Enhanced Pure Pursuit Algorithm & Autonomous Driving (FULL MATLAB CODE AVAILABLE)

Thesis · June 2019

DOI: 10.13140/RG.2.2.16776.44803/2

1 author:

Edoardo Cocconi
University of Nottingham
1 PUBLICATION **0** CITATIONS
SEE PROFILE

School of Mechanical and Manufacturing Engineering

Faculty of Engineering

UNSW Sydney

Enhanced Pure Pursuit Algorithm & Autonomous Driving

by

Edoardo Cocconi

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Mechanical Engineering

Submitted:	June 2019	Student ID: z5229188
Supervisor:	Jay Katupitiya	

Abstract

Pure Pursuit Algorithm is a commonly used path-tracking algorithm that in autonomous driving applications determines the steering rate necessary for the vehicle to stay on track. The aim of the project is to set up a simulation of a moving UGV and experiment with different mathematical solutions to improve the existing technology, making it suitable for a wider spectrum of applications. The main result produced by this research is a Modified Follow the Carrot Algorithm capable of high accuracy even in sharp corners and in slip conditions. This has been achieved by calculating the trajectory that the carrot must assume to allow the vehicle to stay on track. The full MATLAB code used to run the simulations is available in the Solution Method section accompanied by step-by-step comments.

Acknowledgements

Thanks to Dr Jay Katupitiya for his guidance, support and availability. I am thankful for working side to side with such a brilliant professor and learning about such interesting topics with him.

I want to thank Dr David Lyons for going an extra mile to let me start this amazing journey and for supporting me before and throughout my stay here at UNSW.

Abbreviations & Nomenclature

Follow the Carrot Algorithm

$\boldsymbol{k_1}$	=	proportional gain
$\boldsymbol{k_2}$	=	derivative gain
$\boldsymbol{P}$	=	target point
$\boldsymbol{R_0}$	=	look-ahead distance
$\boldsymbol{v_1}$	=	tangential velocity
$\boldsymbol{v_2}$	=	steering rate
$\boldsymbol{xy}$	=	fixed reference frame
$\boldsymbol{XY}$	=	vehicle reference frame
$\boldsymbol{\delta}$	=	steering angle
$\boldsymbol{\theta}$	=	angle offset

Pure Pursuit Algorithm

$\mathbf{e}_{\boldsymbol{\ell}_\mathrm{d}}$	=	offset error
$\boldsymbol{g_x, g_y}$	=	target point coordinates
$\boldsymbol{k}$	=	tunable look-ahead constant
$\boldsymbol{L}$	=	length of the vehicle
$\boldsymbol{\ell}_\mathbf{d}$	=	look-ahead distance
$\boldsymbol{R}$	=	radius of curvature
$\boldsymbol{\delta}$	=	steering angle
$\boldsymbol{\kappa}$	=	curvature

Stanley Method

$\boldsymbol{\theta_e}$	=	orientation error
$\boldsymbol{e_{fa}}$	=	offset error

Contents

Chapter 1

Introduction

THE transition to autonomous vehicles is one of the most awaited advancements in modern technology and will revolutionise the whole transportation system. Road accidents are currently the ninth leading cause of death in the world and the first among people aged 18 to 29. It is estimated that deaths due to car accidents are on average 3,287 per day [1]. These issues and many others could be solved through the diffusion of driverless technology.

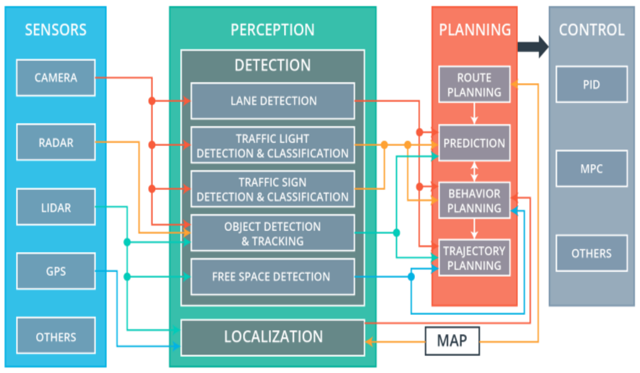

Figure 1. Autonomous vehicle workflow [2].

An autonomous vehicle uses sensors such as LIDAR, GPS, IMU, and cameras to detect the surrounding environment and localize itself. As shown in Fig. 1, it will subsequently calculate the most feasible, safe, legal, and efficient path that will lead it to destination, and use a control system to try to move following this path [2]. The subject of this thesis is Pure Pursuit Algorithm, which is a control algorithm that allows autonomous UGVs to compute the steering angle necessary to follow the previously determined path. The advantage of using such an algorithm instead of hard-coding a sequence of steering angles is that in the eventuality of deviations due to errors and unpredictable circumstances the vehicle can automatically stir itself to get back onto its original route.

2.1 Problem Definition

This increase in flexibility comes at a cost: it is necessary to develop a mathematical model that triggers the right behavior under a wide spectrum of circumstances. In high accuracy applications like self-driving cars, control algorithms are automatically tuned by artificial neural networks (ANN) [11]. However, artificial intelligence can be expensive for a company in terms of time and high level of expertise needed for the collection of data and the creation of a model. These costs are further exacerbated by the constant expenses due to retraining, maintenance, and eventual upgrades that AI algorithms typically need[15]. Therefore, AI is not to be regarded as a silver bullet, but as a last resort to be utilized when simpler alternatives fail. The aim of this thesis is to improve the Pure Pursuit controller so it can be used without ANN in a wider range of applications.

2.1 Introduction to Pure Pursuit Algorithm

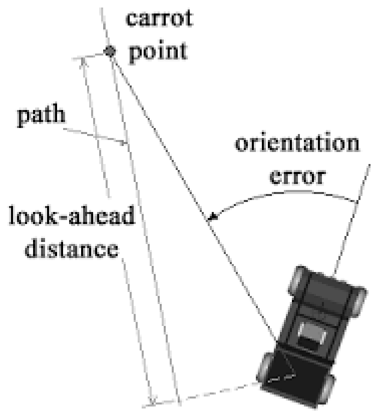

Figure 2. Follow the Carrot algorithm [4].

Over the course of the thesis the Follow the Carrot Algorithm, the traditional Pure Pursuit Algorithm, and the Stanley Method have been analyzed in depth and will be used as a base to come up with possible improvements. Different variants of the Pure Pursuit Algorithm perform better or worse according to the selected course and requirements; none of them is suitable in every single application [3]. All of the algorithms mentioned above are based on a simple underlying principle illustrated in Fig. 2: they detect a point on the path at a certain distance from the UGV and generate a steering angle to direct the vehicle towards the point. The algorithms take into account both the distance from the path and the orientation error of the vehicle, eventually leading the UGV to be aligned to the course. These solutions involve the use of tuning variables that can be regulated so the vehicle reaches the path both quickly and smoothly, characteristics that are usually conflicting in control systems [4]. Despite the possibility to tune the system, the Pure Pursuit Algorithm suffers from an inherent geometrical limitation. Since the vehicle is headed towards a point travelling on the path, it will detect a turn as soon as the point starts turning, even though it is not time for the vehicle to turn yet. This leads to the cutting of corners, which is the main obstacle that has to be overcome to make the algorithm usable in a wider range of applications.

2.1 Solution Method

The different algorithms are tested with a MATLAB simulation of a kinematic model of a UGV [5]. A kinematic model is a model in which it is assumed that the vehicle can instantly reach the desired speed, and it is therefore only accurate at low speeds, typically under 5 m/s [6]. This allows to neglect the effects of accelerations and consequent forces acting on the UGV. A kinematic model is already suited to the design of controllers for agricultural applications, characterized by low speeds and accelerations [16][17]. The effects of slip have been added to the simulation in order to better mimic an off-road environment. In future studies the produced code could be tested on dynamic models and hardware to extend its validity to other types of vehicles.

2.1 Results & Contribution

Follow the Carrot, Pure Pursuit, and Stanley Method have been implemented with and without the effects of slip. An original improvement of the Follow the Carrot Algorithm has been completed and it is ready for more realistic testing. In this modified algorithm, the carrot doesn't travel on the path that the vehicle is required to follow. Instead, it travels on the path that will lead the vehicle in the right position at all times, even during turns. This method and the produced MATLAB code is to be considered the contribution of this thesis.

The upcoming sections of the report will guide the reader through the theory behind Pure Pursuit Algorithm and show in detail how it has been modified for maximum performance.

Chapter 2

Background

Pure Pursuit Algorithm is the most common and effective geometric method to direct a vehicle towards a moving point travelling a path. Pure Pursuit was first devised in a study published in 1969 regarding the pursuit of a missile to a target [7]. In 1985 the Pure Pursuit strategy was applied to the field of robotics for the first time in an algorithm capable of estimating the steering necessary to maintain the vehicle on the road [8][9]. In this section the existing variants of the Pure Pursuit Algorithm are explored in depth. The first to be discussed will be a simplified version commonly called Follow the Carrot Algorithm. Successively, the full Pure Pursuit Algorithm will be explained alongside the assumptions on which it is based. Eventually, this section will go over the most recent variants and improvements of the algorithm made in recent years.

2.1 Simplified Pure Pursuit Algorithm: Follow the Carrot

The Follow the Carrot Algorithm is based on a very simple idea. At every instant the algorithm detects the point on the path ahead of the vehicle with distance $R0$ from the center of its front axle. Then it sets the value of the steering angle $\delta(t)$ to be equal and opposite to the offset error, until the vehicle is directed towards its moving goal and its trajectory converges with the path. To simplify the mathematical aspect of the model, the path indicated in red in Fig. 3 coincides with the x-axis of the fixed reference frame. However, the same principles can be applied to a much wider spectrum of scenarios if the reference frame of the vehicle is adopted instead.

