AGH University of science and technology

Automatyka i Robotyka

Final project report

Embedded Systems II

Student:

JHON VELASQUEZ

Professor:

mgr inż. Dawid Knapik

Kraków, December 26th, 2022

CONTENT INDEX

FIGURE INDEX

Introduction

The present report describes the process of the final project of the course Embedded Systems II. It is taking the project from Embedded Systems I as starting point. The main change in hardware is the improvement in motors and their resolutions.

Regarding the objectives, they were not fulfilled totally. It was obtained a path generator for open loop control of position, and a closed loop position control design and simulation.

Conclusions that can be considered for future projects are shown at the end.

Review of Related Literature

The presented Project takes as starting point the project from Embedded Systems I.

2.1. Robot

The robot is a three-wheeled differentially robot. It has two independent wheels, which permits fair maneuverability. There's one free movement wheel on the front.

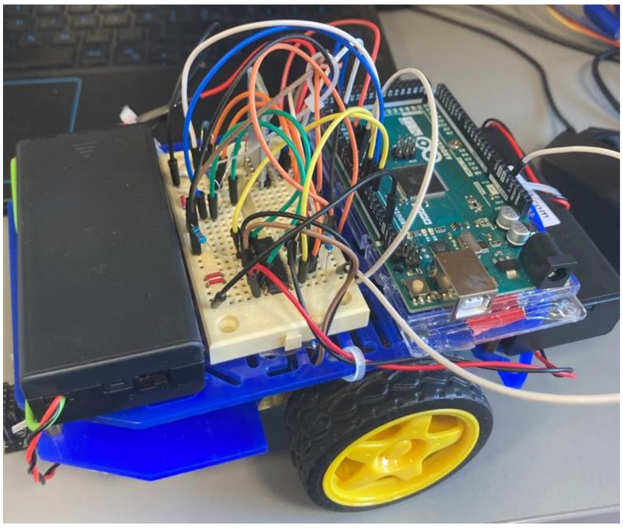


Figure 1. Robot hardware configuration, starting point.

2.2. Software repository

The repository of this project is public:

https://github.com/JhonVelasquez/Car_Speed_Control_2_wheels

Design

3.1. Hardware improvements

Two of the conclusions of the previous report was taken into consideration. Therefore, these was implemented in the car:

- Encoder with two channels so the direction can be known.
- Encoder with resolution 8 connected to the shaft of a motor with 120:1 gearbox. Total sensing resolution in 960.

The motor FIT0450 from DFROBOT was chosen. It uses two hall sensors for detecting the magnetic parts on the wheel connected to the shaft of the motor. Each sensor is one channel, according to the phase of both signals, it can be known the if the motor is going to one direction or the other one.

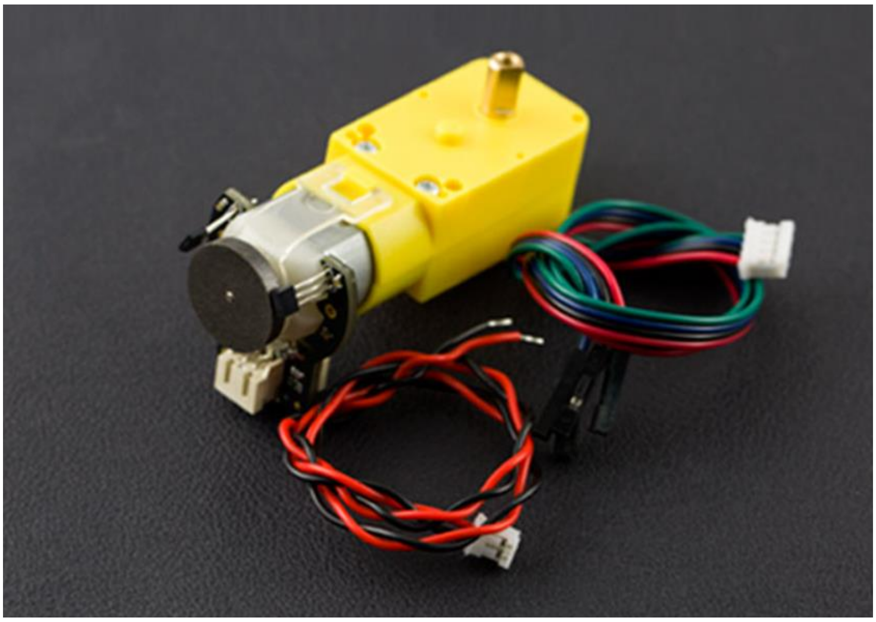


Figure 2. FI045 motor and enconder.

3.2. General schematic

In the figure below is shown the schematic of the system. There are the physical components and how they interact between them.

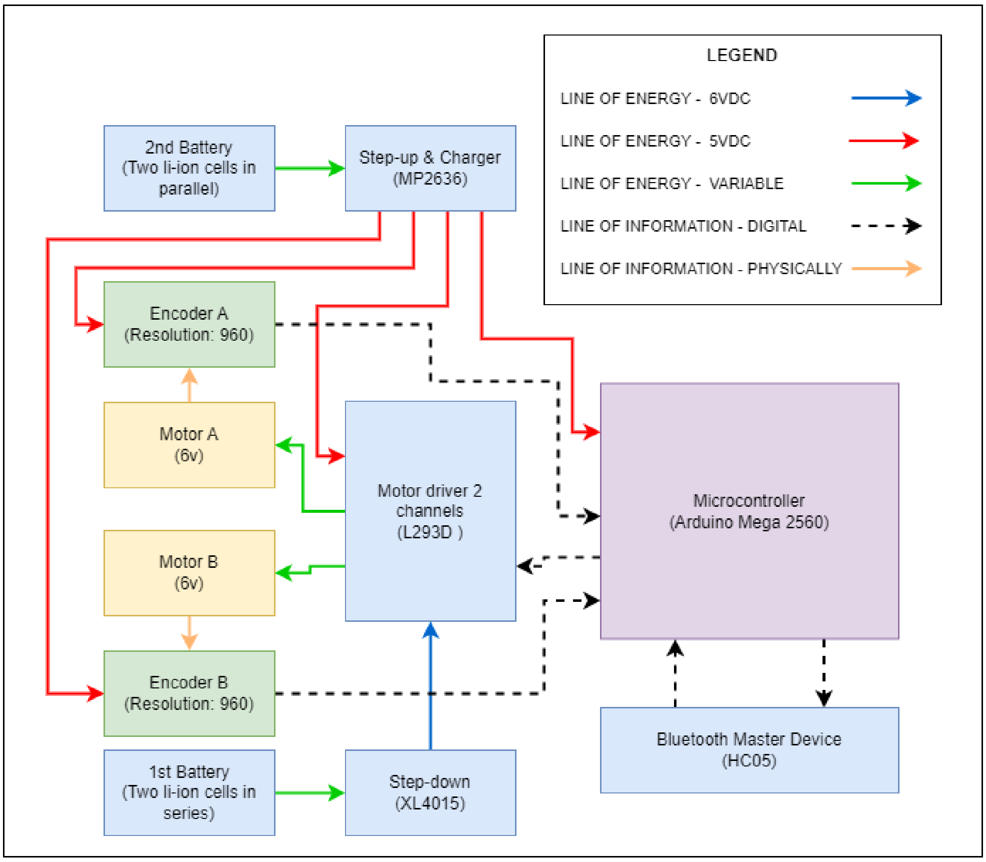


Figure 3. Updated General schematic

3.3. Control system

The design of the motor's speed control is shown in this section. Later it is shown the design for open loop and close loop of position control. These last two use the plant as if the inputs were rotational speeds, which means that when implementing on the car. It is a cascade control.

3.3.1. Motor's speed control

3.3.1.1. Identification

The code used for system identification was used with the new motors.

MotorA	MotorB
0.05	0.04
0.14	0
0.28	0.25
0.41	0.37
0.52	0.48
0.63	0.58
0.73	0.71
0.81	0.78
0.89	0.86
0.99	0.93
1.03	1
1.1	1.06
1.17	1.14
1.22	1.19

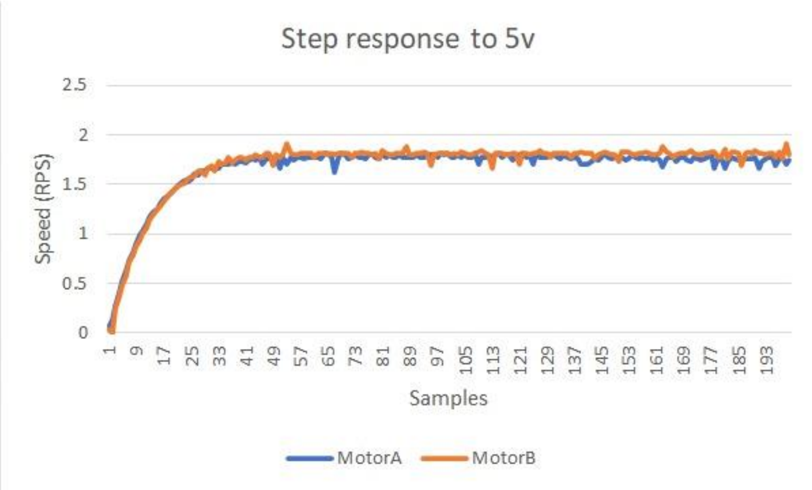


Figure 4. Step responses of the new motors

Using Matlabs System Identification Toolbox it was possible to get each model in continuous-time transfer function, see Figure 5 and Figure 6.

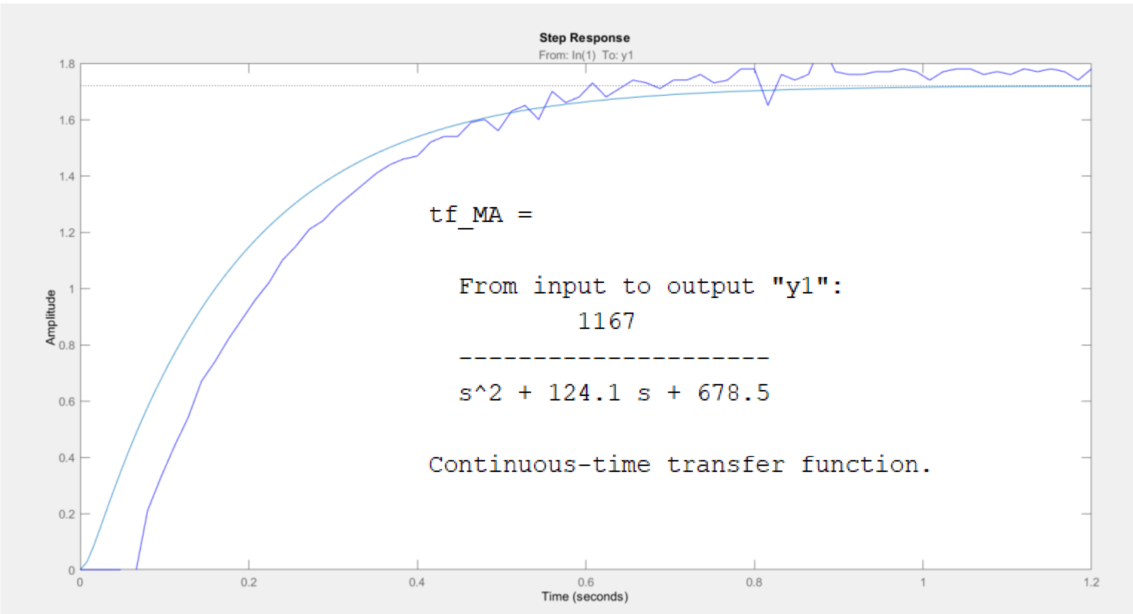


Figure 5. Fitted model for new MotorA and step response

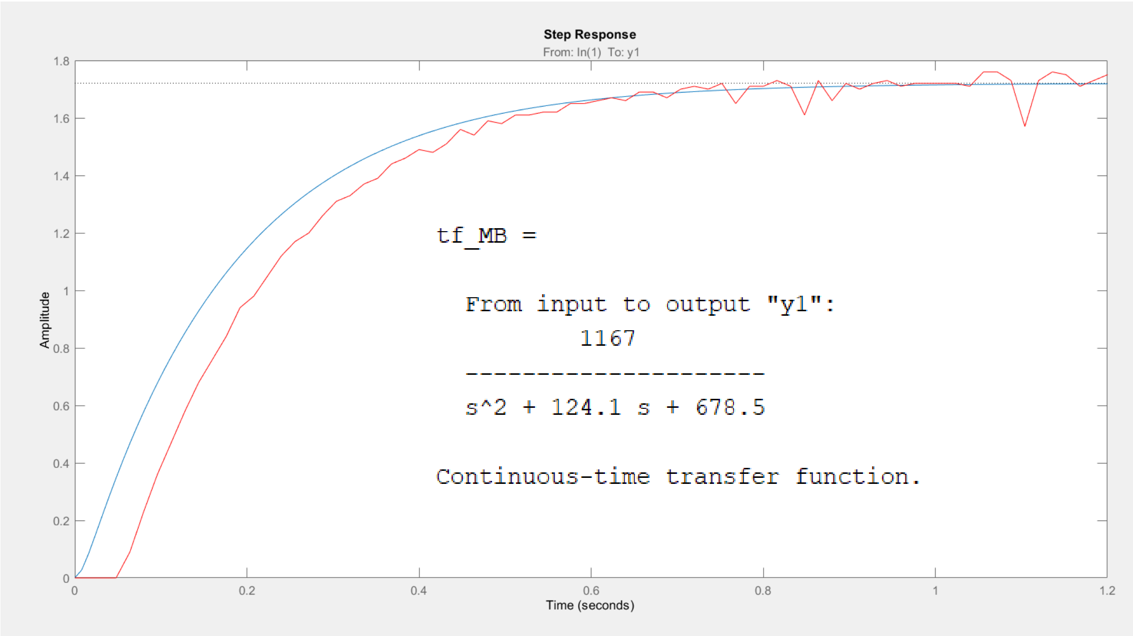


Figure 6. Fitted model for new MotorB and step response

3.3.1.2. Design

This is explained in the previous project. The updated results are shown in this section.

