AMERICAN UNIVERSITY OF SHARJAH

ELE 494-08: AUTONOMOUS ROBOTIC SYSTEMS

PROJECT REPORT: CTE DOCUMENT 2

Robot Way-Point Navigation using Ackermann Steering with Obstacle Avoidance

Submitted By:
Taha AMEEN

Student ID:
@00066555

Submitted To:
Dr. Shayok MUKHOPADHYAY

November 29, 2019

Contents

I Updated Goal Statement 2

II Precise Objective 2

III Plan of Action 2

IV Theoretical Background 2
- IV-A The State Feedback Linearization Model . . . 2
- IV-B The Kalman Filter . . . 3

V Progress and Dissemination of Results 4
- V-A The State Feedback Linearization Model . . . 4
  - V-A1 MATLAB Code . . . 4
  - V-A2 Results . . . 6
- V-B The Kalman Filter . . . 6

VI Conclusion 9

References 9

List of Figures

1 Results of Ackermann Steering with Number of Segmented Points = 2. . . . 6
2 Results of Ackermann Steering with Number of Segmented Points = 20. . . . 6
3 Results of Ackermann Steering with Number of Segmented Points = 20. . . . 6

Robot Way-Point Navigation using Ackermann Steering with Obstacle Avoidance

Taha Ameen ur Rahman

***Abstract*—This document is a continuous time evolution (CTE) document submitted to course instructor Dr. Shayok Mukhopadhyay, in partial fulfillment of the project requirement for the course ELE 49408 - Autonomous Robotic Systems. The objective of this report is to present an intermediate update on the progress, along with information on the expected outline of the project. This project is a practical and hardware oriented project, which deals with way-point navigation of robots. The objective of the project is to develop a robot that is capable of navigating from one point to another, given GPS coordinates when selected from a map, and doing so using the Ackermann steering method, with obstacle detection and avoidance. The report presents precise objectives and examines the methodology that will be implemented over the course of the remaining part of the semester to achieve this goal. In addition, it also presents the theoretical background behind the material, and disseminates the results achieved so far with respect to software and simulation.**

I. Updated Goal Statement

This is a hardware-oriented project with the objective of developing an autonomous robot that navigates from an initial point to a final point as specified by GPS coordinates. The robot is expected to achieve this through measurement of position, filtering the signal, and executing commands to drive itself while avoiding obstacles in its path.

II. Precise Objective

This section discusses the precise objectives of the project. The following list outlines the expected objectives to be fulfilled.

- To build a differential two-wheel robot frame that can house ultrasonic sensors and an Arduino board.
- To use the state feedback linearization technique to allow the robot to autonomously drive itself from an initial point to a final point, as specified by GPS coordinates, which are then converted to NED coordinates.
- To implement a Kalman Filter (or atleast, a complementary filter) to smoothen the measured GPS readings for accurate estimation, and an Inertial Measurement Unit (IMU) to track the error in the navigation procedure.
- To implement obstacle detection and avoidance using multiple ultrasonic sensors, which allow the robot to interrupt and alter its path based on the detection.
- To accompany the robot with an Android based mobile application that allows the user to interface the desired coordinates.
- To demonstrate and test all the above in an outdoor environment.

III. Plan of Action

This section deliberates on the plan of action for me as an individual contributor and the proposed timeline for the completion of the project.

- Develop the MATLAB and Arduino based code for the Ackermann Steering Model.
- Develop the filtering process through a Kalman Filter or complementary filter.
- Simulate the entire process using MATLAB beforehand to ensure accuracy of the model.

Since my contribtutions are software intensive, I have already initiated the process of implementation and simulation. However, compatibility with the robot and further refinement is contingent on the development of the hardware and interfacing which my teammates are simultaneously working on. Once the robot frame is ready, it remains to download the software codes to the Arduino module associated with it, and running the entire system for testing purposes. However, my immediate responsibility is to ensure that all the codes work as expected by simulating the entire procedure and thoroughly debugging the programs.

IV. Theoretical Background

This section presents the theoretical background behind Ackermann steering and Kalman Filter in order to better motivate the choice of the navigation mechanism.