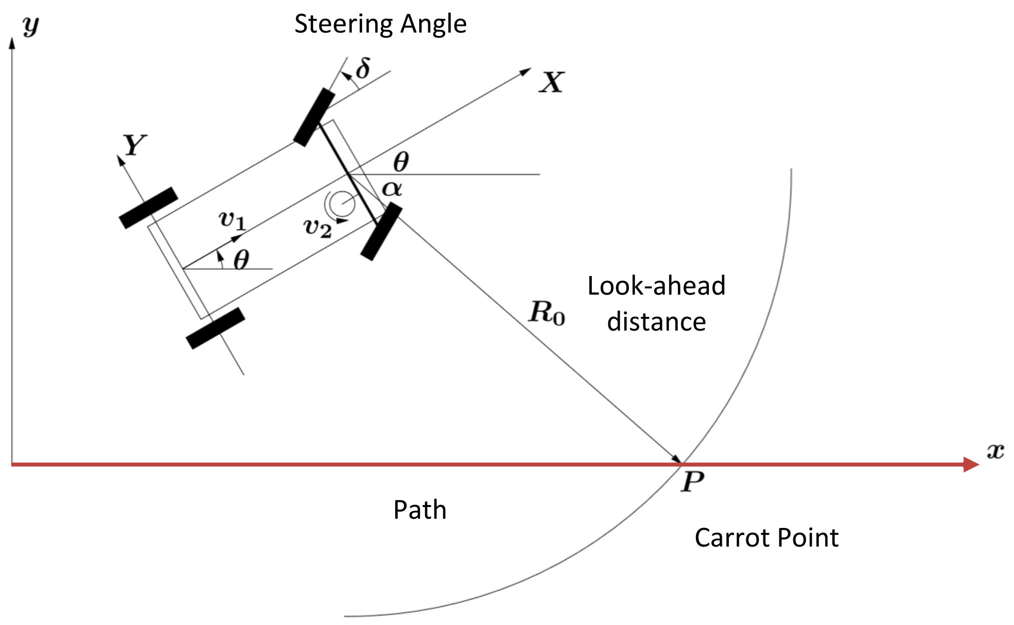

Figure 3. Follow the Carrot geometry [4]

In Fig. 3 the orientation error of the UGV is given by the sum of α and θ.

$$\delta(t) = -(\alpha + \theta) \tag{1}$$

Since the path coincides with the x-axis, it is possible to use the following geometric properties to relate α and θ to the distance of the vehicle from the path y:

$$\alpha = \frac{y}{R_0} \qquad \theta = \frac{dy}{dx} \tag{2}$$

$$\delta(t) = -\left(\frac{1}{R_0}\, y + \frac{dy}{dx}\right) \tag{3}$$

In control systems, the term containing y is generally called Proportional Term and the term containing its derivative is called Derivative Term. The Proportional Term allows to quickly reduce the distance y of the vehicle from the path, while the Derivative Term makes sure that the vehicle reaches the path with the right steering angle and low velocity in the y direction [10].

Tunable parameter k_1 and k_2 can be added to the equation to regulate the influence of these two terms on the steering angle and on the trajectory of the UGV.

$$\delta(t) = -\left(\frac{k_1}{R_0}\, y + k_2\, \frac{dy}{dx}\right) \tag{4}$$

k_1 is called Proportional Gain, and k_2 is called Derivative Gain. Increasing the Derivative Gain too much will result in a very smooth trajectory, but it will keep the vehicle off the path for an unacceptable amount of time. This is wat is commonly referred to as an Over-Damped System. Instead, if the Proportional Gain is too high the system will steer abruptly, and it will oscillate around the path before stabilizing. The same effect could be caused by a small look-ahead distance R_0 or by an increase of the tangential velocity of the vehicle. This type of system is called Under-Damped System. When k_1 and k_2 and R_0 are properly tuned it is possible to achieve a Critically Damped System, where the vehicle quickly reaches the path without incurring in overshooting as indicated in green in Fig. 4 [10].

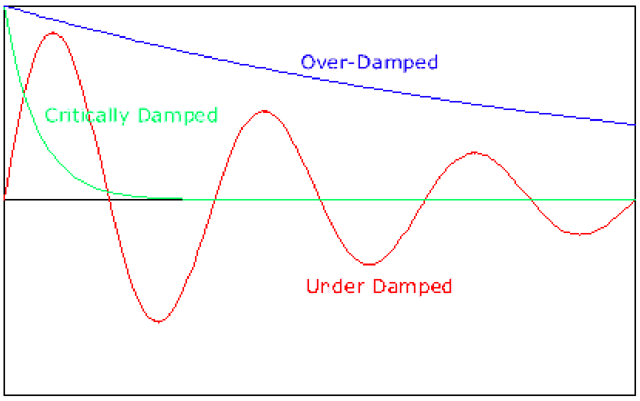

Figure 4. System damping.

3.1 Pure Pursuit Algorithm

The complete version of this algorithm can be better visualized by representing the vehicle using the Bicycle Model. The Bicycle Model simplifies the four-wheel car by combining the two front wheels together and the two rear wheels together to form a two-wheeled model, like a bicycle. This results in a simple geometric relationship between the front wheel steering angle and the curvature that the rear axle will follow [3].

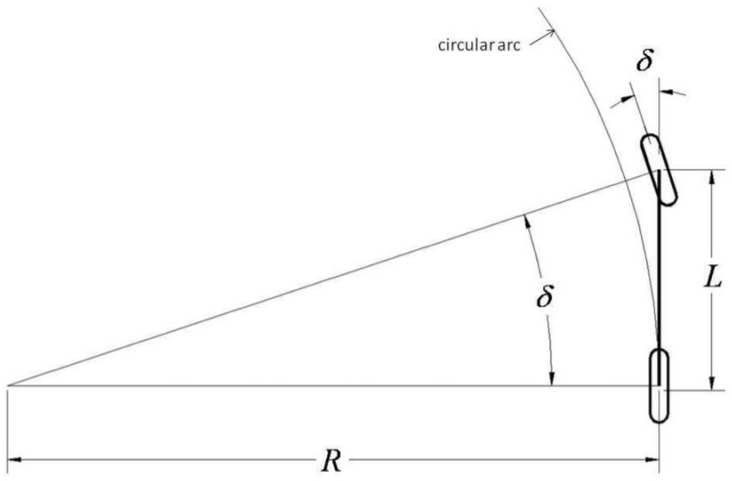

Figure 5. Bicycle Model geometry [3]

As shown in Fig. 5, these quantities can be linked in the geometric relationship

$$tan(\delta) = \frac{L}{R} \tag{5}$$

Where δ is the steering angle of the front wheel, L is the length of the vehicle and R is the radius of the circle that the rear axle will travel along at the given steering angle.

This is particularly useful since the Pure Pursuit Algorithm's aim is to calculate the curvature $1/R$ of a circular arc that connects the rear axle a goal point on the path. Below it will be explained how this curvature can then be translated into a steering angle that will direct the vehicle towards the goal point at distance ℓ_d from the current rear axle position. The goal point (gx, gy) and the Pure Pursuit Algorithm geometry is illustrated in Fig. 6.

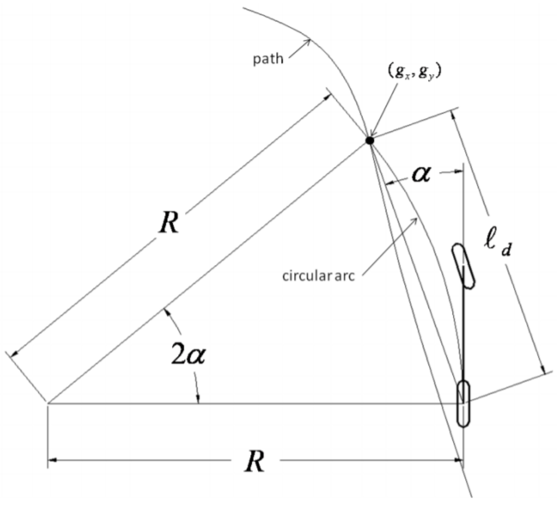

Figure 6. Pure Pursuit Algorithm [3].

The vehicle's steering angle δ can be defined using the goal point location and the angle α between the vehicle's heading vector and the look-ahead vector. By applying the law of sines, it is possible to derive the following equation [3]:

$$\frac{\ell_\mathrm{d}}{\sin(2\alpha)} = \frac{\mathrm{R}}{\sin\left(\frac{\pi}{2} - \alpha\right)} \tag{6}$$

$$\frac{\ell_\mathrm{d}}{2\sin(\alpha)\cos(\alpha)} = \frac{\mathrm{R}}{\cos(\alpha)} \tag{7}$$

$$\frac{\ell_\mathrm{d}}{\sin(\alpha)} = 2\mathrm{R} \tag{8}$$

$$\kappa = \frac{2\sin(\alpha)}{\ell_\mathrm{d}} \tag{9}$$

where κ is the curvature of the circular arc. Using the properties of the geometry of the bicycle model earlier described in Eq. (5), the steering angle can be written as follows:

$$\delta = \tan^{-1}(\kappa L) \tag{10}$$

Combining Eqs. (9) and (10), the Pure Pursuit control law can be expressed as:

$$\delta(\mathrm{t}) = \tan^{-1}\left(\frac{2\,L\sin\left(\alpha(\mathrm{t})\right)}{\ell_\mathrm{d}}\right) \tag{11}$$

A better understanding of this control law can be gained by defining a new variable, e_{ℓ_d} to be the offset error defined by the lateral distance between the heading vector and the goal point [3]. This results in the equation:

$$\sin(\alpha) = \frac{\mathrm{e}_{\ell_\mathrm{d}}}{\ell_\mathrm{d}} \tag{12}$$

. Equation (9) can then be rewritten as:

$$\kappa = \frac{2}{\ell_d^2}\,\mathrm{e}_{\ell_\mathrm{d}} \tag{13}$$

Equation (13) demonstrates that Pure Pursuit is a Proportional Control of the steering angle with a Proportional Gain of $2/\ell_d^2$ [3].