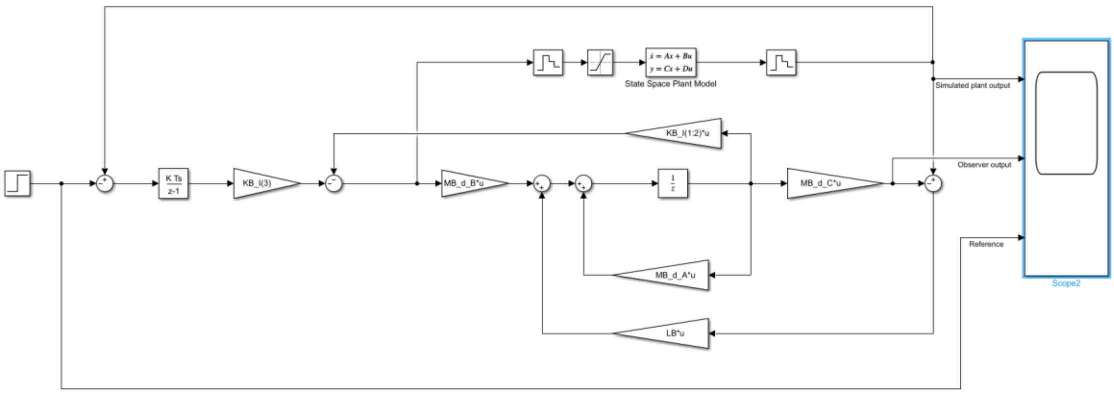


Figure 7. Simulink model of motor speed control

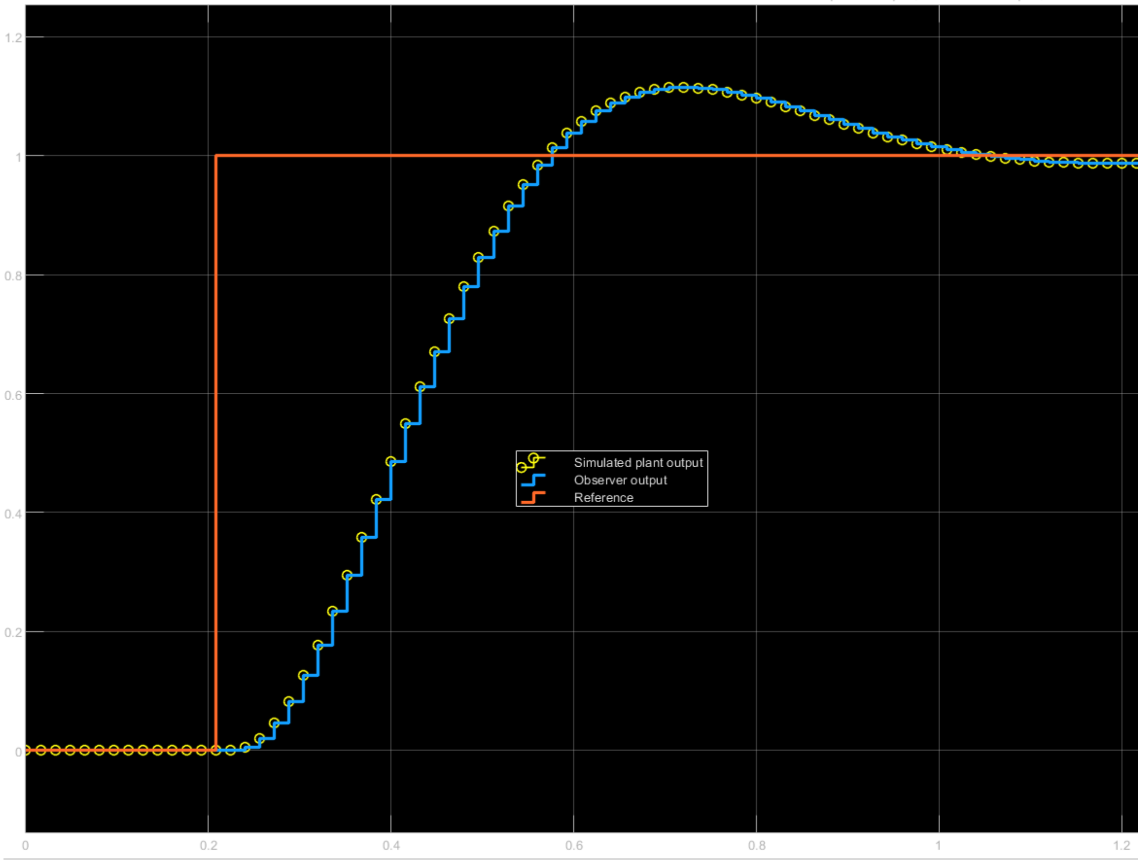


Figure 8. Simulink model response

3.3.2. Open loop position control

This can be considered as a route generator. The idea of this control is to have two vectors of positions in axes X and Y to be achieved by the car with k elements each one. Then it generates three vectors: speed Motor A, speed Motor B, and time, with k elements each one. Therefore, the motor will have to have as reference a speed MA_speed_k and one of MB_speed_k during a time t_k. The figure below shows the movement between two positions.

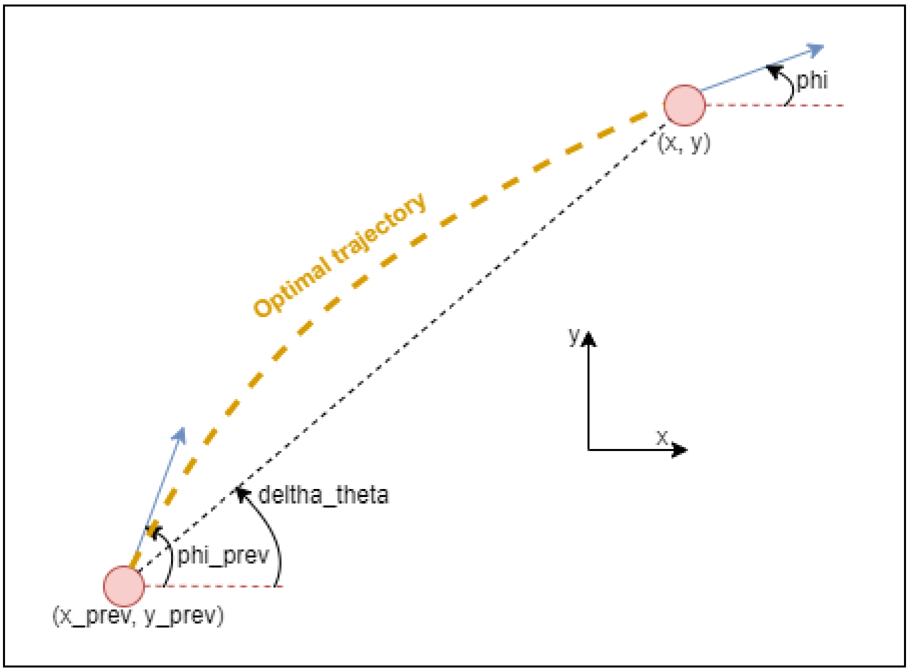


Figure 9. Trajectory between two positions

The optimal trajectory is obtained based on the kinematics of the car when moving a car between a previous position to a desired position. The idea is to obtain fixed rotation velocities of the wheels so the car can get to the desired position.

3.3.2.1. Considerations

- The implantation was done in Matlab. The subsections show in order how the data is obtained from position references to speed references.
- All calculations in the Matlab script require a wheel rotation speed.
- The scripts are in the Generation_path directory:

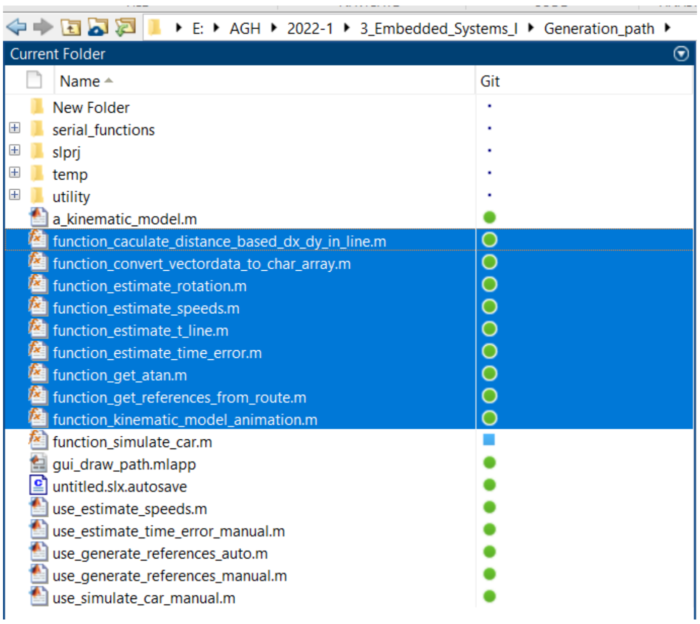


Figure 10. Generation_path directory

- There are scripts that start with "use_", they are for testing the functions. The main one, the use of the main function is explained forward.

