ETSEIB

Escola Tècnica Superior d'Enginyeria Industrial de Barcelona
Universitat Politècnica de Catalunya

Master's Degree in Industrial Engineering

Sliding mode control of a unicycle two type differential-drive mobile robot following a path

Arístides Barea Fernández

supervised by

Arnau Doria Cerezo

September, 2017

Abstract

This thesis only concerns the control design of a unicycle type differential-drive mobile robot following a path, using the Sliding Mode Control techniques. In the first place, the kinematic and dynamical models are found so that the mathematical analysis and the simulations can be performed. The model happens to be non-linear and its control needs two "state" variables of which only one can be measured. A linear observer solves the unmeasured variable. Two different modalities of movement include forward movement and backward movement. These modalities are substantially different due to the geometry of the vehicle, but require slightly different control analysis. The Lyapunov theorem for non-linear stability systems is applied in order to find the proper control action. Other details are simulated such us the sensor characteristics and the motors non linearities. Specifically, the dynamics of the motors are simulated but not implemented in the dynamical model. Future work could continue this thesis trying to design a control policy that acts directly over the electric impulse rather than the velocities of the vehicle.

The verification of the proposed control action is conducted with the Matlab Simulink software. This document includes diagrams and code so that the simulation model can be understood. In addition, a python app has been developed helping to animate the simulations and the important graphs that can proof the correct behaviour.

Summary

List of Figures

1 INTRODUCTION

1.1 Topic

This thesis is a part of a bigger project focused on the investigation around autonomous vehicles and intelligent management of circulation. It includes communication and coordination between vehicles. The project is at a laboratory stage where the vehicles that are being controlled are replicas of themselves. The typology of the vehicle can be described as a laboratory prototipe that follows a curve over the floor. It has two motored wheels and one infrared sensor located at a certain distance from the center point of the wheels axis. It is bound to follow a black trajectory painted over a white backround. The behaviour of the vehicle as an individual part of the whole system is essential to eventually apply the proper management policy for a group of vehicles. The individual behaviour could imply variation of velocity as well as forward and backward movement. Of course, it has to be reliable. It has to make no mistakes, like losing the track, for it is needed to avoid collisions and disorder.

1.2 Objectives

Framed within this topic, this thesis has the aim of designing a control policy based on sliding mode control in order to control the laboratory vehicle in backward and forward movement. The control will be designed over a non-linear system without trying to linearize it. There are other possible secondary objectives that include:

- The addition of the variation of linear velocity in the control design.
- The implementation of the control design in the microcontroller of the laboratory vehicle.
- The modelation of the system so that the electric impulses that excite the motors became the two control actions.
- Design of an app that helps to animate the simulations results.

1.3 Thesis scope

This thesis covers the parametrization, kinematic relations and dynamic model of the system formed by the vehicle and the track. It includes all de mathematica demonstrations of every equation used to model the system and its dynamics. It also covers the simulations with Matlab Simulink and data acquisition for the purpose of analysing results and draw conclusions. All the block diagrams are explained in this thesis as well as part of the equations used in the code. However, it does not cover the explanation of the code itself.

The animator program designed with Python is a secondary objective. Therefore, the document only provides an overview of its logics with no deeper analysis. The module used to make a graphic interface is Pygame. It cannot be expected a tutorial document. The main purpose of this topic is to provide conclusions about the Pygame usefullness in this ambit.

2 STATE OF THE ART

The kind of laboratory vehicle that follows a path is something that exists nowadays in different areas. Automated warehouse, robot competitions and even toys implement different control solutions. However, every solution is adapted in different ways depending on the typology of the robot and its sensors and actuators. The next sections aim to put the reader in the current situation offering an introduction to the needed theory that will be used in this document.

2.1 Previous works

The precedent to this document[3] covers, amongst other things, the modelation of the system and its simplification. It also proposes basic control options.

The two wheels are two actuators that can make the vehicle move and redirect. This is a coupled system due to the fact that the two control actions are not independent. The solution to this problematic is to apply a change of variables so that the linear velocity and the direction change velocity become the two control actions of the model. Hence, they become independent one from another. It provides the oportunity of using only one of the control actions while the other remains constant. While the linear velocity is set to a constant value, the velocity of the direction change is selected to be the control action. This way, the system becomes single input multiple output (SIMO).

Two basic controllers are designed in order to test the vehicle in the real world once it has been built: proportional and integral-proportional. The system is linearized in order to design both controllers. They both work with no significant differences. However, they can only provide a forward movement with asymptotic stability.

2.2 Sliding mode introduction

A variable structure system is composed of two or more continuous subsystems and a certain logic that commutes between them. In the design of the variable structure system, the control action becomes a discontinous function of the states. When the iteration from a subsystem to another occurs at a high frecuency, it is called a sliding mode or regime. It offers some advantages like robustness in front of uncertainty and perturbation, reduced order compensated dynamics and finite-time convergence, amongst others.

Let us consider the continuous system:

$$\begin{aligned}\dot{x} &= f(x) + g(x)u \\ y &= h(x)\end{aligned} \tag{2.1}$$

where x is the states vector and u the control action. Defining the conmutation function $s(x)$ with a $\nabla\sigma(x) \neq 0$ for every state, then the set

$$\sigma = \{x : s(x) = 0\} \tag{2.2}$$

defines an entity of conmutation that is called sliding surface. The control action u can be defined as a function of the sign of the sliding surface σ.

$$u_2 = \begin{cases} u^+ & \text{if} \quad \sigma(x) < 0 \\ u^- & \text{if} \quad \sigma(x) > 0 \end{cases} \tag{2.3}$$

The control action u is a function of the states. The two possible actions u^+ and u^- cannot be equal and they always satisfy $u^+ > u^-$.

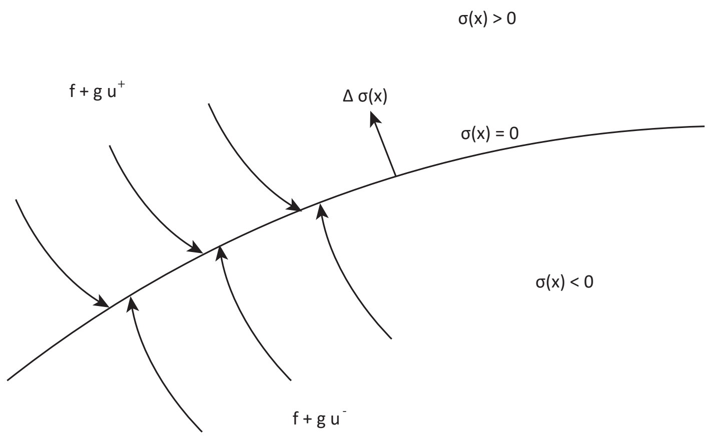

Figure 2.1: *The sign of σ is the logic that makes the control action conmute from one function to another (u^+ and u^-). The system is commuting along the time while the states ensure the oscilation around $\sigma = 0$.*

There is a sliding regime when the system gets to the surface σ and stays locally around it. It is important that the vectorial fields of the two continuous subsystems ($f + gu^+$ and $f + gu^-$) target locally toward the surface σ. Note that this kind of control provides a finit time approach.

The system, operating in the sliding mode, commutes ideally at an infinite frequency. That makes it impossible to find an analitic solution of the state equation. Another way to obtain the dynamics of a continuous system is to find the equivalent control. The equivalent control corresponds to the control action solution that makes the system stay at the sliding surface when it gets there.
In order to find the proper control action that ensures the sliding mode, a necessary condition must be secured. This condition recieves the name of *transversality condition.*

$$\frac{\partial \sigma}{\partial x} g(x) \neg 0 \tag{2.4}$$

A basic methodology can be used to design the sliding mode control:

- Select the sliding surface that provides the desired dynamics.
- Obtain the control law that surely will need a function sign of the sliding surface σ.
- Determine the sliding domain where the system will be stable.
- Analyze the stability of the ideal sliding dynamics.

3 THE MODEL

3.1 Kinematic relation

In the real world, the system is basically made of two elements: the vehicle itself and the trajectory it is going to follow. The vehicle has two motored wheels that can be controlled by the proper electric impulse. The sensor is located at a defined distance from the middle point between the two wheels. A third wheel gives mechanical stability to the chassis. In order to define and simulate any possible control action, the system must be characterized. It is necessary to establish a virtual coordinate system and find the kinematic model. The kinematic model gives us the relation between the velocity and the position of the vehicle. Eventually, it will also be needed the relation between the vehicle position and the desired position so that the dynamic model can be defined. But first we will start by defining the parameters and variables as follows.

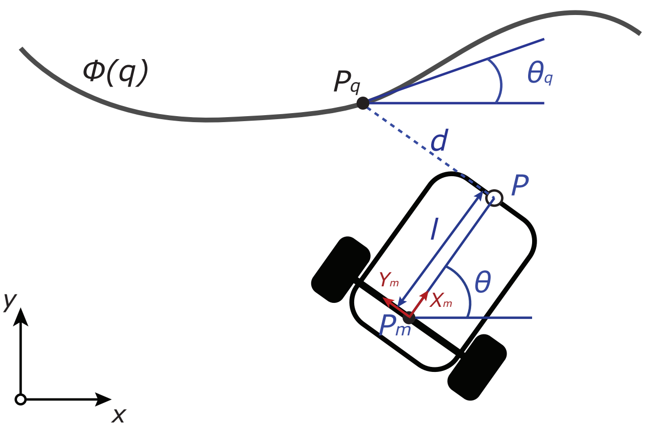

Figure 3.1: *Representation of the vehicle with the needed parameters and variables. l is the distance between the sensor and the wheels axes. P is the position of the sensor, and the point that must be controlled. P_m is the middle point between the wheels. Y_m and X_m are local axes. θ is the angle of the vehicle in global coordinates. d is the distance between P and P_m, it must tend to zero.*

- $Pm(x, y)$ is the middle point of the wheels axis.

- The axes X_m and Y_m are useful to describe the orientation of the vehicle. Their origin is the point Pm.
- $P(x, y)$ is the point where the infrared sensor is located. This is the point that has to be over the trajectory.
- P_q is the point in the trajectory where the point P is wanted to be. P_q is described as the intersection point in the trajectory with a line that goes perpendicular to the direction of the vehicle and begins in the point P.
- $\phi(q)$ is the name of the trajectory parametrized with its arc-length q. It is a vector that contains the position of Pq expressed in global axes.
- $\theta_q(q)$ is the angle between the tangent of the trajectory $\phi(q)$ and the global axis X.
- The variable d is the distance from the point P to P_q.

The kinematic model considers the velocity of the change of direction of the vehicle $\dot{\theta}$, as well as the velocity of the point Pm expressed in global axes (X,Y). The model is simplified using two new variables: u_1 and u_2 (see equation 3.2). This two new variables represent the linear velocity of the vehicle (u_1) and its direction change velocity (u_2). They allow us to control the vehicle by acting on the direction and linear velocity independently.

$$\begin{cases} \dot{x} = \cos(\theta)u_1 \\ \dot{y} = \sin(\theta)u_1 \\ \dot{\theta} = -u_2 \end{cases} \tag{3.1}$$

$$\begin{cases} u_1 = \frac{r}{2}(w_l + w_r) \\ u_2 = \frac{r}{2R}(w_l - w_r) \end{cases} \tag{3.2}$$

Variables w_r and w_l are the angular velocity of the right and left wheels respectively. R is the distance between the two wheels and r their radius.

3.2 Dynamic model

The process of defining the dynamical model starts with the question of which variables must be controlled. Thus, in order to make the vehicle follow the trajectory, two conditions must be met:

- $P_q = P$ or, what is the same, $d = 0$
- $\theta \simeq \theta_q$. We can suspect that the direction of the vehicle will approximate the direction of the tangent of the trajectory. However, this is a secondary condition since we are only interested in making it stable. Thus, the model found will give an answer to this relation.

In order to fulfill these conditions, the model must contemplate the dynamics of d and the relation between θ_q and θ. Firstly, the coordinates of P_q are expressed in terms of θ and P_m, being $R(\theta)$ the rotation matrix in the 2D space:

$$P_q = P_m + R(\theta)\begin{pmatrix} l \\ d \end{pmatrix} = \begin{pmatrix} \phi_x(q) \\ \phi_y(q) \end{pmatrix}$$

The next step is the differentiation of the equation with respect to the time.

$$\dot{P}_q = \dot{P}_m + R(\theta)\begin{pmatrix} 0 \\ \dot{d} \end{pmatrix} + \frac{\partial R(\theta)}{\partial \theta}\dot{\theta}\begin{pmatrix} l \\ d \end{pmatrix} = \dot{q}\begin{pmatrix} \frac{\partial \phi_x(q)}{\partial q} \\ \frac{\partial \phi_y(q)}{\partial q} \end{pmatrix}$$

We continue as follows, replacing $\dot{P}_m$ and $\dot{\theta}$ by the equations of the kinematic model (3.1).

$$\begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} u_1 + R(\theta)\begin{pmatrix} 0 \\ \dot{d} \end{pmatrix} - \frac{\partial R(\theta)}{\partial \theta}\begin{pmatrix} l \\ d \end{pmatrix} u_2 = \dot{q}\begin{pmatrix} \frac{\partial \phi_x(q)}{\partial q} \\ \frac{\partial \phi_x(q)}{\partial q} \end{pmatrix}$$

From now on, let us write $\frac{\partial \phi_x(q)}{\partial q}$ as $\partial \phi_x$ and $\frac{\partial \phi_y(q)}{\partial q}$ as $\partial \phi_y$. Now it is possible to isolate $\dot{d}$.

$$\begin{pmatrix} 0 \\ \dot{d} \end{pmatrix} = R^{-1}(\theta)\left[\begin{pmatrix} \partial \phi_x \\ \partial \phi_y \end{pmatrix}\dot{q} - \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} u_1 - \frac{\partial R(\theta)}{\partial \theta}\begin{pmatrix} l \\ d \end{pmatrix} u_2\right]$$

Where:

$$R^{-1}(\theta) = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}; \frac{\partial R(\theta)}{\partial \theta} = \begin{pmatrix} -\sin(\theta) & -\cos(\theta) \\ \cos(\theta) & -\sin(\theta) \end{pmatrix}$$

Thus:

$$\begin{pmatrix} 0 \\ \dot{d} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}\begin{pmatrix} \partial \phi_x \\ \partial \sigma_y \end{pmatrix}\dot{q} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} u_1 - \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} l \\ d \end{pmatrix} u_2$$

Finally, we extract the resulting two equations from above:

$$\begin{cases} \dot{d} = lu_2 - \Big(\partial \phi_x \sin(\theta) - \partial \phi \cos(\theta)\Big)\dot{q} \\ \dot{q} = \dfrac{u_1 - d \cdot u_2}{\partial \phi_x \cos(\theta) + \partial \phi_y \sin(\theta)} \end{cases} \tag{3.3}$$

Note that $\partial \phi_x = \cos(\theta_q)$ and $\partial \phi_y = \sin(\theta_q)$. It is so because the trajectory $\phi(q)$ is parametrized with its arclength q

$$\phi(q) = [\phi_x(q), \phi_y(q)]$$

and we know the angle θ_q at every point of the trajectory. Consequently

$$\begin{cases} \dot{d} = lu_2 - \Big(\cos(\theta_q) \cdot \sin(\theta) - \sin(\theta_q) \cdot \cos(\theta)\Big)\dot{q} \\ \dot{q} = \dfrac{u_1 - d \cdot u_2}{\cos(\theta_q) \cdot \cos(\theta) + \sin(\theta_q) \cdot \sin(\theta)} \end{cases}$$

Using the trigonometric relation

$$\cos(a - b) = \cos(a)\cos(b) + \sin(a)\sin(b)$$

we get

$$\dot{d} = lu_2 - \dot{q} \cdot \sin(\theta - \theta_q) \tag{3.4}$$

$$\dot{q} = \frac{u_1 - d \cdot u_2}{\cos(\theta - \theta_q)} \tag{3.5}$$

It is also necessary to model the dynamics of θ. In order to simplify it, we will create the variable $\theta_e = \theta - \theta_q$:

$$\dot{\theta}_e = \dot{\theta} - \dot{\theta}_q = \dot{\theta} - \frac{\partial \theta_q}{\partial q}\dot{q} \tag{3.6}$$

Where $\frac{\partial \theta_q}{\partial q}$ is the curvature $c = c(q)$ of the trajectory ϕ. Using the kinematic equation of θ (3.1) and substituting (3.5) into (3.4) and (3.6) we obtain:

$$\begin{cases} \dot{d} = lu_2 - tan(\theta_e)(u_1 + d \cdot u_2) \\ \dot{q} = \dfrac{1}{\cos(\theta_e)}(u_1 + d \cdot u_2) \\ \dot{\theta}_e = -u_2 - c\dfrac{(u_1 + d \cdot u_2)}{\cos(\theta_e)} \end{cases}$$

4 | CONTROL DESIGN

4.1 Model adaptation

We consider a new condition

$$\dot{q} = v$$

that makes the vehicle tend to go at a constant speed v. However, it does not mean that the vehicle will have a constant speed nor it will be v. As seen in the first chapter, the variable q is the length traveled along the path with which P_q is parametrized. Remember that P_q is the objective point where the vehicle point P should be at every moment.

$$P_q = \Big(\phi_x(q), \phi_y(q)\Big)$$

Its derivative $\dot{q}$ is the speed of the objective point P_q. It seems intuitive that if the vehicle follows the path thoroughly, its speed and $\dot{q}$ will be equal. In other words, the speed of P will be equal to the speed of P_q. If the curvature of the path is different from zero, the tangential speed will depend on which point of the vehicle we choose (it will depend on the distance to the center of rotation). Thus, the tangential speed of P (the sensor) will be differernt from u_1 (which is the speed of Pm). Consequently, u_1 may not be equal to $\dot{q}$ if the curvature is different from zero. Let us explain it with the proper equations. First, we assume that the control action u_1 ensures $\dot{q} = v$. We get the simplified model shown in the equation 4.1.

$$\begin{cases} \dot{d} = lu_2 - v\sin(\theta_e) \\ \dot{\theta}_e = -u_2 - c(t)v \end{cases} \tag{4.1}$$

Considering a working point given by any d^* and $\dot{q}^* = v$, the required control values u_1 and u_2 that ensure $\dot{d} = 0$ can be found as follows (note that $\dot{d} = 0$ implies $\dot{\theta}_e = 0$ if curvature c is constant). First we apply the conditions of the working point to the dynamical model:

$$0 = lu_2^* - v\sin(\theta_e^*) \tag{4.2}$$

$$0 = -u_2^* - cv \tag{4.3}$$

$$v = \frac{1}{\cos(\theta_e^*)}(u_1^* + d^* \cdot u_2^*) \tag{4.4}$$

Isolating u_2^* from 4.3 and replacing it in the equations 4.2 and 4.4, we obtain

$$\theta_e^* = \arcsin(-lc) \tag{4.5}$$

$$u_1^* = v\cos(\theta_e^*) + dcv \tag{4.6}$$

Replacing the equation 4.5 in 4.6, we obtain

$$u_1^* = v\cos(\arcsin(-lc)) + dcv = v[\cos(\arcsin(-lc)) + dc]$$

An easy trigonometric relation tells us that $\cos(\arcsin(-lc)) = \sqrt{1 - l^2c^2}$. It is explained as follows. We apply $\cos(t) = \sqrt{1 - sin^2(t)}$:

$$\cos(\arcsin(-lc)) = \sqrt{1 - \sin^2(\arcsin(-lc))}$$

using an auxiliary variable a we conclude

$$\begin{aligned}\sin^2(\arcsin(-lc)) &= a\\ \sin(\arcsin(-lc)) &= \sqrt{a}\\ -lc &= \sqrt{a}\\ a &= l^2c^2\end{aligned}$$

Thus, the required control values u_1^* and u_2^*, that ensure the working point, and the corresponding deviation angle are

$$u_1^* = v(\sqrt{1 - l^2c^2} + cd^*) \tag{4.7}$$

$$u_2^* = -cv \tag{4.8}$$

$$\theta_e^* = \arcsin(-cl) \tag{4.9}$$

As a conclusion of equation 4.7, if the distance d is set to $d^* = 0$, the control signal u_1^* only depends on the distance l and the curvature c. The curvature c is not known. Therefore, we can only approximate it assuming $c = 0$. Whether the distance l equals $l = 0$ or the path has curvature $c = 0$, the speed u_1 of the vehicle equals $u_1 = v$. The equation 4.7 has the term l^2c^2 that entails the maximum curvature constraint given by

$$c_{max} = \frac{1}{l} \tag{4.10}$$

or

$$l_{max} = \frac{1}{c_{max}} \tag{4.11}$$

4.1.1 Moving forward

The sliding mode control consists in making the vehicle go directly to the path and slide along it. We know in advance that the control will be an on/off type. Then, according to the control objective $d = 0$, we define the sliding surface

$$\sigma = d \tag{4.12}$$

The sliding surface is the equation that will be controlled. The equivalent control is the control applied when the vehicle is actually over the line. That means $d = 0$. In that moment, we want $\dot{d} = 0$ in order to make it slide along surface. In other words, we want d to ramain in the value $d = 0$ while the vehicle is moving. Then, the equivalent control can be found by making $\dot{\sigma} = 0$. Isolating u_2^* from the equation 4.1 when $\dot{d} = 0$ we obtain:

$$u_2^* = \frac{v}{l}\sin(\theta_e) \tag{4.13}$$

The signal u_2 equals u_2^* (equation 4.13) when $\dot{\sigma} = \dot{d} = 0$. When the vehicle is not in the objective situation, it must tend to it. Thus, the Lyapunov theorem for non linear systems stability must be applied so that the sliding surface becomes a stability point. The Lyapunov equation chosen (equation 4.14) responds to the two first conditions of Lyapunov (4.15).

$$V = \frac{1}{2}\sigma^2 \tag{4.14}$$

The three conditions of Lyapunov are

$$\begin{cases} V(0) = 0 \\ V(\sigma \neq 0) > 0 \\ \dot{V} < 0 \end{cases} \tag{4.15}$$

The third condition gives us some freedom to find the necessary control action (equations 4.16 and 4.17).

$$\dot{V} = \frac{\partial V}{\partial \sigma}\dot{\sigma} = \sigma\Big(lu_2 - v\sin(\theta_e)\Big) < 0 \tag{4.16}$$

$$u_2 = u_2^* - \frac{\rho}{l}\mathrm{sign}(d) \tag{4.17}$$

The control action (4.17) guarantees the third condition of Lyapunov (inequation 4.16). Thus, it guarantees $\sigma \to 0$ in a finite time. Considering a switching control action, the control policy can be defined such that the control action has two possible values:

$$u_2 = \begin{cases} u_2^{max}, & \text{if} \quad \sigma < 0 \\ u_2^{min}, & \text{if} \quad \sigma > 0 \end{cases}$$

However, the remaining dynamics of θ_e is not necessary stable. Applying $\dot{d} = 0$ in the dynamical model (4.1) we get the dynamics of θ_e as follows:

$$\dot{\theta}_e = -\frac{v}{l}(\sin(\theta_e) + cl)$$

To proof its stability we can use a phase portrait where the X axis is θ_e and the Y axis is $\dot{\theta}_e$. Figure 4.1 shows that the system is stable between $-\pi - \theta_e^*$ and $\pi - \theta_e^*$. The arrows indicate the tendency depending on the sign of θ_e and $\dot{\theta}_e$. Note that this is true as long as $v > 0$. In the case that $v < 0$, the arrows of the diagram will be inversed and the equilibrium point θ_e will not be stable.