A. The State Feedback Linearization Model

We begin by considering a robot with center of mass (x, y), such that the objective is to move (x, y) to the point (x_r, y_r). Ackermann steering procedure introduces another point (x_h, y_h) such that $\lim_{t\to\infty}(x_h, y_h) = (x_r, y_r)$. Let us also introduce an inertial frame of reference, with origin at (x, y) and a mutually orthogonal basis set, such that one of the basis vectors is collinear with the direction of heading of the robot. We then introduce θ as the angle made by the heading with respect to the horizontal, and θ_d as the angle made by the horizontal with respect to the vector that points from the current position to the final desired position. Hence we have

$$\theta_d = \arctan\frac{y_r - y}{x_r - x} \tag{1}$$

Let v represent the linear velocity of the robot, and ω represent its angular velocity. Robot navigation is governed by the following set of non-holonomic equations:

$$\dot{x} = v\cos(\theta) \tag{2a}$$

$$\dot{y} = v\sin(\theta) \tag{2b}$$

$$\dot{\theta} = \omega \tag{2c}$$

If we were interested in Heading control, our objective would be to make θ approach θ_d so that the robot can then follow a straight line to reach $(x_r, y_r$. However, the process of controlling the angular velocity, followed by the linear velocity requires the tuning of many gains assuming the utilization of a PID controller. The Ackermann steering method provides an efficient method to overcome this problem, albeit with consequences of its own.

Our control strategy is to ensure that

$$\lim_{t\to\infty}(x_h, y_h) = (x_r, y_r) \tag{3}$$

Observe the following

$$x_h = x + h\cos(\theta) \tag{4a}$$
$$y_h = y + h\sin(\theta) \tag{4b}$$

Letting $\dot{x}, \dot{y}$ represent the time derivative of x, y respectively, we have

$$\dot{x_h} = \dot{x} - h\sin(\theta)\dot{\theta} \tag{5a}$$
$$\dot{y_h} = \dot{y} + h\cos(\theta)\dot{\theta} \tag{5b}$$

By substituting the non-holonomic constraints, we get

$$\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} = \begin{bmatrix}\cos(\theta) & -h\sin(\theta)\\ \sin(\theta) & h\cos(\theta)\end{bmatrix}\begin{bmatrix}v\\ \omega\end{bmatrix} \tag{6}$$

Let e represent the error signal, which is a measure of the difference between the actual value and the desired value. Hence,

$$e = \begin{bmatrix}x_h - x_r\\ y_h - y_r\end{bmatrix}, \quad \dot{e} = \begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} \tag{7}$$

Next, we let R to be the matrix in (6) and write

$$\dot{e} = R\begin{bmatrix}v\\ \omega\end{bmatrix} \tag{8}$$

Lastly, we pick

$$\begin{bmatrix}\theta\\ \omega\end{bmatrix} = -R^{-1}Ke, \quad K = \begin{bmatrix}K_1 & 0\\ 0 & K_2\end{bmatrix}$$

Here $K_1, K_2 > 0$, and mathematical manipulation yields

$$\dot{e} = -Ke \tag{9}$$

Since $-K$ is a diagonal matrix with negative entries, it is rest assured that the eigenvalues of this matrix are negative and real, and hence the system is LTI stable. This allows us to conclude that $\lim_{t\to\infty} e = 0$. This implies that $(x_h, y_h) \to (x_r, y_r)$. It is also worth noting that the R matrix is invertible when $h \neq 0$. This requirement is a necessary condition for Ackermann steering. However, the choice of (x_h, y_h) is entirely up to the user, and it can be chosen sufficiently close to (x, y) without affecting the invertibility.

The Ackermann steering method is a powerful technique because it guarantees the stability of the system. It is clear that the system will always converge provided that the constraints on K_1, K_2 are met. Ofcourse, this comes with the drawback of losing control over the path being taken. However, with sufficiently small space step, it is possible to ensure that the robot does indeed follow a smooth path.

B. The Kalman Filter

In this section, we review the Kalman Filter and its utility in autonomous robotics. The Kalman filter is required by systems in order to smoothen the measured variables. All decisions are made contingent on the measured values of position and velocity from hardware such as accelerometer, gyroscope and GPS. However, these readings are noisy and successive readings can reflect high fluctuations due to the noise present in the states as well as the output. Consequently, the error propagates through the system and greatly affects its performance.