To simplify tuning, the control law can be rewritten, scaling the look-ahead distance with the longitudinal velocity of the vehicle. Additionally, the look-ahead distance is commonly limited to a minimum and maximum value. This results in:

$$\delta(t) = tan^{-1}\left(\frac{2\,L\sin\,(\alpha)}{k\,v(t)}\right) \tag{14}$$

As k increases, the look-ahead distance is increased and the tracking becomes less and less oscillatory. A short look-ahead distance provides more accurate tracking while a longer distance provides smoother tracking. It is clear that a k value that is too small will cause the system to be unstable and under-damped, and a k value that is too large will generate an Over-Damped System with poor tracking. One must be careful not to over tune on a course and test a variety of courses and speeds to find a k that can perform well over the operating requirements of the vehicle [3]. Another characteristic of Pure Pursuit is that a sufficient look-ahead distance will result in cutting corners while executing turns on the path [4]. This is due to the fact that the algorithm is looking several meters forward, so it will detect turns before the vehicle has reached them it and will start steering accordingly.

4.1 Stanley Method

The method is named after the self-driving car Stanley that is the first autonomous vehicle to have ever completed the notorious DARPA Grand Challenge and used this very algorithm. The Stanley Method have a base steering angle equal to the orientation error θ_e and an additional steering angle proportional to the offset error e_{fa}. This intuitive control law allows to minimize both errors and bring the vehicle on the path. e_{fa} is equal to the distance between the front axle of the vehicle and its closest point on the path indicated by the coordinates (c_x, c_y) (Fig. 7). Instead, θ_e

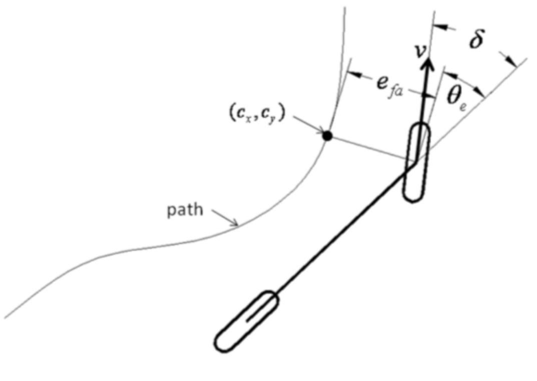

Figure 7. Stanley Method [3].

is given by the angular difference between the direction of the vehicle and the direction of the path at (c_x, c_y). The resulting control law assumes the following form [3]:

$$\delta(t) = \theta_e + tan^{-1}\left(\frac{k \cdot e_{fa}}{v_x(t)}\right) \tag{15}$$

Where k is a tunable parameter and the tangential speed $v_x(t)$ is situated at the denominator to avoid overshoot at higher speeds. Indeed, approaching the course too fast could lead to oscillations, but it can be avoided by minimizing the effects of the offset error term.

The major drawback of this method is that the path must be differentiable in order to calculate its direction at (c_x, c_y), and no discontinuity points are allowed [11]. The algorithm also suffers from the opposite problem of Pure Pursuit: without feed forward the algorithm cannot prevent errors but only react to them. This means that in case of sharp corners the vehicle will find itself off the path and a control effort will successively be generated to bring it back on track. This causes wide turns.

As shown in this section there is plenty of algorithms and ideas to approach the path-tracking issue. The subject topic covered in the next section is how to set up a system to test and improve these solutions.

Chapter 3

Solution Method

In order to test a control system such as the Pure Pursuit Algorithm, it is necessary to implement it into a source code and to set up an infrastructure to test the code produced. This can be done in two ways.
Using a physical UGV: This method can be expensive and dangerous but gives an immediate idea of the actual behavior of the vehicle.
Using a UGV model: This method is cheap and safe but needs the code of the simulation to be written. It is not going to match completely the actual behavior because it is just a mathematical simulation [13]. At the initial stages of a project, a simulation is usually preferred, and it is the method that will be utilized throughout this thesis.

The simulation of a UGV can be based on two different models.
Kinematic Model: This model assumes that it is possible to reach cruise speed instantly and uses the derivative of velocity to find position. It is considerate an accurate model for UGV's that travel at speeds inferior to 5 m/s.
Dynamic Model: This model assumes that it is possible to deliver an instantaneous force and therefore acceleration to the vehicle, and that its speed increases gradually due to this acceleration. It is possible to use the second derivative of force divided by mass to find position. It is considerate an accurate model of UGV's that travel at speeds greater than 5 m/s [13].
Taking the effect of forces into account results in a more complicated model, therefore a Kinematic Model is preferred at this stage. Since a Kinematic Model is used, the tangential velocity can be taken as an input instead of the torque provided by the motor. The other input could be either the steering angle or the steering rate. "Input" is considered whatever influences

the state of the UGV model, however the user of the simulation will not have to manually provide this information since they will be automatically outputted by the control system. The model with tangential velocity and steering rate as inputs is referred to as Standard Model, while the model with tangential velocity and steering angle as inputs is referred to as Reduced Model. The Reduced model is based on the assumption that every possible steering angle can be reached instantaneously. As suggested by the name, this is a simplified model and it has the advantage of requiring less computational power and being easier to understand and visualize in relation to the algorithms explained above. Both models have been successfully set up, but the Reduced Model gave more results because it is easier to visualize, which facilitated control algorithms modification.