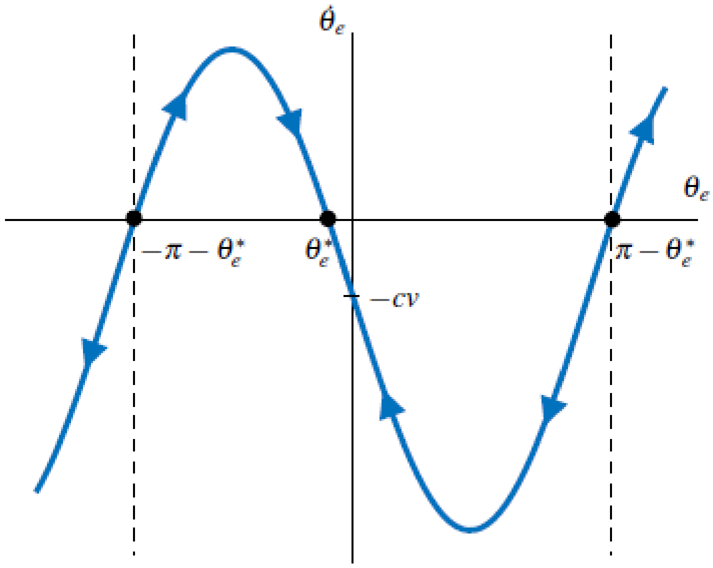

Figure 4.1: $\dot{\theta}_e$ *is plotted as a function of* θ_e*.* θ_e *increases its value when* $\dot{\theta}_e$ *is positive. Likewise when* $\dot{\theta}_e$ *is negative,* θ_e *decreases. This behaviour is shown with the arrows. The stability point is* θ_e^* *for a region limited by the interval* $[-\pi - \theta_e^*, \pi - \theta_e^*]$*.*

4.1.2 Moving backward

As pointed out at the begining of this section, the system has local stability if $v > 0$ in $\theta_e^* = \arcsin(-c \cdot l)$. Otherwise, the system has no stability in that region. To make it stable for $v < 0$ the sliding surface must contain the variable θ_e.

$$\sigma = d + \beta\tilde{\theta}_e \tag{4.18}$$

The new variable $\tilde{\theta}_e$ is defined such that θ_e must be equal to θ_e^* when d $= 0$ to make $\sigma = 0$:

$$\tilde{\theta}_e = \theta_e - \theta_e^* \tag{4.19}$$

The equivalent control is found by making $\dot{\sigma} = 0$

$$\dot{d} = -\beta\dot{\tilde{\theta}}_e \tag{4.20}$$

Replacing the equation 4.20 by the dynamical model, we get

$$l \cdot u_2 - v \cdot \sin(\tilde{\theta}_e) = -\beta[-u_2 - c \cdot v] \tag{4.21}$$

Isolating u_2 we obtain the equivalent control:

$$u_2^* = \frac{v}{l-\beta}(\sin(\tilde{\theta}_e) + \beta c) \tag{4.22}$$

Once again, Lyapunov must be applied in order to find the control action u_2 that makes the system locally stable. The Lyapunov equation

$$V = \frac{1}{2}\sigma^2 \tag{4.23}$$

must have a negative derivative

$$\dot{V} = \sigma\dot{\sigma} < 0 \tag{4.24}$$

Derivating σ from (4.18)and replacing it in (4.24) we obtain

$$\dot{V} = \sigma[lu_2 - v\sin(\tilde{\theta}_e) - \beta(u_2 + cv)] < 0 \tag{4.25}$$

reorganizing the terms of the equations we get

$$\dot{V} = \sigma[u_2(l-\beta) - v(\sin(\tilde{\theta}_e) + \beta c)] < 0 \tag{4.26}$$

Finally, the control action that makes $\dot{V} < 0$ is:

$$u_2 = u_2^* - \frac{\rho}{l-\beta}\text{sign}(\sigma) \tag{4.27}$$

The stability of θ_e can be proven by using the equivalent control (4.22) in the dynamical system

$$\dot{\theta}_e = -\frac{v}{l-\beta}(\sin(\theta_e) + cl) \tag{4.28}$$

As seen in the *Moving Forward* section, a phase portrait can prove the local stability of θ_e. This time the condition to make θ_e^* locally stable is that $\frac{v}{l-\beta} > 0$. Thus, the term β can change its value in order to change from a fordward movement to backward.

4.2 Full state observer

In order to properly apply the control action, the $\tilde{\theta}_e$ should be known. However, it is not sensed and it cannot be known. Consequently, if possible, it must be observed. To do that, the lineal observer of Luenberger has been proposed. First of all, the dynamical model of the vehicle needs to be liniarized. The general equation used for the linearization of multivariable systems at a specific equilibrium point is

$$\begin{aligned} f(x,u) &\approx f(x^*,u^*) + \frac{\partial f}{\partial x_{x=x^*}}(x-x^*) + \frac{\partial f}{\partial u_{u=u^*}}(u-u^*) \\ h(x,u) &\approx h(x^*,u^*) + \frac{\partial h}{\partial x_{x=x^*}}(x-x^*) + \frac{\partial h}{\partial u_{u=u^*}}(u-u^*) \end{aligned} \tag{4.29}$$

In the equilibrium point we know that

$$f(x^*, u^*) = 0$$
$$h(x^*, u^*) = y^*$$

and defining the next new variable of state:

$$X = x - x^*$$
$$U = u - u^*$$
$$Y = y - y^*$$

The new system can be expressed as follows

$$\begin{aligned} \dot{X} = \dot{x} \approx \frac{\partial f}{\partial x_{x=x^*}} X + \frac{\partial f}{\partial u_{u=u^*}} U \\ Y = y - y^* \approx \frac{\partial h}{\partial x_{x=x^*}} X + \frac{\partial f}{\partial u_{u=u^*}} U \end{aligned} \tag{4.30}$$

Hence, linearizing the dinamical model

$$\begin{cases} \dot{d} = l u_2 - v \sin(\theta_e) \\ \dot{\theta}_e = -u_2 - c(t) v \end{cases} \tag{4.31}$$

we obtain

$$\begin{pmatrix} \dot{d} \\ \dot{\theta}_e \end{pmatrix} = \begin{pmatrix} 0 & -v\cos(\theta_e^*) \\ 0 & 0 \end{pmatrix} \begin{pmatrix} d - d^* \\ \theta_e - \theta_e^* \end{pmatrix} + \begin{pmatrix} l \\ -1 \end{pmatrix} (u_2 - u_2^*)$$

In order to know whether the system can be observed or not, we analyze the observability matrix[6] when the output $y = d$:

$$W_0 = \begin{pmatrix} 1 & 0 \\ 0 & -v\cos(\theta_e^*) \end{pmatrix} \tag{4.32}$$

The observability matrix (4.32) has full rank as long as $\theta_e^* \neq \pm\frac{\pi}{2}$. This situation means that the vehicle is going perpendicular to the path and has two possible the system is observable. However, the equilibrium point θ_e^* is not known. It depends on the curvature (see equation 4.9 of the *Sliding Mode Control* section). Considering a curvature $c = 0$, we can say that $\theta_e^* = 0$. Then, the observer is as follows:

$$\begin{pmatrix} \dot{\hat{d}} \\ \dot{\hat{\theta}}_e \end{pmatrix} = \begin{pmatrix} 0 & -v\cos(\theta_e^*) \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{d} \\ \hat{\theta}_e \end{pmatrix} + \begin{pmatrix} l \\ -1 \end{pmatrix} u_2 + \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} (d - \hat{d})$$

It yields the following state matrix:

$$\begin{pmatrix} -L_1 & -v \\ -L_2 & 0 \end{pmatrix}$$

With eigenvalues

$$\lambda = \frac{-L_1 \pm \sqrt{L_1^2 + 4vL_2}}{2}$$

Then, the observer must comply two conditions to be stable:

$$\begin{cases} L_1^2 + 4vL_2 < 0 \\ L_1 > 0 \end{cases}$$

Figure 4.2 shows the observer diagram in Matlab Simulink. The block A is the state matrix of the linearized system. The block L is the feedback matrix

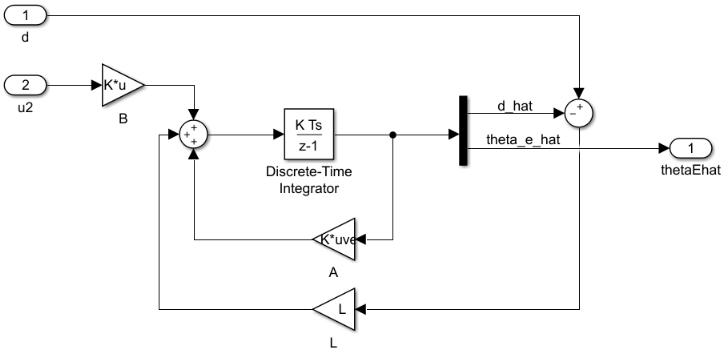

Figure 4.2: *Luenberger observer designed to return the estimation of θ_e (thetahat). The gain A is the matrix that multiply the variables $\hat{d}$ and $\hat{\theta}_e$. The gain B is the matrix that multiplies the control action u_2. L is the gain of the error $d - \hat{d}$.*

5 REALISTIC DETAILS

5.1 Motor Dynamics

As seen in the first section, the two variables u_1 and u_2 are defined in relation to the speed of the wheels:

$$\begin{cases} u_1 = \frac{r}{2}(w_l + w_r) \\ u_2 = \frac{r}{2R}(w_l - w_r) \end{cases}$$

The motors of the wheels have their own dynamics that have not been contemplated so far. In fact, the designed control described in the previous section acts directly on the speed. In the real world, that is impossible. The microcontroller sends an electrical signal (that must be amplified) that stimulates the DC motors inducing the corresponding variation of angular velocity. The electrical signal is not analogycal but a PWM type. The PWM consist in sending pulses with the same voltage and frequency but different prolongation in time. This is a common techic used in controlling DC motors that takes advantage of the rotor inertia. The "duty cycle" of the PWM determines the percentage of time that the signal is in high voltage compared with the cycle period. The cycle period is the time of a whole cycle between two pulses. A non linear relation between the duty cycle and the angular velocity can be found as an equation in a stationary state. However, there will be a delay until the motor achieves the velocity that corresponds to a determined duty cycle. Let us say that the real velocity (Vr) of the motor can be expressed as a function of the duty cycle in a stationary situation.

$$Vr = f(dutycycle) \tag{5.1}$$

Let us say that the duty cycle can be defined as a function of the desired velocity (Vd).

$$DutyCycle = f(Vd) \tag{5.2}$$

Then, the real velocity would be a linear function of the desired velocity. In fact, they are the same. The delay between the specification of the desired velocity and the moment when the motor achives that velocity, can be modeled by a first order transfer function as follows.

$$\frac{1}{\tau s + 1} \tag{5.3}$$

The time constant τ is the time that the motor needs to reach the 63% of the velocity goal. In this case, the constant of both motors is set to $\tau = 0,01s$. This dynamic is not considered neither in the model nor in the control action design. Consequently, the sliding mode control, defined in the previous section, might fail. In order to prove the robustness of the sliding mode control before this new situation, a new block has been added to the simulink diagram. This block emulates the dynamics of the motors by changing the behaviour of the action u_2.

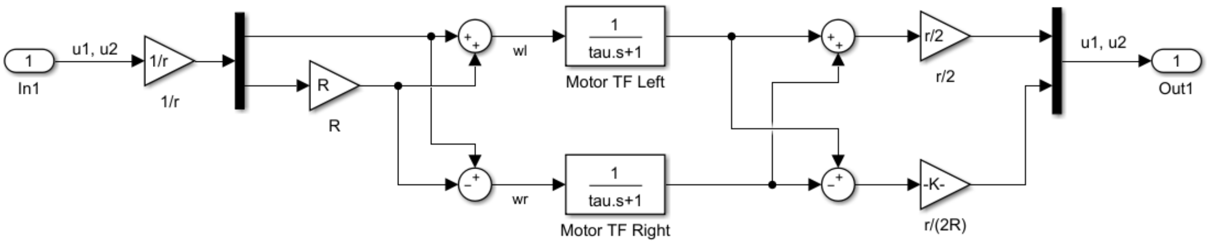

Figure 5.1: *The motor dynamics block emulates the behaviour of the motors by changing the control action.*

Figure 5.1 shows that the control actions u_1 and u_2 are changed in order to emulate the behaviour when the control actions are not the velocities but the electrical impulses.

5.2 Sensor

The sensor uses two light intensity inputs to calculate the relative position of the path. When the path gets closer to one spot, that spot receives more black than the other. It indicates that the variable d is not zero. A specific algorithm can be performed to translate these intensity inputs in a determined distance. However, this algorithm is not discussed in this document. Figure 5.2 is a sketch that shows the behaviour of the sensor according to the distance d. When the sensor gets far enough from the trajectory, it cannot be known whether the path is at the left or the right of the sensor point (P). The control design does not take into account the sensor limitation. Therefore, due to the little margin of the sensor, the control design could fail if the vehicle gets far enough.

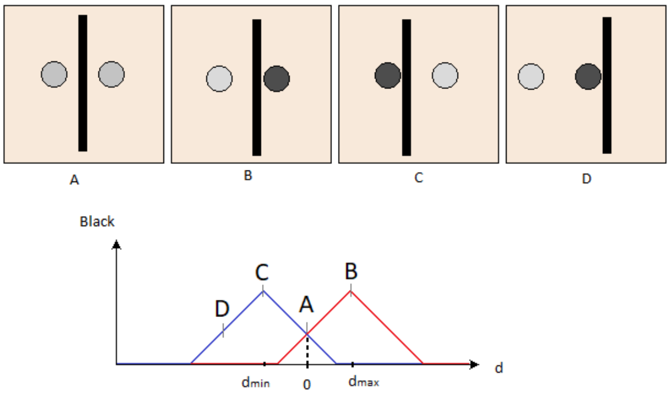

Figure 5.2: *Sketch of the sensor behaviour. The images A, B, C and D show four example situations where the two spots of the sensor recieve different light intensity. The input of the sensor depends on the position of the path between the two spots.*

The Simulink diagram includes the block *Sensor* in order to emulate the limitation of the sensor. However, in this case the value of d equals zero when exceeds the interval $[d_{min}, d_{max}]$.

$$d = d * (d < d_{max}) * (d > d_{min})$$

The terms $(d < d_{max})$ and $(d > d_{max})$ equal zero when they are false. Despite th

6 SIMULATION DIAGRAMS

This chapter aims to explain how the matlab simulink model works. Figure 6.1 gives us an overview of the simulation diagram. Some of the blocks contained in the diagram have already been explained in previous chapters. Thus, they are not covered in this chapter.

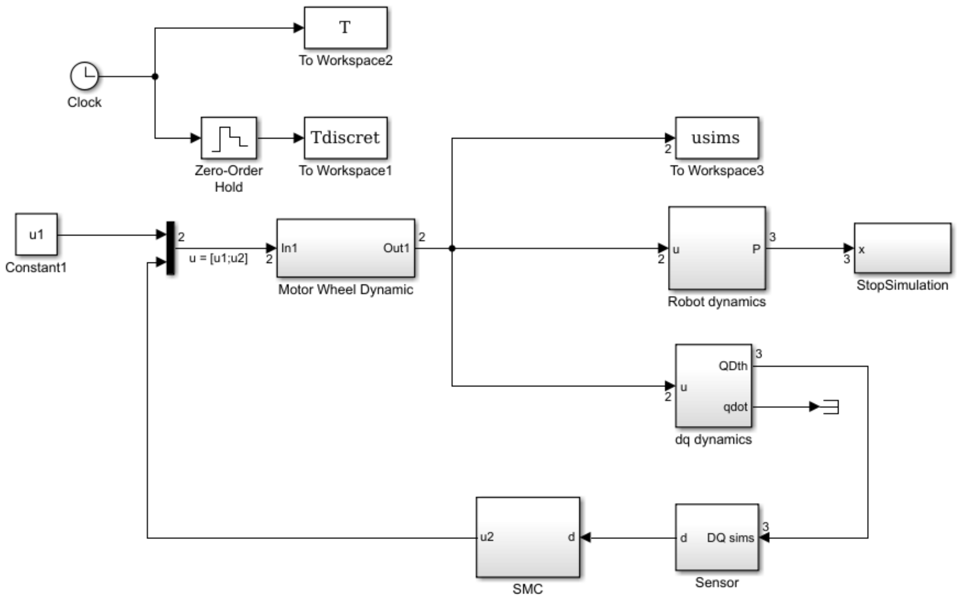

Figure 6.1: *Overview of the matlab diagram.*

6.1 Robot dynamics block

The Robot Dynamics block integrates the kinematic model (equation 6.1) and uses trigonometry relations to completely define the position of the vehicle (it calculates the position of the wheels, the point P and P_m). The integration of u_2 gives θ. With θ and u_1, it is possible to find the values of x and y (that are the coordinates of the point P_m) by integrating them. Once the point P_m is known, the position of the wheels and the point P can be easily found with trigonometry.

$$\begin{cases} \dot{x} = \cos(\theta)u_1 \\ \dot{y} = \sin(\theta)u_1 \\ \dot{\theta} = -u_2 \end{cases} \tag{6.1}$$

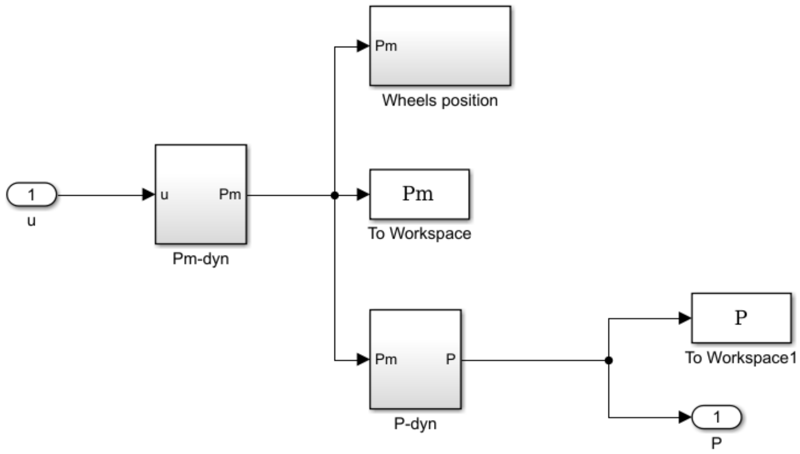

Figure 6.2: *The robot dynamics diagram contains three blocks. The Pm-dyn calculates de coordinates of the point P_m by integating u_1 and u_2 using the kinematic model. The other two blocks use trigonometry to calculate the position of the wheels and the point P* from the calculated point P_m.

6.2 DQdynamics block

The *DQdynamics* block (figure 6.3) uses a script that calculates the derivatives of d and q by using the equations of the dynamical model (equations 6.2). Section 3.2 covers the demonstration of the dynamical model.

$$\begin{cases} \dot{d} = lu_2 - \Big(\partial\phi_x \sin(\theta) - \partial\phi \cos(\theta)\Big)\dot{q} \\ \dot{q} = \dfrac{u_1 - d \cdot u_2}{\partial\phi_x \cos(\theta) + \partial\phi_y \sin(\theta)} \end{cases} \tag{6.2}$$

Different tracks or trajectories (ϕ), expressed in the 2D space coordinates $\big(\phi_x(q), \phi_y(q)\big)$, have different $\partial\phi_x$ and $\partial\phi_y$ (remember that $\partial\phi_x$ and $\partial\phi_y$ are the abreviation for $\frac{\partial\phi_x}{\partial q}$ and $\frac{\partial\phi_y}{\partial q}$). The shape of the track (or trajectory) is defined by setting those partial derivatives into the model with the script already mentioned. As an example, the partial derivatives of the trajectory with a semicircle shape and radius A are

$$\partial\phi_x = A\sin(q)$$
$$\partial\phi_y = A\cos(q)$$

After the calculation of the partial derivatives of the trajectory, the values of $\dot{d}$ and $\dot{q}$ can be found with the dynamical model equations (equations 6.2) and the values of u_1 and u_2.