3.3.2.2. General main script flow

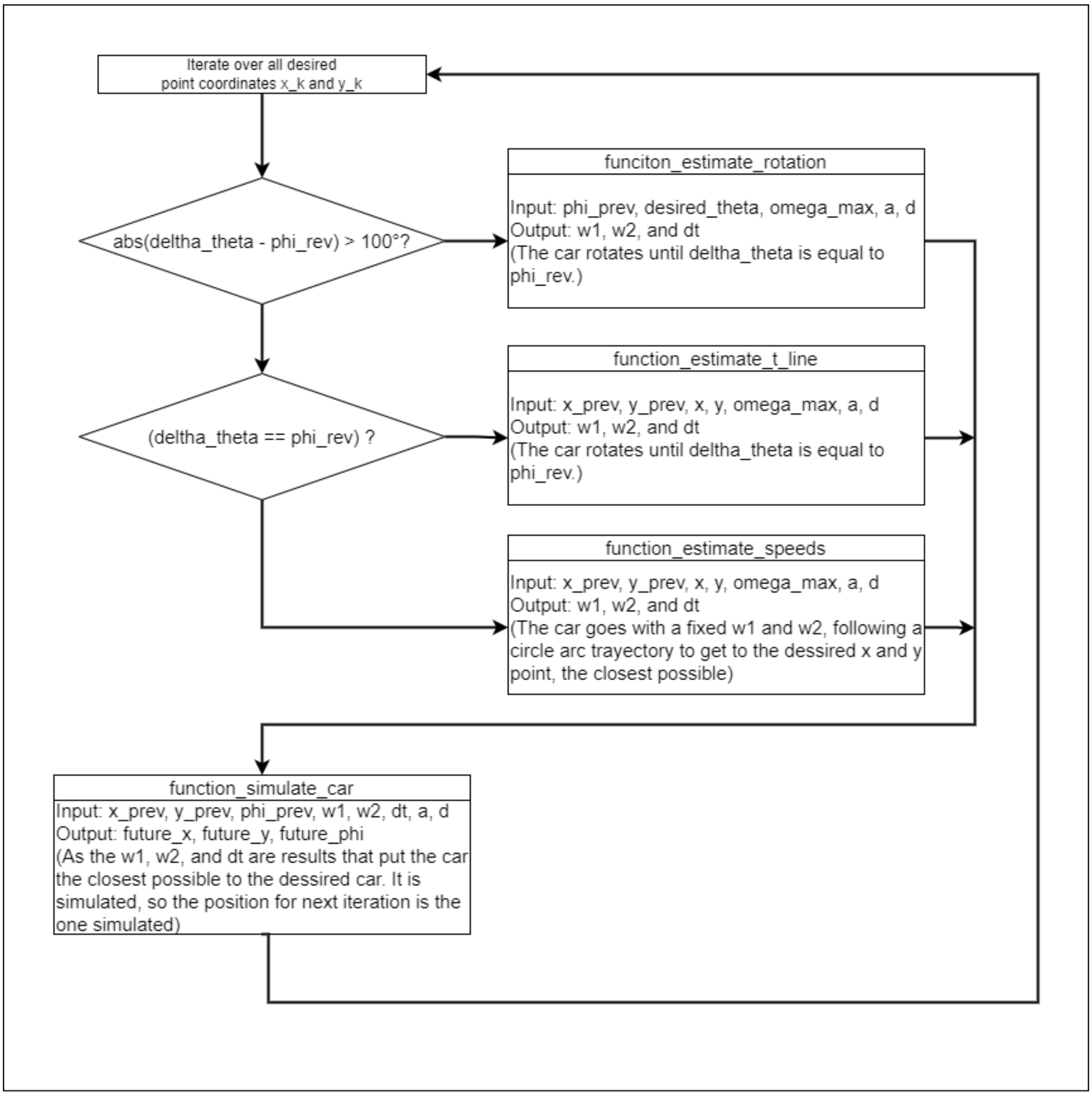


Figure 11. General main script flow

3.3.2.3. Function_estimate_rotation

Based on the kinematics, it gives the wheels rotation velocities and a period in which the car should move an achieve the desired_theta. In the matlab script, the desired_theta is set as deltha_theta plus an offset. The offset might be found on the script as zero.

It uses the input wheel rotation speed omega_max_mov to calculate the other kinematics parameters. As it is rotation, it rotates with one wheel as positive value of the input parameter, and the other wheel with the negative value.

3.3.2.4. Function_estimate_t_line

Based on current position it calculates the wheels rotation speeds and the time so the car achieves the desired position. It uses the linear distance between the two points.

It uses the maximum wheel rotation speed omega_max_mov to calculate the other kinematics parameters. As it is move forward, it gives the wheels speed as the same magnitude as the input parameter.

3.3.2.5. Function_estimate_speeds

About the restricted movement of the car, it has three starting parameters x_prev, y_prev, and phi_prev. It can be only possible to move the car to two parameters as there are two controlling parameters, w1 and w2. Therefore, it is known x_prev, y_prev, and phi_prev. It will be calculated a w1 and a w2 so x and y are achieved; phi is obtained.

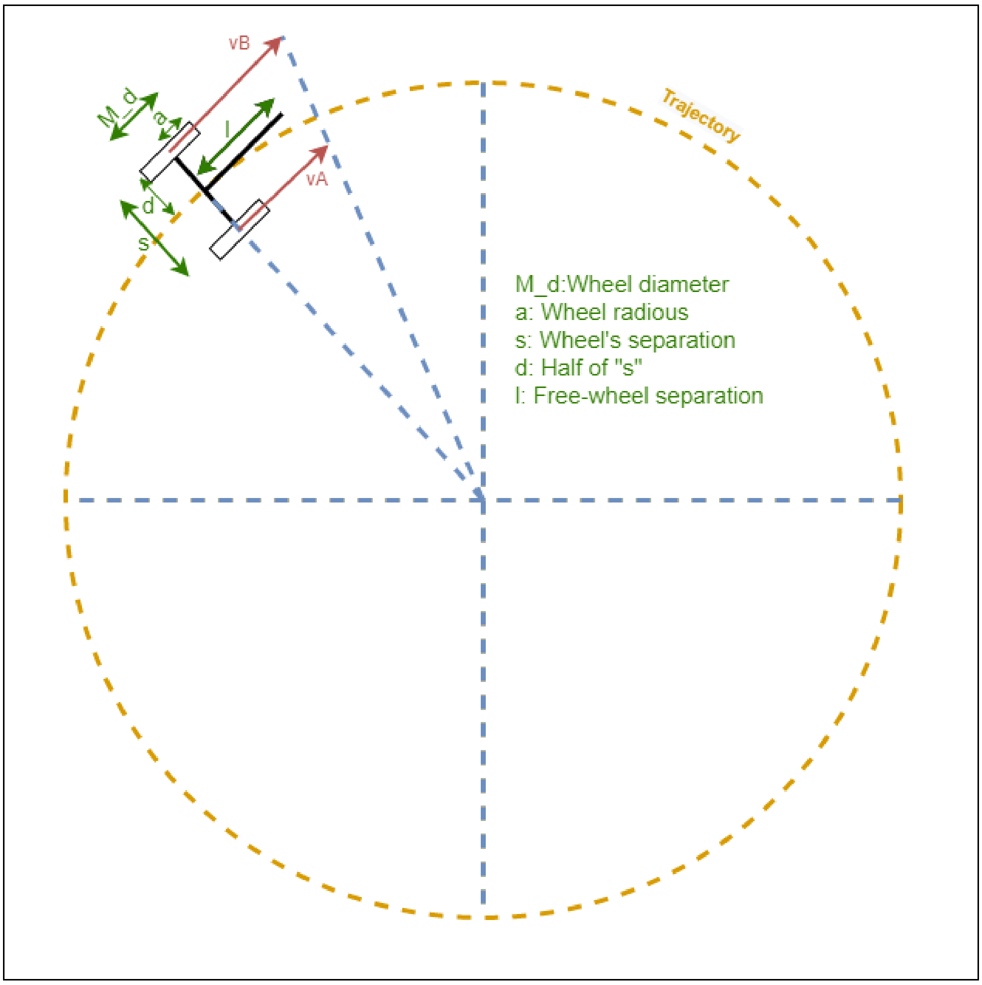


Figure 12. Car kinematics displacement

It uses the input wheel rotation omega_max_mov speed to calculate the other kinematics parameters. In Figure 12, the car rotates clockwise, then wheel B, of linear speed vB and rotation speed wB, takes the value omega_max_mov. If the car needs to go to the anticlockwise, when deltha_theta is more than phi_prev (see Figure 9), wA is fixed.

Then internally, for each 500 values of wA from negative omega_max_mov to positive omega_max_mov, it calculates the time for which it is closer to the desired point (this is the function function_estimate_time_error). Then, it is taken the minimum distance error among all combinations of wA and fixed wB.

It is important to point out that the car not necessarily has a configuration of calculated speeds that makes it get to another point. This is because the numerical calculations for solving the kinematics equation give approximations. Such when wA values are in steps between negative omega_max_mov to positive omega_max_mov, it is split in 500 steps.

3.3.2.6. Function_estimate_time_error

For a given position, x_prev, y_prev, and phi_prev; a desired position x and y; two fixed wheel rotation speeds; an input wheel rotation omega_max_mov speed; and dimensions of the car (for using the kinematic equations). It is iterated 100 times from a time 0 to a max_time (calculated on linear time for reaching desired point multiplied by 1.1 as security factor). In each iteration, it is calculated the distance from the desired point. Then the parameters combination, in which the minimum separation is obtained, are given back such the change in deltha, time, and the error.

It is important to point out that there are two kinematic equations. This simplification is done because “r” is in the denominator. There, some obtained values for “r” tend to be infinite. In this case, it is considered two-wheel speeds have same value and that the car performs a liner movement.

3.3.3. Closed loop position control

The idea of this control is to input a position in x and y axes. Then it should generate two-wheel speed references. First, the equations of the kinematics of the car are shown.

3.3.3.1. Kinematics frame

The Figure 13 shows the frame references used in the modelling of the car. It is shown the parameters that used on the equations.

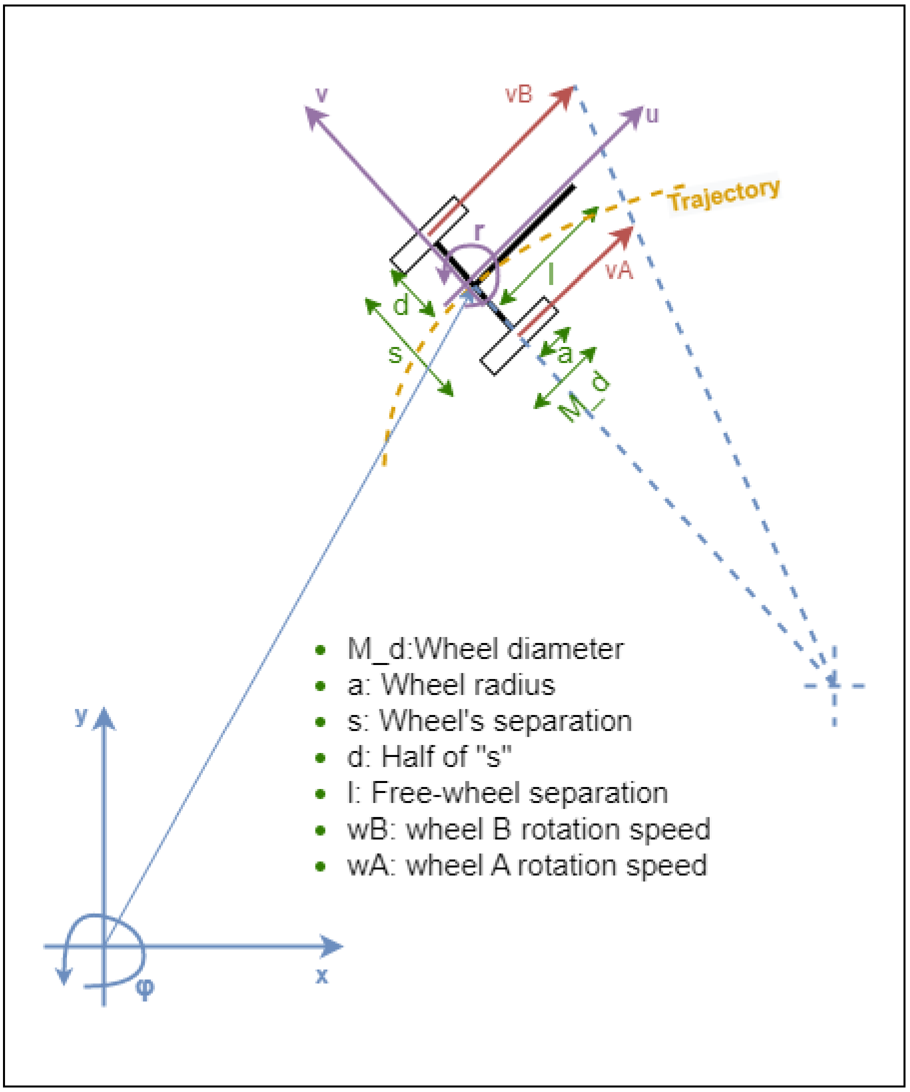


Figure 13. Car kinematic frames

3.3.3.2. Kinematics equations

Given wheel rotation speeds w1 (wB) and w2 (wA), it can be obtained a linear speed u, and a rotation speed r. It is considered there is not lateral gliding, v is 0. Car frame equationes are:

$$u = \left(\frac{a}{2}\right) \times (w_1 + w_2)$$

$$v = 0$$

$$r = \left(-\frac{a}{2d}\right) \times (w_1 - w_2)$$

These coordinates can be converted into an external ground fixed coordinate:

$$\dot{x} = cos\varphi\, u - sin\varphi\, v$$

$$\dot{y} = sin\varphi\, u + cos\varphi\, v$$

$$\dot{\varphi} = r$$

3.3.3.3. Kinematics matrixes

Using the frames seen previously, the following matrixes work for the car:

$$\begin{bmatrix} u \\ v \\ r \end{bmatrix} = \begin{bmatrix} a/2 & a/2 \\ 0 & 0 \\ -a/2d & -a/2d \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} cos\varphi & -sin\varphi & 0 \\ sin\varphi & cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u \\ v \\ r \end{bmatrix}$$

3.3.3.4. Ground frame equations

When $w_1 = w$ and $w_2 = w$; $u = aw$ and $r = 0$, then:

$$\dot{x} = cos\varphi\, u \rightarrow x_{(t)} = cos\varphi_0\, u\, t + x_0$$

$$\dot{y} = sin\varphi\, u \rightarrow y_{(t)} = sin\varphi_0\, u\, t + y_0$$

$$\dot{\varphi} = 0 \rightarrow \varphi_{(t)} = \varphi_0$$

When $w_1 = -w$ and $w_2 = w$; $u = 0$ and $r = \frac{w\,a}{d}$, then:

$$\dot{x} = 0 \rightarrow x_{(t)} = x_0$$

$$\dot{y} = 0 \rightarrow y_{(t)} = y_0$$

$$\dot{\varphi} = r \rightarrow \varphi_{(t)} = r\, t + \varphi_0$$

When w_1, w_2; $u = \left(\frac{a}{2}\right)(w_1 + w_2)$ and $r = \left(-\frac{a}{2d}\right)(w_1 - w_2)$, then:

$$\dot{x} = cos\varphi\, u \rightarrow x_{(t)} = sin\varphi_{(t)} \left(\frac{u}{r}\right) + x_0 - sin\varphi_0 \left(\frac{u}{r}\right)$$

$$\dot{y} = sin\varphi\, u \rightarrow y_{(t)} = -cos\varphi_{(t)} \left(\frac{u}{r}\right) + y_0 + cos\varphi_0 \left(\frac{u}{r}\right)$$

$$\dot{\varphi} = r \rightarrow \varphi_{(t)} = r\,t + \varphi_0$$

3.3.3.5. *Simulink model*

The equations were represented in a Simulink model. It is found the Control_car_position.slx in folder Control_car_position.