In order to allow the reader to reproduce the tests that have been performed, the MATLAB code of the Modified Follow the Carrot Reduced Model simulation with slip has been pasted below. The comments explain its functioning. The base code of the simulation can be retrieved at [5].

```
% Name this file main.m
% Edoardo Cocconi - FOLLOW THE CARROT, REDUCED MODEL, SLIP
% This file simulates the model described in UPDN_UGV_model.m

% Cleaning the previous simulation
clear all
close all
clc

% Creating the course
Create_Path2()

% Initialising the necessary global variables
global x_path_calc_coordinates
global y_path_calc_coordinates
global x_path_coordinates
global y_path_coordinates
global point_sequence
global current_point
global data

% Simulation
tspan = 0.0:0.1:60; %start time : timestep : end time
options = odeset('RelTol',1e-4,'AbsTol',[1e-5 1e-5 1e-5]); % options of
the ODE-solving function
[t,x] = ode45(@UPDN_UGV_model, tspan,[0 2 2*pi/8.0], options);
% [result] = ode45(function to solve, [start-time, end-time],
% initial conditions [x, y, teta, delta], options)
% this will derivate [xdot(1) xdot(2) xdot(3) xdot(4)] and give [x, y,
% theta, delta] of the UGV at any given time
% Obtain the steering angles at each instant so they can be plotted
NM = size(t);
N = NM(1);
point_sequence = zeros(N,1);
steering_angle = zeros(N,1);
current_point = 1;
for i = 1:N
    inputs = UPDN_UGV_control_carrot(t(i,1),x(i,:));
    point_sequence(i,1) = current_point;
    steering_angle(i,1) = inputs(2);
end

% Put the steering angles in the 4th coloumn of x
backup = x;
clear x
x = [backup(:,1), backup(:,2), backup(:,3),steering_angle(:,1)];

% Plotting of the data obtained in the simulation
UPDN_UGV_Path(t,x);
```

```
% Name this file Create_Path2.m
% Edoardo Cocconi - FOLLOW THE CARROT, REDUCED MODEL, SLIP
% This file creates the paths that the vehicle and the carrot have to
% follow

function Create_Path2()

global current_point;
current_point = 0;

% Defining the initial segment of the path
xinitial = 0.0;
xfinal = 10.0;
yinitial = 2.0;
yfinal = 2.0;
length = sqrt((xfinal-xinitial)^2+(yfinal-yinitial)^2);
n = length * 100; %There will be 100 points per meter

slope1 = (yfinal-yinitial)/(xfinal-xinitial);

% Saving the points in the path coordinates. The path is made up by
points.
global x_path_coordinates
x_path_coordinates = linspace (xinitial,xfinal, n);
global y_path_coordinates
y_path_coordinates = linspace (yinitial,yfinal, n);

% x_path_calc_coordinates are the coordinates thet the carrot has to
follow
% to prevent the vehicle from cutting corners. They are automatically
% calculated. The first part of the path is in common between carrot and
vehicle
global x_path_calc_coordinates
x_path_calc_coordinates = x_path_coordinates;
global y_path_calc_coordinates
y_path_calc_coordinates = y_path_coordinates;

% The course is made up of straight lines joined by splines
% The initial coordinates of the spline is the final coordinates of the
line
xinitial_spline = xfinal;
yinitial_spline = yfinal;

fprintf('\n\tENTER NUMERICAL VALUES ONLY')
pause(1)
clc

continue_loop = 1;
% The loop will continue as long as you are building the course
while continue_loop

    xmemo = xfinal;
    ymemo = yfinal;

    answer1 = input('Enter the x coordinate of the starting point of the
next course segment:','s');
    xinitial = str2num(answer1);

    answer1 = input('Enter the y coordinate of the starting point of the
next course segment:','s');
    yinitial = str2num(answer1);

    answer1 = input('Enter the x coordinate of the ending point of the
next course segment:','s');
    xfinal = str2num(answer1);

    answer1 = input('Enter the y coordinate of the ending point of the
next course segment:','s');
    yfinal = str2num(answer1);

    slope2 = (yfinal-yinitial)/(xfinal-xinitial);

    spline_bounds_x = [xinitial_spline xinitial];
    spline_bounds_y = [yinitial_spline yinitial];

    % Get 100 points of a spline that smoothly joins two lines
    n = 100;
    xx = linspace (xinitial_spline,xinitial, n);
    yy = spline(spline_bounds_x, [slope1 spline_bounds_y slope2],xx);
    x_path_coordinates = [x_path_coordinates, xx];
    y_path_coordinates = [y_path_coordinates, yy];

    % Calculate the course that the carrot has to follow
    equation = spline(spline_bounds_x, [slope1 spline_bounds_y slope2]);
    spline_derivative = fnder(equation);
    derivative_at_points = fnval(spline_derivative,xx);
    new_xx = xx+3.3*cos(atan(derivative_at_points));
    new_yy = yy+3.3*sin(atan(derivative_at_points));

    % The following lines allow to equally space the points to be
followed
    % Estimate the length of the carrot spline
    [sz1, sz2] = size(new_xx);
    length_invisible_path = 0;
    for i = 1:sz2-1
        length_invisible_path = length_invisible_path +
sqrt((new_xx(i+1)-new_xx(i))^2+(new_yy(i+1)-new_yy(i))^2);
    end
    length_invisible_path = ceil(length_invisible_path); %round to the
next integer
    n_invisible_path = length_invisible_path*100;

    % interparc allows to obtain equally spaced points
    [pt] = interparc(n_invisible_path,new_xx,new_yy,'spline');
    xx_equal_distance = pt(:,1);
    yy_equal_distance =  pt(:,2);

    % At first the carrot has to go straight even if the path curves
    % This allows to avoid cutting corners
    length = sqrt((xx_equal_distance(1)-xmemo)^2+(yy_equal_distance(1)-
ymemo)^2);
    n = length * 100;
    x_coordinates_segment = linspace (xmemo,xx_equal_distance(1), n);
    y_coordinates_segment = linspace (ymemo,yy_equal_distance(1), n);
    x_path_calc_coordinates = [x_path_calc_coordinates,
x_coordinates_segment];
    y_path_calc_coordinates = [y_path_calc_coordinates,
y_coordinates_segment];

    % The curved part of the calculated carrot path is then saved
    xx_equal_distance = xx_equal_distance.'; % coloumn vector to row
vector
    x_path_calc_coordinates = [x_path_calc_coordinates,
xx_equal_distance];
    yy_equal_distance = yy_equal_distance.'; % coloumn vector to row
vector
    y_path_calc_coordinates = [y_path_calc_coordinates,
yy_equal_distance];

    % The second segment of the vehicle path is saved
    length = sqrt((xfinal-xinitial)^2+(yfinal-yinitial)^2);
    n = length * 100;
    x_path_coordinates_new = linspace (xinitial,xfinal, n);
    y_path_coordinates_new = linspace (yinitial,yfinal, n);

    x_path_coordinates = [x_path_coordinates, x_path_coordinates_new];
    y_path_coordinates = [y_path_coordinates, y_path_coordinates_new];


    % The second segment of the carrot path is saved. However, the line
has
    % new initial coordinates, different from the vehicle path
    [sz3, sz4] = size(xx_equal_distance);
    new_xinitial = xx_equal_distance(sz4);
    new_yinitial = yy_equal_distance(sz4);
    length1 = sqrt((xfinal-new_xinitial)^2+(yfinal-new_yinitial)^2);
    n1 = length1 * 100;
    x_path_coordinates_new = linspace (new_xinitial,xfinal, n1);
    y_path_coordinates_new = linspace (new_yinitial,yfinal, n1);
    x_path_calc_coordinates = [x_path_calc_coordinates,
x_path_coordinates_new];
    y_path_calc_coordinates = [y_path_calc_coordinates,
y_path_coordinates_new];

    answer2 = input('Do you want to insert another segment? 1 for YES, 0
for NO:','s');
    continue_loop = str2num(answer2);
    xinitial_spline = xfinal;
    yinitial_spline = yfinal;
    slope1 = slope2;

end

return;
```

```
% Name this file UPDN_UGV_model.m
% Edoardo Cocconi - FOLLOW THE CARROT, REDUCED MODEL, SLIP
% UGV reduced model in file UPDN_UGV_model.m

function xdot = UPDN_UGV_model(t,x)
% Initialisation
L =0.9; % Length of the vehicle in meters
% State variables
% [x(1) x(2) x(3)]
% x(1) - global x coordinate (m.)
% x(2) - global y coordinate (m.)
% x(3) = \theta - UGV heading (rad.)

% inputs(1) - UGV speed (m/s)
% inputs(2) - UGV steering angle (rad)

% Declarations
xdot = zeros(3,1);
inputs = zeros(2,1); %velocity and stiring angle

%control inputs
inputs = UPDN_UGV_control_carrot(t,x);

% k is a tunable slip constant. It allows to limit the angle of slip to
10°
k = (tan(10.0*pi/180.0))/(42.0*pi/180.0);

% the slip velocity is proportional to how fast the car is going, k, and
the steering angle
v_slip = k*inputs(1)*inputs(2);
slip_angle = atan(v_slip/inputs(1));

% trigonometry to find resultant of v(1) and v_slip
vtot = inputs(1)/cos(slip_angle);

%state equations
xdot(1) = vtot*cos(x(3)-slip_angle);
xdot(2) = vtot*sin(x(3)-slip_angle);
xdot(3) = sin(inputs(2))/(L*cos(inputs(2)-slip_angle))*vtot;

return;
```

```
% Name this file UPDN_UGV_control_carrot.m
% Edoardo Cocconi - FOLLOW THE CARROT, REDUCED MODEL, SLIP
% UGV controller in file UPDN_UGV_control.m

function inputs = UPDN_UGV_control_carrot(t,x)
% Initialisatio
% --- none
% State variables
% [x(1) x(2) x(3)]
% x(1) - global x coordinate (m.)
% x(2) - global y coordinate (m.)
% x(3) = \theta - UGV heading (rad.)

global current_point;
global x_path_calc_coordinates
global y_path_calc_coordinates
R0 = 4.0; % Fixed look-ahead distance

% Getting the coordinates of the point currently considered by the
algorithm
current_point = current_point + 35;
current_x = x_path_calc_coordinates(current_point);
current_y = y_path_calc_coordinates(current_point);

X_prime =  x(1) - current_x; %horizontal offset
Y_prime =  x(2) - current_y; % vertical offset

distance = sqrt(X_prime^2+Y_prime^2);

count = 0;

% The while loop makes sure the point is always at distance R0
while distance > R0 && count < 35
    current_point = current_point-1;

    current_x = x_path_calc_coordinates(current_point);
    current_y = y_path_calc_coordinates(current_point);

    X_prime =  x(1) - current_x; %horizontal offset
    Y_prime =  x(2) - current_y; % vertical offset

    distance = sqrt(X_prime^2+Y_prime^2);

    count = count + 1;

end

% Coordinates of the point in vehicle in vehicle reference frame
X1 = X_prime*cos(x(3)) + Y_prime*sin(x(3));
Y1 = Y_prime*cos(x(3)) - X_prime*sin(x(3));

% inputs(1) - UGV speed (m/s.)
% inputs(2) - UGV steering rate (rad/s.)
% Declarations
inputs = zeros(2,1);

%control inputs for the close loop case.
inputs(1) = 2.0;
k1 = 1.0;

% Control system Eq. (4)
inputs(2) =  (k1 * (atan2(Y1,X1)));
inputs(2) =  inputs(2) - sign(inputs(2))*pi();

% The steering angle is limited to 42 degrees
if(abs(inputs(2)) > 42.0*pi/180.0)
    inputs(2) = sign(inputs(2))*42.0*pi/180.0;
```

```
end

return;
```

```
% Name this file UPDN_UGV_Path.m
% Edoardo Cocconi - FOLLOW THE CARROT, REDUCED MODEL, SLIP
% Path data plot function in file UPDN_UGV_Path.m

% This function plots the necessary graphs after the simulation
function [] = UPDN_UGV_Path(t,x)
global x_path_calc_coordinates
global y_path_calc_coordinates
global x_path_coordinates
global y_path_coordinates
global point_sequence
figure; % Plot figure 1
UGV = [ 0          0       -0.9000     -0.9000     0           0
        0    -0.2500     -0.2500      0.2500      0.2500      0]; % UGV
dimensions

%Contol file test.
NM = size(t);
N = NM(1);
Controls = zeros(N,2);
for i = 1:N % Obtain the steering angles at each instant so they can be
plotted
    inputs = UPDN_UGV_control_carrot(t(i,1),x(i,:));
    Controls(i,1) = inputs(1);
    Controls(i,2) = inputs(2); % The steering angle are now in the
second row of the array
end

for i = 1:size(t)% Real time animation of UGV motion and trajectory
    UGV_ugv = [cos(x(i,3)) -sin(x(i,3));sin(x(i,3)) cos(x(i,3))];
    ugv = UGV_ugv* UGV;
    plot(x(i,1)+ugv(1,:), x(i,2)+ugv(2,:)); hold on; % Plot the UGV
    plot(x_path_coordinates(1,:),y_path_coordinates(1,:),'k') % Plot the
path
    % Plot the path percurred by the carrot
    plot(x_path_calc_coordinates(1,:),y_path_calc_coordinates(1,:),'b')
    plot(x(1:i,1),x(1:i,2),'r') % Plot the actual path already percurred
    % Plot the carrot in real time

plot(x_path_calc_coordinates(1,point_sequence(i,1)),y_path_calc_coordina
tes(1,point_sequence(i,1)),'r.')
    %L is the length
    %angle is alpha
    x2=x(i,1)+(4*cos(x(i,4)+x(i,3)));
    y2=x(i,2)+(4*sin(x(i,4)+x(i,3)));
    plot([x(i,1) x2],[x(i,2) y2],'g') % Shows where the car is pointing
    % Scale the axis
    axis equal;
    axis([min(x(:,1))-5.0, max(x(:,1))+5.0,min(x(:,2))-
5.0,max(x(:,2))+5.0]);
    % Plot the axis
    ax = gca;
    ax.XAxisLocation = 'origin'; % Plot x axis
    ax.YAxisLocation = 'origin'; % Plot y axis
    pause(0.001); % Slow down the output so it can be seen
    hold off % Cancels the effect of "hold on" and allows the new
position of the UGV to be plotted
end


figure(2); % Plot the steering angle
delta = Controls(:,2).*180/pi; % Converts angles from radians to degrees
plot(t, delta,'k'); % Plot the steering angle vs time in degrees
axis([0.0, max(t),min(delta)-20.0,max(delta)+20.0]); % scale axis
ax = gca;
ax.XAxisLocation = 'origin'; %display x axis
xlabel(' Time, s'); % label x axis
ylabel(' Steering Angle, degrees'); % label y axis

return;
```

These five files need to be in the same directory to work properly. The interparc function can be found at [18] and also needs to be saved in the same directory. The simulation can be started by running main.m. Follow the instructions displayed on the screen. If you get an error, it is likely that the time allocated for the simulation is too long. You can either increase the size of the course by entering new values or decrease the simulation time by modifying tspan in the main function. The graphs produced by the code have been analyzed in an iterative process to evaluate and modify the different algorithms. The results of this process can be found in the following section.