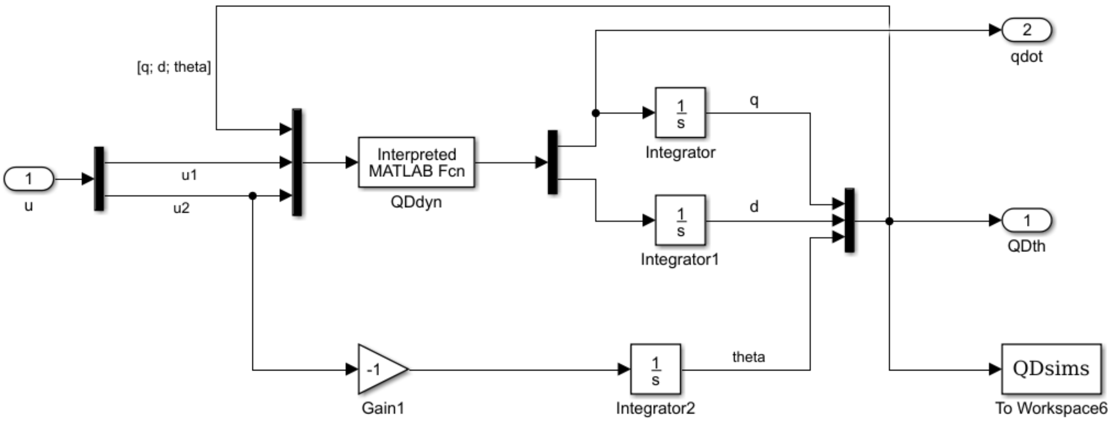

Figure 6.3: *The DQdynamics block calculates the current values of d, q and θ by integrating the dynamical model and setting the parameters of the trajectory. The QDdyn script calculates the derivatives $\dot{d}$ and $\dot{q}$ that will be integrated.*

To sum up, as shown shown in figure 6.3, and for each iteration:

- The values of q, d and θ are found by integrating $\dot{q}$, $\dot{d}$ and u_2 respectively.
- The values of $\dot{q}$ and $\dot{d}$ are calculated in the the script *QDdyn* with the equations of the dynamical model and setting the proper partial derivatives of the path.

6.3 Sliding mode control block

The control design has been explained in previous chapters, as well as the observer needed. Therefore, this section only explains its modelation in Matlab Simulink. It is recomended to previously read the chapter *Control design*.

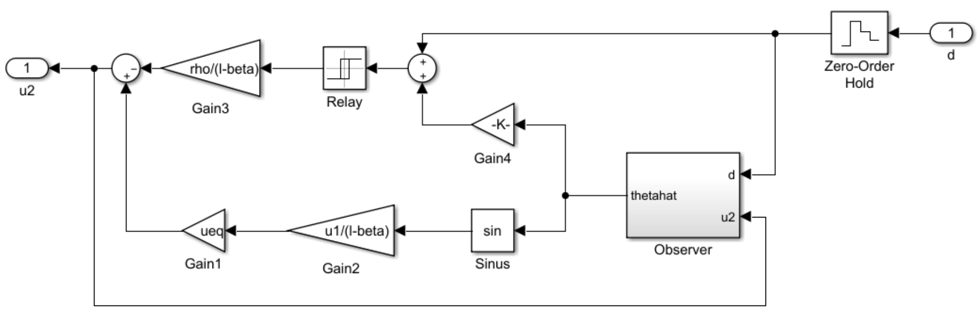

Figure 6.4: *The SMC block simulates the sliding mode control applyied in discrete time. Gain4: β. Gain1: equivalent control.*

As shown in figure 6.4, a zero-order hold must be included so that the controller works in discrete time. The frequency of the holder has been set to $5 \cdot 10^{-4}$ in order to adjust to a microcontroller frequency. When the velocity of the vehicle is set to positive the vehicle goes forward. In that case, the paramter β of the control action is set to zero. In the case that the velocity is negative, which means that the vehicle goes backwards, the paramter β will need to be modified depending on the characteristics of the vehicle and the path. A relay is added to work as the function $sign(\sigma)$. The behaviour of the relay is as follows:

$$\begin{cases} -1 & \text{if } \sigma < -0,001 \\ 1 & \text{if } \sigma > 0,001 \end{cases}$$

7 SIMULATIONS RESULTS

7.1 Methodology

The simulation tests consist in four different tracks (or trajectories) and two directions for each one: forward and backward. The different tracks are parametrized as follows.

Track 1: linear trajectory
$\phi(q) = \big(q, 0\big)$

Track 2: linear trajectory with sudden deviation
$\phi(q) = \big(q, m(q - q_r)(q > q_r)\big)$

Track 3: circular trajectory
$\phi(q) = \big(D - D\cos(q), D\sin(q)\big)$

Track 4: sinusoidal trajectory
$\phi(q) = \big(q, A\sin(2\pi q)\big)$

The term $(q > q_r)$ of the second track equals 0 when false and 1 when true. The term q_r is the value of q when the deviation starts. In the third track, the term D is the diametre of the circle. The parameters set in the simulations are also important:

- The perpendicular length from the wheels axis to the sensor l.
- The distance from each wheel to the middle point of the axis R.
- The radius of the wheels r.
- The parameter $beta$, used in the calculation of the control action u_2.

In order to extract conclusions from the simulations, different conditions must be applied regardless of the size of the real vehicle that we could prove at the laboratory. The following sections show some of the different conditions that have been simulated.

7.2 Forward movement with sensor emulator

Figures 7.1, 7.2, 7.3 and 7.4 show the trajectory followed by the vehicle in forward movement. Despite the oscilations, the control design works properly in forward movement.

The next parameters have been applied: $l = 0,05$, $R = 0,05$, $r = 0,02$ and $beta = 2,5l$. The three black spots represent the wheels, the sensor (point P) and the middle point between the wheels (point P_m) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).

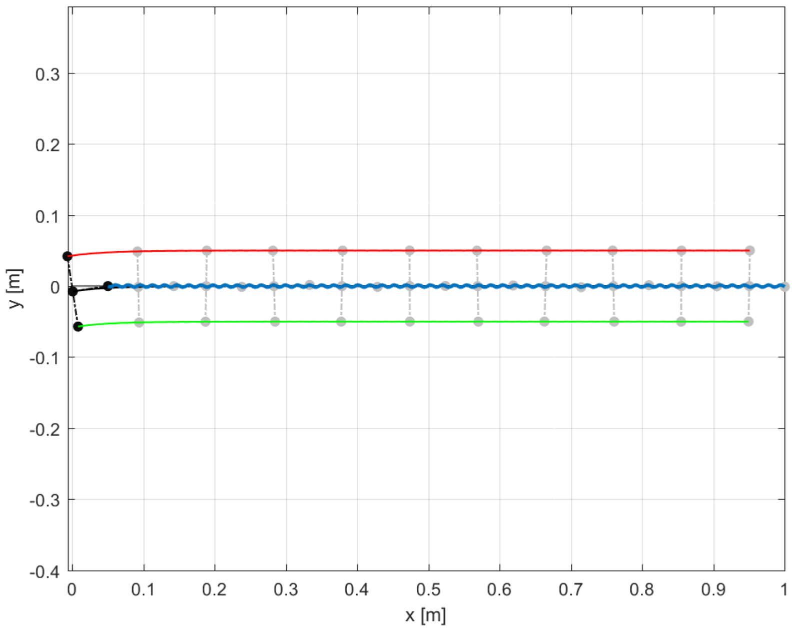

Figure 7.1: *Forward movement of the vehicle over a linear trajectory. The sensor emulator is activated.*

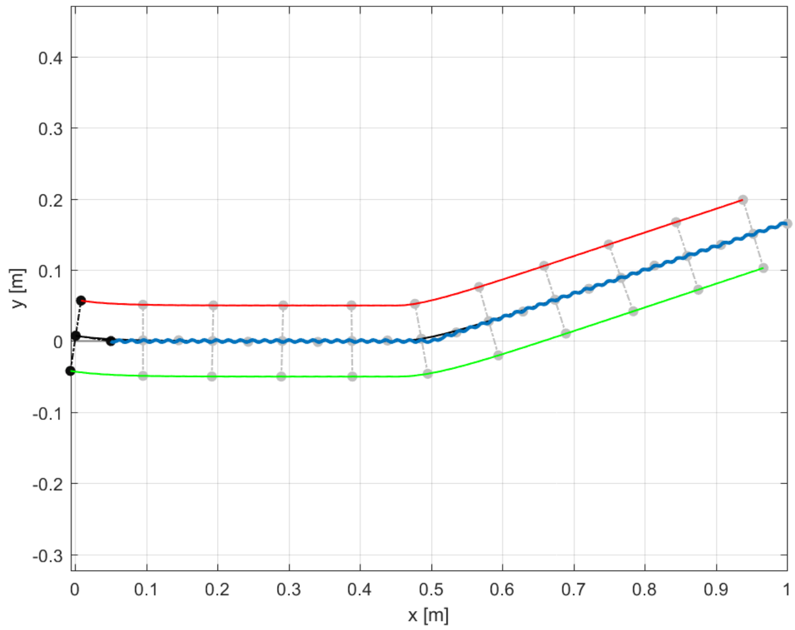

Figure 7.2: *Forward movement of the vehicle over a linear trajectory with an spontaneous change of direction. The sensor emulator is activated.*

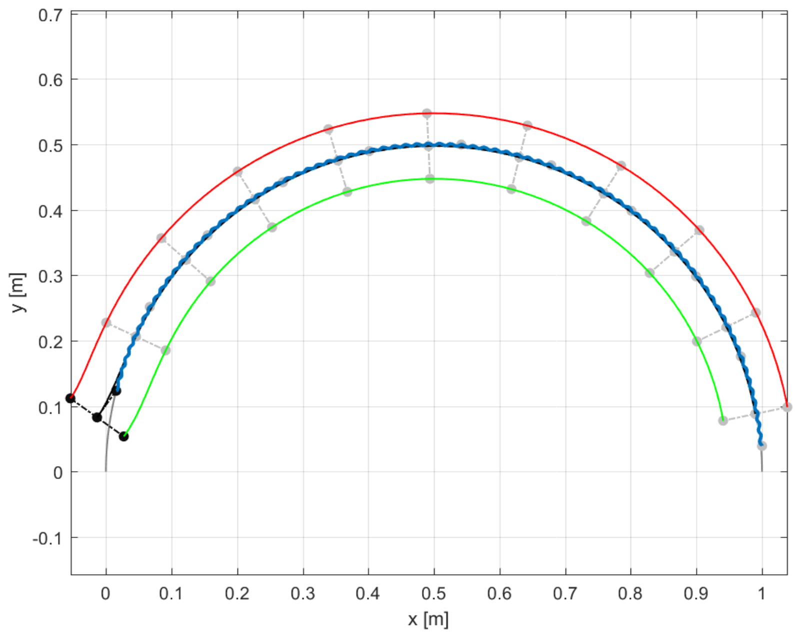

Figure 7.3: *Forward movement of the vehicle over a circular trajectory. The sensor emulator is activated.*

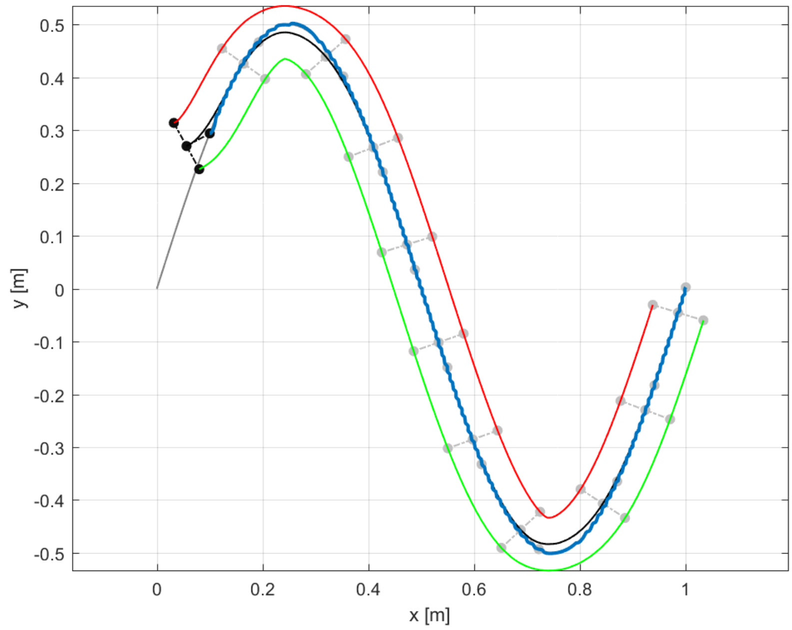

Figure 7.4: *Forward movement of the vehicle over a sinusoidal trajectory. The sensor emulator is activated.*

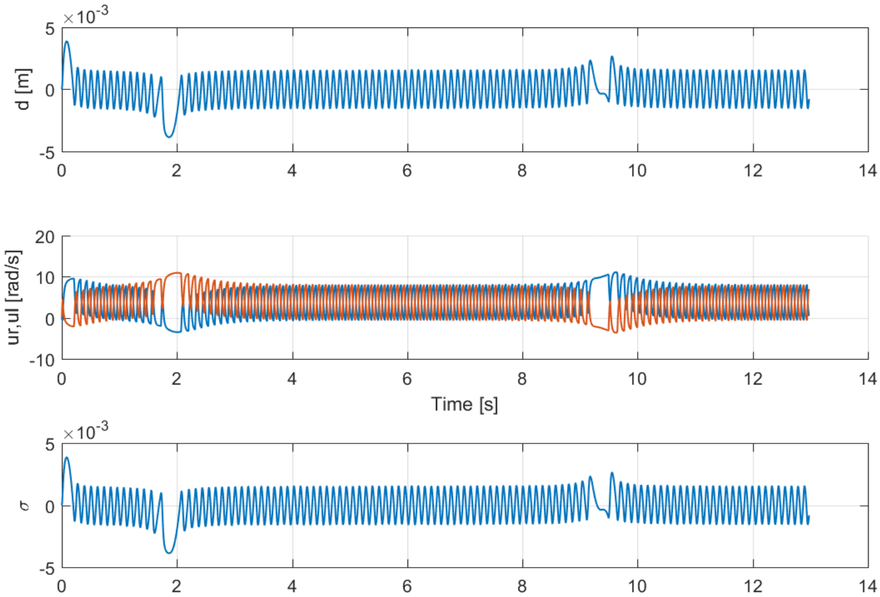

Figure 7.5: *Forward movement of the vehicle over a sinusoidal trajectory. The sensor emulator is activated.*

7.3 Backward movement with no sensor emulator

If the sensor emulator was activated the simulation results wouldn't tell us how the vehicle would behave in the cases that it gets out of the path. The behaviour of the vehicle can be completely seen if the sensor emulator is erased from the simulation. This way, the vehicle knows exactly its perpendicular distance to the path (d) with no restrictions.

Figure 7.6 show two situations where the distance d is too big to be sensed properly. The two situations correspond to a reorientation of the vehicle due to the lack of information about the future curvature.

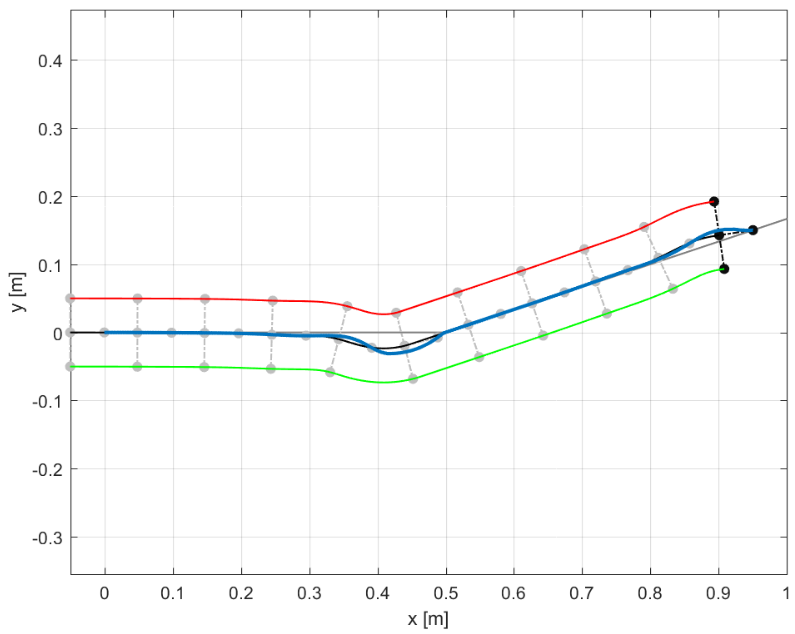

Figure 7.6: *The vehicle follows the path. It indicates the stability of the control. However, the deviation d is to high in the points where the vehicle has to redirect itself. The next parameters have been applied:* $l = 0,05$, $R = 0,05$, $r = 0,02$ *and* $beta = 2,5l$. *The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

Figure 7.7 shows how the vehicle follows the circular path with a constant deviation $d \neq 0$. A possible explanation to that behaviour is that the curvature of the circle is bigger than the maximum. However, this theory has been disregarded because the maximum curvature equals $c_{max} = 20$ while the curvature of this circle equals $c = 1/2$.

In the forward movement, the sensor knows the path before the wheels get to it. In the backward movement, on the other hand, the wheels preceed the sensor. This makes the sensor lose contact with the track when the wheels turn to change the direction. It explains the behaviour of figure 7.7 where a constant curvature makes the vehicle have a constant deviation d from the track. In other words, the vehicle is constantly reorientating itself and, therefore, losing contact with the track permanently.

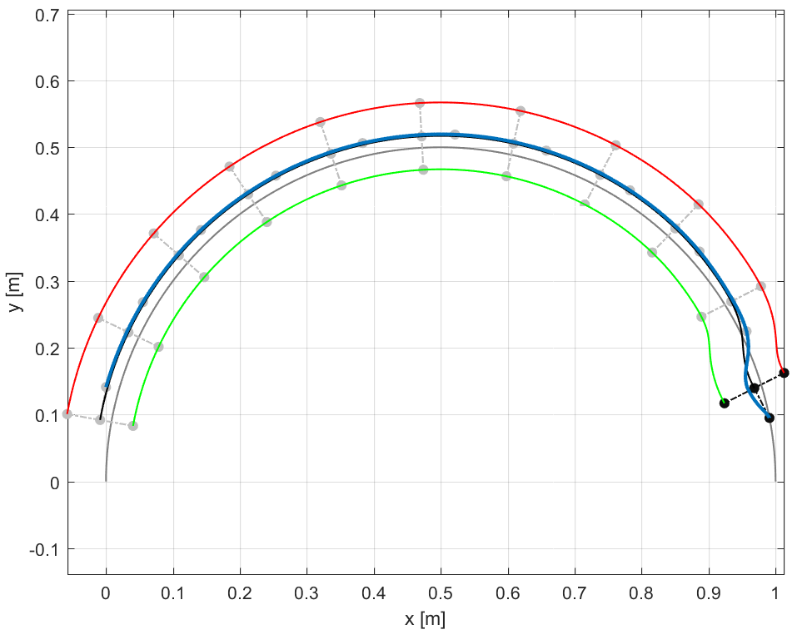

Figure 7.7: *Backward movement of the vehicle over a circular trajectory. The sensor emulator is not activated. Therefore, the vehicle can follow the trajectory with a remarkable and constant deviation d. The next parameters have been applied:* $l = 0,05$*,* $R = 0,05$*,* $r = 0,02$ *and* $beta = 2,5l$*. The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

Figure 7.8 shows the sinusoidal trajectory when the vehicle moves backwards. Note that the vehicle tends to follow the maximum curvature but it can only describe a pseudo circle around it. In conclusion, the sliding mode control designed for backward movement is stable. However, it does not behave properly due to the curvature or the starting position. We can easily say by looking at figures 7.6, 7.7 and 7.8 that it has a non acceptable deviation from the objective trajectory. A possible solution is to increase the gain of the control action u_2. That could be achieved by modifying the parameter β. Its important to remark that the complexity of the dynamics make it impossible to determine the parameters of the controller by specifying any properties of the compensated dynamics.

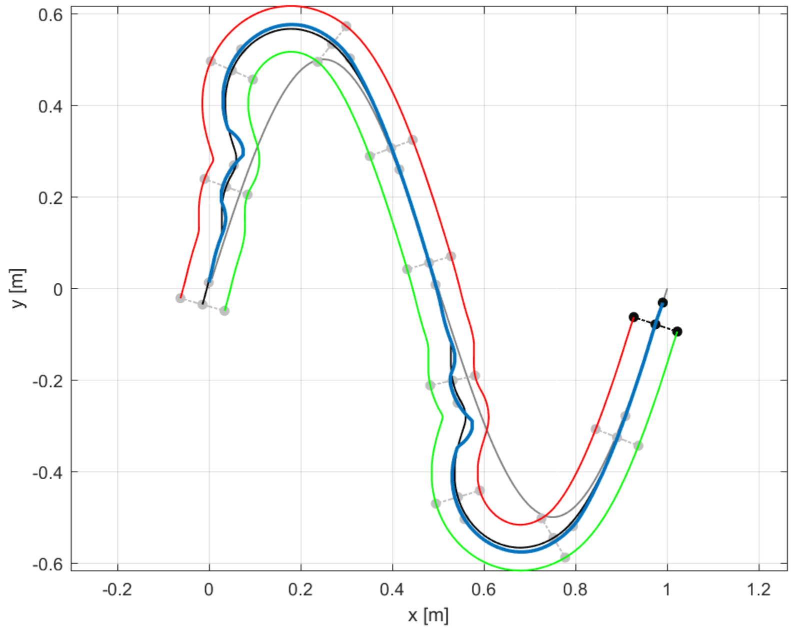

Figure 7.8: *Backward movement of the vehicle over a sinusoidal trajectory. The sensor emulator is turned off. The vehicle cannot follow the trajectory when the curvature is too high. The maximum curvature that the vehicle can theoretically follow is* $c_{max} = 20$ *while the maximum curvature of the trajectory is* $c = 19,73$*. The next parameters have been applied:* $l = 0,05$*,* $R = 0,05$*,* $r = 0,02$ *and* $beta = 2,5l$*. The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

7.4 Backward movement with sensor emulator

In this set of simulations, the next parameters have been applied: $l = 0,05$, $R = 0,05$, $r = 0,02$ and $beta = 2,5l$. The sensor is set to fail when its deviation from the line exceeds the interval

$$-0.005 < d < +0.005$$

Therefore, any possible deviation shown in the graphics is smaller than the maximum allowed.