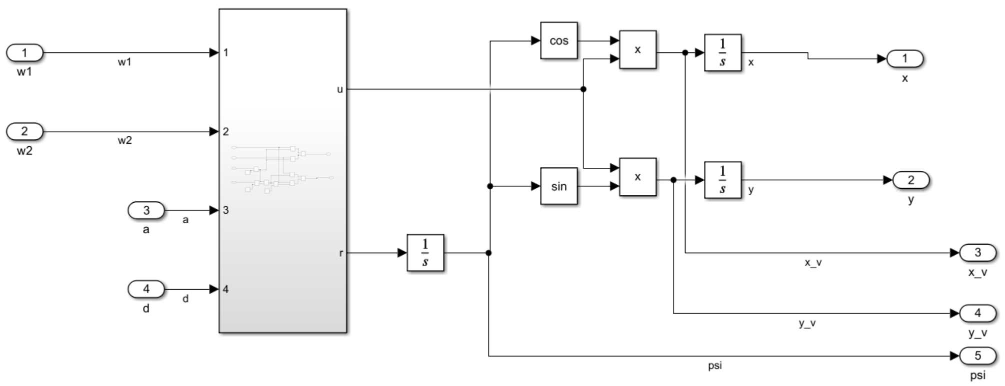
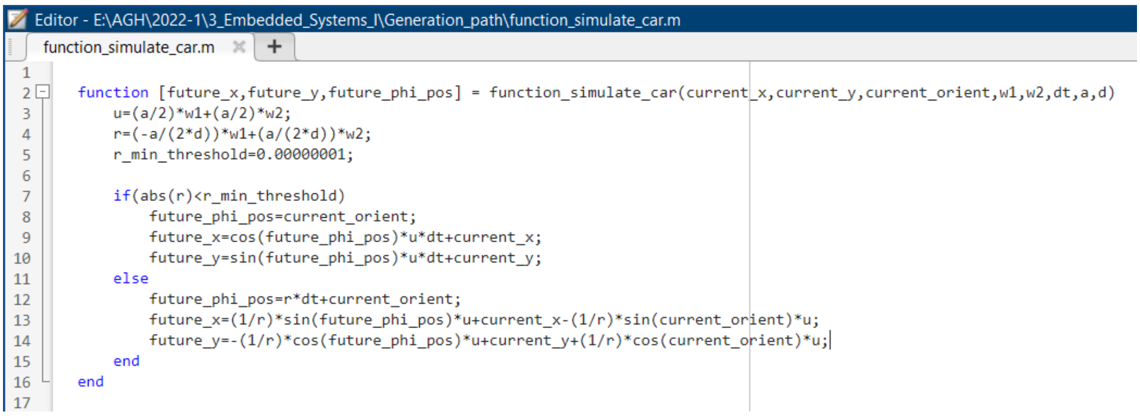


Figure 14. Simulink model

3.3.3.6. *Matlab script model*

The equations were represented in the script function_simulate_car.m in the folder Generation_path. When r is zero or insignificant, less than 0.00000001, it is considered to go straight. This is because limitations in the calculations while dividing by small values.



Editor - E:\AGH\2022-1\3_Embedded_Systems_I\Generation_path\function_simulate_car.m

function_simulate_car.m

```
function [future_x,future_y,future_phi_pos] = function_simulate_car(current_x,current_y,current_orient,w1,w2,dt,a,d)
    u=(a/2)*w1+(a/2)*w2;
    r=(-a/(2*d))*w1+(a/(2*d))*w2;
    r_min_threshold=0.00000001;

    if(abs(r)<r_min_threshold)
        future_phi_pos=current_orient;
        future_x=cos(future_phi_pos)*u*dt+current_x;
        future_y=sin(future_phi_pos)*u*dt+current_y;
    else
        future_phi_pos=r*dt+current_orient;
        future_x=(1/r)*sin(future_phi_pos)*u+current_x-(1/r)*sin(current_orient)*u;
        future_y=-(1/r)*cos(future_phi_pos)*u+current_y+(1/r)*cos(current_orient)*u;
    end
end
```

Figure 15. Matlab script model

3.3.3.7. *Results*

This control was tested, and it was working as expected. The car follows the expected road shapes. As there was no way for sensing position, results cannot be shown through plots. Due to slipping in the speed of the wheels, it won't follow exactly the expected path.

3.3.3.8. Position control simulink

3.3.3.8.1. Transform references

In Figure 16, it is shown the kinematic diagram for the closed loop control. The idea od the control is to have error_d and error_h as 0 when there is a point reference.

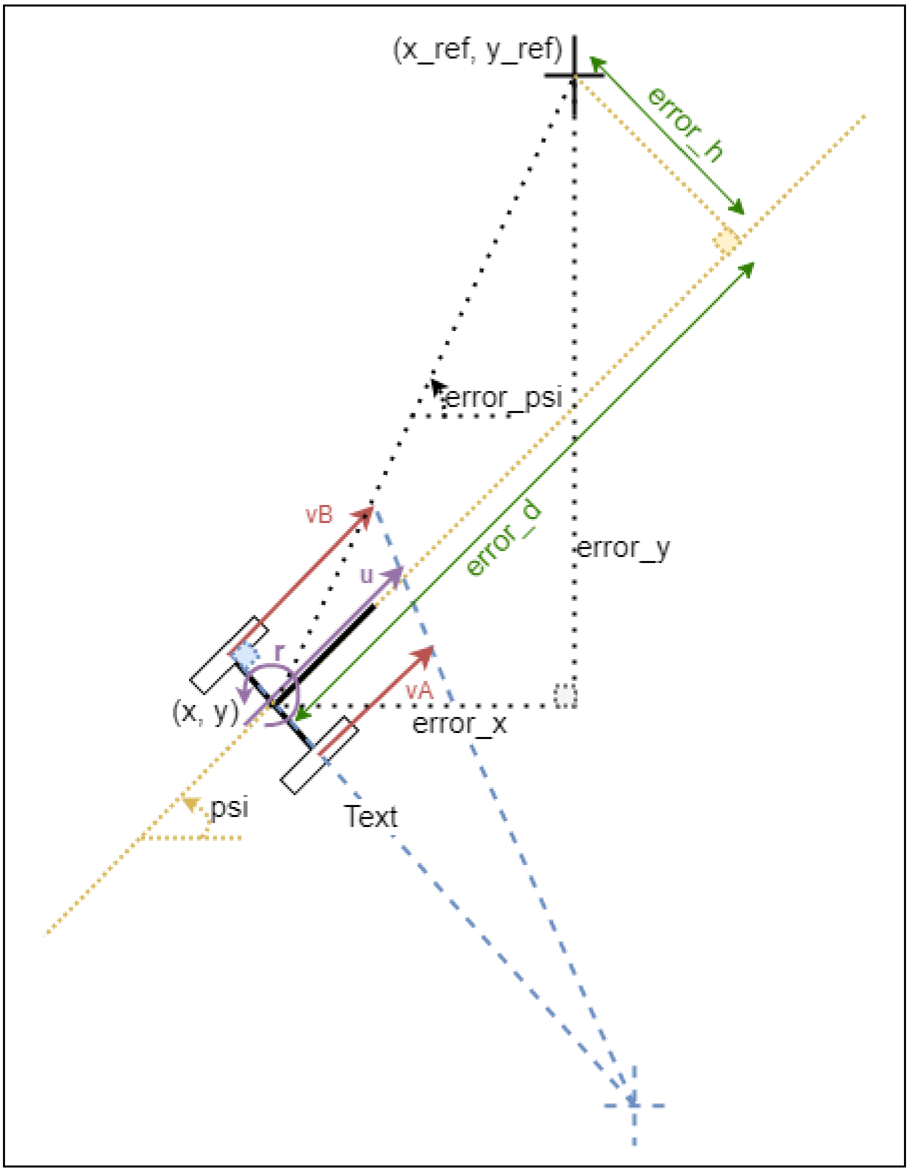


Figure 16. Closed loop control kinematic diagram

These values are obtained through these equations based on error_y, error_x, and psi:

$$error\_psi = \arctan\left(\frac{error\_y}{error\_x}\right)$$

$$error\_h = sin(error\_psi - psi) * \sqrt{|error\_x|^2 + |error\_y|^2}$$

$$error\_d = cos(error\_psi - psi) * \sqrt{|error\_x|^2 + |error\_y|^2}$$

A Simulink block was prepared for this operation (see Figure 17Figure 18).

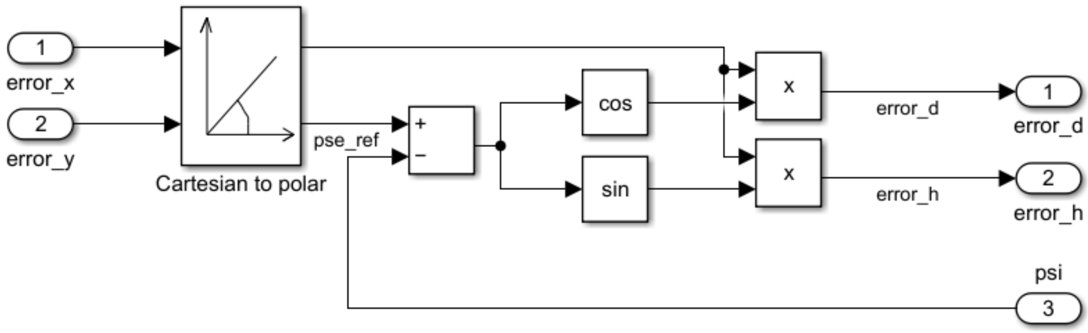


Figure 17. Simulink block for transforming references

3.3.3.8.2. Mixing control signals

The plant that is meant to be controlled with w1 and w2 wheel motor speeds. However, because of complexity of the plant, these are obtained from two control signals, one of them is a forward signal, and the other is a turn signal. The equations for this is:

$$w1 = u\_forward - u\_turn$$

$$w2 = u\_forward + u\_turn$$

The car would do a pure rolling movement when u_forward is 0 and u_turn has some value. When u_turn is 0, the car would move in straight line.

A Simulink block was prepared for this operation (see Figure 18).