Chapter 4

Results & Discussion

In this section, the performance of the algorithms coded will be analyzed and compared. The first algorithm is a Follow the Carrot Algorithm with the goal of following a path made out of segments joined by splines.

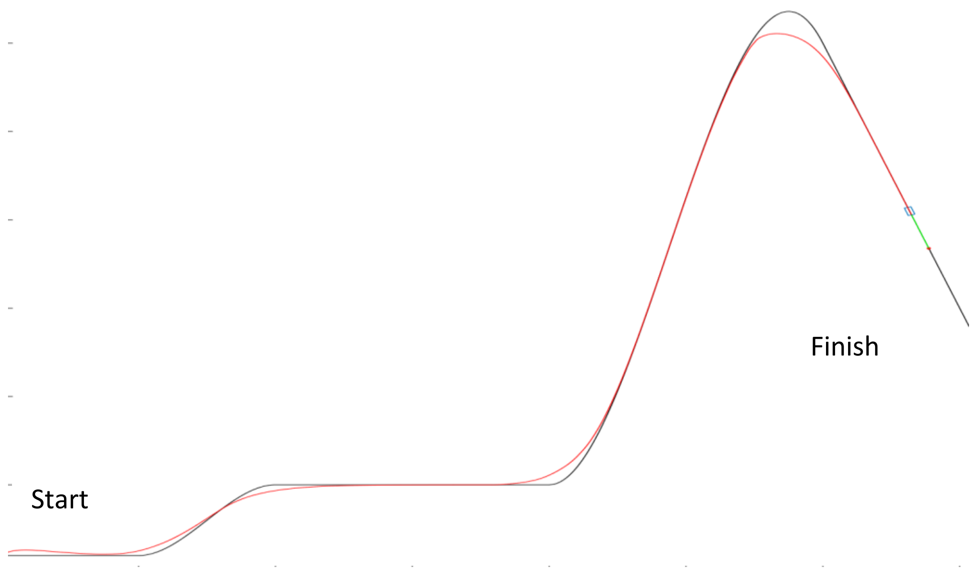

Figure 8. Follow the Carrot Algorithm.

As is possible to observe from Fig. 8, the vehicle trail indicated in red smoothly converges with the black path without overshooting or oscillating. The red and the black lines completely overlap in the straightway, meaning that no steady state error is present. However, the typical flaw of Pure Pursuit algorithms can be observed: the vehicle cuts corners. This issue is evident especially on sharp corners and this algorithm is therefore non satisfactory in high accuracy applications.

One way to tackle this issue is to come up with different error terms to use in the controller, but there is an easier and more effective way to improve the algorithm. It is sufficient to calculate the path that the carrot would have to follow for the vehicle to stay on the chosen path. At every moment in time, the desired position of the carrot can be calculated by following these procedures:

1. Find the closest point to the vehicle that is part of the path and call it point A.
2. Draw the tangent to the path that touches point A.
3. Find points B and C that lie on the tangent at distance $R0$ from A.
4. Discard point B on the back of the vehicle and keep point C.
5. Point C is where the carrot point should be to keep the vehicle on track.

If the direction of travel of the UGV is known in advance, it is possible to calculate beforehand the path that the carrot has to follow. This is done by carrying out steps 1 to 5 for every point in the chosen path. The result of this operation is illustrated in Fig. 9. The chosen path is presented

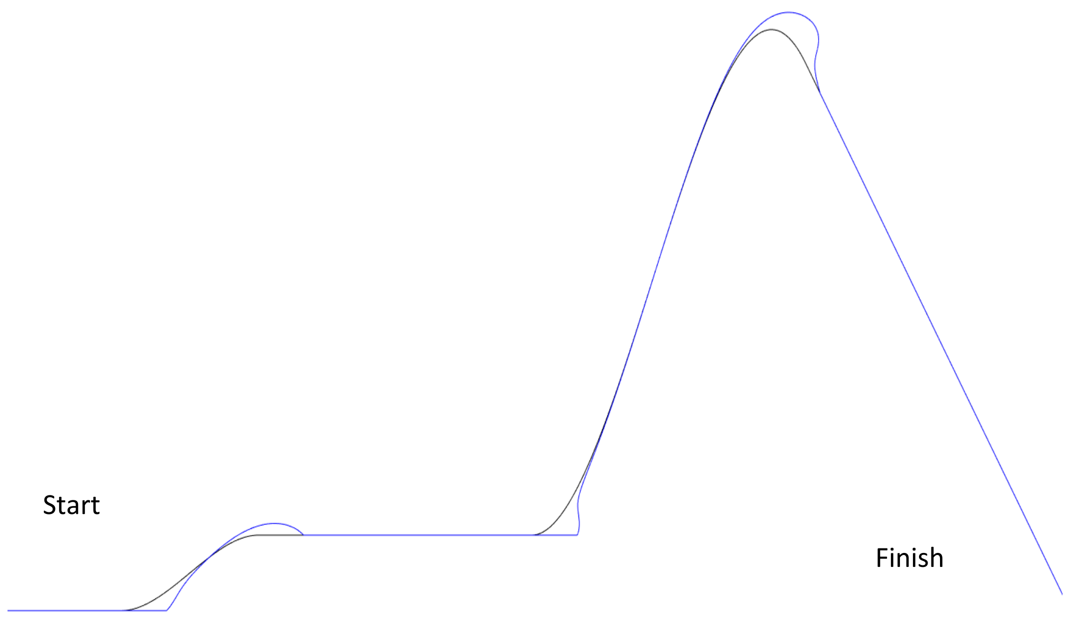

Figure 9. Carrot Path (Blue) & Vehicle Path (Black).

in black and the path to be travelled by the carrot is presented in blue.

This modification prevents the vehicle from cutting corners because it is allowed to go straight until it actually reaches a turn. The results obtained by the Modified Follow the Carrot Algorithm are shown in Fig. 10.

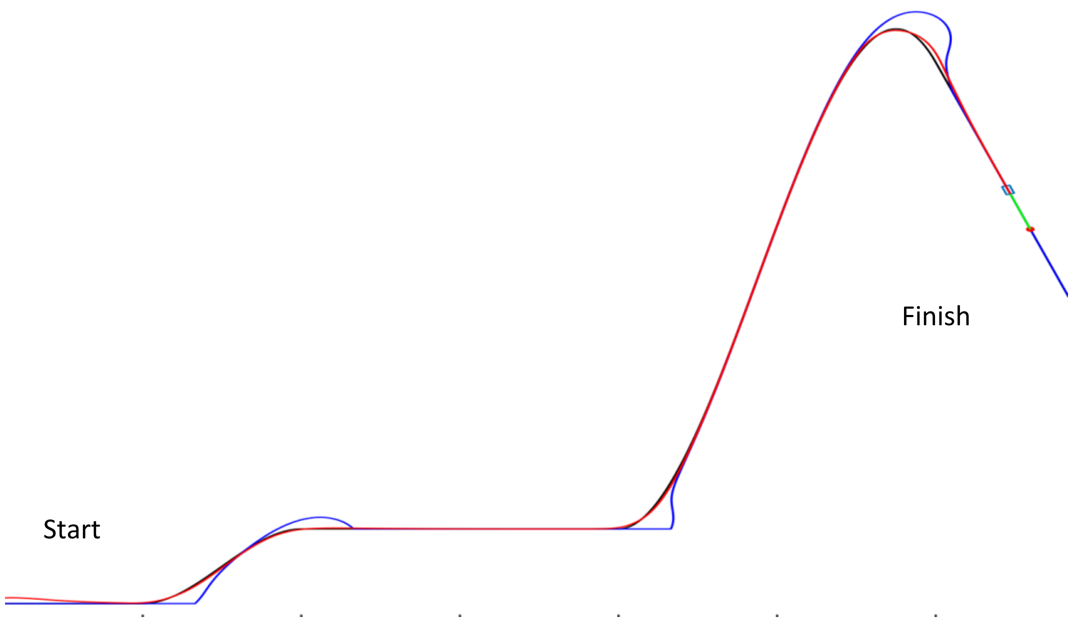

Figure 10. Modified Follow the Carrot Algorithm with Slip.

It is observable from Fig.10 that red line and the black line are almost completely superimposed. The minor discrepancies are due to the mechanical limitations of the vehicle, which can stir at a maximum angle of 42°. The plateau in Fig. 11 underlines the impossibility of the vehicle to follow the sharpest turns.

