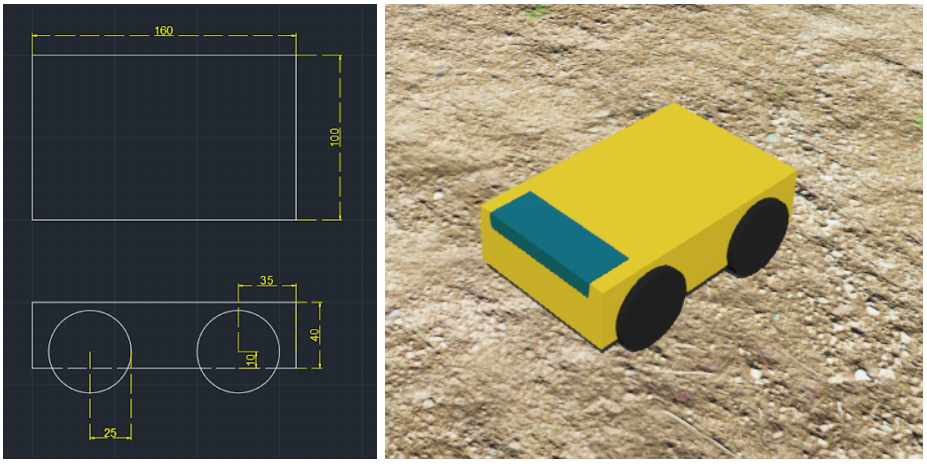

Fig. 2.1: Robot Model. All dimensions are given in mm.

Fig. 2.2: Components of the robot.

2.2 Actuation and Sensors

All four wheels of the robot are independently driven with one rotational motor each, giving it the capability of 4 wheel drive which is advantageous on uneven terrain since it provides higher power and greater accuracy of motion. In-addition, the two front wheels of the robot can be steered independently in either direction by two servos. The two back wheels also have brakes whose damping can be set to retard the robot at different rates. Various sensors have also been attached to the robot which will be discussed below.

2.2.1 Rotary encoder

A rotary-encoder is an electro-mechanical device that converts the angular position or motion of a shaft or axle to analog or digital output signals. Rotary-encoders are used to measure the total angle spanned by the device. The can be used for wheel-odometry by using this angle to find the distance that each wheel travels. These can then be fit into the mathematical model derived by us to get the position estimation of the robot (discussed in Kinematics and Navigation). Limitations of wheel-odometry through rotary encoders include errors due to slipping along and perpendicular to the wheel and bouncing of wheels. As a result it suffers from positional drift that accumulates over time. Although it is easy and inexpensive to implement, it is not reliable where precise and long-term localisation is needed.

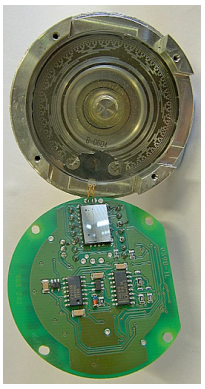

Fig. 2.3: Rotary Encoder.

2.2.2 Accelerometer and IMU

An accelerometer is a device that measures the proper acceleration along all 3–axes. Modern mechanical accelerometers are often small micro–electro-mechanical systems (MEMS), and are often very simple MEMS devices, consisting of little more than a cantilever beam with a proof mass (also known as seismic mass). The acceleration values can be integrated once to give the velocity of the robot, and then once again to get the position of the robot. We have implemented this using the Runge–Kutta method. Accelerometer based localization suffers from drift due to double integration of finite timesteps. An IMU is just an accelerometer integrated with a gyroscope that can in turn measure acceleration as well as rotation along all 3–axes. The measurements of roll, pitch and yaw also suffer from the same drift and need to be corrected.