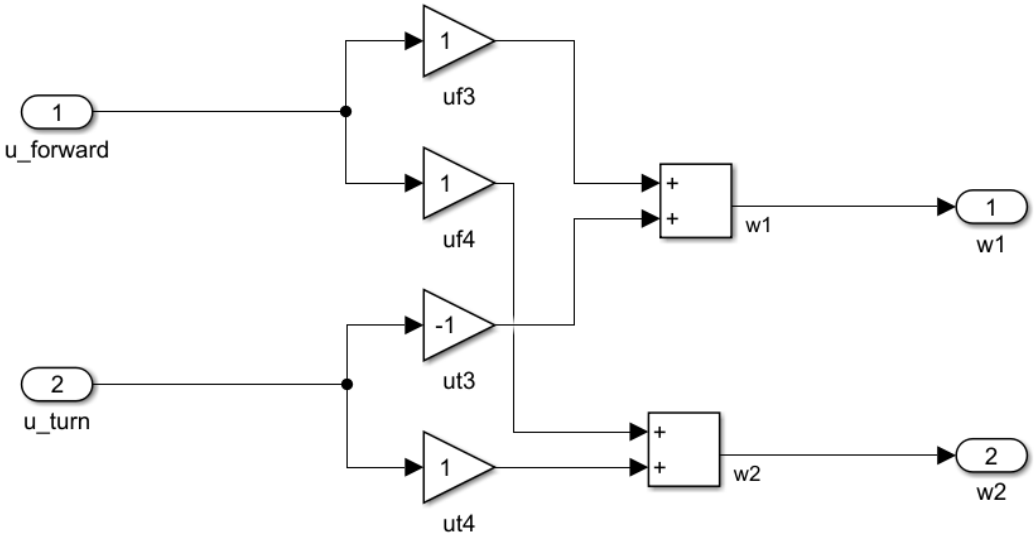


Figure 18. Mixing control signals forward and turning

3.3.3.8.3. Integration

The control strategy was to control only the error_d all the time. While the error_h is activated when it is over 0.0001 meters. The plant oscillates indefinitely when this second control is not connected, this is due to the non-linearity of the plant and the fact that there's only two control signals for controlling three states, position x, position y and orientation psi. All states are not controllable at the same time.

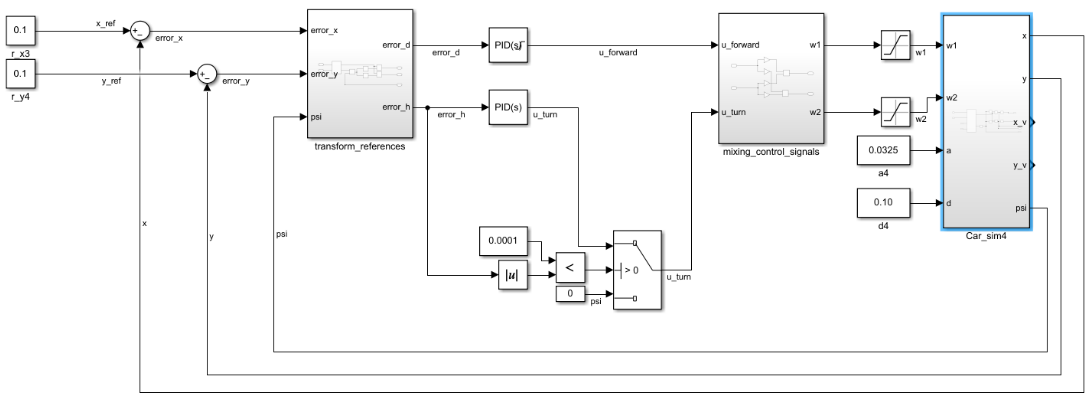


Figure 19. Position control closed loop simulation

3.3.3.8.4. Results

The previous schematic was modified to test if it is working (see Figure 20). It was added a step change in the reference, and the responses are plotted.

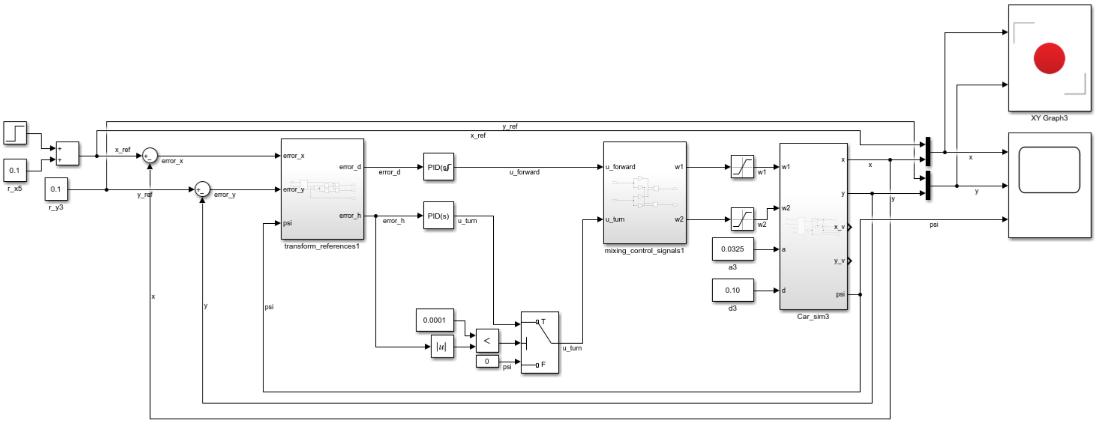


Figure 20. Testing closed loop position control

The plots of references and sensed values are shown in Figure 21. It can be shown that the car starts in position x=0 and y=0; then the reference is x=0.1 and y = 0.1; later, only x reference is changed to 0.6. The orientation gets closer to 1 rad, which is closer to 45°, as the point of reference is at 45° from the starting point.

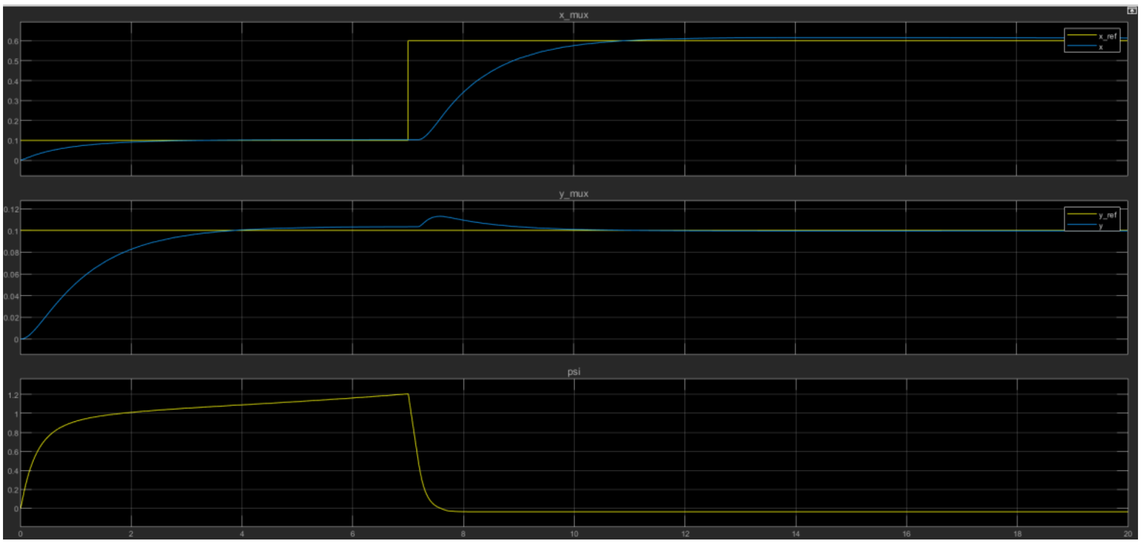


Figure 21. Plots of testing closed loop position control

In Figure 22, it is observed the path the car would follow when it has two different reference points from a starting point x=0 and y=0. When it has the update of the second reference point, it can be observed in Figure 21 that the car reorientates it goes to the right, just as in the starting point.

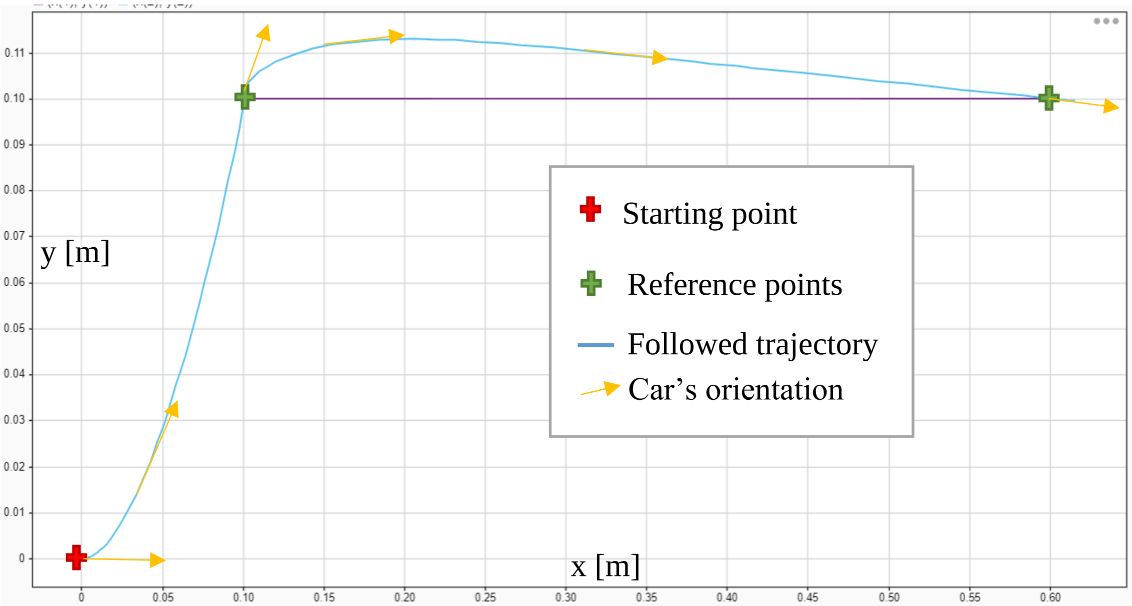


Figure 22. y vs. x position plotting in closed loop control

3.4. Commands protocol

The commands sent to the car by Bluetooth are text, which can be understood as array of ASCII characters that start with "$" and end with ";", the values sent are divided by ":". It has three sections mainly, see Figure 23.

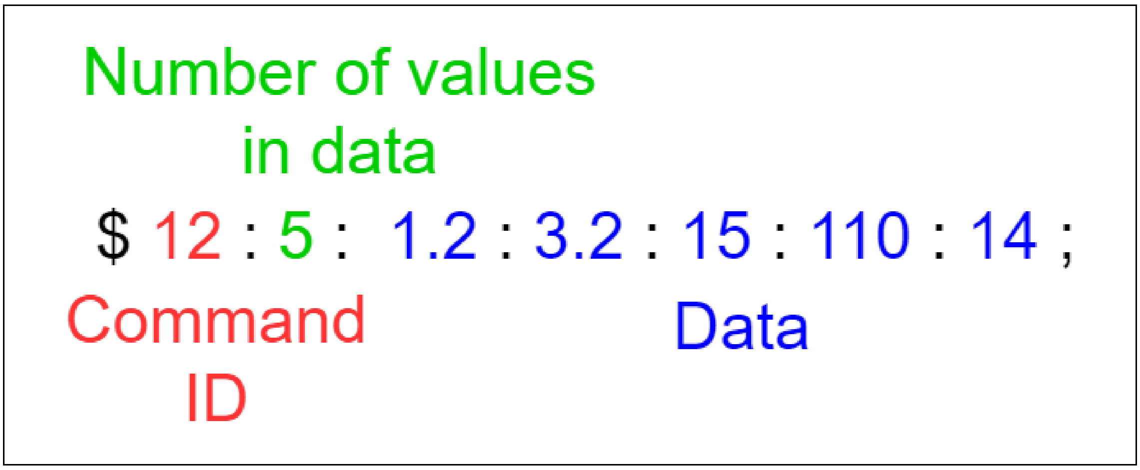


Figure 23. Commands protocol

The commands that do not need data can be only Command ID such "$12;".

3.4.1. Car commands

The buttons in this section send commands described in this section. Consider that all units of rotational wheel speed is in RPS.

- Stop: "$0;", set the rotational wheel speeds as 0.

- Line: “$1:1:0.5;”, the data sent is the wheel speed, in this case, 0.5 RPS.

- Circle: “$2:2:30:70;”, 30 is in cm and it is the radius of the circle to be drawn by the card. 70 is the percentage of speed the car can follow, the maximum is 100, it uses the omega_max_mov as maximum constrain in one wheel of the car.

- Enable plotting: “$3:1:1;”, data 1 enables the plotting, while 0 turn it off. The plotting enabled is interpretated by Arduino IDE. The data send back though the terminal has this structure (without spaces): “wA /t wB /t ref_A /t ref_B”.