The Kalman Filter is an effective way to ensure that the noise is filtered so as to smoothen the output. This results in a continuous and well-behaved dependency of the measured variables with respect to time, and ameliorates the stability of the system. Although the mathematical background for the Kalman Filter is intense, we present a basic overview of the equations associated with the process.

Consider a system whose state-space representation is given by:

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \tag{10}$$
$$z_k = H_k x_k + v_k \tag{11}$$

Here, x_k refers to the state variables at time instant k, and u is the input to the system. F, G, H are the state matrices, and w, v represent the noise in the states and output respectively, both of which are assumed to be i.i.d Gaussian Random Variables. Let Q_k and R_k represent the covariance matrix of w and v respectively. Hence, $w_k \sim (0, Q_k)$ and $v_k \sim (0, R_k)$.

The Kalman Filter estimates the covariance apriori and aposteriori as follows:

$$P_k^- = F_{k-1}P_{k-1}^+F_{k-1}^T + Q_{k-1} \tag{12a}$$
$$K_k = P_k^- H_k^T(H_k P_k^- H_k^T + R_k)^{-1} \tag{12b}$$
$$\hat{x}_k^- = F_{k-1}\hat{x}_{k-1} + G_{k-1}u_{k-1} \tag{12c}$$
$$\hat{x}_k^+ = \hat{x}_k^- + K_k(z_k - H_k\hat{x}_{k-1}) \tag{12d}$$
$$P_k^+ = \left[(P_k^-)^{-1} + H_k^T R_k^{-1} H_k\right]^{-1} \tag{12e}$$

Here, K_K is the Kalman Gain, which does not need to be explicitly tuned as the filter is capable of choosing it as shown above. This procedure allows us to estimate the states in a smooth manner.

A major drawback of this system is that it assumes a linear system. Most robots are not linear in nature due to their non-holonomic constraints. Consequently, an extended Kalman Filter was developed to account for these non-linearities. Consider a general system, where

$$x_k = f_{k-1}(x_{k-1}, v_{k-1}, w_{k-1}) \tag{13a}$$
$$z_k = h_k(x_k, v_k) \tag{13b}$$
$$w_k \sim (0, Q_k) \tag{13c}$$
$$v_k \sim (0, R_k) \tag{13d}$$
$$\tag{13e}$$

Then, the matrices are calculated as

$$F_{k-1} = \frac{\partial f_{k-1}}{\partial x} |_{\hat{x}^+_{k+1}} \qquad L_{k-1} = \frac{\partial f_{k-1}}{\partial w} |_{\hat{x}^+_{k+1}} \tag{14a}$$

$$H_k = \frac{\partial h_k}{\partial x} |_{\hat{x}^-_k} \qquad M_k = \frac{\partial h_k}{\partial v} |_{\hat{x}^-_k} \tag{14b}$$

Lastly, the estimation is done in a similar fashion, with the exception of the presence of L and M matrices.

$$P^-_k = F_{k-1} P^+_{k-1} F^T_{k-1} + L_{k-1} Q_{k-1} L^T_{k-1} \tag{15a}$$

$$K_k = P^-_k H^T_k (H_k P^-_k H^T_k + M_k R_k M^T_k)^{-1} \tag{15b}$$

$$\hat{x}^-_k = f_{k-1}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{15c}$$

$$\hat{x}^+_k = \hat{x}^-_k + K_k(z_k - h_k(\hat{x}_{k-1}, 0)) \tag{15d}$$

$$P^+_k = \left[(P^-_k)^{-1} + H^T_k R^{-1}_k H_k\right]^{-1} \tag{15e}$$

This allows us to achieve the objective of smoothening the output based on the measured quantities.

V. Progress and Dissemination of Results

This section presents the progress of the team until midterm II. Although advancements have been made on the hardware side such as procuring the body of the robot and preparing the frame, as well as on the software side such as preliminary code for the obstacle avoidance algorithm and a mobile application to import the GPS coordinates, the focus of this section will be on the feedback linearization model and the Kalman Filter. This is to highlight my individual contribution to the project.