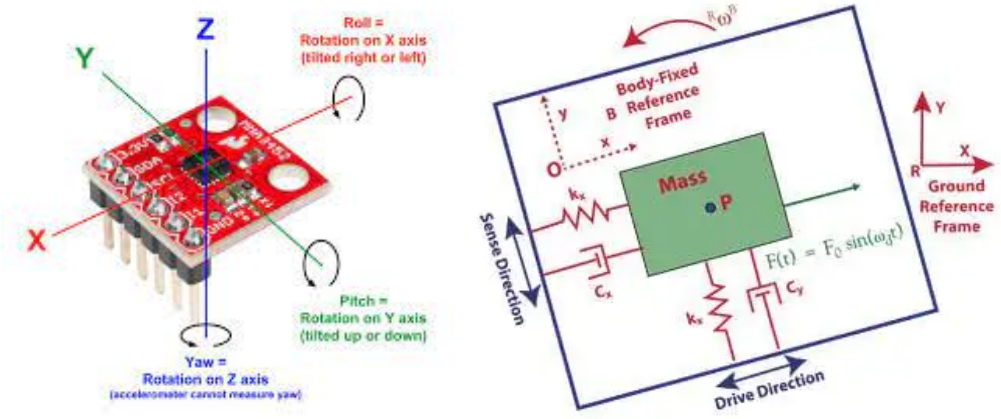

Fig. 2.4: Integrated Accelerometer and IMU

2.2.3 GPS

In the Webots simulator a GPS device essentially allows a platform to retrieve the exact position of the robot in all 3 coordinates. This device **has NOT** been used to estimate the position of the robot in any scenario. It's only use has been to dynamically measure the mean and the variance of the errors in position estimation done by the robot.

2.3 Steering

We have implemented the Ackermann Steering mechanism for the robot. The intention of Ackermann geometry is to avoid the need for tires to slip sideways when following the path around a curve. The geometrical solution to this is for all wheels to have their axles arranged as radii of circles with a common centre point. As the rear wheels are fixed, this centre point must be on a line extended from the rear axle. Intersecting the axes of the front wheels on this line as well requires that the inside front wheel be turned, when steering, through a greater angle than the outside wheel. The robot implements this mechanism through the following formulae, which we have derived, enabling the robot to travel in an arc. The implementation of this in the controller is given in A.1.

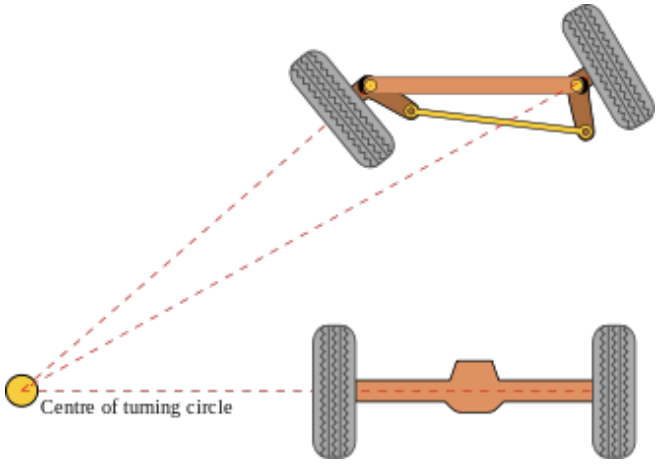

Fig. 2.5: Ackermann steering geometry

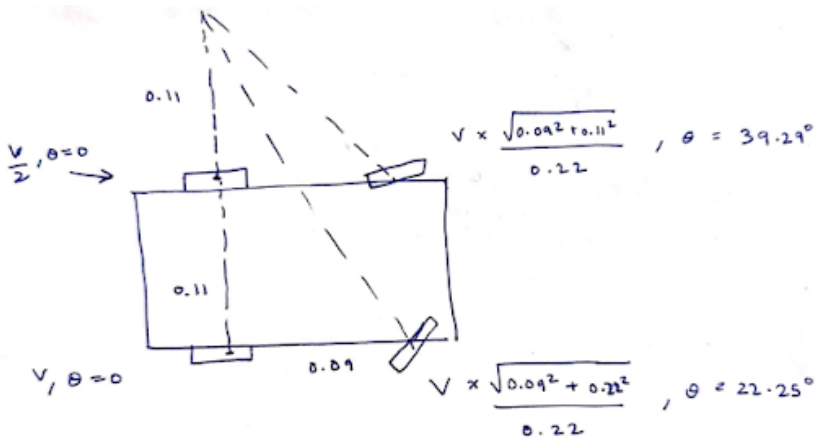

Fig. 2.6: Our implementation of the mechanism in our robot.