Figure 24. Example of plotting in arudino IDE.

- Load left motor: “$4:4:1.5:2:1.5:2;”, it updates the vector of references for the left motor. In this case, 4 points of reference. Values are in RPS.

- Load right motor: “$5:4:2:1.5:2:1.5;”, it updates the vector of references for the right motor. In this case, 4 points of reference. Values are in RPS.

- Load time motor: “$6:4:1000:500:2000:500;”, it updates the vector of time, values are meant to be in milliseconds.

- Start track: “$7;”, starts the iteration over the vectors uploaded with commands 4, 5, and 6. It takes one value of each speed vector, and use it as reference for the motors control speed, during a time taken from the time vector.

- Print track param: “$8;”, it prints the values of the vectors in the right and left motor speed references vector and time vector.

- Motor speeds: “$9:2:1.5:2;”, it sets the first data value as the motor A (right motor) reference speed in RPS, while the second value is for the motor B (left motor).

- Ping: “$10;”, sends back “$OK;” as positive response.

- Enable PID: “$11:1:1;”, data 1 enables the PID control loop, while 0 turn it off.

- Sets MA and MB voltages: “$12:2:5:5;”, it sets the first data value as the motor A (right motor) voltage in V, while the second value is for the motor B (left motor).

- Record step response for MA and MB: “$13:2:5:5;”, samples the step responses for the motors for the voltages set in data. First value for MA, and second for MB. The values are recorded in a vector of 100 elements.

- Record MA and MB: “$14;”, starts the recording of the motor’s speeds in a vector of 100 elements (it might be changed on the code).

- Print recorded values: “$15;”, sends back the recorded values from the motors.

- Enable ECHO commands: “$16:1:1;”, data 1 enables an echo response, while 0 turn it off. This means that the car sends back the received command as an acknowledge.

3.5. GUI

In order to interact with the car and its different functions. The Car Control Interface was implemented with use of Matlab App Designer. It is found in the Generation_path folder as “gui_draw_path.mlapp”, see Figure 10.

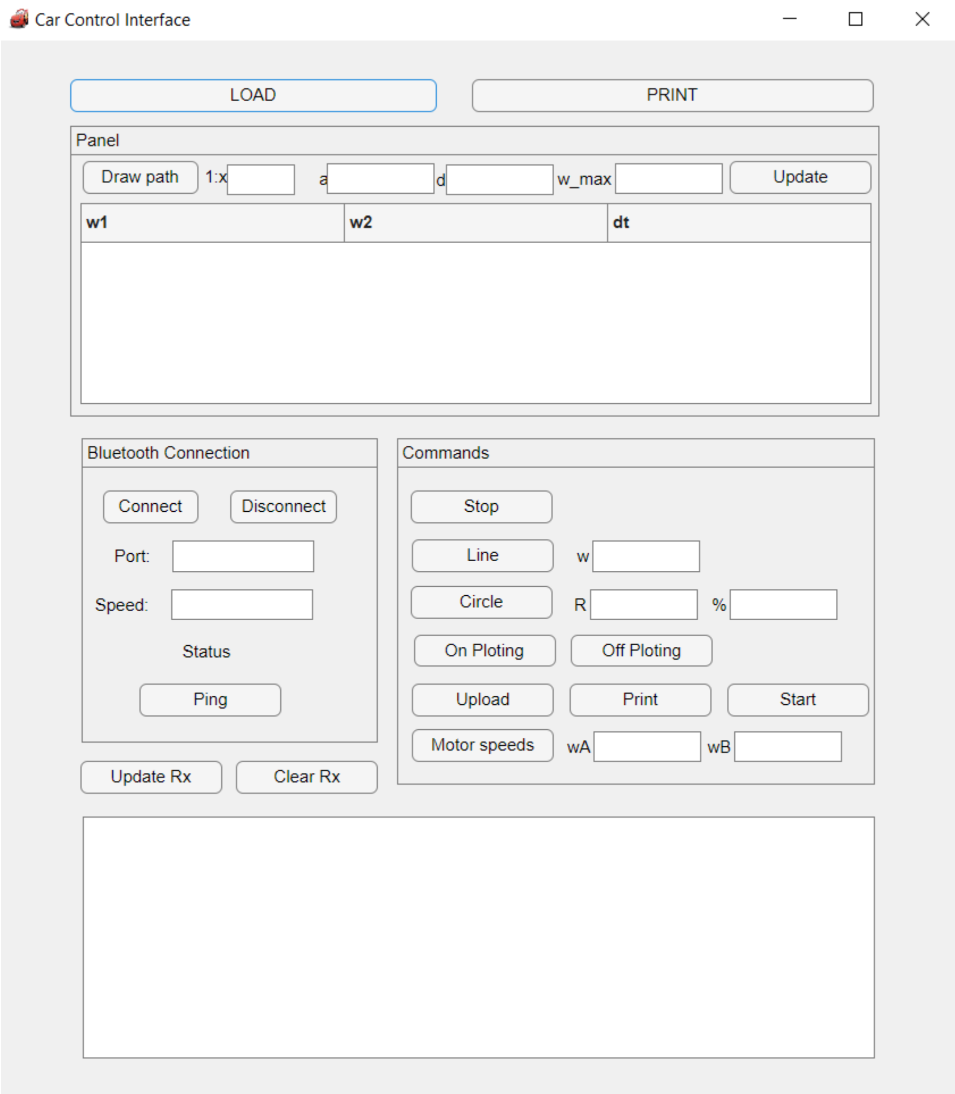


Figure 25. Car Control Interface (GUI)

3.5.1. Generate route references

The buttons related to this section do the following actions:

- LOAD: loading default parameters to the interface, such x, a, w_max, among others.
- PRINT: updating the loaded GUI values.

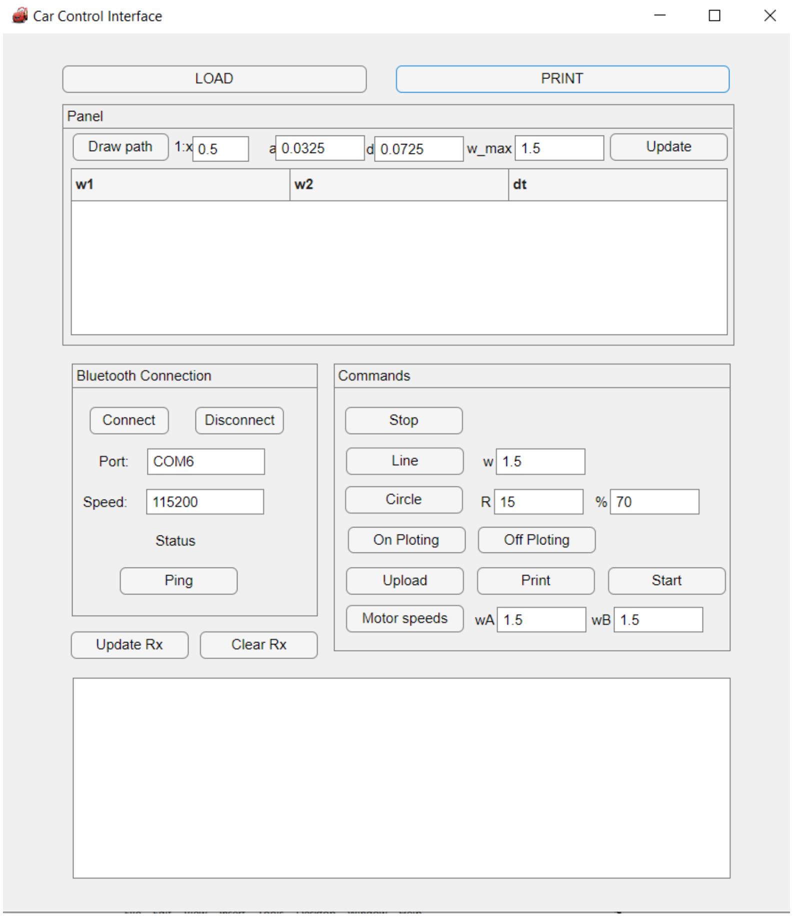


Figure 26. Car Control Interface loaded with default data

- Draw path: Opens a window, in which the path to follow is input (see Figure 27). The maximum value of the new window is given by "x", it is in meters. Then after finishing the clicking, and pressing enter, the program calculates the references and plot the clicked points with the expected path (see Figure 27).

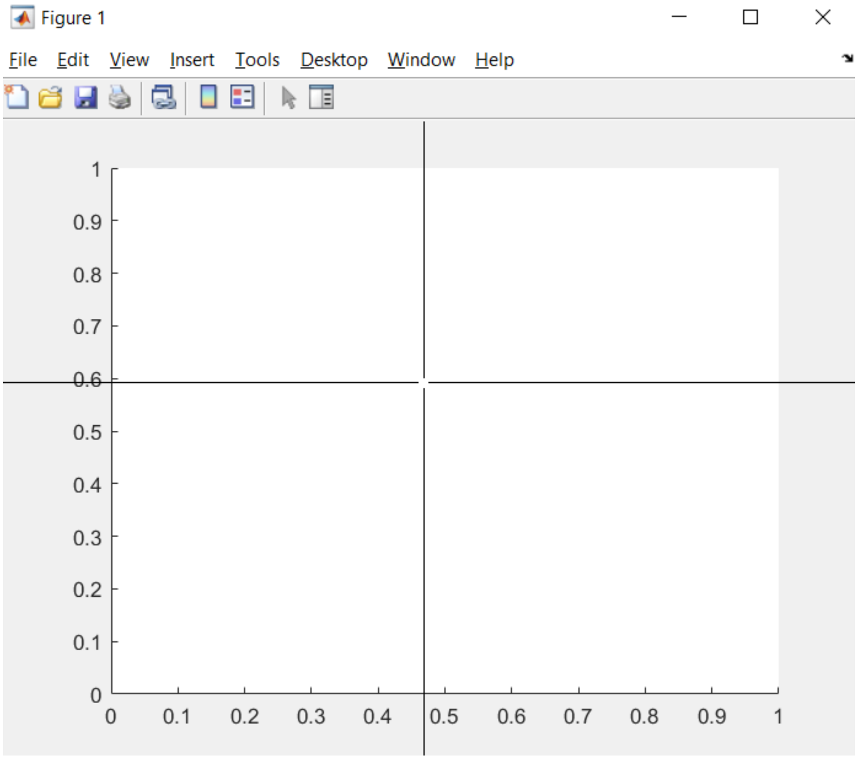


Figure 27. Screen for input of desired points.

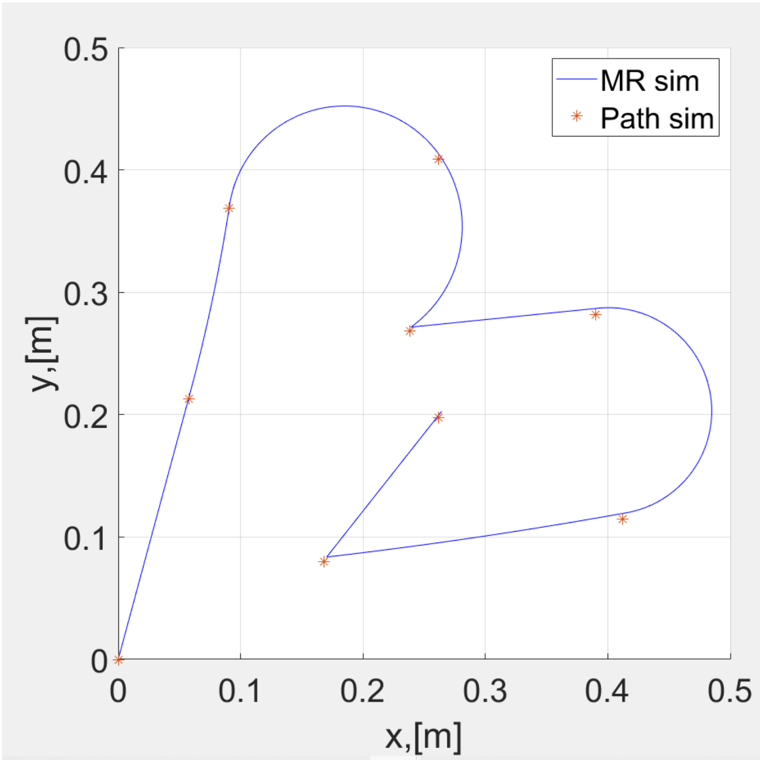


Figure 28. Screen with desired points and expect path

- Update: Updates the values obtained from "Draw path" into the environment and the table that has w1, w2, and dt as columns (see Figure 29). These are the vectors uploaded into the car.