A. The State Feedback Linearization Model

In this section, we illustrate the implementation of the Ackermann Steering model. It is emphasized that the basic version to navigate from point A to point B was already implemented as a homework in partial fulfilment for the requirements of this course.

In contrast, this model allows for greater degrees of freedom, and introduces the additional flexibility to allow for path planning. Although the implementation is not a rigorous path planning approach, it allows the achievement of a linear path and also allows for ensuring navigation through a pre-specified set of intermediate points. This is achieved by segmenting the path between initial and final coordinate and implementing the Ackermann Steering Model on each of the intermediate points. If these intermediate points are relatively close to each other, then the robot achieves a relatively linear path that it follows. This can be further enforced by breaking up the path into linearly spaced intermediate points. This process of segmentation of path and iterative state feedback linearization is simulated in MATLAB.

1) MATLAB Code: This section presents the code for the segmented state feedback linearization model. It consists of a function to allow for navigation between a start and end point, and a main function that creates a linear path segment and performs waypoint navigation between the points.

Listing 1: MATLAB Code: Point to Point Navigation.m

```
function [P,Ph,j] = Ack_Steer(Pin, Pfi,
    Pg, Phg, h, j1)
%% Step I: Initialization of Variables

dt = 0.1;       % Time Step (Global)
t = 0:dt:200;   % Discretization of Time

P  = Pg; % Matrix of Initial States
% P(:,1) = X Coordiante of Robot
% P(:,2) = Y Coordinate of Robot
% P(:,3) = Angular Position, theta, of
    Robot

P(j1,1) = Pin(1); %Initial x
P(j1,2) = Pin(2); %Initial y
P(j1,3) = Pin(3); %Initial theta

%% Parameters in Ackermann Steering Model
Pr = [Pfi(1), Pfi(2)];
% Pr(1) = x_r (Desired Final x coordinate
    )
% Pr(2) = y_r (Desired Final y coordinate
    )

K = [0.5 0; 0 0.5]; % Matrix (Problem
    Statement)

% v = 0; % Initial Linear Velocity
% w = 0; % Initial Angular Velocity

Ph = Phg; % Matrix of Positions
% Ph(:,1) = x_h
% Ph(:,2) = y_h

for j = j1:j1+length(t)
    Ph(j,1) = P(j,1) + h*cos(P(j,3)); %
        Definition of x_h
    Ph(j,2) = P(j,2) + h*sin(P(j,3)); %
        Definition of y_h

    E = Ph(j,:) - Pr; E = E'; % Error
        Vector

    R = [cos(P(j,3)) -h*sin(P(j,3)); sin(
        P(j,3)) h*cos(P(j,3))]; % R Matrix
    A = -inv(R)*K*E;

    v = A(1); % Linear Velocity
    w = A(2); % Angular Velocity

    % Numerical Integration to get x,y
        from \dot{x}, \dot{y}
    P(j+1,1) = v*cos(P(j,3))*dt + P(j,1);
    P(j+1,2) = v*sin(P(j,3))*dt + P(j,2);
    P(j+1,3) = w*dt + P(j,3);
   if max(abs(E)) < 0.001
       break % No Point executing after
```

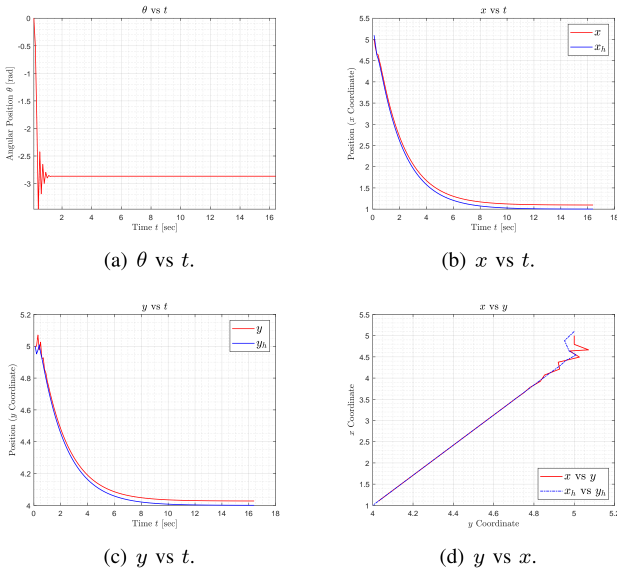
```
            saturation
        % This condition is needed to prevent
            robot from lurking around an
        % intermediate point. It should
            directly move to the next point.
      end
   end
   P = P(1:end-1,:); %Ensures vector length
      compatibility

   end
```