2.4 Kinematics and Localization in Navigation

2.4.1 Navigation and Kinematics

The documentation for kinematics for a four wheeled robot is scarce. The problem of kinematics is more complex and involved compared to a 2 wheeled differential drive robot since the four wheeled robot is a non-holonomic system. We need to divide the problem into two segments namely; traveling straight and moving in an arc. Traveling in a straight line involves setting the velocity of all wheels to be the same, however for traveling in an arc of radius 110 mm, the implementation from Fig. 2.6. is utilized. To travel from any one coordinate to another the robot has to orient itself onto the line joining the two points while also facing the destination point. The robot must traverse on 2 different arcs to reach this pose as seen in Fig. 2.7. The implementation of this in the controller is given in A.2.

$$\mathrm{a} = \theta - \arcsin \frac{\theta^2}{2}$$

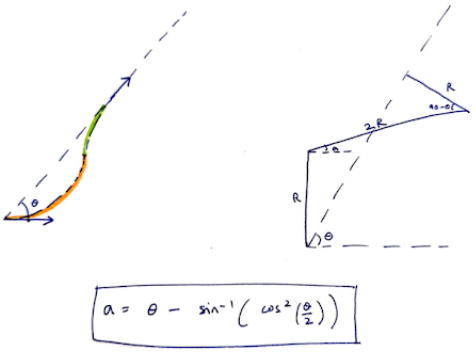

Fig. 2.7: The path the robot must follow to orient itself facing the waypoint while being on the line joining them.

2.4.2 Navigation and Localization

Our code allows us the capability of simply entering the waypoints in a list resulting in the robot traveling to each one of these waypoints in that order. In order to do this, the robot must be able to travel to any given point, know that it has reached its destination and then begin traveling to the next point.

Localization can be done using the wheel odometry values as well as acceleration readings.

For Wheel odometry, we derived the following formula to compute the x and y increment in every timestep. See A.3.

$$
\begin{aligned}
dx &= \frac{(dR + dL)}{2} \times \cos\frac{\theta}{2} \\
dy &= \frac{(dR + dL)}{2} \times \sin\frac{\theta}{2} \\
\theta &= 2(sin^{-1}(\frac{dR - dL}{2}) - \phi)
\end{aligned}
$$

Here $\phi \equiv yaw$

For IMU, we use the Runge-Kutta method to double integrate the acceleration in every iteration of the time loops to get the x and y increments. It should be noted that the values we obtain have significant errors due to drift because Webots does not allow timesteps smaller than 0.032 seconds. See A.6.

Chapter 3

Extended Kalaman Filter

A Kalman Filter is a Bayes' Filter for Gaussian Linear Case that relies on recursive state estimation to make it a popular estimation algorithm used for a variety of purposes ranging from temperature control to sensor fusion. The algorithm is relatively simple to implement and requires minimal computational power making it suitable for this project. However a fundamental understanding of the basic concepts is very necessary for its implementation. For this project we have used an Extended Kalman Filter to incorporate multiple degrees of freedom of our sensors. A detailed description of the code used is given below:

- **Initialization:**
  For the first iteration of the Extended Kalman Filter, we start at time k. In other words, for the first run of EKF, we assume the current time is k. We initialize the state vector and control vector for the previous time step $k-1$. We have also initialized the control state variables and assigned them an initial value of 0.

$$\hat{x}_{k-1|k-1} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \gamma_{k-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$u_{k-1} = \begin{bmatrix} v_{k-1} \\ w_{k-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

  Where v_{k-1} is the forward velocity in the robot frame at time $k-1$ and ω_{k-1} is the angular velocity around the z–axis at time $k-1$ or yaw.

- **Predicted State Estimation:**
  This estimated state prediction for time k is currently our best guess of the current state of the bot. We can also add some static noise values to make the sensor measurements more

realistic. However, for the purpose of implementing localization with maximum possible precision, we have assumed noise from measurements to be negligible. Our code does have provision to introduce some noise.

$$\begin{bmatrix} x_k \\ y_k \\ \gamma_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \gamma_{k-1} \end{bmatrix} + \begin{bmatrix} \cos\gamma_{k-1} * dk & 0 \\ \sin\gamma_{k-1} * dk & 0 \\ 0 & dk \end{bmatrix} \begin{bmatrix} v_{k-1} \\ w_{k-1} \end{bmatrix} + \begin{bmatrix} noise_{k-1} \\ noise_{k-1} \\ noise_{k-1} \end{bmatrix}$$