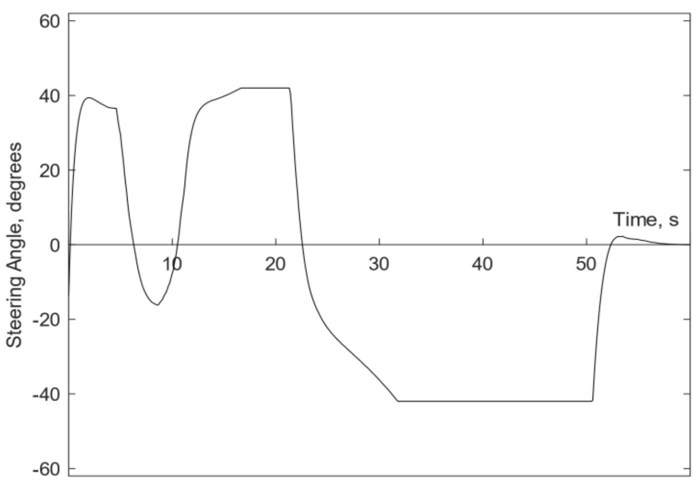

Figure 11. UGV steering angle vs time.

The effects of slip can also be overcome by modifying the course travelled by the carrot. It is just necessary to shorten the distance AC so that the vehicle starts to turn earlier. This expedient allows to obtain results indistinguishable from the ones shown in Fig. 10.

Pure Pursuit Algorithm and Stanley Method have also been tested but did not produce results as positive as the Follow the Carrot Algorithm. The Pure Pursuit Algorithm can also benefit from the method explained above, but its greater complexity does not allow to calculate the specific path that the carrot should follow. It can be obtained empirically but this places this algorithm one step below the Follow the Carrot. The Stanley method has been successfully implemented, but a way has not been found to improve it or mix it with the previous algorithms. The code for both algorithms can be found in the Appendix along with a graphical representation of their results.

The Modified Follow the carrot algorithm can be considered successful so and should be brought forward to be tested in a dynamic model. None of the research papers that have been examined for the creation of this thesis used this approach to improve path tracking algorithms. Therefore, the method and the MATLAB code described above can be considered an original contribution of this thesis.

Chapter 5

Conclusion

Pure Pursuit Algorithm is a powerful control system widely used in different fields like aerospace engineering and autonomous driving. This thesis has started from the most basic versions of the algorithm and expanded to a solution that could be a promising alternative to AI in low-end applications. Being able to contrast the effects of slip it is also a good candidate for farming vehicles. As stated in the results section, the Modified Follow the Carrot Algorithm can be considered successful and ready to be tested in a dynamic model.

6.1 Future Work

Future modifications could find a way to calculate the exact path that the carrot should follow for the Pure Pursuit Algorithm to obtain similar results to the ones shown above.

The Pure Pursuit Algorithm could be improved by establishing an inverse relationship between the tangential speed of the vehicle v and the curvature κ to mimic the behavior of human drivers that slow down before a sharp corner. The Follow the Carrot Algorithm could also be further improved by reducing the look-ahead distance during turns and increase it during straightway. This solution is again inspired to the behavior of human drivers.

When the algorithms will be applied to hardware, an Integral Term could be added to the equation to detect if the vehicle is spending more time on one side or the other of the path. This allows to take into account systematic errors due to defects in the hardware utilized and to correct the steering angle accordingly. This would create a combination of three that is referred to as PID Control (Proportional, Integral, Derivative Control) [10].

More complex algorithms could also be tested. Vector Pursuit is an algorithm that has been developed in the attempt to not only have the vehicle arrive at the goal point, but to arrive also with the correct orientation and curvature. This translates into the vehicle joining the course with the proper steering angle [4].

Bibliography

[1] Association for Safe International Road Travel (ASIRT), URL: https://www.asirt.org/safe-travel/road-safety-facts/ [cited 10 October 2018].

[2] D. Rose, "THE FOUR PILLARS OF SELF-DRIVING CARS," STRATEGYWISE, Mar. 2018, URL: https://strategywise.com/the-four-pillars-of-self-driving-cars/ [cited 14 October 2018].

[3] J. M. Snider, "Automatic Steering Methods for Autonomous Automobile Path Tracking," Robotics Institute Carnegie Mellon University, Pittsburgh, PE, 2009.

[4] M. Lundgren and T. Hellström, "Path Tracking for a Miniature Robot," Excerpt from master's thesis, Dept. of Computing Science, Umeå University, Sweden, 2003.

[5] J. Katupitiya "Modelling and Simulation of a Kinematic Model of a Car-like UGV," University of New South Wales, Australia, 2014.

[6] P. Polack, F. Altché, B. D'Andréa-Novel, and A. De La Fortelle "The Kinematic Bicycle Model: a Consistent Model for Planning Feasible Trajectories for Autonomous Vehicles?" HAL Polytechnique, France, 11 May 2017.

[7] L. Scharf, W. Harthill, and P. Moose, "A comparison of expected flight times for intercept and pure pursuit missiles," IEEE Transactions on Aerospace and Electronic Systems, vol. 4, pp. 672-673, 1969.

[8] M. Samuel, "A Review of some Pure-Pursuit based Path Tracking Techniques for Control of Autonomous Vehicle," International Journal of Computer Applications (0975 – 8887) Volume 135 – No.1, February 2016.

[9] R. Wallace, A. Stentz, C. E. Thorpe, H. Maravec, W. Whittaker, and T. Kanade, "First Results in Robot Road Following," IJCAI, pp. 1089-1095, 1985.

[10] A. C. Ritchie, "Systems modelling and control," Dynamics (MM2DYN), University of Nottingham, Dept. of Mechanical, Materials and Manufacturing Engineering, 2018.

[11] Y. Shan, W. Yang, C. Chen, J. Zhou, L. Zheng and B. Li "CF-Pursuit: A Pursuit Method with a Clothoid Fitting and a Fuzzy Controller for Autonomous Vehicles," International Journal of Advanced Robotic Systems, 12:134 doi: 10.5772/61391, 2015.

[12] T. Hellström and O. Ringdahl, "Autonomous Path Tracking Using Recorded Orientation and Steering Commands," Dept. of Computer Science Ume˚a University, Sweden, 2005.

[13] J. Katupitiya, "Computer Applications in Mechatronic Systems," MTRN3500-S2 2018, University of New South Wales, Australia.

[14] J. Katupitiya, "Computer Applications in Mechatronic Systems," MTRN3500-S2 2018, University of New South Wales, Australia.

[15] M. Yao, M Jia and A. Zhou "Applied Artificial Intelligence: A Handbook for Business Leaders," 2017.

[16] J. Taghia, S. Lam and J. Katupitiya "Path Following Control of an Off-Road Track Vehicle Towing a

Steerable Driven Implement," IEEE International Conference on Advanced Intelligent Mechatronics (AIM), Busan, Korea, 2015.

[17] R. Werner, S. Muller, and K. Kormann, "Path tracking control of tractors and steerable towed implements based on kinematic and dynamic modeling," in 11th International Conference on Precision Agriculture, Indianapolis, Indiana, USA, 2012.

[18] J. D'Errico, MathWorks File Exchange, URL: https://au.mathworks.com/matlabcentral/fileexchange/34874-interparc [cited 10 October 2018].