Listing 2: MATLAB Code: Segmented Ackermann Navigation.m

```
function [P,Ph,t] = Robot_Path_Ack(P_in,
    P_fi)
% P_in = Initial [x,y,theta]'
% P_fi = Final   [x,y,theta]'
close all;

NP = 20;  %Number of Points to break path
     into
h = 0.05;
dt = 0.1; %Time Step

Pvec = zeros(NP,3); %Initializing vector
    of positions

% Creating Array of Positions to navigate
     to in steps
for i = 1:3
   Pvec(:,i) = linspace(P_in(i), P_fi(i),
       NP);
end
j1 = 1;
Pg = zeros(1,3); Phg = zeros(1,2);

%Pvec = [P_in, [-4;10;0], [6;-20;0], P_fi
    ]'
%NP = 4
%% Ackermann Steering for each segment
for i = 1:NP-1
    % Navigate from previous to next
        intermediate point
    [P,Ph,j] = Ack_Steer(Pvec(i,:),Pvec(i
        +1,:),Pg,Phg,h,j1);
    Pvec(i+1,:) = P(end,:);
    Pg = P; Phg = Ph; j1 = j;
end

t = dt*(1:j); %Time Vector from
    differential time step

%% Plotting Theta vs. t
figure(1);
plot(t,P(:,3),'r','LineWidth',1);
xlabel('Time $t$ [sec]', 'interpreter', '
    latex','FontSize',12);
ylabel('Angular Position $\theta$ [rad]',
     'interpreter', 'latex','FontSize',12)
    ;
title('$\theta$ vs $t$', 'interpreter','
    latex','FontSize',14);
grid on; grid MINOR;
axis tight;

%% Plotting x vs. t and x_h vs t
figure(2);
plot(t,P(:,1),'r','LineWidth',1);
hold on
plot(t,Ph(:,1),'b','LineWidth',1);
legend({'$x$' '$x_h$'},'interpreter','
    latex', 'FontSize',16);
xlabel('Time $t$ [sec]', 'interpreter', '
    latex','FontSize',12);
ylabel('Position ($x$ Coordinate)', '
    interpreter', 'latex','FontSize',12);
title('$x$ vs $t$', 'interpreter','latex'
    ,'FontSize',14);
grid on; grid MINOR;

%% Plotting y vs. t and y_h vs t
figure(3);
plot(t,P(:,2),'r','LineWidth',1);
hold on
plot(t,Ph(:,2),'b','LineWidth',1);
legend({'$y$' '$y_h$'},'interpreter','
    latex', 'FontSize',16);
xlabel('Time $t$ [sec]', 'interpreter', '
    latex','FontSize',12);
ylabel('Position ($y$ Coordinate)', '
    interpreter', 'latex','FontSize',12);
title('$y$ vs $t$', 'interpreter','latex'
    ,'FontSize',14);
grid on; grid MINOR;

%% Plotting x vs. y and x_h vs y_h
figure(4);
plot(P(:,2),P(:,1),'r','LineWidth',1.1);
hold on
plot(Ph(:,2),Ph(:,1),'b-.','LineWidth',1)
    ;
legend({'$x$ vs $y$' '$x_h$ vs $y_h$'},'
    interpreter','latex', 'FontSize',16, '
    location', 'southeast');
xlabel('$y$ Coordinate', 'interpreter', '
    latex','FontSize',12);
ylabel('$x$ Coordinate', 'interpreter', '
    latex','FontSize',12);
title('$x$ vs $y$', 'interpreter','latex'
    ,'FontSize',14);
grid on; grid MINOR;

%% Saving Data as .fig and .pdf
```

```
for i = 1:4
    figure(i)
    fig = gcf;
    fig.PaperPositionMode = 'auto';
    fig_pos = fig.PaperPosition;
    fig.PaperSize = [fig_pos(3) fig_pos(4)];
    MG_string_name = char(['C:\Users\Lenovo\Desktop\CTE2\F_' num2str(i)]);
    savefig(MG_string_name)
    print(fig,MG_string_name,'-dpdf','-r0');
end
end
```