- **Predicted Covariance of State Estimate:**
  Our current predicted state estimate for time k is not completely accurate. So here, we predict the state covariance matrix $P_{k|k-1}$ for the current time step . Since our robot car has three states [x, y, $yaw\ angle$] in the state vector, P is a 3×3 matrix. The P matrix has variances on the diagonal and covariances on the off-diagonal. The last term in the predicted covariance of the state equation is Q_k which is the state model noise covariance matrix. Q represents the deviation of the actual motion from the space state model. It is a square matrix that has the same number of rows and columns as there are states. In this case it is 3×3.

$$Q_k = \begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,\gamma) \\ Cov(y,x) & Cov(y,y) & Cov(y,\gamma) \\ Cov(\gamma,x) & Cov(\gamma,y) & Cov(\gamma,\gamma) \end{bmatrix}$$

- **Innovation/Measurement Residual:**
  z_k is the observation vector. It is a vector of the actual readings from our sensors at time k. $h(\hat{x}_{k|k-1})$ is our observation model. It represents the predicted sensor measurements at time k given the predicted state estimate at time k from Step 2.

$$\tilde{\mathbf{y}}_\mathbf{k} = \mathbf{z}_\mathbf{k} - h(\hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k-1}})$$

- **Innovation (or residual) Covariance:**
  Here we have used the predicted covariance of the state estimate $P_{k|k-1}$ from Step 3 and the measurement matrix H_k and its transpose, and R_k (sensor measurement noise covariance matrix) to calculate S_k, which represents the measurement residual covariance (also known as measurement prediction covariance).

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} {\mathbf{H}_k}^T - h(\hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k-1}}) + \mathbf{R}_k$$

- **Near-Optimal Kalman Gain:**
  This is effectively our parameter to compute the weighted average between our actual readings and the predicted values. When the sensor noise is large, it approaches 0 and when the control state noise is large it approaches 1. Thus, it allows us to estimate the most accurate readings and uses those values to update itself in the next iteration.

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}_k{}^T\mathbf{S}_k{}^{-1}$$

- **Updated State Estimation:**
  Here we finally compute the weighted average mentioned in the previous step. We use the predicted state values, the Kalman gain and the measurement residual to do so.

$$\hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k}} = \hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k-1}} + \mathbf{K}_k\tilde{\mathbf{y}}_{\mathbf{k}}$$

- Updated Covariance of the State Estimate: In the final step we correct the estimated covariance for the timestep k by updating it. We use the near-optimal Kalman gain, the measurement matrix and the initially assumed values of the covariance for this timestep.

$$\mathbf{P}_{\mathbf{k}|\mathbf{k}} = (\mathbf{I} - \mathbf{K}_\mathbf{k}\mathbf{H}_\mathbf{k})\mathbf{P}_{\mathbf{k}|\mathbf{k-1}}$$

The Python code for each step of the Extended Kalman Filter is provided in A.5 of Appendix A.

Chapter 4

Path Planner

Path-planning is an important primitive for autonomous mobile robots that lets robots and the shortest - or otherwise optimal - path between two points. Otherwise optimal paths could be paths that minimize the amount of turning, the amount of braking or whatever a specific application requires. In this project, the path is optimized in terms of the distance traveled by the path.

For our path planning algorithm and navigation, we have put together some code from some open source libraries to implement the A^* algorithm.

Here, n is the current node, and n_0 is the adjacent node. The accumulated cost from the start node to any given node n with $g(n)$. The cost from a node n to an adjacent node n_0 becomes $c(n; n_0)$, and the expected cost (heuristic cost) from a node n to the goal node is described with $h(n)$. The total expected cost from start to goal via state n can then be written as

$$f(n) = g(n) + h(n)$$

Where is a parameter that assumes algorithm-dependent values is the weight given by the algorithm to the heuristic cost $h(n)$. For the A^* algorithm, the value is 0.

```
let openList equal empty list of nodes
let closedList equal empty list of nodes
put startNode on the openList (leave it's f at zero)
while openList is not empty
    let currentNode equal the node with the least f value
    remove currentNode from the openList
    add currentNode to the closedList
    if currentNode is the goal
        You've found the exit!
    let children of the currentNode equal the adjacent nodes
    for each child in the children
        if child is in the closedList
            continue to beginning of for loop
        child.g = currentNode.g + distance b/w child and current
        child.h = distance from child to end
        child.f = child.g + child.h
        if child.position is in the openList's nodes positions
            if child.g is higher than the openList node's g
                continue to beginning of for loop
        add the child to the openList
```

Fig. 4.1: This is pseudo-code for the A^* algorithm.