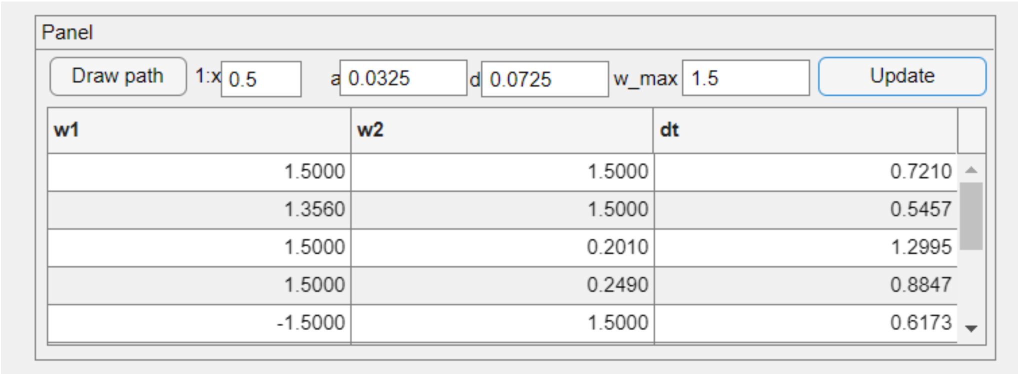


Figure 29. Table with generated parameter for path following

3.5.2. Communication

The buttons related to this section do the following actions:

- Connect: Connect via Bluetooth.
- Disconnect: Disconnect the Bluetooth communication.
- Ping: Sends "$10;" and expects a "$OK;" as positive response.
- Update Rx: Reads the serial buffer and updates the text box below it.
- Clear Rx: Clears the serial buffer.

3.5.3. Car movement control

The following buttons send commands, explained in section 3.4.1 Car commands.

- Stop: "$0;"
- Line: "$1:1:w;"
- Circle: "$2:2:R:%;"
- On plotting: "$3:1:1;"
- Off plotting: "$3:1:0;"

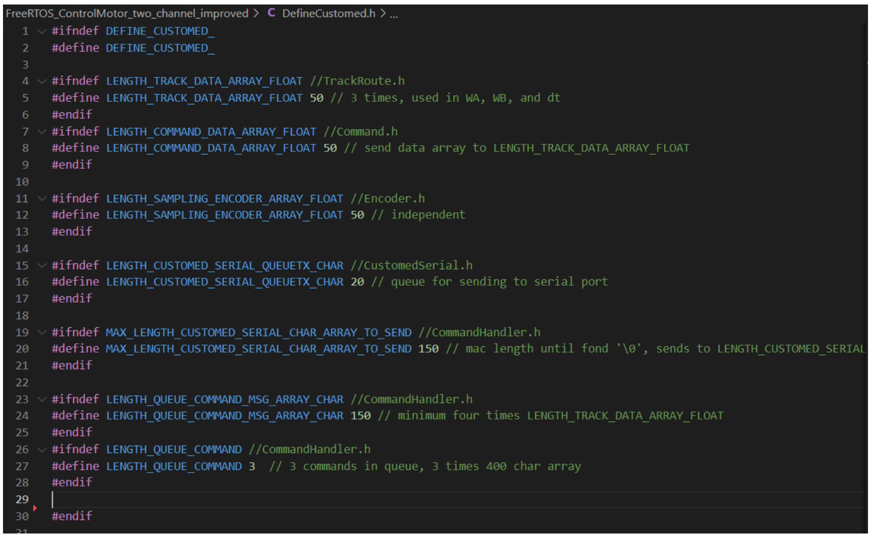
- Upload: “$4:n_data:data;” , “$5:n_data:data;” and “$6:n_data:data;”
- Print: “$8;”
- Start: “$7;”
- Motors speeds: “$12:2:wA:wB;”

3.6. Embedded software implementation

The name of the project is FreeRTOS_ControlMotor_two_channel_improved (see Figure 31). The code was upgraded: Parametrizing variables; creating classes for a better organization; and using FreeRTOS, a Real Time Operating System platform for embedded systems.

3.6.1. Parametrizing variables

and putting all of them in the DefineCustemd.h file (see Figure 30).


FreeRTOS_ControlMotor_two_channel_improved > C DefineCustomed.h > ...

```
#ifndef DEFINE_CUSTOMED_
#define DEFINE_CUSTOMED_

#ifndef LENGTH_TRACK_DATA_ARRAY_FLOAT //TrackRoute.h
#define LENGTH_TRACK_DATA_ARRAY_FLOAT 50 // 3 times, used in WA, WB, and dt
#endif
#ifndef LENGTH_COMMAND_DATA_ARRAY_FLOAT //Command.h
#define LENGTH_COMMAND_DATA_ARRAY_FLOAT 50 // send data array to LENGTH_TRACK_DATA_ARRAY_FLOAT
#endif

#ifndef LENGTH_SAMPLING_ENCODER_ARRAY_FLOAT //Encoder.h
#define LENGTH_SAMPLING_ENCODER_ARRAY_FLOAT 50 // independent
#endif

#ifndef LENGTH_CUSTOMED_SERIAL_QUEUETX_CHAR //CustomedSerial.h
#define LENGTH_CUSTOMED_SERIAL_QUEUETX_CHAR 20 // queue for sending to serial port
#endif

#ifndef MAX_LENGTH_CUSTOMED_SERIAL_CHAR_ARRAY_TO_SEND //CommandHandler.h
#define MAX_LENGTH_CUSTOMED_SERIAL_CHAR_ARRAY_TO_SEND 150 // mac length until fond '\0', sends to LENGTH_CUSTOMED_SERIAL
#endif

#ifndef LENGTH_QUEUE_COMMAND_MSG_ARRAY_CHAR //CommandHandler.h
#define LENGTH_QUEUE_COMMAND_MSG_ARRAY_CHAR 150 // minimum four times LENGTH_TRACK_DATA_ARRAY_FLOAT
#endif
#ifndef LENGTH_QUEUE_COMMAND //CommandHandler.h
#define LENGTH_QUEUE_COMMAND 3  // 3 commands in queue, 3 times 400 char array
#endif

#endif
```

Figure 30. DefinedCustomed.h file

3.6.2. Classes

Splitting the project into header files according to the classes (see Figure 31).

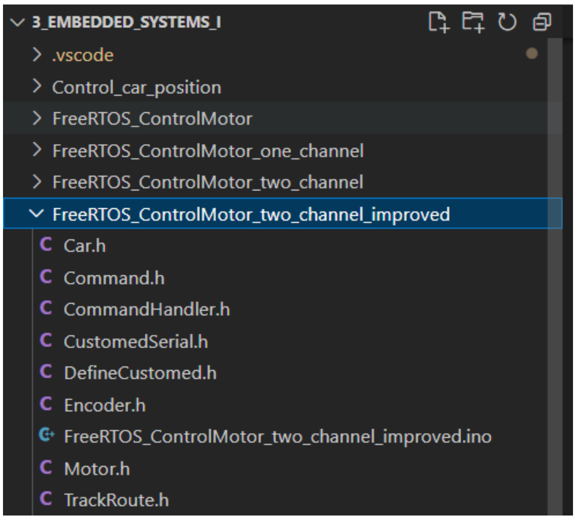


Figure 31. Upgraded 32roject location

3.6.2.1. Car

It contains the physical features of the car (see Figure 32).

```
class Car{
    private:
        float s;
        float M_d;
        float M_vel_min;
        float M_vel_max;
    public:
        Car(float s, float M_d, float M_vel_min, float M_vel_max);
        float get_S();
        float get_M_d();
        float get_M_vel_min();
        float get_M_vel_max();
};
```

Figure 32. Car class.

3.6.2.2. Command

It describes how a command is, it has a command_id, number_data, and a pointer to an array of float values that are the data. Review section 3.4 Commands protocol for understanding the protocol of the commands.

```
class Command{
    public:
        int command_id;
        int number_data;
        float *data_float_array;
        Command();
};
```

Figure 33. Command class

3.6.2.3. CommandHandler

This class has the attributes required for handling the commands in the system. The handleCommand() checks if there is a new command in the queue_commands_serial. There is a function to convert the string command into a command object. When it is used, the command obtained is stored in the command object attribute.

```
class CommandHandler{
    public:
        CommandHandler(Encoder* e_A, Encoder* e_B, Motor* ma, Motor* mb, TrackRoute* tr);
        void handleCommand();
        void receiveCommandString();
        void convertCommandString2Command(uint8_t* message_pointer, Command* command_to_write);
        void printReceievedCommand(Command* c);
        void InterpretateCommand(Command* c);
    private:
        char rxBuffer;
        uint8_t cnt_rx;
        uint8_t state_rx;
        queue_commandRx queueCommandRx;
        QueueHandle_t queue_commands_serial;
        char * stringRecievedMessage;
        char * command_string;
        char * number_data_string;
        Command command;
        bool enable_echo_command;
        bool enable_plot;
        bool enable_PID_A;
        bool enable_PID_B;
        Encoder* encoder_MA;
        Encoder* encoder_MB;
        Motor* MA;
        Motor* MB;
        TrackRoute* trackRoute_pointer;
};
```

Figure 34. CommandHandler class

3.6.2.3.1. Interpretate command

It receives a command object and execute it. In Figure 35, it can be shown how it proceeds according to the 3.4 Commands protocol.

The commands that must be executed by the TrackRoute object are set through adding commands to its queue of commands. The parameters are sent writing into its attributes by its methods.

Additionally, it is seen how the motor A wheel rotation speed reference of the trackRoute_pointer receives the array of data of the command, see the command id 4 in the Figure 35. It is done similarly done for wB speeds and time array dt.

```
void CommandHandler::InterpretateCommand(Command* c){
  int id = c->command_id;
  if ( id == 0){ //  STOP MOTORS //  $0; stop motors

    trackRoute_pointer->sendMotorCommandQueue(STOP);

  }else if ( id == 1){ //  LINE  //  $1:1:0.5; // go straight with 0.5 of velocity

    trackRoute_pointer->setTransferVar1(c->data_float_array[0]);
    trackRoute_pointer->sendMotorCommandQueue(LINE);

  }else if ( id == 2){ //  CIRCLE  //  $2:2:30:70; // make a cricle of 30cm with 50% of speed

    trackRoute_pointer->setTransferVar1(c->data_float_array[0]);
    trackRoute_pointer->setTransferVar2(c->data_float_array[1]);
    trackRoute_pointer->sendMotorCommandQueue(CIRCLE);

  }else if ( id == 3){ // EN_PLOTING //  $3:1:1; //enable ploting

    if(c->data_float_array[0] != 0.0) {
      enablePlot();
    }else{
      disablePlot();
    }

  }else if ( id == 4){ //  LOAD_LEFT_MOTOR //  $4:4:1.5:2:1.5:2; //left motor

    trackRoute_pointer->sendMotorCommandQueue(STOP);
    trackRoute_pointer->set_number_data_motion_WA(c->number_data);
    trackRoute_pointer->set_wA_array(c->data_float_array, c->number_data);

  }else if ( id == 5){ //  LOAD_RIGHT_MOTOR  //  $5:4:2:1.5:2:1.5; //right motor
```

Figure 35. Interpretate command object (first 3 commands)

3.6.2.4. *CustomedSerial*

This class has attributes and methods for managing the reception and emission of bytes.

```
class CustomedSerial{
    private:
        char tx_buffer_recv;
        QueueHandle_t queue_rx_serial; // BUG: Commenting this lines makes de freertos to crash aparently
        QueueHandle_t queue_tx_serial;
    public:
        CustomedSerial();
        void mainTask();
        void printNumber(unsigned long n, uint8_t base);
        void printNumber(unsigned long n);
        void printFloat(double number, uint8_t digits);
        void printFloat(double number);
```

Figure 36. CustomedSerial class

This class was implemented in order to give the code independency from Arduino libraries. The serial methods for reading and writing bytes to serial from Arduino are only used once in specific methods from the CustomedSerial class. The explanation is giving the followed sections.