Appendix 1

A.1 Pure Pursuit Algorithm

```
% Name this file main.m
% Edoardo Cocconi - PURE PURSUIT ALGORITHM, REDUCED MODEL, SLIP
% This file simulates the model described in UPDN_UGV_model.m

% Cleaning the previous simulation
clear all
close all
clc

Create_Path2()

global x_path_calc_coordinates
global y_path_calc_coordinates
global x_path_coordinates
global y_path_coordinates
global point_sequence
global current_point
global data

[sz1,sz2] = size(x_path_calc_coordinates);
tfinal = sz2/270;

% Simulation
tspan = 0.0:0.1:60;
options = odeset('RelTol',1e-4,'AbsTol',[1e-5 1e-5 1e-5]); % options of
the ODE-solving function
[t,x] = ode45(@UPDN_UGV_model, tspan,[0 2 2*pi/8.0], options);
% [result] = ode45(function to solve, [start-time, end-time],
% initial conditions [x, y, teta, delta], options)
% this will derivate [xdot(1) xdot(2) xdot(3) xdot(4)] and give [x, y,
% theta, delta] of the UGV at any given time


%Contol file test.
NM = size(t);
N = NM(1);
point_sequence = zeros(N,1);
steering_angle = zeros(N,1);
current_point = 1;
```

```
for i = 1:N % Obtain the steering angles at each instant so they can be
plotted
    inputs = UPDN_UGV_control_pure_pursuit(t(i,1),x(i,:));
    point_sequence(i,1) = current_point;
    steering_angle(i,1) = inputs(2);
end

backup = x;
clear x
x = [backup(:,1), backup(:,2), backup(:,3),steering_angle(:,1)];

% Plotting of the data obtained in the simulation
UPDN_UGV_Path(t,x);
```

```
% Name this file Create_Path2.m
% Edoardo Cocconi - FOLLOW THE CARROT, REDUCED MODEL, SLIP
% This file creates the paths that the vehicle and the carrot have to
% follow

% +++++++++++++++++++SAME AS FOLLOW THE CARROT+++++++++++++++++++++ %
```

```
% Name this file UPDN_UGV_model.m
% Edoardo Cocconi - PURE PURSUIT ALGORITHM, REDUCED MODEL, SLIP
% UGV reduced model in file UPDN_UGV_model.m


% +++++++++++++++++++SAME AS FOLLOW THE CARROT+++++++++++++++++++++ %
```

```
% Name this file UPDN_UGV_control_pure_pursuit.m
% Edoardo Cocconi - PURE PURSUIT ALGORITHM, REDUCED MODEL, SLIP
% UGV controller in file UPDN_UGV_control.m

function inputs = UPDN_UGV_control_pure_pursuit(t,x)
% Initialisatio
% --- none
% State variables
% [x(1) x(2) x(3)]
% x(1) - global x coordinate (m.)
% x(2) - global y coordinate (m.)
% x(3) = \theta - UGV heading (rad.)

global current_point;
global x_path_calc_coordinates
global y_path_calc_coordinates

% Getting the coordinates of the point currently considered by the
algorithm
current_point = current_point + 35;
current_x = x_path_calc_coordinates(current_point);
current_y = y_path_calc_coordinates(current_point);

X_prime =  x(1) - current_x; %horizontal offset
Y_prime =  x(2) - current_y; % vertical offset

distance = sqrt(X_prime^2+Y_prime^2);

count = 0;
```

```
while distance > 4.0 && count < 35
    current_point = current_point-1;

    current_x = x_path_calc_coordinates(current_point);
    current_y = y_path_calc_coordinates(current_point);

    X_prime =  x(1) - current_x; %horizontal offset
    Y_prime =  x(2) - current_y; % vertical offset

    distance = sqrt(X_prime^2+Y_prime^2);

    count = count + 1;

end

% Coordinates of the point in vehicle in vehicle reference frame
X1 = X_prime*cos(x(3)) + Y_prime*sin(x(3));
Y1 = Y_prime*cos(x(3)) - X_prime*sin(x(3));

% inputs(1) - UGV speed (m/s.)
% inputs(2) - UGV steering rate (rad/s.)
% Declarations
inputs = zeros(2,1);

%control inputs for the close loop case.
inputs(1) = 2.0;
k1 = 1.0;

Ld = 3.0; % Fixed look-ahead distance
L = 0.9;
curvature = 2/(Ld^2)*Y1;

inputs(2) =  atan2(curvature*L,1); % Control system Eq. (4)
inputs(2) =  inputs(2) - sign(inputs(2))*pi();

if(abs(inputs(2)) > 42.0*pi/180.0) % The steering angle is limited to 42
degrees
    inputs(2) = sign(inputs(2))*42.0*pi/180.0;
end

return;
```

```
% Name this file UPDN_UGV_Path.m
% Edoardo Cocconi - PURSUIT ALGORITHM, REDUCED MODEL
% Path data plot function in file UPDN_UGV_Path.m

function [] = UPDN_UGV_Path(t,x)% This function plots the necessary
graphs after the simulation

global x_path_calc_coordinates
global y_path_calc_coordinates
global x_path_coordinates
global y_path_coordinates
global point_sequence
figure; % Plot figure 1
UGV = [ 0         0      -0.9000     -0.9000      0           0
        0   -0.2500     -0.2500      0.2500      0.2500      0]; % UGV
dimensions
```

```
%Contol file test.
NM = size(t);
N = NM(1);
Controls = zeros(N,2);
for i = 1:N % Obtain the steering angles at each instant so they can be
plotted
    inputs = UPDN_UGV_control_pure_pursuit(t(i,1),x(i,:));
    Controls(i,1) = inputs(1);
    %Controls(i,2) = inputs(2)+pi/90; % The steering angle are now in
the second row of the array
end

%sz = size(point_sequence);
for i = 1:size(t)% Real time animation of UGV motion and trajectory
    UGV_ugv = [cos(x(i,3)) -sin(x(i,3));sin(x(i,3)) cos(x(i,3))];
    ugv = UGV_ugv* UGV;
    plot(x(i,1)+ugv(1,:), x(i,2)+ugv(2,:)); hold on; % Plot the UGV
    plot(x_path_coordinates(1,:),y_path_coordinates(1,:),'k') % Plot the
path
    plot(x_path_calc_coordinates(1,:),y_path_calc_coordinates(1,:),'b')
    plot(x(1:i,1),x(1:i,2),'r') % Plot the actual path already percurred

plot(x_path_calc_coordinates(1,point_sequence(i,1)),y_path_calc_coordina
tes(1,point_sequence(i,1)),'r.')
    % Scale the axis
    axis equal;
    axis([min(x(:,1))-5.0, max(x(:,1))+5.0,min(x(:,2))-
5.0,max(x(:,2))+5.0]);
    % Plot the axis
    ax = gca;
    ax.XAxisLocation = 'origin'; % Plot x axis
    ax.YAxisLocation = 'origin'; % Plot y axis
    pause(0.001); % Slow down the output so it can be seen
    hold off % Cancels the effect of "hold on" and allows the new
position of the UGV to be plotted
end


figure(2); % Plot figure 2
delta = Controls(:,2).*180/pi; % Converts angles from radians to degrees
plot(t, delta,'k'); % Plot the steering angle vs time in degrees
axis([0.0, max(t),min(delta)-20.0,max(delta)+20.0]); % scale axis
ax = gca;
ax.XAxisLocation = 'origin'; %display x axis
%title('Steering angle vs time'); % Give a title to the figure
xlabel(' Time, s'); % label x axis
ylabel(' Steering Angle, degrees'); % label y axis


return;
```

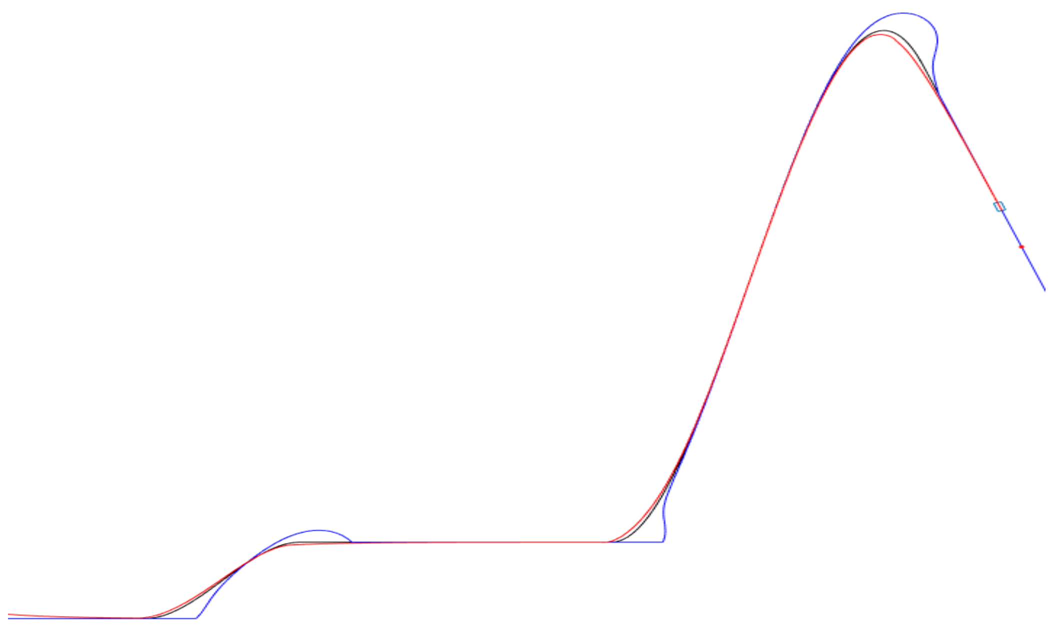

Figure 10. Modified Pure Pursuit Algorithm.