A^* works by making a lowest-cost path tree from the start node to the target node. A^* algorithm begins at the start (yellow star), and considers all adjacent cells. Once the list of adjacent cells has been populated, it filters out those which are inaccessible (walls, obstacles, out of bounds). It then picks the cell with the lowest cost, which is the estimated $f(n)$. This process is recursively repeated until the shortest path has been found to the target (red star). The computation of $f(n)$ is done via a heuristic that usually gives good results. A^* differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct.

Famous examples of heuristics are the "Euclidean" and the "Manhattan" distance. The heuristic function must be admissible, which means it can never overestimate the cost to reach the goal. The value of $h(n)$ would ideally equal the exact cost of reaching the destination. This is, however, not possible because the best path is not known. The Manhattan distance is usually preferred for grid based map representations, with Von Neumann's neighborhood (only four neighbours at right angles). It is given by

$$h(n) = jn_x goal_x j + jn_y goal_y j.$$

We have implemented this algorithm to generate a dataset of waypoints to determine the accuracy of wheeled locomotion in our robot. The inclination of a hypothetical line segment joining the start and end-points from the horizontal would determine the efficacy of the 4 wheel drive system including our EKF and unique steering algorithm.

The Python implementation of the A^* path planner is provided in A.6 of Appendix A.

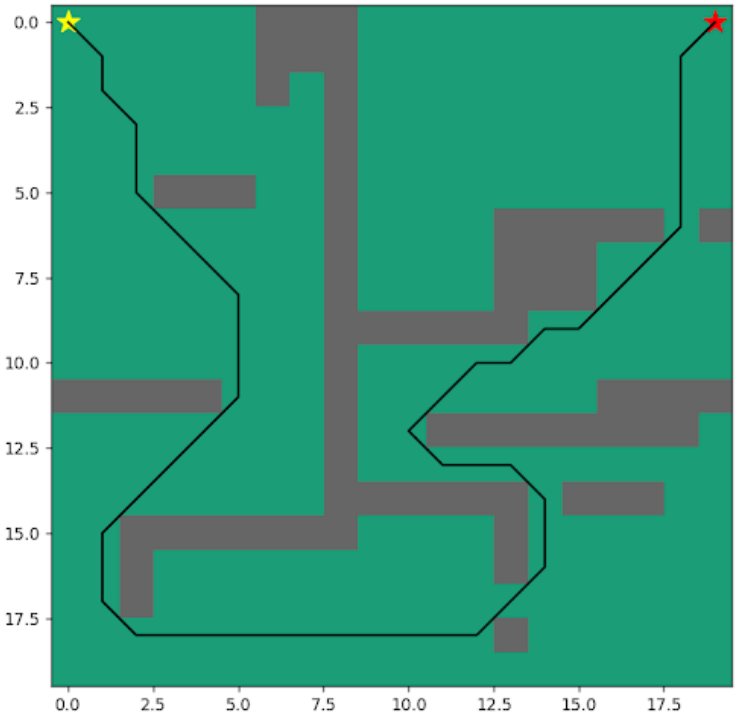

Fig. 4.2: This is a MATLAB plot of our sample path along with obstacles that are to be avoided.

Chapter 5

Conclusion and Future Work

Our simulation is able to complete all the objectives we set off with in the beginning. We have modelled a robust 4-wheeled robot capable of traveling on uneven rigid surfaces without drifting too much off a pre-decided path, with reasonable accuracy. We have made a controller that can make the 4-wheeled, non-holonomic robot travel on any path by just giving it a list of waypoints. The controller can also incorporate the A^* Algorithm for avoiding obstacles. This in-turn makes the robot capable of autonomously navigating through any course made out of obstacles, as long as the course has been mapped. We have developed an Extended Kalman Filter suitable for accurate localization and pose estimation of our 4-wheeled robot, granting it the power to overcome the physical limitations of its sensors, making it powerful enough to travel in statically uneven and non-ideal environments.