3.6.2.4.1. *Emiting bytes*

The function write sends a byte to the queue_tx_serial.

```
//receiving chars from user
void CustomedSerial::write(char rx){
    char temp_char= rx;
    xQueueSend(queue_tx_serial, &temp_char, portMAX_DELAY);
}
```

Figure 37. CustomedSerial::write

Then, as the mainTask is executed constantly in the CharCommunication_mainTask_thread, it checks if there's a new byte in the queue_tx_serial and sends it through the Serial2 class. This can be overwrite by other method that sends bytes through the serial peripheral in any other platform.

```
void CustomedSerial::mainTask(){
    //send to the exterior
    if(xQueueReceive(queue_tx_serial, &tx_buffer_recv, 0) == pdPASS ){
        Serial2.print(tx_buffer_recv);
    }
}
```

Figure 38. CustomedSerial::mainTask

3.6.2.4.2. Receiving bytes

These is done using two methods, they are basically the same as Serial object from Arduino. The available() method tells if there is a new incoming byte, if so, the read() method should be call so the byte is read.

```
bool CustomedSerial::available(){
    if(Serial2.available()){
     return true;
    }else{
     return false;
    }
}
```

Figure 39. CustomedSerial::available()

```
char CustomedSerial::read(){
    char r = Serial2.read();
    return r;
}
```

Figure 40. CustomedSerial::read()

3.6.2.5. Encoder

This class contains the attributes and methods that enable obtaining the speed in RPS of the motors. See Figure 41.

```
class Encoder{
    private:
        char * tag;
        int direction_enc;
        int pinEnc_1channel;
        int pinEnc_2channel;
        float timeEncDiference;
        unsigned long timeEncNow;
        unsigned long timeEncBef;
        float last_sensed_speed;

        float vector_sampling[LENGTH_SAMPLING_ENCODER_ARRAY_FLOAT];
        float size_of_sampling;
        int counter_sampling;
        int correction_factor_direction;
    public:
        Encoder(int pinEnc_1channel,int pinEnc_2channel, char * tag, int correction_factor_direction);
        int getPinOfInterrupt();
        void encoderFunction();
        float sense_speed();

        void startSampling();
        void sample();
        void plotSampled();
};
```

Figure 41. Encoder class.

It is initialized with the pins of the two channels of the encoder. The encoderFunction() is the method that is called from the interruptions connected to the first channel. The attribute timeEncDiference is the one that contains indirectly the speed of the motor. It is updated in every interruption.

The method sense_speed converts the timeEncDiference into speed in RPS.

3.6.2.6. Motor

This class has the attributes and methods required for controlling voltage and rotation speed of one motor.

```
class Motor {
    private:
        float d_A11 ;   float d_A12 ;   float d_A21 ;  float d_A22 ;
        float d_B11 ;   float d_B21 ;
        float K_11 ;    float K_12 ;    float K_13 ;   float L_11 ;    float L_21 ;
        float dt;

        float Vm;
        int pin_enable;
        int pin_pwm;

        float speed; float u; float u_saturated; float error; float integral_error;
        float acc_bef; float speed_bef; int is_equal; int sign_equal; int is_saturated;

        float counter_sampling;
    public:
        Motor(  float d_A11 ,float d_A12 , float d_A21 , float d_A22 ,
                float d_B11 , float d_B21 ,
                float K_11 , float K_12 ,  float K_13 ,  float L_11 ,  float L_21 , float dt,
                float Vm, int pin_enable, int pin_pwm
                );
        void controlSpeed(float speed_ref, float speed_sensed);
        void motorVoltaje(float voltaje);
        void motorVoltaje(int pinEnable, int  pinPWM, float  voltaje, float Vm);
        void plotMotorParameters(float speed_MA, float speed_MB, float ref_MA, float ref_MB);
        int getIsSaturated();
        float getError();
        float getU();

};
```

Figure 42. Motor class.

3.6.2.7. TrackRoute

This class contains attributes and methods for updating the references for the PID control of the motors. It can be seen there are three arrays, for wA, wB, and dt.

```
class TrackRoute{
    public:
        TrackRoute();
        float get_wA();
        float get_wB();
        void mainExecution(Car* car_pointer);
        void draw_line_constant_velocity(float variable);
        void actualizarReferenciasCirculo(float rt, float circle_speed, Car* car);
        void sendMotorCommandQueue(int command);
    private:
        float MA_w_ref;
        float MB_w_ref;
        bool first_execute_motion;
        int motion_command_state;
        unsigned long nowTimeMs;
        unsigned long previousTimeMs;

        float  wA_float_array[LENGTH_TRACK_DATA_ARRAY_FLOAT]; //set externally
        int wA_array_length;
        float  wB_float_array[LENGTH_TRACK_DATA_ARRAY_FLOAT]; //set externally
        int wB_array_length;
        float  dt_float_array[LENGTH_TRACK_DATA_ARRAY_FLOAT]; //set externally
        int dt_array_length;

        int counter_motion_data;

        float transfer_var_1; //set externally
        float transfer_var_2; //set externally

        QueueHandle_t queue_commands_motor;
};
```

Figure 43. TrackRoute class

3.6.2.7.1. Main execution

This method is the main one, first it is check if there is a new track command, if so, the motion_command_state is updated. Next, there is logic for following states. The idle state is EXECUTED; after every command is done, the EXECUTED state is set (see Figure 44).

```
void TrackRoute::mainExecution(Car* car_pointer){

    if(xQueueReceive(queue_commands_motor, &motion_command_state, 0) == pdPASS ){
      if(motion_command_state == TRACK){
          first_execute_motion=false;
      }
    }

    switch(motion_command_state){
        case EXECUTED:
          break;
        case STOP:
          MB_w_ref=0;
          MA_w_ref=0;
          first_execute_motion=false;
          motion_command_state=EXECUTED;
          break;
        case LINE:               // $1:1:1.5; // go straight with 1.5 of velocity
          draw_line_constant_velocity(transfer_var_1);
          motion_command_state=EXECUTED;
          break;
        case CIRCLE:
          actualizarReferenciasCirculo(transfer_var_1, transfer_var_2, car_pointer); // $2:2:30,50;
          motion_command_state=EXECUTED;
          break;
```

Figure 44. TrackRoute::mainExecution

3.6.2.7.2. TRACK command

When the TRACK command is set for the first time, the flag first_execute_motion is set to false, and after its execution, it is set to true. It resets the counter_motion_data. The idea in general is to have one MA_w_ref and MB_w_ref until the dt_float_array is achieved, then the counter_motion_data increases in one (see Figure 45).

```
case TRACK:
  if(number_data_motion>0){
    if(first_execute_motion==false){
      previousTimeMs=millis();
      MA_w_ref= wA_float_array[0];
      MB_w_ref= wB_float_array[0];
      counter_motion_data=0;
      first_execute_motion=true;
    }
    nowTimeMs=millis();

    if((nowTimeMs-previousTimeMs)>(dt_float_array[counter_motion_data])){
      counter_motion_data++;
      if(counter_motion_data >= number_data_motion){
        motion_command_state=STOP;
      }else{
        MA_w_ref= wA_float_array[counter_motion_data];
        MB_w_ref= wB_float_array[counter_motion_data];
        previousTimeMs=nowTimeMs;
      }
    }
  }
  break;
```

Figure 45. TRACK command

3.6.3. Threads

The 4 threads executed in FreeRTOS are explained in this section.

```
// FreeRTOS Tasks are defined
xTaskCreate(CharCommunication_mainTask_thread, "Task-1", 512, NULL, 0, NULL);
xTaskCreate(CommandHandler_thread, "Task2", 256, NULL, 0, NULL);
xTaskCreate(ControlMotors_thread, "Task4", 256, NULL, 3 , NULL);
xTaskCreate(TrackRoute_thread, "Task1", 256, NULL, 2, NULL);
xTaskCreate(Plot_thread, "Task3", 256, NULL, 1, NULL);
```

Figure 46. Created tasks in FreeRTOS

3.6.3.1. CharCommunication_mainTask_thread

It is executed constantly, but with a low priority of 0, so it doesn't interfere with other tasks. The communication input and output use this thread for sending values of the queues. customedSerial.mainTask() sends the internal char queue to the "exterior", in this case, the serial2, which is connected to the bluetooth. commandHandler.receiveCommandString() receives char from the "exterior" (serial2), while receiving checks if a command has been detected and send it to a queue of commands.

```
//Rutina manejo de mensajes
void CharCommunication_mainTask_thread(void* pvParameters) {
  while (1) {
    customedSerial.mainTask();
    commandHandler.receiveCommandString();
  }
}
```

Figure 47. CharCommunication_mainTask_thread

3.6.3.2. CommandHandler_thread

This thread is for executing the main function of the commandHandler object. It is checked if there's a new command in the queue of commands every 49 ms and it has the lowest priority, so it does not interfere with other tasks. If so, it executes it.

```
void CommandHandler_thread(void* pvParameters) {
  TickType_t delayTime = *((TickType_t*)pvParameters);
  while (1) {
    commandHandler.handleCommand();
    vTaskDelay(49 / portTICK_PERIOD_MS);
  }
}
```

Figure 48. CommandHandler_thread

3.6.3.3. ControlMotors_thread

This thread is for the control of the motors. According to the current speed, it is calculated the voltage it should have to get to the desired speed. It is executed every 16 ms and it has the highest priority of 3.

```
void ControlMotors_thread(void* pvParameters){
  TickType_t delayTime = *((TickType_t*)pvParameters);
  while (1) {
    float s_A = encoder_MA.sense_speed();
    float r_A = trackRoute.get_wA();
    if(commandHandler.getEnablePIDA()) MA.controlSpeed( r_A, s_A);

    float s_B = encoder_MB.sense_speed();
    float r_B = trackRoute.get_wB();
    if(commandHandler.getEnablePIDB()) MB.controlSpeed( r_B, s_B);
    vTaskDelay(16 / portTICK_PERIOD_MS);
  }
}
```

Figure 49. ControlMotors_thread

3.6.3.4. TrackRoute_thread

This thread is for executing the main function of the trackRoute object. The commands related to the wheel's rotational speed reference are executed such: STOP (setting PID control speeds to 0), LINE (setting PID control speeds to the same fixed value) or START_TRACK (updating speeds after every time in the vector of times for following a track). It is executed every 16 ms and it has a medium priority of 2.

```
void TrackRoute_thread(void* pvParameters){
  TickType_t delayTime = *((TickType_t*)pvParameters);
  while (1) {
    trackRoute.mainExecution(&car);
    vTaskDelay(16 / portTICK_PERIOD_MS);
  }
}
```

Figure 50. TrackRoute_thread

3.6.3.5. Plot_thread

This thread is for plotting the values recorded in compatibility of Arduino IDE plotting interface. It basically sends the speeds and references of the motors. The task is executed every 112 ms and it has a low priority of 1.

```
void Plot_thread(void* pvParameters) {
  TickType_t delayTime = *((TickType_t*)pvParameters);
  while (1) {
    if(commandHandler.getEnablePlot()){
      float s_A = encoder_MA.sense_speed();
      float r_A = trackRoute.get_wA();
      float s_B = encoder_MB.sense_speed();
      float r_B = trackRoute.get_wB();
      //MA.plotMotorParameters(customedSerial, s_A, s_B, r_A, r_B);
      //MB.plotMotorParameters(customedSerial, s_A, s_B, r_A, r_B);
      customedSerial.print(s_A);
      customedSerial.print("\t");
      customedSerial.print(r_A);
      customedSerial.print("\t");
      customedSerial.print(s_B);
      customedSerial.print("\t");
      customedSerial.print(r_B);
      customedSerial.print("\t");
      customedSerial.println("");
    }
    vTaskDelay(112 / portTICK_PERIOD_MS);
  }
}
```

Figure 51. Plot_thread

3.6.4. Problems encountered

- At the beginning, it was planned to use an IMU for obtaining position in x and y of the car. However, it was found that the common libraries for using the

IMU in Arduino platform is not compatible with FreeRTOS. Due to time constraints, it was not implemented.

Conclusions

- The conclusions from the previous stage of this project were taken into consideration, it was used a RTOS platform and encoders with higher resolution of 960 and direction sense.

- It was implemented an upgrade on the software. It was migrated to FreeRTOS and using classes. The classes were writing so it doesn't depend on Arduino libraries, and they can be used with another microcontrollers.

- An open loop position control, or a track generator, was implemented. It uses MATLAB for generating the reference data for the car from desired position references. It worked as expected.

- A simple protocol for interacting with the car was created over serial communication and ASCII, there are codes for specific functions that the car can perform.

- A GUI was implement using MATLAB to interact with the car. It is connected to almost all the protocol codes the car has.

- A closed loop position control was designed and simulated. However, due to time constraints, it was not implemented.

- Future stages of this project can consider implementing the designed closed loop position control. This can be done using indirect states measure technicism such observers or integrate a position and orientation sensors.