Figures 7.9, 7.10, 7.11, 7.12 and 7.13 show the backward movement of the vehicle when the sensor emulator is activated. Note that only the linear trajectory (figure 7.9) works properly. The curvature is set to $c = 0$ in the model simulations because it is an unknown parameter.

Figure 7.10 is similar to figure 7.9 with the addition of a sudden change of deviation. In this case, it should follow the trajectory until the deviation is met. However, the failure is at the begining (see figure 7.10). The possible explanation to this behaviour is that, when the vehicle turns to follow the linear trajectory, the sensor gets too far from the trajectory. Figure 7.11 shows the same trajectory as figure 7.10 with a different starting position. With the new starting position, it works fine until the deviation is met. The extreme curvature of the deviation might be the issue this time. The maximum curvature is a concept explained in the *Sliding Mode Control* section of the *Control Design* chapter. It applies not only for the forward movement but also for the backward movement. Therefore, the same rule that makes the control work in the forward movement (figure 7.2) should not be a problem in the backward movement (figure 7.11).

Figure 7.12 shows that the control fails when going backwards in the circular trajectory. It has already been mentioned in the previous section. Figure 7.12 demonstrates that the sensor emulator works as expected. Figure 7.13 corresponds to the fourth track, the sinusoidal track. It shows how the vehicle follows the trajectory as the curvature increases until a determined point is reached. As it has already been said, the maximum curvature equals $c_{max} = 20$ when $l = 0,05$. The curvature of the sinusoidal trajectory can be calculated by derivating the angle of a tangential line of the curve. Remember that the parametrized curve is

$$\phi(q) = \big(q, A\sin(2\pi q)\big)$$

Then, the slope of a tangential line is

$$\frac{\partial \phi_y}{\partial q} = \frac{\partial A\sin(2\pi q)}{\partial q} = A2\pi\cos(2\pi q)$$

and the angle with respect to the X axis is

$$\theta_q = \arctan\big(A2\pi\cos(2\pi q)\big)$$

The cruvature is the derivative of θ_q

$$\frac{\partial \theta_q}{\partial q} = -\frac{19.7392\sin(2\pi q)}{9.8696\cos^2(2\pi q) + 1}$$

The variable q equals $q = 1/4$ when the curvature of the curve is maximum. In that position the curvature equals $c = 19,7392$. The length l is chanched to $l = 0.005$ in order to increase the difference between the maximum curvature of the curve and the maximum curvature allowed ($c_{max} = 200$ for $l = 0.005$). Figure 7.14 shows the result of a length ten times smaller. Comparing figures 7.13 and 7.14 it can be seen that the vehicle takes longer to lose control in the case with $l = 0.005$ than the case with $l = 0.05$. Still, it does not behave properly. To sum up, the maximum curvature concept explained in the *Sliding mode control* section is not enough to explain the backward movement failures. It only covers the curvature that geometrically or physically could the vehicle follow.

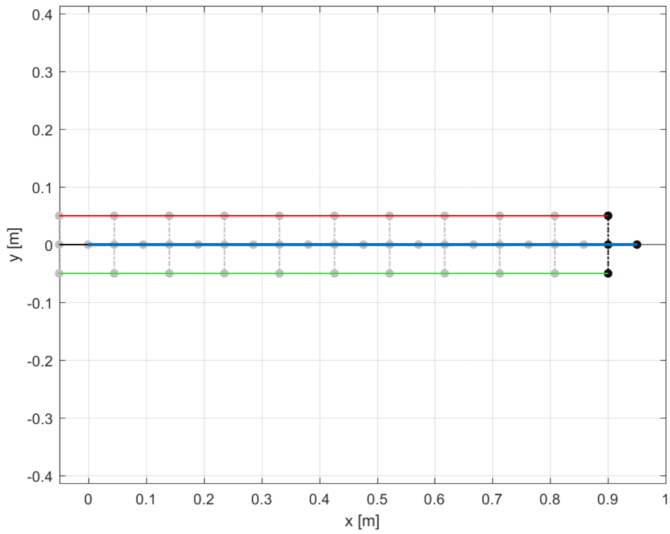

Figure 7.9: *Backward movement of the vehicle with sensor emulator. The next parameters have been applied:* $l = 0,05$*,* $R = 0,05$*,* $r = 0,02$ *and* $beta = 2,5l$*. The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

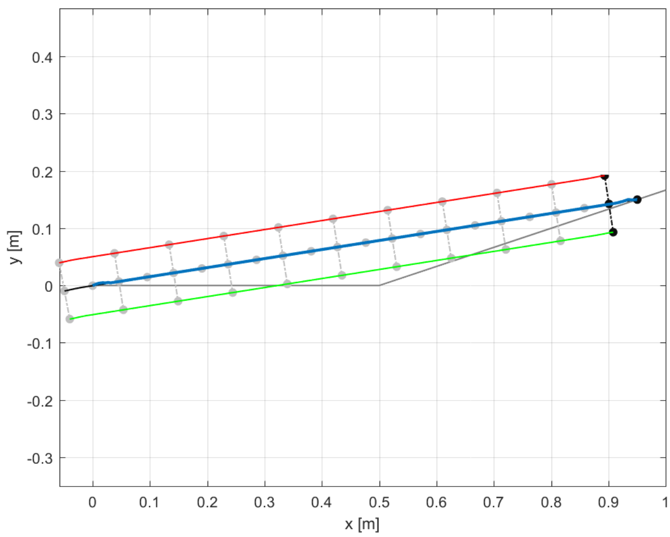

Figure 7.10: *Backward movement of the vehicle over linear track with spontaneous change of direction. The control fails from the beginning due to the starting position. When the vehicle turns to change the direction, the sensor gets out of the path and loses control. The next parameters have been applied:* $l = 0,05$*,* $R = 0,05$*,* $r = 0,02$ *and* $beta = 2,5l$*. The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

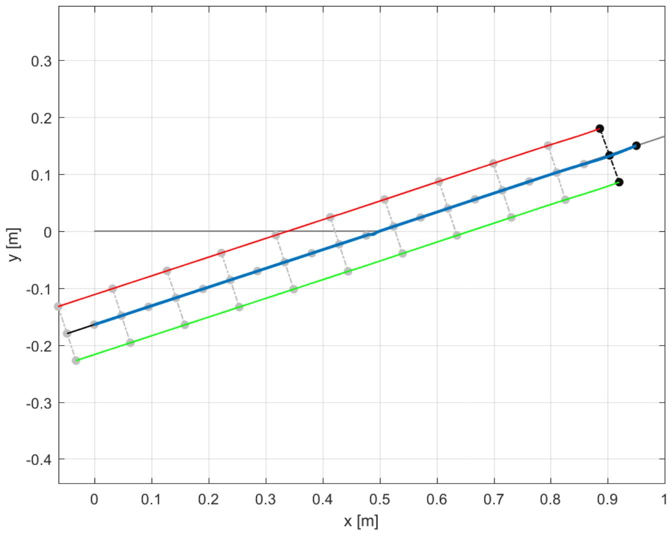

Figure 7.11: *Backward movement of the vehicle over a linear track with spontaneous change of direction. The control fails when the deviation of the path is met. The failure is not due to the discontinuity of the path, nor the starting position. The failure comes when the vehicle turns to follow the new direction and the sensor gets out of the path, losing control. The next parameters have been applied:* $l = 0,05$*,* $R = 0,05$*,* $r = 0,02$ *and* $beta = 2,5l$*. The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

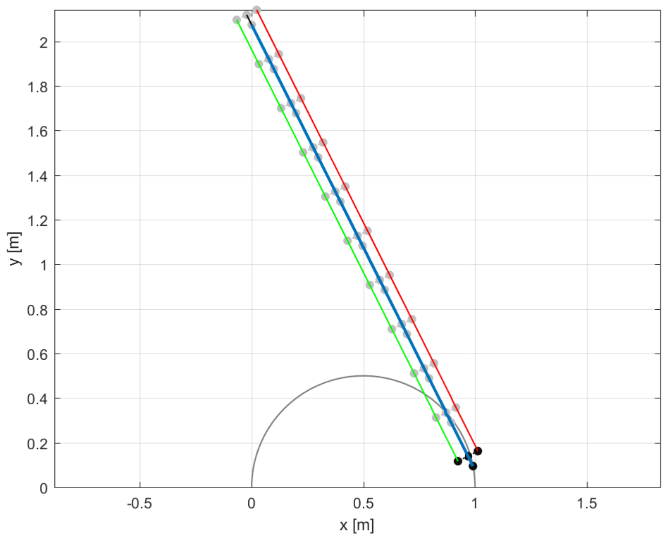

Figure 7.12: *Backward movement of the vehicle over a circular trajectory. The vehicle fails from the beginning in the starting position. The next parameters have been applied:* $l = 0,05$, $R = 0,05$, $r = 0,02$ *and* beta $= 2,5l$. *The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

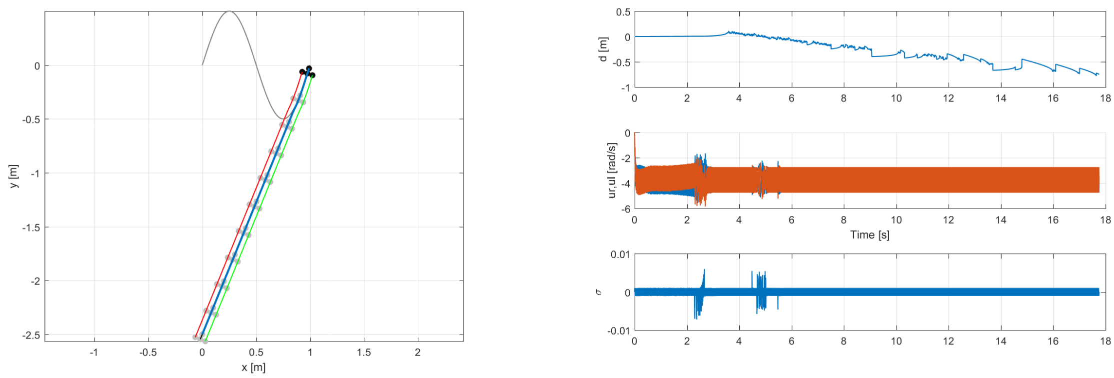

Figure 7.13: *Forward movement of the vehicle over a sinusoidal trajectory. The vehicle follows the trajectory until a determined point is reached where the curvature is too high. The next parameters have been applied:* $l = 0,05$, $R = 0,05$, $r = 0,02$ *and* beta $= 2,5l$. *The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

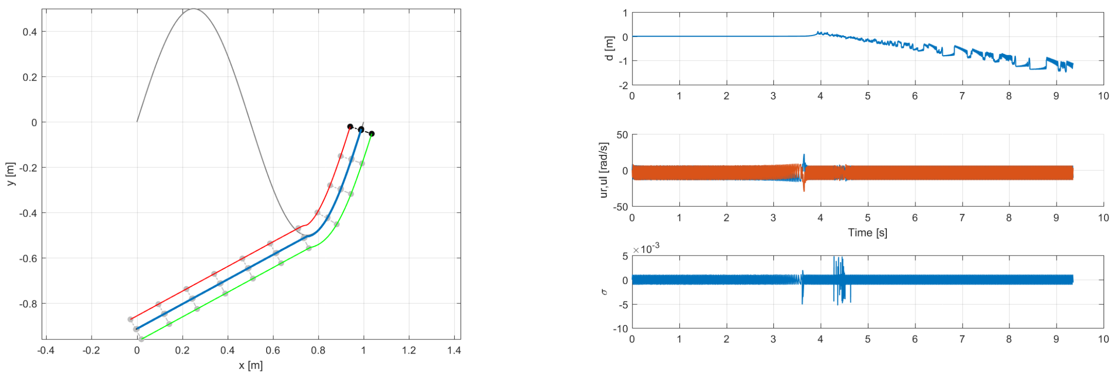

Figure 7.14: *Backward movement of the vehicle over a sinusoidal trajectory with sensor emulator activated. The length has been modified to increase the theoretical maximum curvature ten times over the maximum curvature of the path. However, the vehicle loses track. The next parameters have been applied:* $l = 0,005$, $R = 0,05$, $r = 0,02$ *and* $beta = 2,5l$. *The three black spots represent the wheels, the sensor (point* P*) and the middle point between the wheels (point* P_m*) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

7.5 Length effect in backward movement

The sensor emulator has been activated and the next parameters have been set in this section:

- $l = 0,005$
- $\beta = 1,5l$

The paramter β is used in the control action gain. The proper value of the gain is found by trial and error due to the complexity of the model.

$$u_2 = u_2^* - \frac{\rho}{l - \beta}\text{sign}(\sigma)$$

Figures 7.15, 7.16 and 7.17 show a proper behaviour in backward movement for the parameters applied. However, the length $l = 0,005$ is ten times smaller than the real one at the laboratory.

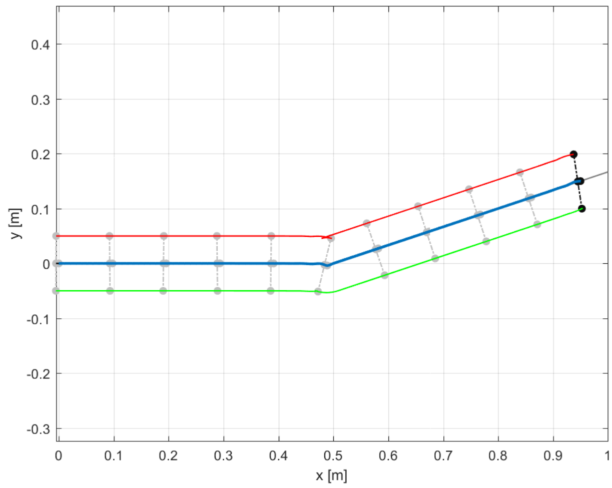

Figure 7.15: *Backward movement of the vehicle following a linear trajectory with spontaneous change of direction. The vehicle length l has been set to $l = 0,005$ and the sensor emulator is activated. The vehicle follows the path correctly. The next parameters have been applied: $l = 0,005$, $R = 0,05$, $r = 0,02$ and beta $= 2,5l$. The three black spots represent the wheels, the sensor (point P) and the middle point between the wheels (point P_m) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

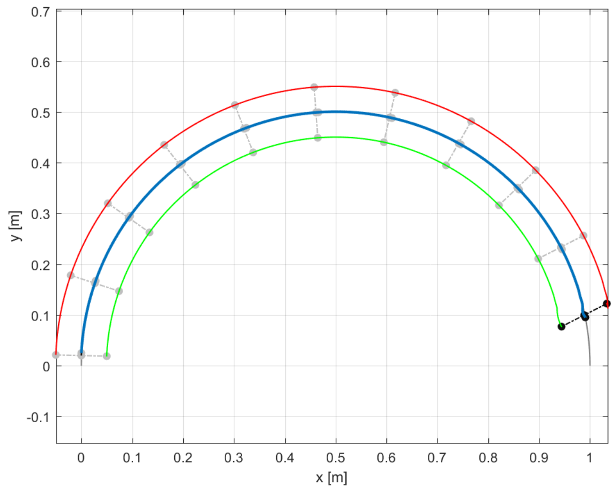

Figure 7.16: *Backward movement of the vehicle following a circular trajectory with spontaneous change of direction. The vehicle length l has been set to $l = 0,005$ and the sensor emulator is activated. The vehicle follows the path correctly. The next parameters have been applied: $l = 0,005$, $R = 0,005$, $r = 0,02$ and beta $= 2,5l$. The three black spots represent the wheels, the sensor (point P) and the middle point between the wheels (point P_m) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

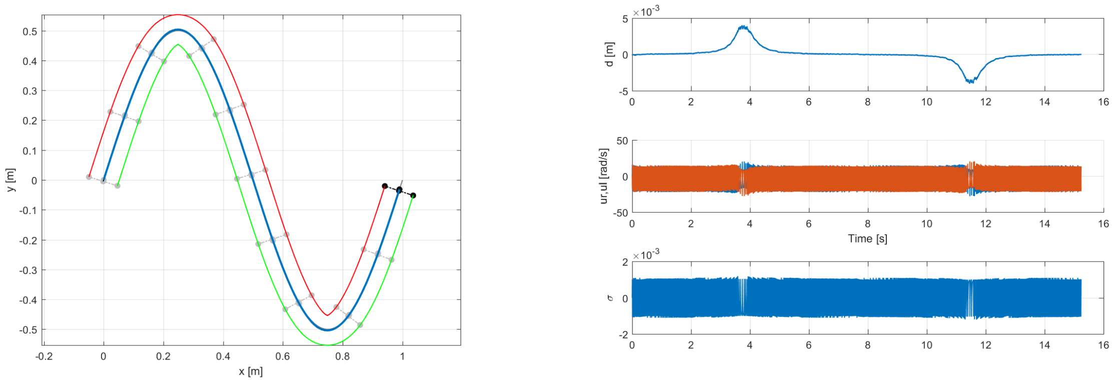

Figure 7.17: *Backward movement of the vehicle following a sinusoidal trajectory with spontaneous change of direction. The vehicle length l has been set to $l = 0,005$ and the sensor emulator is activated. The vehicle follows the path correctly. The next parameters have been applied: $l = 0,005$, $R = 0,05$, $r = 0,02$ and beta $= 2,5l$. The three black spots represent the wheels, the sensor (point P) and the middle point between the wheels (point P_m) in their starting position. The red (green, blue) line is the path that makes the left wheel (right wheel, sensor).*

7.6 Solution summary

As seen in the previous section, the control is stable but does not behave as it is wanted to. The forward movement works properly but not the backward movement. In this last case, the location of the sensor is important as well as the term $\frac{\rho}{l-\beta}$ that multiplies the sign function of the sliding surface.

$$u_2 = u_2^* - \frac{\rho}{l-\beta}\text{sign}(\sigma)$$

In the backward movement, the main problem is that the wheels are located ahead while the sensor goes on the tail. It makes the sensor lose contact with the path when the vehicle turns to follow a determined curvature. As long as the path or the future curvature cannot be predicted, the vehicle cannot move backwards with the sensor too far from the middle point of the wheels (P). It is likely to work for a determined configuration where the mentioned parameters are set properly. However, the next step should be a careful test in the laboratory to confirm it.

8 PYGAME ANIMATIONS

The aim of this chapter is to provide conclusions of the experience of programming a tracking animator with Python and its module Pygame. This is an alternative to the animation tools of Matlab. It has been proven that the matlab animations are too slow to be attractive and the language is less powerful than Python. The next section gives some explanations of the program logics with no description of the code itself.

8.1 Python possibilities

Among the whole programming languages and aplications that may be useful to create a graphical user interface (GUI), python is one of the most powerful as a consequence of being open source and a high level interpreter. Lots of modules provide the tools to create GUI applications, and graph ploting, as well as math packages and image treatment and creation. Some of the most common GUI libraries that could help are Tkinter, PyQt, wxPython and PyGTK. However, the Pygame module is another option that has not been designed to create GUIs, but to create 2D games.
The final program must be described before any choice can be made.

- It has to be illustrative. It needs to show the vehicle following the trajectory and the tracks left by the points of interest (P_m,P and left and right wheels).
- The graphics of the vehicle could include, if possible, 3D space and textures. Some other tools like the possibility of zooming a region of the animation could be interesting.
- The program must have the possibility to import data from the matlab simulation tests and plot an animation. Therefore, the animation should be able to be paused, reinitiate or reproduced in a loop. These functionalities might include buttons and text inputs.
- At the same time, some graph parameters like the evolution of the distance d or the angular speed of the wheels should be shown.

With the requirements given, the chosen library is Pygame. With the understanding that a 2D game engine can provide as well any GUI options. Although the tools that provides might be at a lower level of programming than the other GUI libraries.

8.2 Animator logics overview

The first thing to do is to export the data from the Matlab simulation tests. A way of doing it is with a ".txt" file:

```
save('PM.txt','Pm','-ascii')
```

This example is repeated for all the variables needed. Once the data is saved in a directory in different text files, it must be readed and saved into a global variable in the Python environment. The length of the arrays that contain the data is around 55 thousand positions. That amount of data is too heavy to be ploted one by one in Pygame. Hence, the data must be reduced. Two functions have been written to load data and make it shorter: *loadData()* and *shortenData()*. Figure 8.1 shows the list of events in blue, at the left and the main actions, mostly buttons, that characterize the program. The shortened data arrays contain data that is separated in time by a same period. For example, position one is separated 0,05 seconds from position 2, and position 2 is separated 0,05 seconds from position three, and so on.