Future improvements and augmentations in our project may include: Inclusion of Trajectory error and Pose error matrices to quantify the performance of Localization through the Kalman-filtering algorithm. Capability of detecting and avoiding transient obstacles through techniques such as SLAM. This can be done through the incorporation of a LIDAR sensor or a camera and would be helpful for extending the capabilities of the robot to transient and deformable environments. Incorporation a PID control setup for smoother steering of the robot. Currently, the robot has to stop every time it changes the steering angle of its wheels. The implementation of a PID controller in our algorithm will allow the robot to change direction without stopping. This will also prevent sudden jerks and vibrations that might be noisy for the accelerometer. There are various assumptions made by the software during a simulation. Real world application will involve many other considerations. Fabrication of a real-life model based on our current one will help us tackle a lot of these first hand and should be the next step in further development of our project.

Appendix A

Code

A.1 Algorithm for Ackermann Steering

```
# Turn
    if state == 2:

          # Turn left
        if not flip:

            left_speed = max_speed / 2
            right_speed = max_speed

            front_left_speed = max_speed * (0.09**2 + 0.11**2)**0.5 / 0.22
            front_right_speed = max_speed * (0.09**2 + 0.22**2)**0.5 / 0.22

          # Turn right
        else:

            left_speed = max_speed
            right_speed = max_speed / 2

            front_left_speed = max_speed * (0.09**2 + 0.22**2)**0.5 / 0.22
            front_right_speed = max_speed * (0.09**2 + 0.11**2)**0.5 / 0.22
```