A.2 Stanley Method

```
% Name this file main.m
% Edoardo Cocconi - STANLEY METHOD, REDUCED MODEL, SLIP
% This file simulates the model described in UPDN_UGV_model.m

% Cleaning the previous simulation
clear all
close all
clc

Create_Path2()

%global x_path_calc_coordinates
%global y_path_calc_coordinates
global x_path_coordinates
global y_path_coordinates
global slope_path
global point_sequence
global current_point
global data

% Simulation
```

```
tspan = 0.0:0.1:60;
options = odeset('RelTol',1e-4,'AbsTol',[1e-5 1e-5 1e-5]); % options of
the ODE-solving function
[t,x] = ode45(@UPDN_UGV_model, tspan,[0 2 2*pi/8.0], options);
% [result] = ode45(function to solve, [start-time, end-time],
% initial conditions [x, y, teta, delta], options)
% this will derivate [xdot(1) xdot(2) xdot(3) xdot(4)] and give [x, y,
% theta, delta] of the UGV at any given time


%Contol file test.
NM = size(t);
N = NM(1);
point_sequence = zeros(N,1);
steering_angle = zeros(N,1);
current_point = 1;
for i = 1:N % Obtain the steering angles at each instant so they can be
plotted
    inputs = UPDN_UGV_control_stanley(t(i,1),x(i,:));
    point_sequence(i,1) = current_point;
    steering_angle(i,1) = inputs(2);
end

backup = x;
clear x
x = [backup(:,1), backup(:,2), backup(:,3),steering_angle(:,1)];

% Plotting of the data obtained in the simulation
UPDN_UGV_Path(t,x);
```

```
% Name this file Create_Path2.m
% Edoardo Cocconi - STANLEY METHOD, REDUCED MODEL, SLIP
% This file creates the paths that the vehicle and the carrot have to
% follow

function Create_Path2()

global current_point;
current_point = 0;

% Defining start and end points of the path
xinitial = 0.0;
xfinal = 10.0;
yinitial = 2.0;
yfinal = 2.0;
length = sqrt((xfinal-xinitial)^2+(yfinal-yinitial)^2);
n = length * 100;
n = ceil(n); %round to the next integer

slope1 = (yfinal-yinitial)/(xfinal-xinitial);

% Defining coordinates vectors
global x_path_coordinates
x_path_coordinates = linspace (xinitial,xfinal, n);
global y_path_coordinates
y_path_coordinates = linspace (yinitial,yfinal, n);

global slope_path
```

```
for i = 1:n
    slope_path = [slope_path, slope1];
end

xinitial_spline = xfinal;
yinitial_spline = yfinal;

fprintf('\n\tENTER NUMERICAL VALUES ONLY')
pause(1)
clc

continue_loop = 1;

while continue_loop

    xmemo = xfinal;
    ymemo = yfinal;

    answer1 = input('Enter the x coordinate of the starting point of the
next course segment:','s');
    xinitial = str2num(answer1);

    answer1 = input('Enter the y coordinate of the starting point of the
next course segment:','s');
    yinitial = str2num(answer1);

    answer1 = input('Enter the x coordinate of the ending point of the
next course segment:','s');
    xfinal = str2num(answer1);

    answer1 = input('Enter the y coordinate of the ending point of the
next course segment:','s');
    yfinal = str2num(answer1);

    slope2 = (yfinal-yinitial)/(xfinal-xinitial);

    spline_bounds_x = [xinitial_spline xinitial];
    spline_bounds_y = [yinitial_spline yinitial];

    n = 100;
    xx = linspace (xinitial_spline,xinitial, n);
    yy = spline(spline_bounds_x, [slope1 spline_bounds_y slope2],xx);

    [sz1, sz2] = size(xx);
    length_path = 0;
    for i = 1:sz2-1
        length_path = length_path + sqrt((xx(i+1)-xx(i))^2+(yy(i+1)-
yy(i))^2);
    end

    length_path = ceil(length_path); %round to the next integer
    n_path = length_path*100;

    [pt] = interparc(n_path,xx,yy,'spline');
    xx_equal_distance = pt(:,1);
    yy_equal_distance =  pt(:,2);

    xx_equal_distance = xx_equal_distance.'; % coloumn vector to row
vector
    x_path_coordinates = [x_path_coordinates, xx_equal_distance];
    yy_equal_distance = yy_equal_distance.'; % coloumn vector to row
vector
    y_path_coordinates = [y_path_coordinates, yy_equal_distance];

    equation = spline(xx_equal_distance, [slope1 yy_equal_distance
slope2]);
    spline_derivative = fnder(equation);
    slope_path = [slope_path,
fnval(spline_derivative,xx_equal_distance)];

    length = sqrt((xfinal-xinitial)^2+(yfinal-yinitial)^2);
    n = length * 100;
    x_path_coordinates_new = linspace (xinitial,xfinal, n);
    y_path_coordinates_new = linspace (yinitial,yfinal, n);

    x_path_coordinates = [x_path_coordinates, x_path_coordinates_new];
    y_path_coordinates = [y_path_coordinates, y_path_coordinates_new];

    for i = 1:n
        slope_path = [slope_path, slope2];
    end

    answer2 = input('Do you want to insert another segment? 1 for YES, 0
for NO:','s');
    continue_loop = str2num(answer2);

    xinitial_spline = xfinal;
    yinitial_spline = yfinal;
    slope1 = slope2;

end

return;
```

```
% Name this file UPDN_UGV_model.m
% Edoardo Cocconi - STANLEY METHOD, REDUCED MODEL, SLIP
% UGV reduced model in file UPDN_UGV_model.m

% +++++++++++++++++++SAME AS FOLLOW THE CARROT+++++++++++++++++++++ %
```

```
% Name this file UPDN_UGV_control_stanley.m
% Edoardo Cocconi - STANLEY METHOD, REDUCED MODEL, SLIP
% UGV controller in file UPDN_UGV_control.m

function inputs = UPDN_UGV_control_stanley(t,x)
% Initialisatio
% --- none
% State variables
% [x(1) x(2) x(3)]
% x(1) - global x coordinate (m.)
% x(2) - global y coordinate (m.)
% x(3) = \theta - UGV heading (rad.)

global current_point
global x_path_coordinates
global y_path_coordinates
global slope_path
```

```
% Getting the coordinates of the point currently considered by the
algorithm
current_point = current_point + 51;
current_x = x_path_coordinates(current_point);
current_y = y_path_coordinates(current_point);

X_prime =  x(1) - current_x; %horizontal offset
Y_prime =  x(2) - current_y; % vertical offset

distancemem = sqrt(X_prime^2+Y_prime^2);

current_x = x_path_coordinates(current_point-1);
current_y = y_path_coordinates(current_point-1);

X_prime =  x(1) - current_x; %horizontal offset
Y_prime =  x(2) - current_y; % vertical offset

distance = sqrt(X_prime^2+Y_prime^2);

count = 0;

while distance < distancemem && count < 50
    current_point = current_point-1;

    current_x = x_path_coordinates(current_point);
    current_y = y_path_coordinates(current_point);

    X_prime =  x(1) - current_x; %horizontal offset
    Y_prime =  x(2) - current_y; % vertical offset

    distance = sqrt(X_prime^2+Y_prime^2);

    count = count + 1;

end

% Coordinates of the point in vehicle in vehicle reference frame
X1 = X_prime*cos(x(3)) + Y_prime*sin(x(3));
Y1 = Y_prime*cos(x(3)) - X_prime*sin(x(3));

% inputs(1) - UGV speed (m/s.)
% inputs(2) - UGV steering rate (rad/s.)
% Declarations
inputs = zeros(2,1);

%control inputs for the close loop case.
inputs(1) = 2.0;
k1 = 0.1;

orientation_error = atan2(slope_path(current_point),1) - x(3);
offset_error = distance;

inputs(2) =  orientation_error - sign(Y1)*atan2(k1*offset_error,2); %
Control system Eq. (4)

if(abs(inputs(2)) > 42.0*pi/180.0) % The steering angle is limited to 42
degrees
    inputs(2) = sign(inputs(2))*42.0*pi/180.0;
end
```

```
return;
```

```
% Name this file UPDN_UGV_Path.m
% Edoardo Cocconi - FOLLOW THE CARROT, REDUCED MODEL, SLIP
% Path data plot function in file UPDN_UGV_Path.m

% This function plots the necessary graphs after the simulation
function [] = UPDN_UGV_Path(t,x)
global x_path_calc_coordinates
global y_path_calc_coordinates
global x_path_coordinates
global y_path_coordinates
global point_sequence
figure; % Plot figure 1
UGV = [ 0          0      -0.9000     -0.9000     0           0
        0   -0.2500     -0.2500     0.2500     0.2500      0]; % UGV
dimensions

%Contol file test.
NM = size(t);
N = NM(1);
Controls = zeros(N,2);
for i = 1:N % Obtain the steering angles at each instant so they can be
plotted
    inputs = UPDN_UGV_control_carrot(t(i,1),x(i,:));
    Controls(i,1) = inputs(1);
    Controls(i,2) = inputs(2); % The steering angle are now in the
second row of the array
end

for i = 1:size(t)% Real time animation of UGV motion and trajectory
    UGV_ugv = [cos(x(i,3)) -sin(x(i,3));sin(x(i,3)) cos(x(i,3))];
    ugv = UGV_ugv* UGV;
    plot(x(i,1)+ugv(1,:), x(i,2)+ugv(2,:)); hold on; % Plot the UGV
    plot(x_path_coordinates(1,:),y_path_coordinates(1,:),'k') % Plot the
path
    % Plot the path percurred by the carrot
    plot(x_path_calc_coordinates(1,:),y_path_calc_coordinates(1,:),'b')
    plot(x(1:i,1),x(1:i,2),'r') % Plot the actual path already percurred
    % Plot the carrot in real time

plot(x_path_calc_coordinates(1,point_sequence(i,1)),y_path_calc_coordina
tes(1,point_sequence(i,1)),'r.')
    %L is the length
    %angle is alpha
    x2=x(i,1)+(4*cos(x(i,4)+x(i,3)));
    y2=x(i,2)+(4*sin(x(i,4)+x(i,3)));
    plot([x(i,1) x2],[x(i,2) y2],'g') % Shows where the car is pointing
    % Scale the axis
    axis equal;
    axis([min(x(:,1))-5.0, max(x(:,1))+5.0,min(x(:,2))-
5.0,max(x(:,2))+5.0]);
    % Plot the axis
    ax = gca;
    ax.XAxisLocation = 'origin'; % Plot x axis
    ax.YAxisLocation = 'origin'; % Plot y axis
    pause(0.001); % Slow down the output so it can be seen
    hold off % Cancels the effect of "hold on" and allows the new
position of the UGV to be plotted
```

```
end


figure(2); % Plot the steering angle
delta = Controls(:,2).*180/pi; % Converts angles from radians to degrees
plot(t, delta,'k'); % Plot the steering angle vs time in degrees
axis([0.0, max(t),min(delta)-20.0,max(delta)+20.0]); % scale axis
ax = gca;
ax.XAxisLocation = 'origin'; %display x axis
xlabel(' Time, s'); % label x axis
ylabel(' Steering Angle, degrees'); % label y axis

return;
```

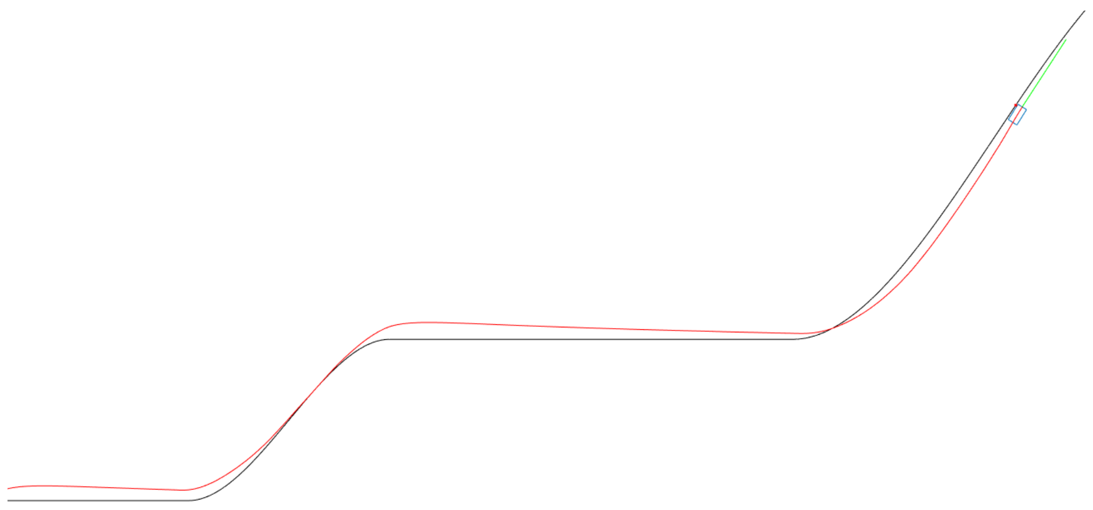

Figure 10. Stanley Method with Slip.