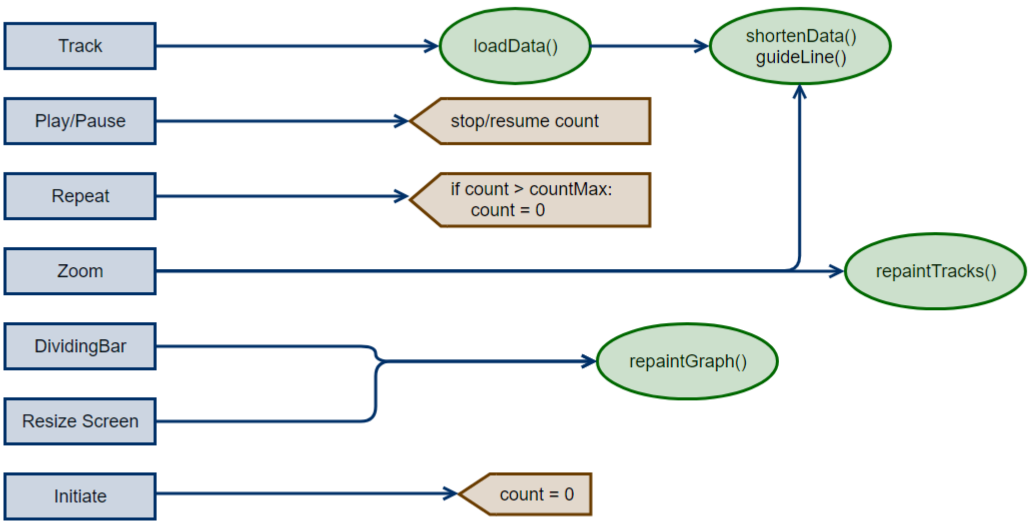

Figure 8.1: *Basic diagram of the behaviour of the program in front of the possible events (blue blocks). The track event is triggered when any of the four buttons are pressed. They are used to change from one kind of track to another. The variable count is the index that decides what position on the data arrays has to be ploted at the current iteration of the program.*

The function *guideLine()* draws the path guide that must be followed by the vehicle. It creates the data from scratch with the parameters applied in the Matlab Simulations. All of the points created by this function are connected with lines giving the aspect of a curve. Except for the guide line, which is ploted in black (8.2), every other plot is being painted step by step, depending on the time of simulation. In other words, each iteration of the Pygame clock, does not make the whole plot repaint again. To do this, different alpha (transparent) images have been created. It is over these images that the plots are painted. The images hold the plots when the screen is reinitiated at every iteration. However, if the zoom event is applied, all the plots have to be repainted from the beginning to be scaled according to the zoom. It takes a little delay almost imperceptible, but enough to slower the program if repeated each iteration. To repaint the plots, two functions have been written: *repaintTracks()* and *repaintGraphs()*. The first one repaints the plot of the tracks and the vehicle on the left (figure 8.2). The second one is explained ahead.

The most important global variable is the one that decides which line of the data arrays has to be ploted at the current iteration of the program. The name for that variable is *count*. This variable is an integer that increases each iteration by one unless the buttons *pause* and *initiate* are activated. The maximum value of the variable *count* is the length of the arrays that contain the shortened data. When the button *initiate* is pressed (see figure 8.1) all the plots have to be erased so that the plots start over.

As shown in figure 8.2, there is a bar that divides the vehicle tracks region on the left from the graphs region on the right. The movement event of this bar recieves the name *dividingBar* in figure 8.1. The width of the left and the right parts of the interface are modified by dragging this bar to the left or the right.

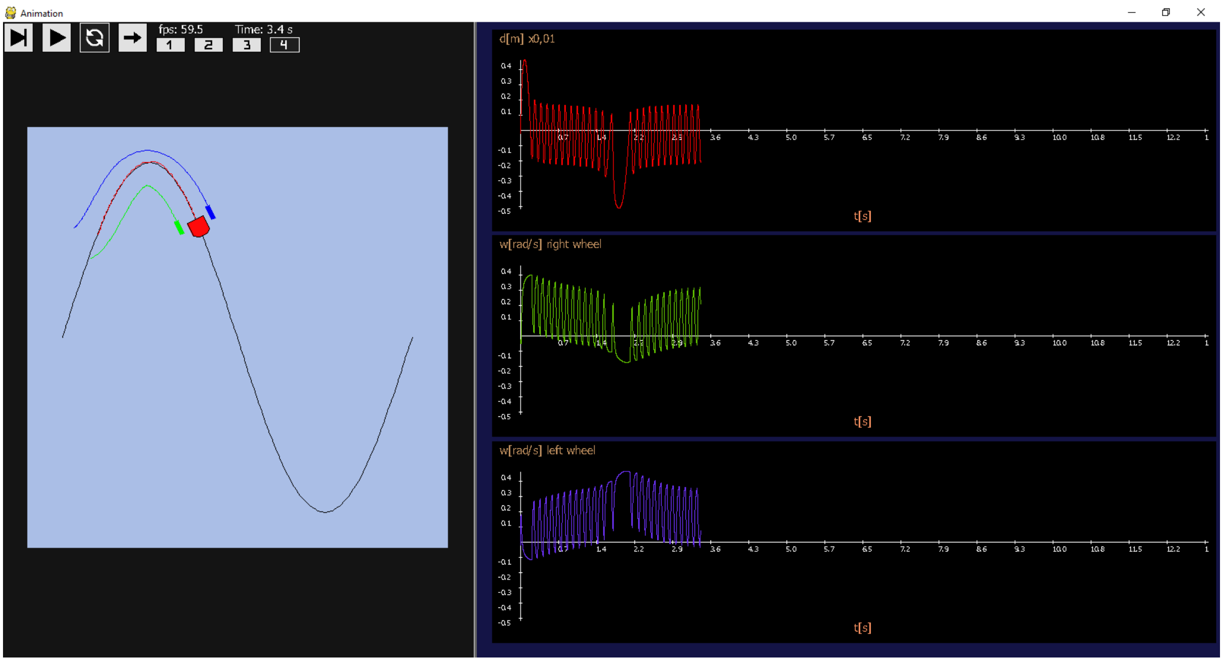

Figure 8.2: *Animator program made with Python and its Pygame module.*

One of the main problematics was the time control. In other words, how the time of the animation is controlled. Pygame gives the possibility to set the frames per second. Like in the cinema, the minimum would be 24 frames per second (FPS). If we know how many iterations of the program occur in one second, we also know the period. Thus, a variable can control the time in every iteration from the beginning by making the summatory of the period of every iteration.

```
#we ask what is the FPS in the current iteration
fps2 = fpsTime.get_fps()
# if the program is not paused, and it has not reached the end of the track(lenPshort)
# update the time variable
```

```
if not pause and count<lenPshort:
  try: time +=  1/fps2#1/fps
  except: time += 1/fps
```

Once the time is known, we need to find in the data arrays which one corresponds to that time so that it can be ploted on time. When the data arrays were made shorter, the array that contains the clock of the Matlab simulations was also made shorter. They were made shorter and equidistant in time. That is, any position of the array from the next one have the same time interval. Thus, we can control the varible int variable *count* comparing it with the float variable *time* as follows.

```
# If the animation is not paused nor it has reached the end,
# and the float variable "time" is bigger than the interval
# multiplied by the "count" variable, increase the "count"
# variable by one.
if time >= interval*count and count<lenPshort and not pause:
    count += 1
```

This code makes the animation be consequent with the real time and the behaviour in the Matlab simulation tests.

8.3 Conclusions

Despite that the Pygame library provides basic tools with which complex programs can be created, there is a lack that makes it less atractive. The fact that the functions that should provide anti-aliasing painting do not work properly. Thus, all the lines and text shown in figure 8.2 have aliasing. In the graphics environment, the aliasing is a problem that apears when the pixels are too big to plot a line without appearing rough to our eye. There are algorithms that change the color of the painting depending on which pixel of a line is going to be painted, but Pygame does not include them.

On the other hand, the freedom that offers Pygame could be of good used to experienced users. As shown in the previous chapter, every specification has been accomplished. However, the buttons and text input could be solved easily with other GUI libraries. A good combination of a GUI library and Math libraries might be the best solution. Thus, the final recomendation is to explore other libraries in case that an application of this kind is needed.

9 BUDGET

The creation of this document and all the time spent in regarding its creation has taken 300 hours of a superior engineer. A cost per hour of 45 €/h corresponds to a total cost of:

$$\text{Cost} = 300h \cdot 45\frac{€}{h} = 13500\ €$$

10 CONCLUSIONS

The sliding mode control makes the control robust in front of uncertainty. It works perfectly when the vehicle moves forward. However, it does not behave properly when the vehicle moves backwards. Different tests prove that the behaviour depends on the control action gain and the geometry of the vehicle. As a definitive conclusion, it must be said that the current sensor implemented in the laboratory vehicle is not enough to provide a proper backward movement. A possible solution is to make the distance from the middle point of the wheels axes to the sensor smaller. Still, that situation works only in the simulation but could not work in the real world where the system is yet more complex than its modelation.

Other solutions could be implemented. For example the usage of one more sensor: one backwards and another forwards. The sensor has a thin area of work and that is a handicap. The best option might be to implement a sensor with a bigger range of work or a camera. However, the implementation of a camera, despite being quite more interesting, is far more complicated. It implies image processing and, maybe, a more powerful microcontroller.

Regarding to the Python program, the Pygame module for Python is not the best option to create the kind of programs that require data plotting and graphic user interface. That is so because of the lack of a anti-aliasing solution to any shapes and text painted. On the other hand, the flexibility that provides the language and its library (Pygame) can be of good use for an experienced user.

Bibliography

[1] Q.Zhang, L. Lapierre, and X. Xiang. Distributed control of coordinated path tracking for networked nonholonomic mobile vehicles. *IEEE Trans. on Insutrial Informatics*, 9(1):472-484, 2013.

[2] F. Garelli. Sistemas de estructura variable. Aplicación al control con restricciones. *Departamento de Electrotecnia Facultad de Ingeniería Universidad Nacional de La Plata*, 35-52, 2007.

[3] I. Prats-Martinho. Control design and implementation for a line tracker vehicle, TFG. *Universitat politécnica de Catalunya*, 2016.

[4] P. Morin, C. Samson. Motion control of wheeled mobile robots. *In B. Siliciano and O Khatib, editors, Handbooks of Robotics*, 779-826. Springer, 2008.

[5] A. Doria-Cerezo, D. Biel, V. Repecho. Sliding mode control of a unicycle type differential-drive mobile robot following a path, tech report. 2017

[6] K. Ogata. Ingeniería de control moderna. 682-688, 751-778. Prentice Hall, Pearson. 2010

A Simulation results track 1

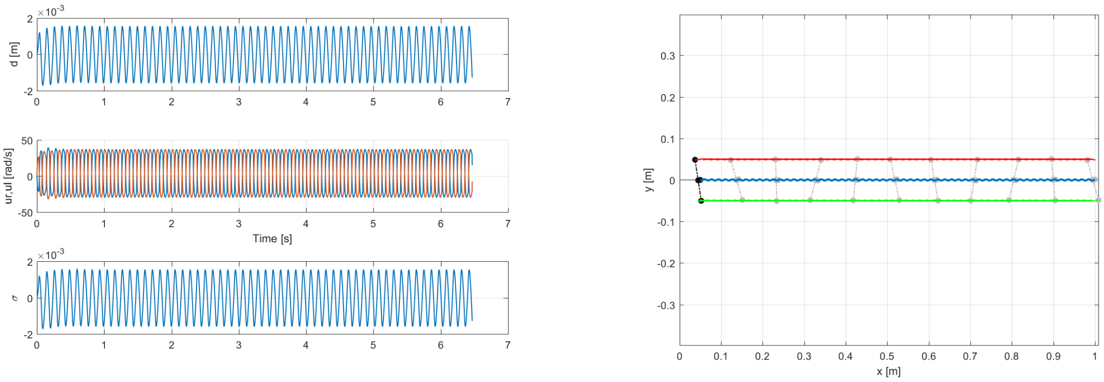

Figure A.1: $l = 0,005$, *sensor emulator off, forward movement.*

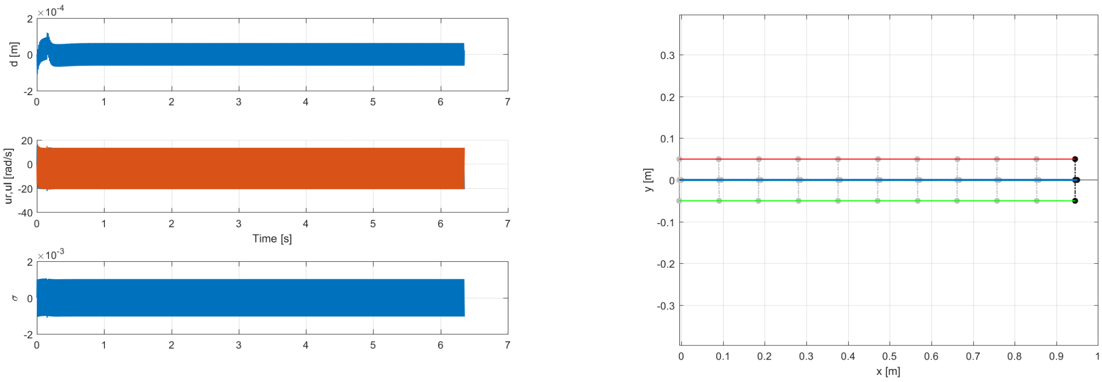

Figure A.2: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 1,5l$.

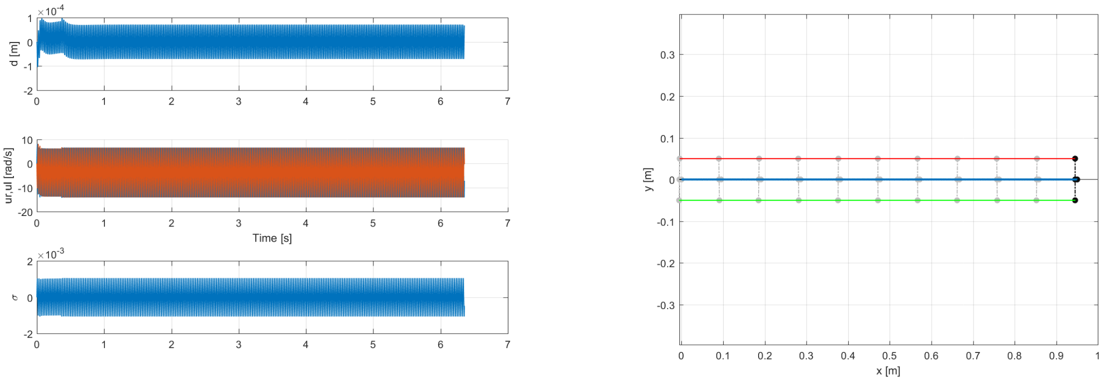

Figure A.3: $l = 0,005$, *sensor emulator off, backward movement,* $\beta = 2,5l$

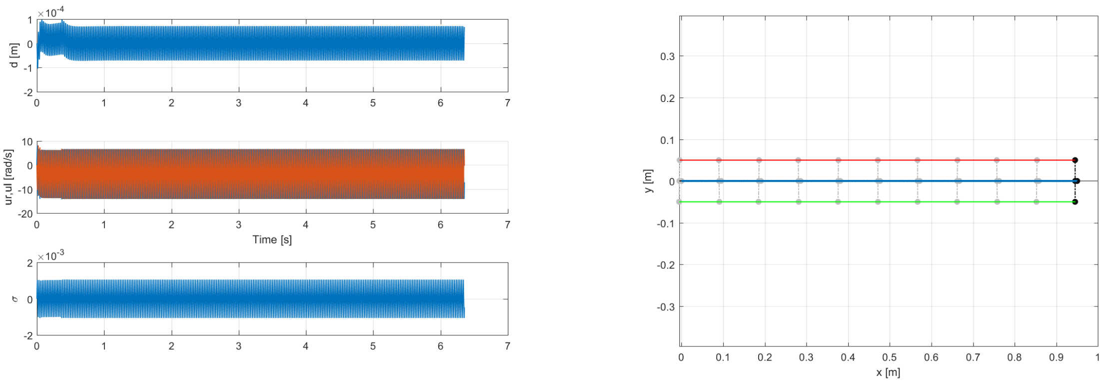

Figure A.4: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

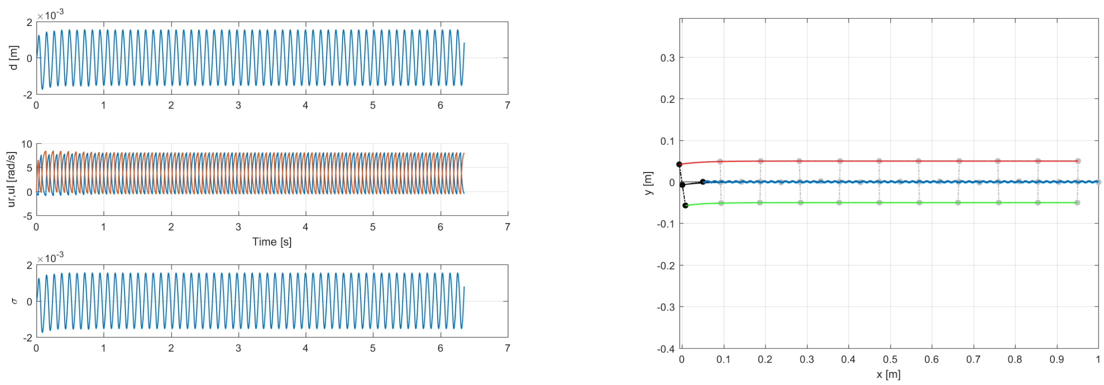

Figure A.5: $l = 0,05$, *sensor emulator off, forward movement.*

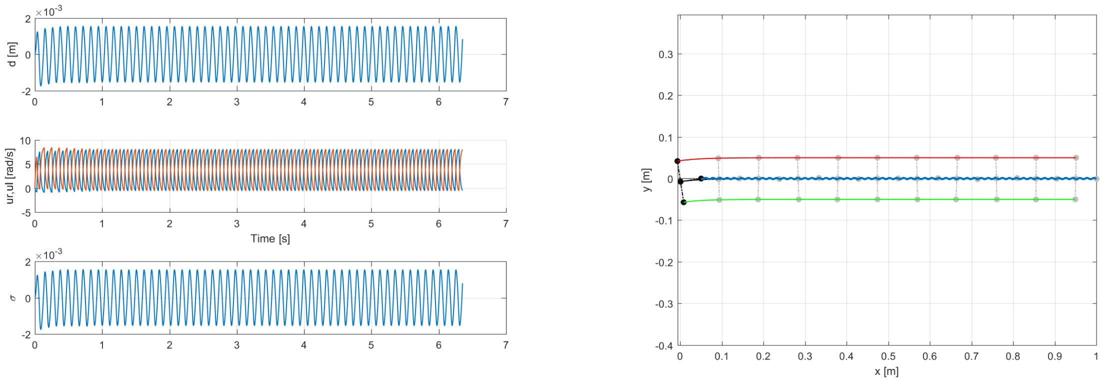

Figure A.6: $l = 0,05$, *sensor emulator on, forward movement.*

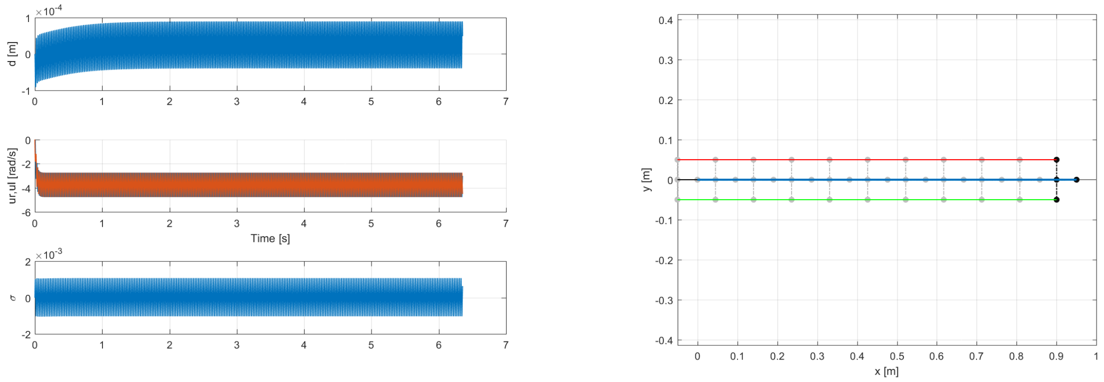

Figure A.7: $l = 0,05$, *sensor emulator off, backward movement,* $\beta = 2,5l$.