A.2 Move to a Waypoint

```
def setState(destx, destz):

    global tp, state, range

    # print(str(destx) + "\t" + str(destz))

    # Check if reached
    if (destx <= range and destx >= -range) \
```

```
and (destz <= range and destz >= -range):

    state = 9

    print("reached")

# Move straight
elif (tp >= 3 * brake_period + 4 * steer_period + \
(math.pi / 2 + theta - math.asin(math.cos(theta / 2) ** 2))/ turn_speed \
+ (math.acos(math.cos(theta/2) ** 2) / turn_speed)):

    state = 8

    tp += timestep / 1000

# Steer wheels straight
elif (tp >= 3 * brake_period + 3 * steer_period + \
(math.pi / 2 + theta - math.asin(math.cos(theta / 2) ** 2))/ turn_speed \
+ (math.acos(math.cos(theta/2) ** 2) / turn_speed)):

    state = 7

    tp += timestep / 1000

# Brake third time
elif (tp >= 2 * brake_period + 3 * steer_period + \
(math.pi / 2 + theta - math.asin(math.cos(theta / 2) ** 2))/ turn_speed \
+ (math.acos(math.cos(theta/2) ** 2) / turn_speed)):

    state = 6

    tp += timestep / 1000

# Turn right
elif (tp >= 2 * brake_period + 3 * steer_period + \
(math.pi / 2 + theta - math.asin(math.cos(theta / 2) ** 2))/ turn_speed):

      state = 5

      tp += timestep / 1000

# Steer wheels right
elif (tp >= 2 * brake_period + steer_period + \
(math.pi / 2 + theta - math.asin(math.cos(theta / 2) ** 2))/ turn_speed):

    state = 4

    tp += timestep / 1000

# Brake second time
elif (tp >= brake_period + steer_period + \
(math.pi / 2 + theta - math.asin(math.cos(theta / 2) ** 2))/ turn_speed):

    state = 3
```

```
        tp += timestep / 1000

    # Turn left
    elif (tp >= brake_period + steer_period):

        state = 2

        tp += timestep / 1000

    # Steer wheels left
    elif (tp >= brake_period):

        state = 1

        tp += timestep / 1000

    # Brake first time
    elif (tp <= brake_period):

        state = 0

        tp += timestep / 1000


def moveToWaypoint(destx, destz, currentTime, right_distance, left_distance, yaw):

    global theta, state, left_speed, right_speed, front_left_speed, front_right_speed, leftdamp, rightda


    if (tp < timestep / 1000):

        # Assign phi
        if yaw < 0:

            phi = 2 * math.pi + yaw

        else:

            phi = yaw

        # Assign alpha
        if destx > 0:

              alpha = math.atan(destz / destx)

        elif destx < 0:

              alpha = math.atan(destz / destx) + math.pi

        else:

              alpha = math.atan(destz / 0.001)

        # Assign theta
        if alpha - phi > math.pi:
```

```
            theta = -(2 * math.pi - (alpha - phi))

        elif alpha - phi < -math.pi:

            theta = 2 * math.pi + (alpha - phi)

        else:

            theta = alpha - phi

        print(str(theta / math.pi))

        # Alter theta
        if theta >= 0 and theta <= math.pi:

            pass

        elif theta >= -math.pi and theta <= 0:

            theta = -theta
            flip = True

    setState(destx, destz)

    # Brake
    if state == 0:

        left_speed = 0
        right_speed = 0

        front_left_speed = 0
        front_right_speed = 0

        leftdamp = brake_power
        rightdamp = brake_power

    # Steer left
    if state == 1:

        leftdamp = 0
        rightdamp = 0

        if not flip:

            leftpos = math.sin(39.29 / 360 * max_speed)
            rightpos = math.sin(22.25 / 360 * max_speed)

        else:

            leftpos = math.sin(-22.25 / 360 * max_speed)
            rightpos = math.sin(-39.29 / 360 * max_speed)

    # Turn left
    if state == 2:
```

```
        if not flip:

            left_speed = max_speed / 2
            right_speed = max_speed

            front_left_speed = max_speed * (0.09**2 + 0.11**2)**0.5 / 0.22
            front_right_speed = max_speed * (0.09**2 + 0.22**2)**0.5 / 0.22

        else:

            left_speed = max_speed
            right_speed = max_speed / 2

            front_left_speed = max_speed * (0.09**2 + 0.22**2)**0.5 / 0.22
            front_right_speed = max_speed * (0.09**2 + 0.11**2)**0.5 / 0.22

    # Brake
    if state == 3:

        left_speed = 0
        right_speed = 0

        front_left_speed = 0
        front_right_speed = 0

        leftdamp = brake_power
        rightdamp = brake_power

    # Steer right
    if state == 4:

        leftdamp = 0
        rightdamp = 0

        if not flip:

            leftpos = math.sin(-22.25 / 360 * max_speed)
            rightpos = math.sin(-39.29 / 360 * max_speed)

        else:

            leftpos = math.sin(39.29 / 360 * max_speed)
            rightpos = math.sin(22.25 / 360 * max_speed)

    # Turn right
    if state == 5:

        if not flip:

            left_speed = max_speed
            right_speed = max_speed/2

            front_left_speed = max_speed * (0.09**2 + 0.22**2)**0.5 / 0.22
            front_right_speed = max_speed * (0.09**2 + 0.11**2)**0.5 / 0.22
```

```
        else:

            left_speed = max_speed / 2
            right_speed = max_speed

            front_left_speed = max_speed * (0.09**2 + 0.11**2)**0.5 / 0.22
            front_right_speed = max_speed * (0.09**2 + 0.22**2)**0.5 / 0.22

    # Brake
    if state == 6:

        left_speed = 0
        right_speed = 0

        front_left_speed = 0
        front_right_speed = 0

        leftdamp = brake_power
        rightdamp = brake_power

    # Make wheels straight
    if state == 7:

        leftdamp = 0
        rightdamp = 0

        leftpos = math.sin(0)
        rightpos = math.sin(0)

    # Move straight
    if state == 8:

        left_speed = max_speed
        right_speed = max_speed

        front_left_speed = max_speed
        front_right_speed = max_speed

        flip = False

    # Stop at waypoint and start over
    if state == 9:

        print("bleh")

        left_speed = 0
        right_speed = 0

        front_left_speed = 0
        front_right_speed = 0

        leftdamp = brake_power
        rightdamp = brake_power
```

```
        if (wp_counter < num_points - 1):

            tp = 0
            wp_counter += 1
```

A.3 Position Estimation with Wheel Odometry

```
 # Odometry to position
    dR = right_distance - prevdistR
    dL = left_distance - prevdistL
    prevdistR = right_distance
    prevdistL = left_distance
    halfcurve = math.asin( dR - dL / 0.22) - roll_pitch_yaw[2]

    currx += ((dR + dL) / 2) * math.cos(halfcurve)
    currz += ((dR + dL) / 2) * math.sin(halfcurve)
```