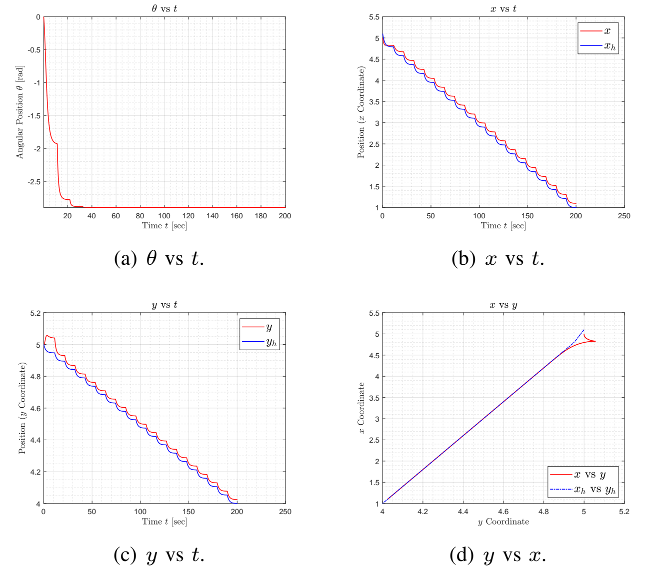
2) Results: In this section, we present the results of segmenting the path into a number of intermediate points and simulating the navigation. Figure 1 presents the robot navigation from $(x, y, \theta) = (5, 5, 0)$ to $(x, y) = (1, 4)$ after segmenting the path into two intermediate points. It can be seen that the system is really fast, but at the expense of involving rapid fluctuations. This translates to a large energy expenditure and can compromise stability by inducing large and rapid jerks in the robot motion.



(a) θ vs t. (b) x vs t. (c) y vs t. (d) y vs x.

Figure 1. Results of Ackermann Steering with Number of Segmented Points = 2.

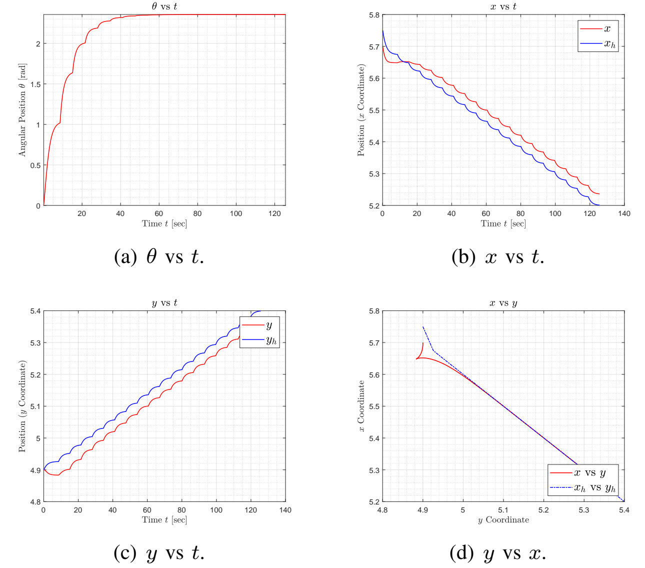
It is reasonable then to expect that increasing the number of segmentations in the path would allow for smoother movements and reduce any rapid fluctuations. Figure 2 shows the results when the path is segmented into 20 intermediate points. This results in a relatively linear path as can be seen in fig. 2(d).



(a) θ vs t. (b) x vs t. (c) y vs t. (d) y vs x.

Figure 2. Results of Ackermann Steering with Number of Segmented Points = 20.