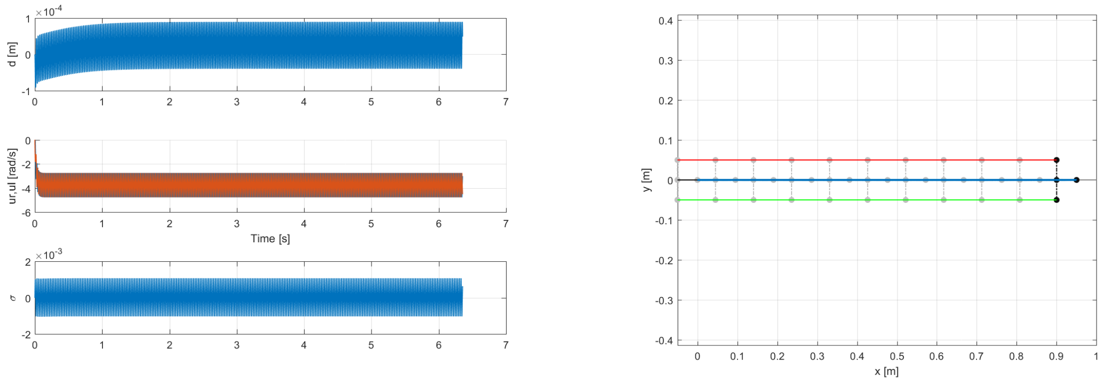

Figure A.8: $l = 0,05$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

B | Simulation results track 2

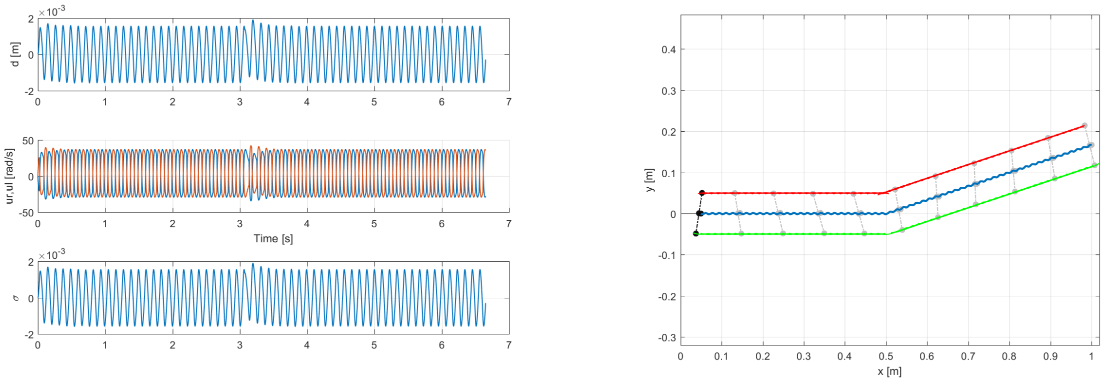

Figure B.1: $l = 0,005$, *sensor emulator off, forward movement.*

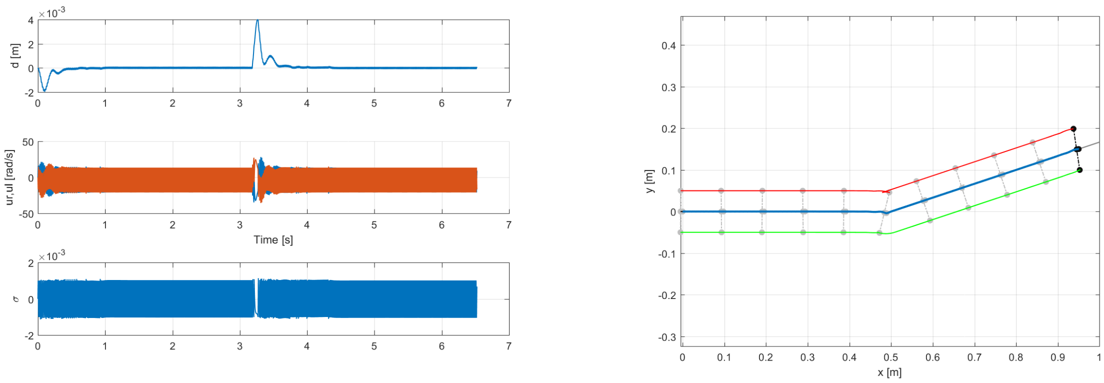

Figure B.2: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 1,5l$.

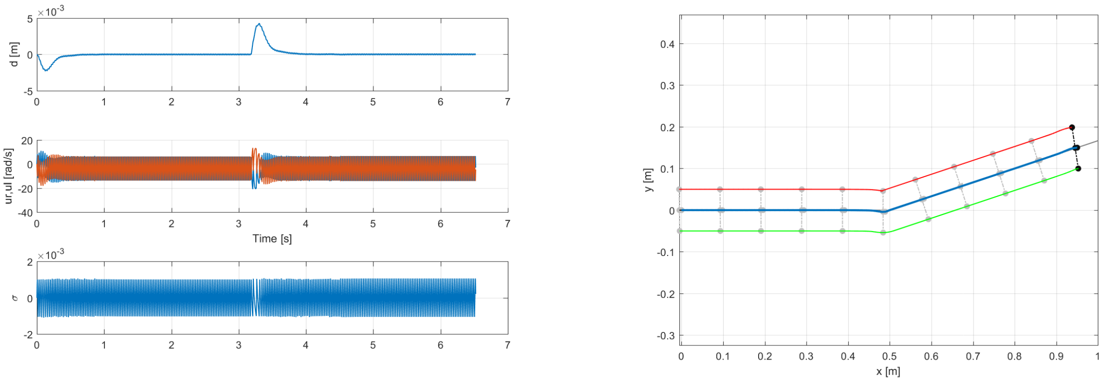

Figure B.3: $l = 0,005$, *sensor emulator off, backward movement,* $\beta = 2,5l$

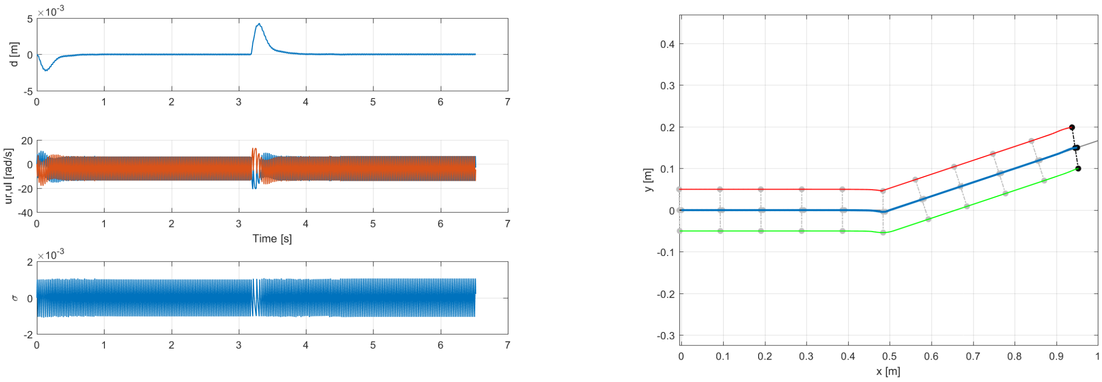

Figure B.4: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

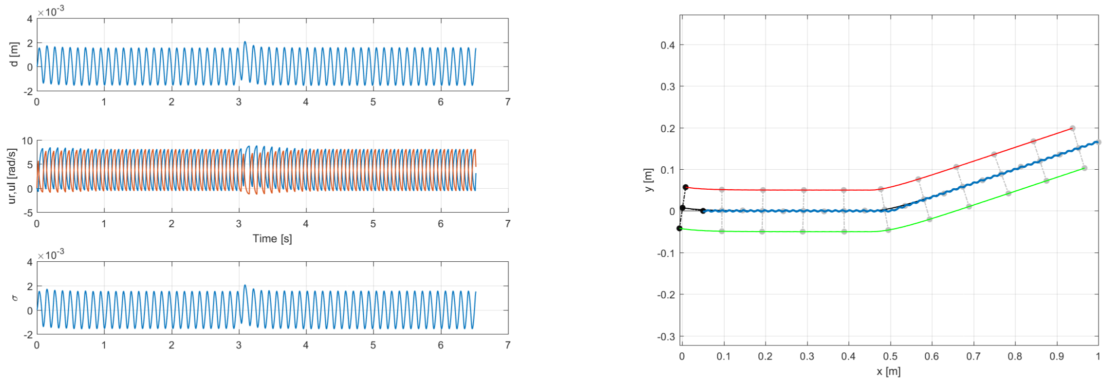

Figure B.5: $l = 0,05$, *sensor emulator off, forward movement.*

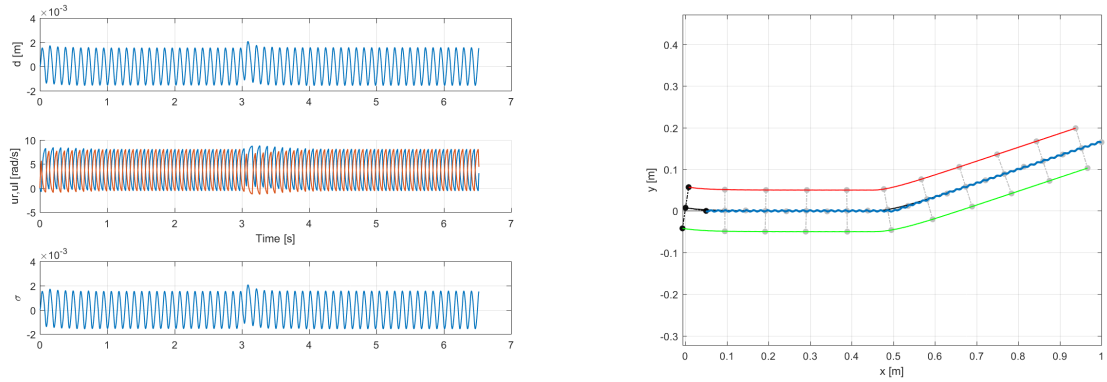

Figure B.6: $l = 0,05$, *sensor emulator on, forward movement.*

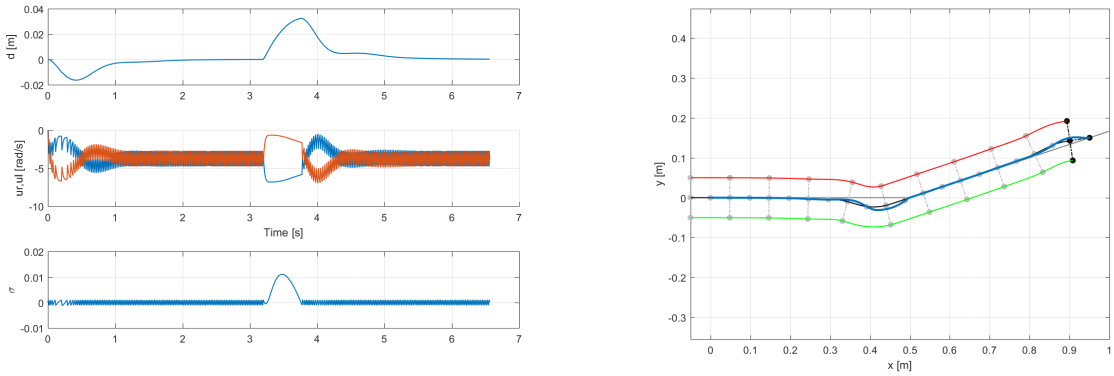

Figure B.7: $l = 0,05$, *sensor emulator off, backward movement,* $\beta = 2,5l$.

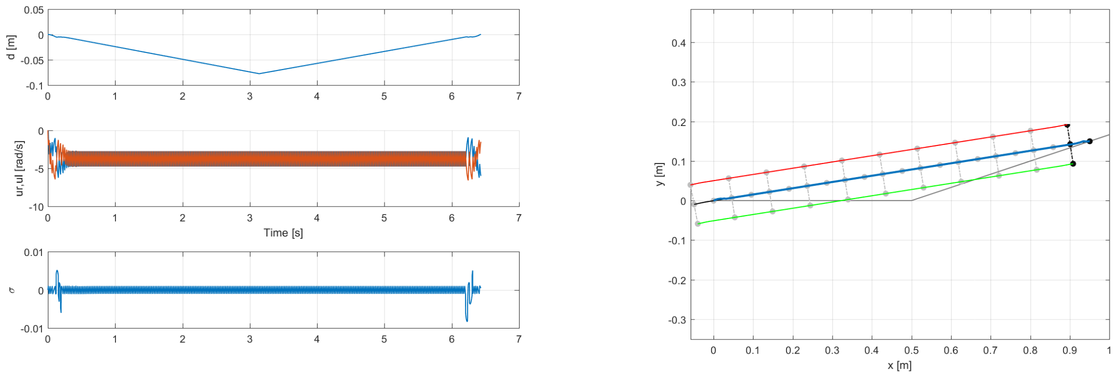

Figure B.8: $l = 0,05$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

C Simulation results track 3

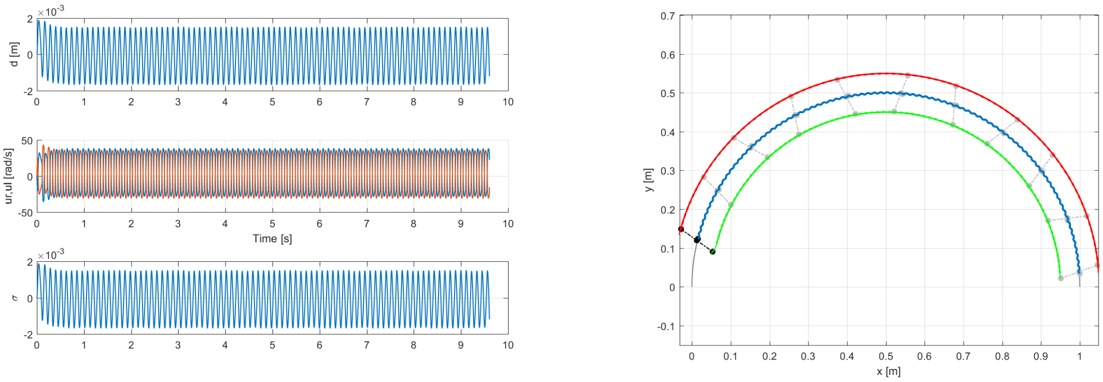

Figure C.1: $l = 0,005$, *sensor emulator off, forward movement.*

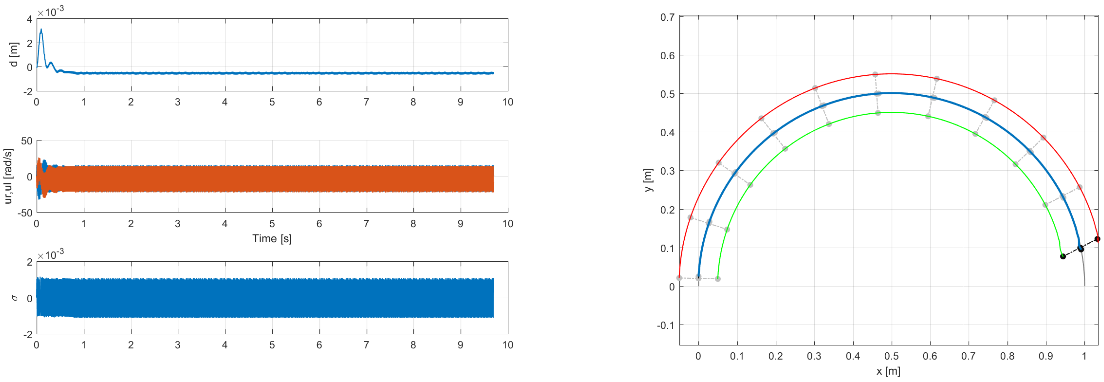

Figure C.2: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 1,5l$.

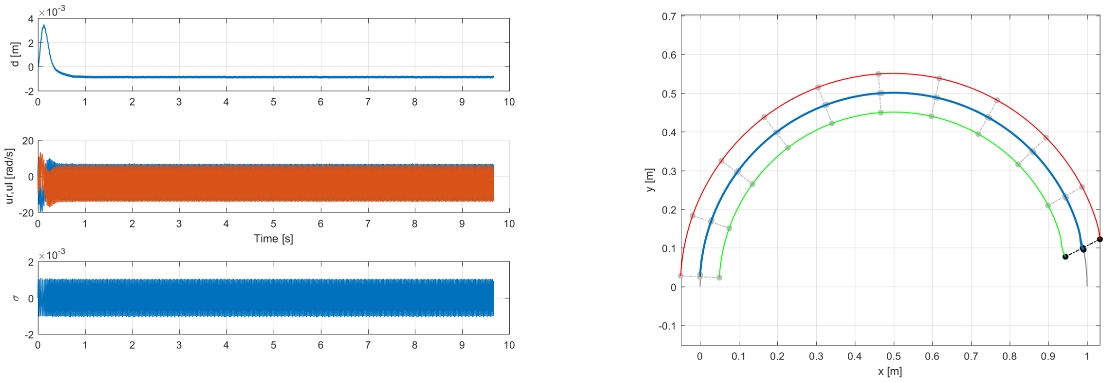

Figure C.3: $l = 0,005$, *sensor emulator off, backward movement,* $\beta = 2,5l$

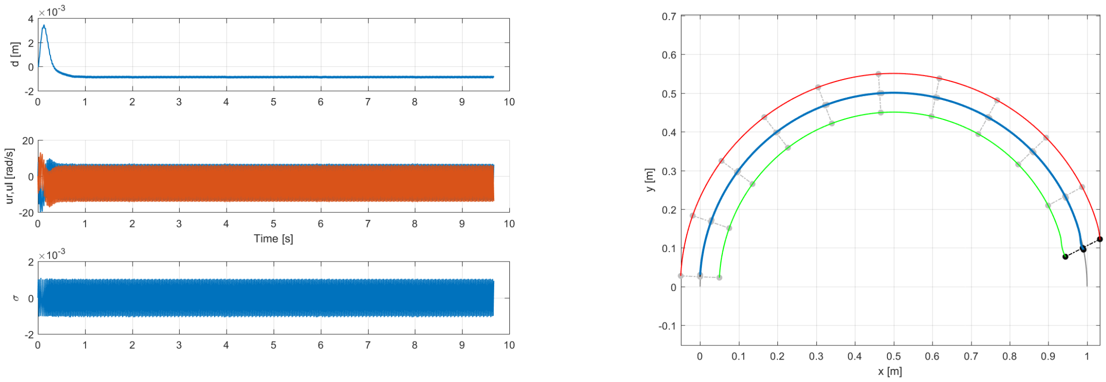

Figure C.4: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

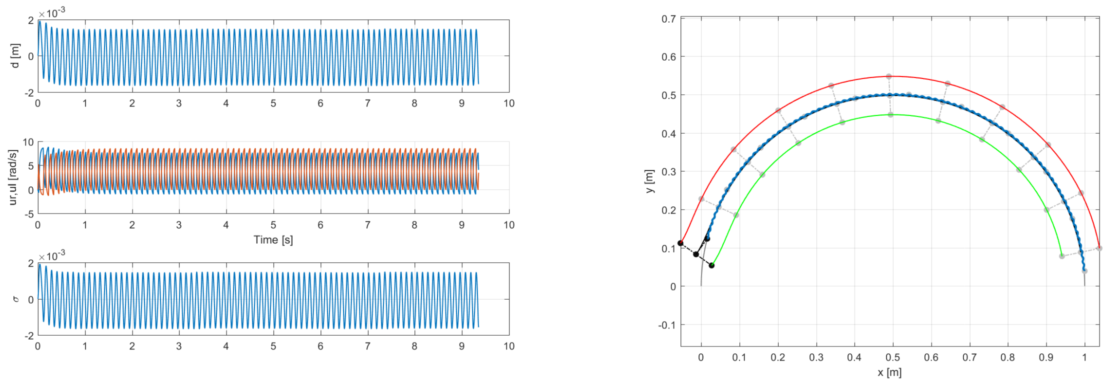

Figure C.5: $l = 0,05$, *sensor emulator off, forward movement.*

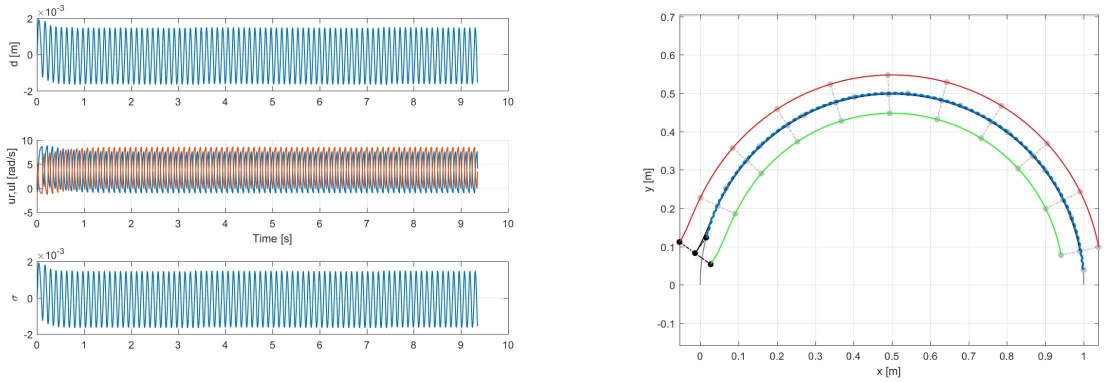

Figure C.6: $l = 0,05$, *sensor emulator on, forward movement.*

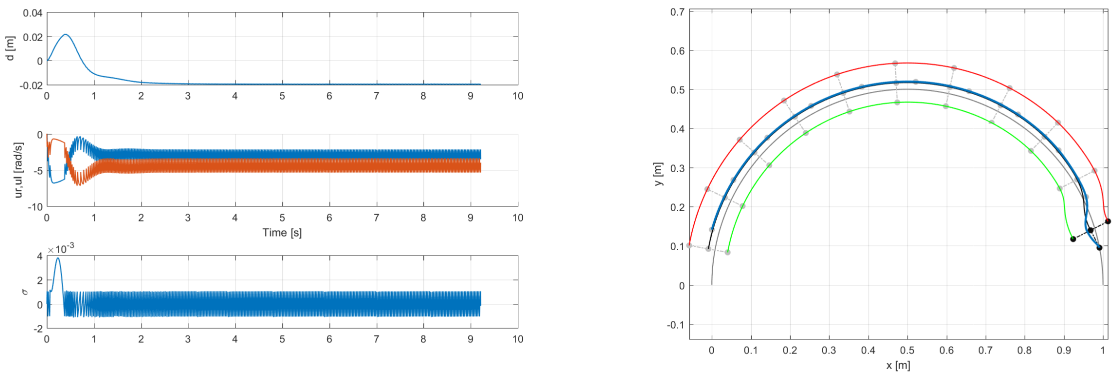

Figure C.7: $l = 0,05$, *sensor emulator off, backward movement,* $\beta = 2,5l$.