A.4 Velocity Estimation from Accelerometer Runge-Kutta

```
    # Acceleration to velocity
    velx += acc[0] * timestep / 1000
    velz += acc[2] * timestep / 1000
```

A.5 Extended Kalman Filter

```
def getB(yaw, deltak):
    B = np.array([[np.cos(yaw)*deltak, 0],[np.sin(yaw)*deltak, 0],[0, deltak]])
    return B

def ekf(z_k_observation_vector, state_estimate_k_minus_1,
        control_vector_k_minus_1, P_k_minus_1, dk):

    global A_k_minus_1, process_noise_v_k_minus_1, Q_k, H_k, R_k

    state_estimate_k = A_k_minus_1 @ (state_estimate_k_minus_1) + (getB(state_estimate_k_minus_1[2],dk)
            process_noise_v_k_minus_1)

    print(f'State Estimate Before EKF={state_estimate_k}')


    P_k = A_k_minus_1 @ P_k_minus_1 @ A_k_minus_1.T + (Q_k)

    measurement_residual_y_k = z_k_observation_vector - ((H_k @ state_estimate_k) + (sensor_noise_w_k))
```

```
    print(f'Observation={z_k_observation_vector}')

    S_k = H_k @ P_k @ H_k.T + R_k

    K_k = P_k @ H_k.T @ np.linalg.pinv(S_k)

    state_estimate_k = state_estimate_k + (K_k @ measurement_residual_y_k)

    P_k = P_k - (K_k @ H_k @ P_k)

    print(f'State Estimate After EKF={state_estimate_k}')

    return state_estimate_k, P_k
```

A.6 A* Path Planning to generate List of wayponints. Forked from here

```
import numpy as np
import heapq
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure


############################################################################
# plot grid
############################################################################


grid = np.array([
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
# start point and goal
start = (0,0)
goal = (0,19)

###########################################################################
# heuristic function for path scoring
###########################################################################


def heuristic(a, b):
    return np.sqrt((b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2)


###########################################################################
```

```
# path finding function

##############################################################################


def astar(array, start, goal):

    neighbors = [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]

    close_set = set()

    came_from = {}

    gscore = {start:0}

    fscore = {start:heuristic(start, goal)}

    oheap = []

    heapq.heappush(oheap, (fscore[start], start))


    while oheap:

        current = heapq.heappop(oheap)[1]

        if current == goal:

            data = []

            while current in came_from:

                data.append(current)

                current = came_from[current]

            return data

        close_set.add(current)

        for i, j in neighbors:

            neighbor = current[0] + i, current[1] + j

            tentative_g_score = gscore[current] + heuristic(current, neighbor)

            if 0 <= neighbor[0] < array.shape[0]:

                if 0 <= neighbor[1] < array.shape[1]:

                    if array[neighbor[0]][neighbor[1]] == 1:

                        continue
```

```
                else:
                    # array bound y walls
                    continue
            else:
                # array bound x walls
                continue

            if neighbor in close_set and tentative_g_score >= gscore.get(neighbor, 0):
                continue

            if  tentative_g_score < gscore.get(neighbor, 0) or neighbor not in [i[1]for i in oheap]:
                came_from[neighbor] = current
                gscore[neighbor] = tentative_g_score
                fscore[neighbor] = tentative_g_score + heuristic(neighbor, goal)
                heapq.heappush(oheap, (fscore[neighbor], neighbor))

    return False
route = astar(grid, start, goal)
route = route + [start]
route = route[::-1]
print(route)

##########################################################################
# plot the path
##########################################################################


#extract x and y coordinates from route list
x_coords = []
y_coords = []
```

```
for i in (range(0,len(route))):

    x = route[i][0]

    y = route[i][1]

    x_coords.append(x)

    y_coords.append(y)

# plot map and path

fig, ax = plt.subplots(figsize=(20,20))

ax.imshow(grid, cmap=plt.cm.Dark2)

ax.scatter(start[1],start[0], marker = "*", color = "yellow", s = 200)

ax.scatter(goal[1],goal[0], marker = "*", color = "red", s = 200)

ax.plot(y_coords,x_coords, color = "black")

plt.show()
```