It is useful to observe that similar patterns are followed when navigating between small distances. This is expected as the GPS reading is not expected to change in the most significant figures. Consequently, figure 3 displays the plots followed by the segmented Ackermann model with initial point $(5.2, 5.4)$ and final point $(5.7, 4.9)$.



(a) θ vs t. (b) x vs t. (c) y vs t. (d) y vs x.

Figure 3. Results of Ackermann Steering with Number of Segmented Points = 20.

Here, the translational difference between the red and blue curves is more evident due to the relatively comparable value of h with respect to the distance traversed.

B. The Kalman Filter

This section presents the preliminary efforts in developing the Kalman Filter using Arduino IDE. The Filter is required

to ensure a smooth signal after measuring from the sensors. Hence, the Arduino will take an input using its AnalogRead() function, and pass the measured values to the Kalman Filter Function which returns an estimate that fitlers the noise to the best it can.

Here, we present two different Arduino codes. The first is inspired from a Kalman Filter Code that is available on online repositories. Due credit has been given to the author in the code headers [1]. The code has, however, been adjusted to account for some of the specifics as demanded by the model at hand.

The second Arduino code is an effort to generalize the Kalman Filter for arbitrary inputs by directly using the associated equations presented in this report. It is emphasized that the code is a draft and is a work in progress.

```
/* Copyright (C) 2012 Kristian Lauszus,
   TKJ Electronics. All rights reserved.
This code has been inspired by a publicly
     available online repository.
*/

#ifndef _Kalman_h
#define _Kalman_h

// Defining the Filter as a Class
class Kalman {
public:
    Kalman() {
        /* We will set the variables like
             so, these can also be tuned
           by the user */
        Q_angle = 0.001;
        Q_bias = 0.0001;
        R_measure = 0.11;

        angle = 0;// Reset the angle
        bias = 0; // Reset bias

        // Since we assume that the bias
           is 0 and we know the starting
           angle, the error covariance
           matrix is set as follows.
        P[0][0] = 0;
        P[0][1] = 0;
        P[1][0] = 0;
        P[1][1] = 0;
    };
    /* Units:
     * 1. Angle: Degrees
     * 2. Rate: Degrees per second
     * 3. dt: Seconds    */

    //Function to get the angle from the
       measured readings
    double getAngle(double newAngle,
       double newRate, double dt) {
        // Discrete Kalman filter time
           update equations - Time Update
            ("Predict")
        // Update xhat - Project the
           state ahead
        /* Step 1 */
        rate = newRate - bias;
        angle += dt * rate;

        // Update estimation error
           covariance - Project the error
            covariance ahead
        /* Step 2 */
        P[0][0] += dt * (dt*P[1][1] - P
           [0][1] - P[1][0] + Q_angle);
        P[0][1] -= dt * P[1][1];
        P[1][0] -= dt * P[1][1];
        P[1][1] += Q_bias * dt;

        // Discrete Kalman filter
           measurement update equations -
            Measurement Update ("Correct
           ")
        // Calculate Kalman gain -
           Compute the Kalman gain
        /* Step 4 */
        S = P[0][0] + R_measure;
        /* Step 5 */
        K[0] = P[0][0] / S;
        K[1] = P[1][0] / S;

        // Calculate angle and bias -
           Update estimate with
           measurement zk (newAngle)
        /* Step 3 */
        y = newAngle - angle;
        /* Step 6 */
        angle += K[0] * y;
        bias += K[1] * y;

        // Calculate estimation error
           covariance - Update the error
           covariance
        /* Step 7 */
        P[0][0] -= K[0] * P[0][0];
        P[0][1] -= K[0] * P[0][1];
        P[1][0] -= K[1] * P[0][0];
        P[1][1] -= K[1] * P[0][1];

        return angle;
    };
    void setAngle(double newAngle) {
       angle = newAngle; }; // Used to
       set angle, this should be set as
       the starting angle
    double getRate() { return rate; }; //
        Return the unbiased rate
```

```
    /* These are used to tune the Kalman
       filter */
    void setQangle(double newQ_angle) {
       Q_angle = newQ_angle; };
    void setQbias(double newQ_bias) {
       Q_bias = newQ_bias; };
    void setRmeasure(double newR_measure)
        { R_measure = newR_measure; };

    double getQangle() { return Q_angle;
       };
    double getQbias() { return Q_bias; };
    double getRmeasure() { return
       R_measure; };

private:
    /* Kalman filter variables */
    double Q_angle; // Process noise
       variance for the accelerometer
    double Q_bias; // Process noise
       variance for the gyro bias
    double R_measure; // Measurement
       noise variance - this is actually
       the variance of the measurement
       noise

    double angle;   // The angle
       calculated by the Kalman filter -
       part of the 2x1 state vector
    double bias;    // The gyro bias
       calculated by the Kalman filter -
       part of the 2x1 state vector
    double rate;    // Unbiased rate
       calculated from the rate and the
       calculated bias - you have to call
        getAngle to update the rate

    double P[2][2]; // Error covariance
       matrix - This is a 2x2 matrix
    double K[2];    // Kalman gain - This
        is a 2x1 vector
    double y;       // Angle difference
    double S;       // Estimate error
};
#endif
```