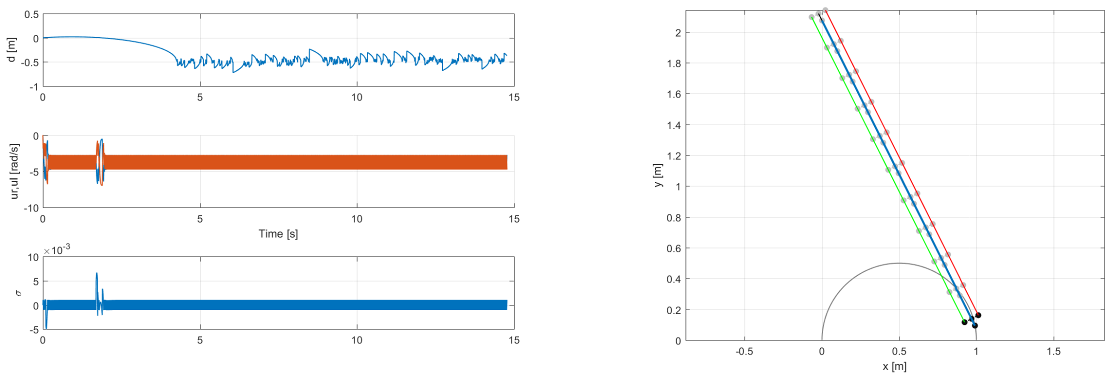

Figure C.8: $l = 0,05$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

D Simulation results track 4

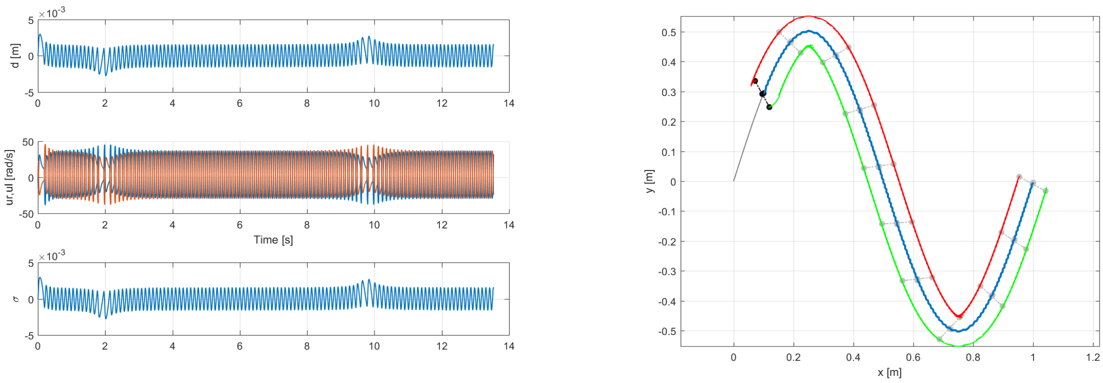

Figure D.1: $l = 0,005$, *sensor emulator off, forward movement.*

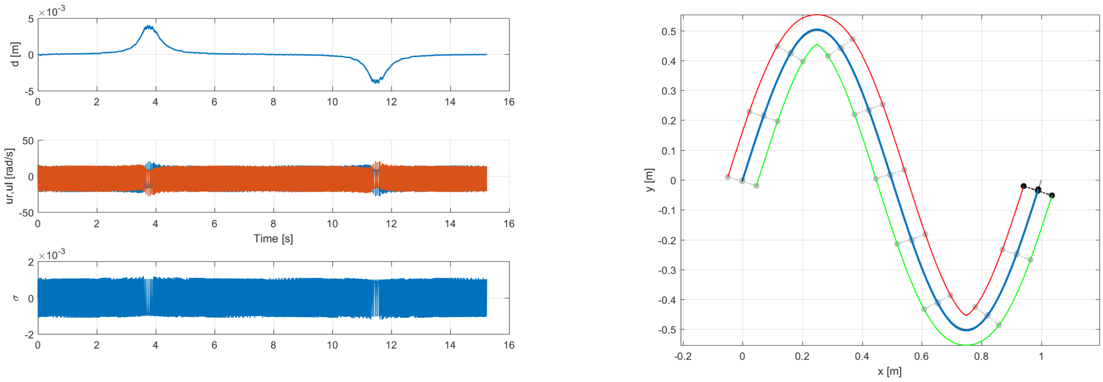

Figure D.2: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 1,5l$.

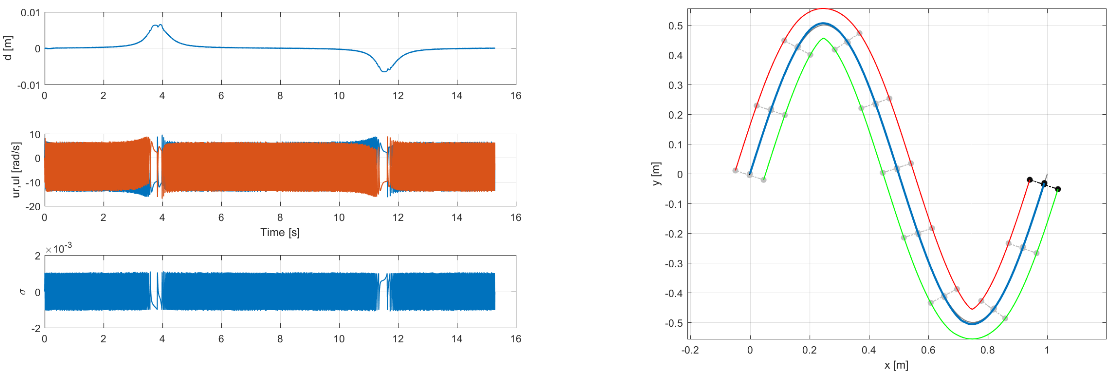

Figure D.3: $l = 0,005$, *sensor emulator off, backward movement,* $\beta = 2,5l$

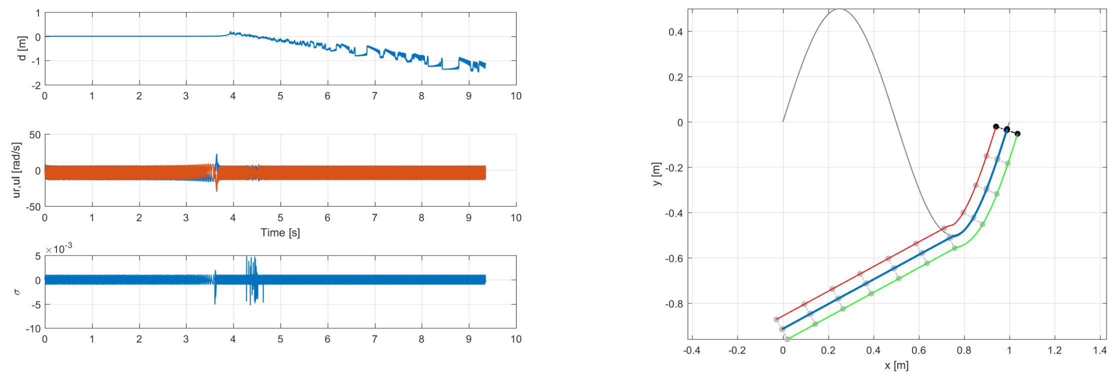

Figure D.4: $l = 0,005$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

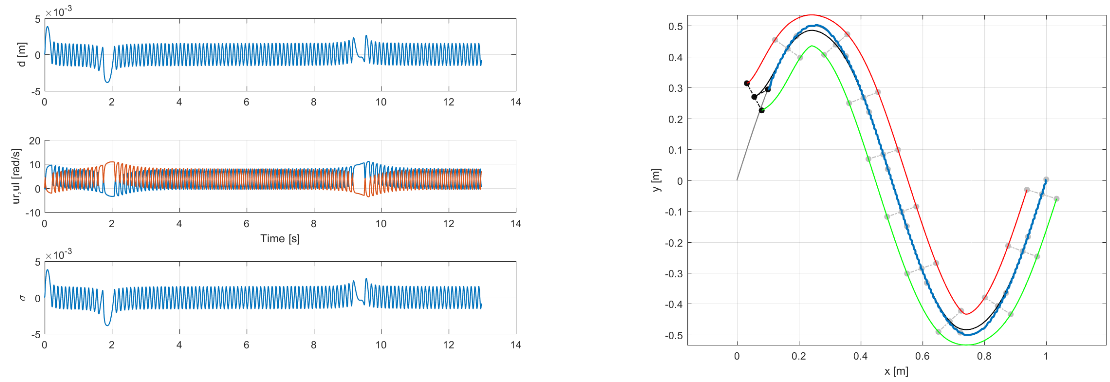

Figure D.5: $l = 0,05$, *sensor emulator off, forward movement.*

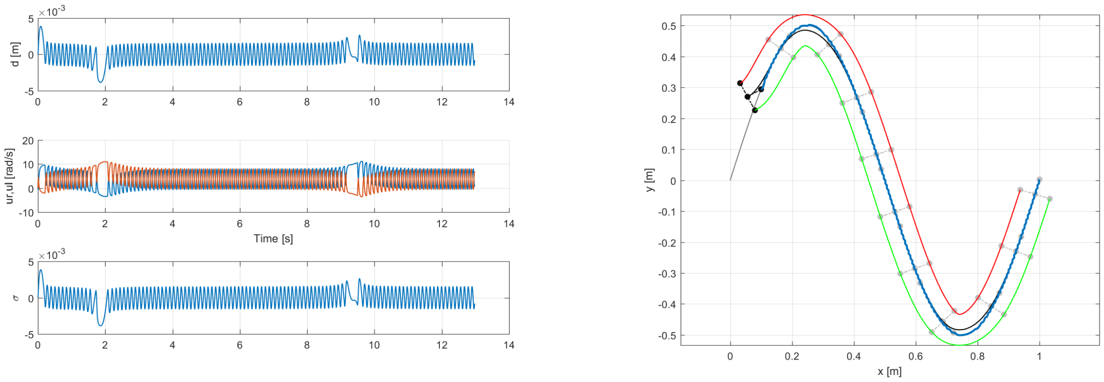

Figure D.6: $l = 0,05$, *sensor emulator on, forward movement.*

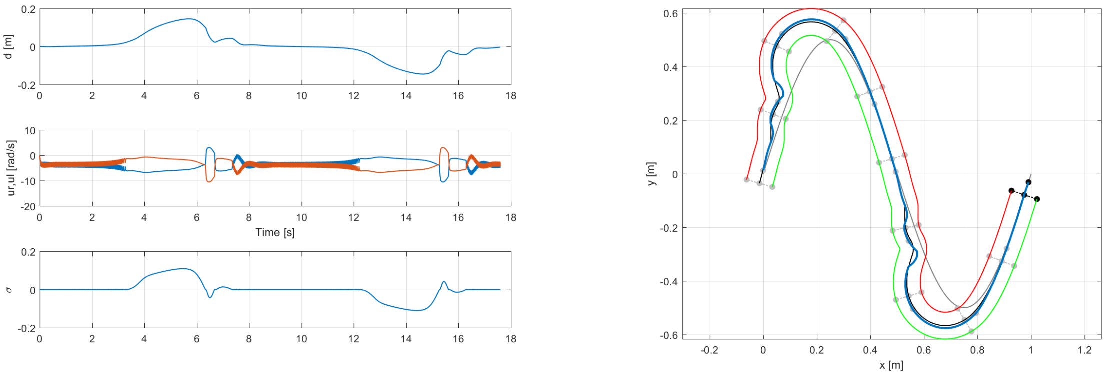

Figure D.7: $l = 0,05$, *sensor emulator off, backward movement,* $\beta = 2,5l$.

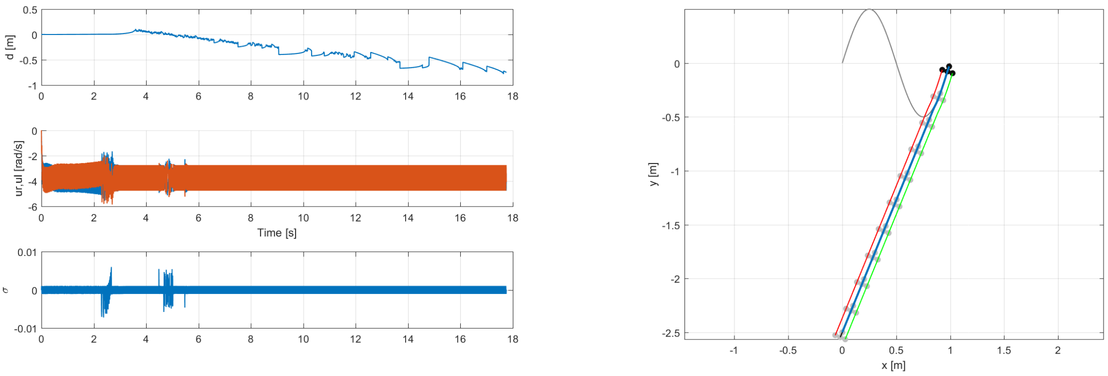

Figure D.8: $l = 0,05$, *sensor emulator on, backward movement,* $\beta = 2,5l$.

E | Pygame code

```
import pygame, sys, random, math
from pygame.locals import *
from math import *
import pygame.gfxdraw

import os
os.environ['SDL_VIDEO_WINDOW_POS'] = "%d,%d" % (100,100)

size = [1000,800]
pygame.init()
fpsTime = pygame.time.Clock()
fps = 60
font = pygame.font.SysFont("tahoma",15)
fontLittle = pygame.font.SysFont("tahoma",10)
pygame.display.set_caption('Animation')
screen = pygame.display.set_mode(size,HWSURFACE|DOUBLEBUF|RESIZABLE)
overScreen = pygame.Surface((2000,2000), pygame.SRCALPHA, 32)
overScreen = overScreen.convert_alpha()
color = 255,255,255


##### SIMULATION PARAMETERS #####
track = 4
A = 0.5
R = 0.05
l = 0.05
r = 0.02

Tmax = 0
dMax = 0
urMax = 0
ulMax = 0

##### SPRITES CLASSES ########
class graph(pygame.sprite.Sprite):
  def __init__(self,xname,yname,type):
```

```
        global viewerWindow, Ngraphs
        self.scale = 1
        self.xname = xname
        self.yname = yname
        self.width = int(screen.get_size()[0]*(1-viewerWindow))
        self.height = int((screen.get_size()[1]-60)/Ngraphs)
        self.image = pygame.Surface((self.width-10,self.height))
        self.imageAxis = pygame.Surface((self.width-10,self.height), pygame.SRCALPHA, 32)
        self.color = (0,0,0)
        self.axesColor = (255,255,255)
        self.image.fill(self.color)
        self.vgap = 20 #pixels
        self.hgap = 50 #pixels
        self.leftMargin = 20
        self.rightMargin = 5
        self.upMargin = 10
        self.downMargin = 10
        self.axisMarginLeft = 37
        self.axisMarginRight = 10
        self.axisMarginUp = 40
        self.axisMarginDown = 30
        self.type = type
        if type == 'd':
            self.pos = int(screen.get_size()[0]*viewerWindow)+5, self.upMargin
        elif type == 'ur':
            self.pos = int(screen.get_size()[0]*viewerWindow)+5, self.upMargin + self.height
        elif type == 'ul':
            self.pos = int(screen.get_size()[0]*viewerWindow)+5, self.upMargin + 2*self.height
        elif type == 'th':
            self.pos = int(screen.get_size()[0]*viewerWindow)+5, self.upMargin + 3*self.height

    def axes(self):
        global Tmax, dMax, urMax, ulMax
        #Paint horizontal line of the axis
        pygame.draw.line(self.imageAxis,self.axesColor,(self.axisMarginLeft,self.height/2),(self.width-self.axisMarginRight,self.height/2))
        #Paint vertical line of the axis
        pygame.draw.line(self.imageAxis,self.axesColor,(self.axisMarginLeft,self.axisMarginUp),(self.axisMarginLeft,self.height-self.axisMarginDown))
        #Paint horizontal marks on the axis
        xLength = (self.width-self.axisMarginLeft-self.axisMarginRight)/self.hgap
        n = Tmax/xLength
        for i in range(int(xLength)):
            pygame.draw.line(self.imageAxis,self.axesColor,(self.axisMarginLeft+(i+1)*self.hgap,self.height/2-2), (self.axisMarginLeft+(i+1)*self.hgap,self.height/2+2))
            text = fontLittle.render(str(round(n*(i+1),1)),0,(255,255,255))
            self.imageAxis.blit(text,(self.axisMarginLeft+(i+1)*self.hgap,self.height/2+3))
```

```
    #Paint vertical marks on the axis
    yLength = int((self.height/2-self.axisMarginUp)/self.vgap)
    n = dMax/yLength
    for i in range(yLength):
      pygame.draw.line(self.imageAxis,self.axesColor,(self.axisMarginLeft-2,self.height/2-self.vgap-i*self.vgap), (self.axisMarginLeft+2,self.height/2-self.vgap-i*self.vgap))
      text = fontLittle.render(str(round(n*100*(i+1),1)),0,(255,255,255))
      self.imageAxis.blit(text,(self.axisMarginLeft-25,self.height/2-self.vgap-i*self.vgap-11))
    for i in range(int((self.height/2-self.axisMarginDown)/self.vgap)):
      pygame.draw.line(self.imageAxis,self.axesColor,(self.axisMarginLeft-2,self.height/2+self.vgap+i*self.vgap), (self.axisMarginLeft+2,self.height/2+self.vgap+i*self.vgap))
      text = fontLittle.render(str(round(-n*100*(i+1),1)),0,(255,255,255))
      self.imageAxis.blit(text,(self.axisMarginLeft-29,self.height/2+self.vgap+i*self.vgap))
    #Paint name of the vertical axis
    text = font.render(self.xname,0,(200,150,100))
    self.imageAxis.blit(text,(10,2))
    #Paint name of the horizontal axis
    text = font.render(self.yname,0,(255,150,100))
    self.imageAxis.blit(text,(self.width/2,self.height-self.axisMarginDown))
    #Blit alpha axes image on the graph image
    self.blitAxes()

  def blitAxes(self):
    self.image.blit(self.imageAxis,(0,0))

  def relocate(self):
    global viewerWindow
    if self.type == 'd':
      self.pos = int(screen.get_size()[0]*viewerWindow)+self.leftMargin, self.upMargin
    elif self.type == 'ur':
      self.pos = int(screen.get_size()[0]*viewerWindow)+self.leftMargin, self.upMargin + self.height + 5
    elif self.type == 'ul':
      self.pos = int(screen.get_size()[0]*viewerWindow)+self.leftMargin, self.upMargin + 2*self.height + 5*2
    elif self.type == 'th':
      self.pos = int(screen.get_size()[0]*viewerWindow)+self.leftMargin, self.upMargin + 3*self.height + 5*3

  def resize(self):
    global viewerWindow
    self.width = screen.get_size()[0]
    self.width *=(1-viewerWindow)
    self.width -= self.leftMargin + self.rightMargin
    self.height = (screen.get_size()[1]-self.upMargin)/Ngraphs-self.downMargin
```

```
    self.image = pygame.Surface((self.width, self.height))
    self.image.fill(self.color)
    self.imageAxis = pygame.Surface((self.width-10,self.height), pygame.SRCALPHA, 32)
    self.axes()

  def graphPaint(self,pos1,pos2):
    global dMax,urMax,ulMax

    if self.type == 'd':
      MAX = dMax
      color = (255,0,0)
    elif self.type == 'ur':
      MAX = urMax
      color = (100,200,0)
    elif self.type == 'ul':
      MAX = ulMax
      color = (110,50,250)

    pos1 = list(pos1)
    pos2 = list(pos2)
    pos1[0] = self.axisMarginLeft+pos1[0]/Tmax*(self.width-self.axisMarginLeft-self.axisMarginRight)
    pos1[1] = pos1[1]/MAX*((self.height-self.axisMarginUp-self.axisMarginDown)/2)
    pos2[0] = self.axisMarginLeft+pos2[0]/Tmax*(self.width-self.axisMarginLeft-self.axisMarginRight)
    pos2[1] = pos2[1]/MAX*((self.height-self.axisMarginUp-self.axisMarginDown)/2)
    if int(pos1[0])!= int(pos2[0]) or int(pos1[1])!= int(pos2[1]):
      pygame.draw.line(self.image,color,(pos1[0],int((self.height-self.axisMarginUp-self.axisMarginDown)/2)+self.axisMarginUp-pos1[1]),(pos2[0],int((self.height-self.axisMarginUp-self.axisMarginDown)/2)+self.axisMarginUp-pos2[1]),1)
    #self.blitAxes()
  def repaint(self):
    global graphTime
    graphTime = 0
    self.resize()
    self.relocate()
    self.axes()

class dividingBar(pygame.sprite.Sprite):
  def __init__(self):
    global viewerWindow
    self.height = screen.get_size()[1]+50
    self.width = 4
    self.image = pygame.Surface((self.width, self.height))
    self.image.fill((100,100,100))
    pygame.draw.line(self.image,(200,200,200),(0,0),(0,self.height),1)
    pygame.draw.line(self.image,(200,200,200),(self.width,0),(self.width,self.height),1)
```

```
    self.pos = screen.get_size()[0]*viewerWindow-self.width, 0
    self.rect = pygame.Rect((self.pos[0],self.pos[1]),(self.width,self.height))
    self.active = False

  def relocate(self):
    global viewerWindow, dGraph, urGraph, ulGraph, graphPanel#, thGraph
    if self.active:
      viewerWindow = pygame.mouse.get_pos()[0]/screen.get_size()[0]
    self.pos = [screen.get_size()[0]*viewerWindow-4, 0]
    self.rect = pygame.Rect((self.pos[0],self.pos[1]),(self.width,self.height))
    dGraph.repaint()
    urGraph.repaint()
    ulGraph.repaint()
    #thGraph.repaint()
    graphPanel = pygame.Surface((screen.get_size()[0]*(1-viewerWindow),screen.get_size()
    [1]))
    graphPanel.fill(graphPanelColor)

  def toggle(self):
    global activeGraph
    if self.active:
      self.active = False
      activeGraph = True
    else:
      self.active = True
      activeGraph = False

  def update(self):
    if self.active:
      self.relocate()
    pos = pygame.mouse.get_pos()
    if self.rect.collidepoint(pos):
      pygame.mouse.set_cursor(size,hotspot,*cursor)
    else:
      if B[0] == 0:
        pygame.mouse.set_cursor(*pygame.cursors.arrow)

class button(pygame.sprite.Sprite):
  def __init__(self,name,pos):
    self.imageNorm = pygame.image.load('skins/'+name+'.png')
    self.image = self.imageNorm
    self.imageInv = pygame.image.load('skins/'+name+'_inv'+'.png')
    if name == 'pause':
      self.imagePlay = pygame.image.load('skins/play.png')
      self.imagePlayInv = pygame.image.load('skins/play_inv.png')
      self.imagePause = self.imageNorm
      self.imagePauseInv = self.imageInv
```

```
        if name == 'forward':
            self.imageBackward = pygame.image.load('skins/backward.png')
            self.imageBackwardInv = pygame.image.load('skins/backward_inv.png')
            self.imageForward = self.imageNorm
            self.imageForwardInv = self.imageInv
        self.pos = pos
        self.rect = self.image.get_rect()
        self.rect[0] = pos[0]
        self.rect[1] = pos[1]
        self.name = name
        if name[:-1] == 'track':
            global track
            self.track = int(self.name[-1])
            if track == self.track:
                self.image = self.imageInv
    def restart(self):
        global count, repeat, pause, time, Pshort, thetaMatrix, track, Ppath, RWpath, LWpath, graphTime, dGraph, urGraph, ulGraph, test #, thGraph
        Ppath = []
        RWpath = []
        LWpath = []
        track = self.track
        loadData()
        loadGraphData()
        pathGuide()
        graphTime = 0
        dGraph.repaint()
        ulGraph.repaint()
        urGraph.repaint()
        #thGraph.repaint()
    def action(self):
        global count, repeat, pause, time, Pshort, thetaMatrix, track, Ppath, RWpath, LWpath, graphTime, dGraph, urGraph, ulGraph, test #, thGraph
        if self.name == 'repeat':
            if repeat == True:
                repeat = False
            else:
                repeat = True
        elif self.name == 'pause':
            if not pause:
                pause = True
                self.imageNorm = self.imagePlay
                self.imageInv = self.imagePlayInv
            else:
                pause = False
                self.imageNorm = self.imagePause
                self.imageInv = self.imagePauseInv
```

```
    elif self.name == 'forward':
      Ppath = []
      RWpath = []
      LWpath = []
      graphTime = 0
      dGraph.repaint()
      urGraph.repaint()
      ulGraph.repaint()
      #thGraph.repaint()
      count = 0
      time = 0
      theta = -Pshort[count][2]
      thetaMatrix = Rz(pi/2+theta)
      if test == 2:
        test = 1 #moving forward
        self.imageNorm = self.imageForward
        self.imageInv = self.imageForwardInv
        loadData()
        loadGraphData()
      else:
        test = 2 #moving backwards
        self.imageNorm = self.imageBackward
        self.imageInv = self.imageBackwardInv
        loadData()
        loadGraphData()

    elif self.name == 'playInit':
      Ppath = []
      RWpath = []
      LWpath = []
      graphTime = 0
      dGraph.repaint()
      urGraph.repaint()
      ulGraph.repaint()
      #thGraph.repaint()
      count = 0
      time = 0
      theta = -Pshort[count][2]
      thetaMatrix = Rz(pi/2+theta)

    elif self.name == 'track1' or self.name == 'track2' or self.name == 'track3' or self.name == 'track4':
      self.restart()

##### MATH FUNCTIONS NEEDED ######
#SCALAR PRODUCT
def SxV(s,v):
```

```
  v2 = []
  for i in range(len(v)):
    v2+=[s*v[i]]
  return(v2)

#ROTATION FUNCTIONS
def Rx(a):
  Rx = [[1,0,0],[0,cos(a),-sin(a)],[0,sin(a),cos(a)]]
  return(Rx)
def Ry(a):
  Ry = [[cos(a),0,-sin(a)],[0,1,0],[sin(a),0,cos(a)]]
  return(Ry)
def Rz(a):
  Rz = [[cos(a),-sin(a),0],[sin(a),cos(a),0],[0,0,1]]
  return(Rz)

#MULTIPLICATION of ROTATION-MATRIX and a VECTOR
def MxV(M,v):
  v2 = [0,0,0]
  v2[0] = M[0][0]*v[0] + M[0][1]*v[1] + M[0][2]*v[2]
  v2[1] = M[1][0]*v[0] + M[1][1]*v[1] + M[1][2]*v[2]
  v2[2] = M[2][0]*v[0] + M[2][1]*v[1] + M[2][2]*v[2]
  return(v2)

#MATRIX MULTIPLICATION
def mxM(m1,m2):
 M = [[0,0,0],[0,0,0],[0,0,0]]
  for i in [0,1,2]:
    for j in [0,1,2]:
      M[i][j] = m1[i][0]*m2[0][j] + m1[i][1]*m2[1][j] + m1[i][2]*m2[2][j]
  return(M)

#WHEEL POINTS: calculates all the vertexes of a polygon respect to a generic path
def polygon(path,points):
  global theta, count
  c = []
  for i in range(len(points)):
    c+=[(MxV(thetaMatrix,SxV(scale,points[i]))[0]+path[count][0],MxV(thetaMatrix,SxV(scale
    ,points[i]))[1]+path[count][1],MxV(thetaMatrix,SxV(scale,points[i]))[2]+path[count
    ][2])]
    n = MxV(GRM,(c[i][0]-origin_pos[0],c[i][1]-origin_pos[1],c[i][2]))
    c[i] = n[0]+origin_pos[0],n[1]+origin_pos[1]
  return(c)

def polygon2(path,points):
  global theta, count
  c = []
```

```
  for i in range(len(points)):
    c+=[(MxV(thetaMatrix,SxV(scale,points[i]))[0]+path[count][0],MxV(thetaMatrix,SxV(scale,points[i]))[1]+path[count][1],MxV(thetaMatrix,SxV(scale,points[i]))[2]+path[count][2])]
    n = MxV(GRM,(c[i][0]-origin_pos[0],c[i][1]-origin_pos[1],c[i][2]))
    c[i] = n[0],n[1],n[2]
  return(c)

#UPDATE FOLLOWING TRACKS OF WHEELS AND POINT P
def updateTracks():
  global Ppath, RWpath, LWpath, count
  Ppath = []
  RWpath = []
  LWpath = []
  count_dump = count
  count = 0
  for i in range(count_dump):
    Ppath += polygon(Pshort,[[0,0,0]])
    RWpath += polygon(RWshort,[[0,0,0]])
    LWpath += polygon(LWshort,[[0,0,0]])
    count+=1
  count = count_dump


##### PROGRAM VARIABLES & PARAMETERS#####
wasd = [False,False,False,False] #Moving up, left , down and/or right
origin_pos = [900,900] #Origin displacement
origin = [-700,-700]
origin2 = tuple(origin)
scale = 200 #scale of image: 1u = 500px
time = 0 #Measure of time from "frames per second"
interval = 0.02 #Period between positions
count = 0 #Position iteration
reset = 0
path = []
Pshort = []
RWshort = []
LWshort = []
test = 1 # 1: forward movement 2: backward movement

Ngraphs = 3
viewerWindow = 0.5
graphPanel = pygame.Surface((screen.get_size()[0]*viewerWindow,screen.get_size()[1]))
graphPanelColor = (20,20,70)
graphPanel.fill(graphPanelColor)
graphCount = 0

```

```

dGraph = graph('d[m] x0,01','t[s]','d')
dGraph.axes()
urGraph = graph('w[rad/s] right wheel','t[s]','ur')
urGraph.axes()
ulGraph = graph('w[rad/s] left wheel','t[s]','ul')
ulGraph.axes()
graphTime = 0
activeGraph = True

bar = dividingBar()

theta = 0
thetaMatrix = [[1,0,0],[0,1,0],[0,0,1]]


h1 = 0
h2 = 0.015
chPoints =[]
chPoints += [[[0,0,h2],[R/4,0,h2],[R/2,l/5,h2],[R/2,l,h2],[-R/2,l,h2],[-R/2,l/5,h2],[-R/4,0,h2]]] #chassis points with respect to point p


whPoints = [[-r/4,r,0],[r/4,r,0],[r/4,-r,0],[-r/4,-r,0]] #wheels points with respect to the wheel centre

Ppath = []
RWpath = []
LWpath = []
d = []
ul = []
ur = []

cursor = pygame.cursors.compile(pygame.cursors.sizer_x_strings,black='.',white='X')
size = len(pygame.cursors.sizer_x_strings[0]), len(pygame.cursors.sizer_x_strings)
hotspot = (6,6)

buttons = [button('playInit',(1,1)),button('pause',(51,1)),button('repeat',(101,1)),button('forward',(151,1))]
buttons2 = []
for i in [1,2,3,4]:
  buttons2 += [button('track'+str(i),(151+50*i,20))]
collision = False
B = [0,0,0]

pause = False
repeat = False
```

```

GRM = [[1,0,0],[0,1,0],[0,0,1]] #Global Rotation Matrix
ax = 0
ay = 0
az = 0
Ax = 0
Ay = 0
Az = 0

##### LOAD DATA ######
# P location list of the vehicle for each iteration
def loadData():
  global Pshort, RWshort, LWshort, interval, count, time, track, T, P, RW, LW, Tmax, test
  time = 0
  count = 0
  P = []
  T = []
  RW = []
  LW = []
  file1 = open('P'+str(track)+'_'+str(test)+'.txt','r')
  file2 = open('T'+str(track)+'_'+str(test)+'.txt','r')
  file3 = open('RW'+str(track)+'_'+str(test)+'.txt','r')
  file4 = open('LW'+str(track)+'_'+str(test)+'.txt','r')
  line1 = file1.readline()
  line2 = file2.readline()
  line3 = file3.readline()
  line4 = file4.readline()
  while line1 != '':
    line1 = str.split(line1)
    line2 = str.split(line2)
    line3 = str.split(line3)
    line4 = str.split(line4)
    line1[0] = float(line1[0])
    line1[1] = float(line1[1])
    line1[2] = float(line1[2])
    line3[0] = float(line3[0])
    line3[1] = float(line3[1])
    line3[2] = float(line3[2])
    line4[0] = float(line4[0])
    line4[1] = float(line4[1])
    line4[2] = float(line4[2])
    P += [line1]
    T += [float(line2[0])]
    RW += [line3]
    LW += [line4]
    line1 = file1.readline()
    line2 = file2.readline()
```

```
    line3 = file3.readline()
    line4 = file4.readline()
  shortenData()
  Tmax = max(T)

def shortenData():
  global P, Pshort, RW, RWshort, LW, LWshort, T, Tshort
  # SHORTENED PATHS:
  # Short and fixed interval between positions in P list and RW list
  Pshort = []
  RWshort = []
  LWshort = []
  QDsimsshort = []
  Tshort = []
  t = 0
  for i in range(len(P)):
    if T[i] >= t:
      Pshort += [(P[i][0]*scale + origin_pos[0], -P[i][1]*scale + origin_pos[1], P[i][2])]
      RWshort += [(int(RW[i][0]*scale) + origin_pos[0], int(-RW[i][1]*scale) + origin_pos[1], RW[i][2])]
      LWshort += [(int(LW[i][0]*scale) + origin_pos[0], int(-LW[i][1]*scale) + origin_pos[1], LW[i][2])]
      QDsimsshort += [(int(LW[i][0]*scale) + origin_pos[0], int(-LW[i][1]*scale) + origin_pos[1], LW[i][2])]
      Tshort += [T[i]]
      t += interval

def loadGraphData():
  global d, ul, ur,dMax,urMax,ulMax,test
  d = []
  ul = []
  ur = []
  file1 = open('d'+str(track)+'_'+str(test)+'.txt','r')
  file2 = open('ur'+str(track)+'_'+str(test)+'.txt','r')
  file3 = open('ul'+str(track)+'_'+str(test)+'.txt','r')
  line1 = file1.readline()
  line2 = file1.readline()
  line3 = file1.readline()
  while line1 != '':
    line1 = str.split(line1)
    line2 = str.split(line2)
    line3 = str.split(line3)
    line1[0] = float(line1[0])
    line2[0] = float(line2[0])
    line3[0] = float(line3[0])
    d+=line1
    ur+=line2
```

```
        ul+=line3
        line1 = file1.readline()
        line2 = file2.readline()
        line3 = file3.readline()
    dMax = max(abs(max(d)),abs(min(d)))
    urMax = max(abs(max(ur)),abs(min(ur)))
    ulMax = max(abs(max(ul)),abs(min(ul)))


loadData()
loadGraphData()

##### CREATE BLACK PATH LINE #####
#Creates a list with the objective points to follow.
#And draws the point onto pathImage
def pathGuide():
    global path, A, track
    path = []
    if track == 2:
        m = 0
        qr = 1/2
        j = 0
        for i in range(101):
            if i >= qr*100 and j==0:
                m = 1/3
                j = i
            path += [(int(i/100*scale)+origin_pos[0],int(-m*(i-j)/100*scale+origin_pos[1]))]
            n = MxV(GRM,(path[i][0]-origin_pos[0],path[i][1]-origin_pos[1],0))
            path[i] = n[0]+origin_pos[0],n[1]+origin_pos[1]
    elif track == 3:
        for i in range(180):
            path += [(int(0.5*(cos(i*pi/180)*scale+scale))+origin_pos[0],int(0.5*(-sin(i*pi/180)
        *scale))+origin_pos[1])]
            n = MxV(GRM,(path[i][0]-origin_pos[0],path[i][1]-origin_pos[1],0))
            path[i] = n[0]+origin_pos[0],n[1]+origin_pos[1]
    elif track == 4:
        for i in range(101):
            path += [(int(i/100*scale)+origin_pos[0],-int(A*sin(2*pi*i/100)*scale)+origin_pos
        [1])]
            n = MxV(GRM,(path[i][0]-origin_pos[0],path[i][1]-origin_pos[1],0))
            path[i] = n[0]+origin_pos[0],n[1]+origin_pos[1]
pathGuide()

while True:
############### BEGIN EVENT MANAGEMENT ###############
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

```
            sys.exit()

        elif event.type==VIDEORESIZE:
            screen=pygame.display.set_mode(event.dict['size'],HWSURFACE|DOUBLEBUF|RESIZABLE)
            bar.relocate()

        elif event.type == KEYDOWN:
            key_down = pygame.key.name(event.key)

            #Asignar valor binario True al vector binario de movimiento
            if key_down == 'left ctrl':
                left_control = True
            elif key_down == 'r':
                reset = True
                GRM = [[1,0,0],[0,1,0],[0,0,1]]
            elif key_down == 'escape':
                sys.exit()

        elif event.type == KEYUP:
            key_up = pygame.key.name(event.key)

            #Asignar valor binario False al vector binario de movimiento
            if key_up == 'left ctrl':
                left_control = False
            elif key_down == 'r':
                reset = False

        elif event.type == MOUSEBUTTONDOWN:
            B = pygame.mouse.get_pressed()
            mouseDown = pygame.mouse.get_pos()
            collision = False
            if B[0]:
                for i in buttons:
                    if i.rect.collidepoint(mouseDown):
                        collision = True
                        if i.name == 'repeat':
                            if repeat:
                                i.image = i.imageNorm
                                i.action()
                            else:
                                i.image = i.imageInv
                                i.action()
                        else:
                            i.image = i.imageInv
                            i.action()
                for i in buttons2:
                    if i.rect.collidepoint(mouseDown):
```

```
            collision = True
            if track != i.track:
              i.action()
              i.image = i.imageInv
              for j in buttons2:
                if j != i:
                  j.image = j.imageNorm
        if bar.rect.collidepoint(mouseDown):
          bar.toggle()

      if event.button == 4:
        scale+=20
        shortenData()
        pathGuide()
        updateTracks()
      elif event.button == 5:
        scale -=20
        shortenData()
        pathGuide()
        updateTracks()

    elif event.type == MOUSEBUTTONUP:
      origin2 = tuple(origin)
      B = pygame.mouse.get_pressed()
      mouseUp = pygame.mouse.get_pos()
      if not B[0]:
        collision = False
        for i in buttons:
          if i.name != 'repeat':
            i.image = i.imageNorm
      if bar.active:
        bar.toggle()


###### PROGRAM LOGIC #####
  lenPshort = len(Pshort)-1
  if time >= interval*count and count<lenPshort and not pause:
    count += 1
    theta = Pshort[count][2]
    thetaMatrix = Rz(pi/2-theta)

  Ppath += polygon(Pshort,[[0,0,0]])
  RWpath += polygon(RWshort,[[0,0,0]])
  LWpath += polygon(LWshort,[[0,0,0]])

  if count>=lenPshort and repeat:
    count = 0
```

```
    time = 0
    Ppath = []
    RWpath = []
    LWpath = []
    graphTime = 0
    dGraph.repaint()
    urGraph.repaint()
    ulGraph.repaint()
    theta = Pshort[count][2]
    thetaMatrix = Rz(pi/2-theta)

if ax or ay or az or reset:
    if ax:
        Ax+= ax
        m = Rx(ax)
    elif ay:
        Ay+= ay
        m = Ry(ay)
    elif az:
        Az+= az
        m = Rz(az)
    GRM = mxM(m,GRM)
    m = [[1,0,0],[0,1,0],[0,0,1]]
    pathGuide()
    updateTracks()

if not collision and B[0]:
    origin[0] = origin2[0]+pygame.mouse.get_pos()[0]-mouseDown[0]
    origin[1] = origin2[1]+pygame.mouse.get_pos()[1]-mouseDown[1]

if activeGraph:
    #Update graphs
    while T[graphTime] < interval*count:
        pos1 = (T[graphTime],d[graphTime])
        pos2 = (T[graphTime+1],d[graphTime+1])
        dGraph.graphPaint(pos1,pos2)
        pos1 = (T[graphTime],ur[graphTime])
        pos2 = (T[graphTime+1],ur[graphTime+1])
        urGraph.graphPaint(pos1,pos2)
        pos1 = (T[graphTime],ul[graphTime])
        pos2 = (T[graphTime+1],ul[graphTime+1])
        ulGraph.graphPaint(pos1,pos2)
        graphTime+=1

#Update bar
bar.update()
```

```
############### SCREEN FILLING #####################
  screen.fill(color)
  overScreen.fill((20,20,20))

  #Draw blue base (base)
  base = []
  for i in [[-0.1,-0.6,0],[1.1,-0.6,0],[1.1,0.6,0],[-0.1,0.6,0]]:
    i = SxV(scale,i)
    b = MxV(GRM,i)
    b = b[0]+origin_pos[0], b[1]+origin_pos[1]
    base += [b]
  pygame.draw.polygon(overScreen,(170,190,230),base)

  #Draw path guide (path)
  for i in path:
    pygame.draw.lines(overScreen,(0,0,0),False,path,1)

  #Draw paths of P, RW and LW (Ppath, RWpath, LWpath)
  if len(Ppath)>1:
    pygame.draw.lines(overScreen,(250,0,0),False,Ppath,1)
    pygame.draw.lines(overScreen,(0,250,0),False,RWpath,1)
    pygame.draw.lines(overScreen,(0,0,250),False,LWpath,1)

  #Draw wheels on the scene
  if count<=lenPshort:
    pygame.draw.polygon(overScreen,(0,250,0),polygon(RWshort,whPoints)) #right wheel
    pygame.draw.polygon(overScreen,(0,0,250),polygon(LWshort,whPoints)) #left wheel

  #Calculate the printing order of each chassis polygon
  chassisPoints = {}
  priority = []
  for i in range(len(chPoints)):
    polypoly = polygon2(Pshort,chPoints[i])
    a = 0
    for j in range(len(polypoly)):
      a+=polypoly[j][2]
    a/=len(polypoly)
    priority+=[a]
    chassisPoints[a]=polygon(Pshort,chPoints[i])
  priority.sort()
  priority2 = []
  for i in range(len(priority)):
    priority2+=[priority[i]+abs(min(priority))]

  #Draw chassis on the screen
  if count<=lenPshort:
    for i in range(len(chPoints)):
```

```
    pcolor = abs(priority2[i])/abs(max(priority2))
    pygame.draw.polygon(overScreen,(pcolor*255,10,10),chassisPoints[priority[i]])
    pygame.draw.polygon(overScreen,(0,0,0),chassisPoints[priority[i]],1)

#Blit overScreen on the screen
screen.blit(overScreen,origin)


#Draw graphPanel
screen.blit(graphPanel,(int(screen.get_size()[0]*viewerWindow),0))

if activeGraph:
  #Draw dGraph
  screen.blit(dGraph.image,dGraph.pos)
  #Draw urGraph
  screen.blit(urGraph.image,urGraph.pos)
  #Draw ulGraph
  screen.blit(ulGraph.image,ulGraph.pos)
  #Draw thGraph
  #screen.blit(thGraph.image,thGraph.pos)

#Draw bar
screen.blit(bar.image,bar.pos)

#Draw buttons on the interface
for i in buttons:
  screen.blit(i.image,i.pos)
for i in buttons2:
  screen.blit(i.image,i.pos)

#Pause if necessary and advance iteration
fps2 = fpsTime.get_fps()
if not pause and count<lenPshort:
  try: time +=  1/fps2#1/fps
  except: time += 1/fps

#Draw text on screen
text = font.render('fps: '+str(round(fps2,1)),0,(255,255,255))
screen.blit(text,(205,0))
text = font.render('Time: '+str(round(time,1))+' s',0,(0,0,0))
text = font.render('Time: '+str(round(Tshort[count],1))+' s',0,(255,255,255))
screen.blit(text,(305,0))

#frame flip
fpsTime.tick(fps)
pygame.display.flip()
```