Now, we present the second code that is being developed to implement the Kalman Filter Equations direclty.

```
void setup() {
// Hardware Interfacing
int SensorPin1 = A0;
int SensorPin2 = A1;
int SV = 0;

// Declaring the Variables of Interest
float v_k, w_k, R_k;
float Q_k[2][2], F_k[2][2], Readings[100]
    = {0};
float xhkplus[2];

// Defining the Variance in Noise (Noise
   Power)
v_k = 0.8; // Variance in Meaurement
   Noise
w_k = 0.1; // Variance in State Noise
R_k = v_k;

//Initialization of Error Covariane
   Matrix
Q_k[0][0] = wk;
Q_k[0][1] = 0;
Q_k[1][0] = 0;
Q_k[1][1] = wk;

//Initialization of the F Matrix
F_k[0][0] = 1;
F_k[0][1] = 0;
F_k[1][0] = 0;
F_k[1][1] = 1;
}

void loop() {
  SV1 = AnalogRead(SensorPin1);
  SV2 = AnalogRead(SensorPin2);
// The Kalman Filter Equations to be
   inserted here for automatic estimation
    and update.
}

// Function to multiply m1[][2] and m2
   [][2], and store result in res[][]
void multiply(float m1[][N],
              float m2[][N],
              float res[][N])
{
    int i, j, k;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            res[i][j] = 0;
            for (k = 0; k < N; k++)
                res[i][j] += m1[i][k] *
                             m2[k][j];
        }
    }
}
// Function to transpose given matrix
void T(float a[][N], float transpose[][N
   ])
{
    int i, j, k, r=2,c=2;

    for(i=0; i<r; ++i)
    for(j=0; j<c; ++j) {
      transpose[j][i] = a[i][j];
```

```
    }
}
// Function to Invert given matrix
void inv(float a[][N], float inv_a[][N])
{
    int det, k;
    det = a[1][1]*a[0][0] - a[0][1]*a
       [1][0];
    k = 1/det;
        inv_a[0][0] = k*a[1][1];
        inv_a[0][1] = -k*a[0][1];
        inv_a[1][0] = -k*a[1][0];
        inv_a[1][1] = k*a[0][0];
    }
}
```

VI. Conclusion

In this report, we explored the precise objectives and expectations of the project with emphasis on the methodology that will be adopted to implement an autonomous robot that uses the state feedback linearization model for way-point navigation. Details on how the robot is expected to achieve this are presented in terms of the theoretical background as well as MATLAB based simulations. The simulations support the purpose of the project and add an extra degree of freedom. This is achieved by the process of segmentation of path, which is useful in determining intermediate points that the robot must navigate through to reach the desired endpoint. This is expected to motivate efficient obstacle avoidance and interruption path executions, the details of which are covered in the CTE documents of my team-mates. The report also presented the preliminary progress on Arduino based codes that are being deevloped to achieve the purpose of implementing the Kalman Filter. This will then ensure that the inputs to the robot are clean and devoid of noise, by filtering the signal.

References

[1] K. Lauszus, *Kalman Filter: Arduino Code*. TKJ: Electronics, 1st ed., 2013.