People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronic Engineering

Department of Electronics

Final Year Project Report Presented in Partial Fulfilment of the Requirements for the Degree of

MASTER

In **Control**

Option: **Control**

Title:

Safe Navigation of a Differential Drive Mobile Robot using a PID Controller

Presented by:

- **Tarek NEKKACHE**
- **Sofiane HACIANE**

Supervisor:

Dr. O. HACHOUR

Registration Number: ...../2020

ACKNOWLEDGEMENT

First and foremost, all praises and thanks to Allah, the Almighty, the merciful and the most compassion, for His showers of blessings throughout our research work to complete it successfully and throughout our entire lives.

We would like to express our deep and sincere gratitude to our research supervisor, Dr. O. HACHOUR for giving us the opportunity to do research and providing valuable guidance and support. Her dynamism, vision, sincerity and motivation have deeply inspired us. We learned from her the methodology to carry out the research and to present the research works as clearly as possible. It was a great privilege and honor to work and study under her guidance.

We are extremely grateful to our beloved parents for their love, prayers, care and sacrifices to educate and prepare us for our future, we ask Allah to repay them with the highest price that none can pay but him. Also, we express our thanks to our sisters and brothers for their support along the years and for our friends and comrades that made this journey unforgettable, we wish for them success and happiness from this time until the end of time.

Finally, our thanks go to all the people who have supported us to complete the research work directly or indirectly.

Abstract

The autonomous wheeled mobile robots are very interesting subject both in scientific research and practical applications. They are considered from several different perspectives mainly, engineering and computer science levels. This project deals with the modeling and control of mobile robots combining the differential drive robot and unicycle models which will be simulated using a PID controller. The PID controller is based on feedback and tries to minimize the error using well-tuned parameters. The Odometry has been used to identify the distance traveled by the robot. The sensing circuitry mounted on the robot provides the feedback data to assure a safe outdoor navigation in a hostile environment. The Hybrid Automata principle provides a switching logic between the designed controllers. In this report, linear algebra is applied to develop a satisfying and stable model which is simulated using a MATLAB based simulator called "Sim.I.am" that allows the design and implementation of controllers on the robot.

Table of contents

CHAPTER 2 Simulation

List of Figures

GENERAL INTRODUCTION

The following introduction highlights the general description of our work including both project's motivation and objectives.

Overview

In recent years, the robotics and control of robotic systems is still an actual theme. In past, the static robots were used mostly in industrial tasks as manipulators, but the mobile robots were almost exclusively applied in research. The investigation and development of the autonomous mobile robot are increasing gradually in many fields such as in military, industries, and hospital. The mobile robots were designed with large size, heavy and require a high cost computer system which need to be connected via cable or wireless devices.

Nowadays, the trend is to evolve with a small mobile robot which is reduced in size, weigh, and cost of the system by using sensors, numerous actuators, and the controller are carried on-board the robot. Mobile robots are built based on a good relation of both hardware and software. There is one more thing that mobile robot really needs is a good navigation system such as vision camera or sensing components which allows the robot to perform successfully its tasks depending on the knowledge it has about the initial configuration of the workspace, but also the ones obtained during its evolution.

There are certain problems that arise in mobile robots, such as: determining the position and orientation in the environment, avoiding collision with different obstacles, planning an optimal movement path. The robot navigation is influenced by several methods, such as measuring the number of rotations made by the motor wheels, using gyros and accelerometers, but usually determines the pose of the robot in relation to a fixed coordinate system. When developing an autonomous mobile robot, to carry out the specific navigation tasks, the robot must be equipped with a suitable locomotion system. But the mobile robot would be nowhere near as effective, if it were not supported by an adequate control system.
For that it is proposed a closed loop control by using a PID controller that allows adjustment of the speed of the brushless DC motors. The reaction system is ensured through two rotary incremental encoders.

Motivation

Applying the ‘Control Principles’ which are the fundamental concepts for the design and analysis of mathematical models to implement suitable controllers for a mobile robot. The availability of various sensors and the efficiency of a PID(Proportional–Integral-Differential) motivated us to design an controller which enhances performance such as: Proportional Control that provides an immediate action to the control error which improves the rise time, Integral Control which to minimize the steady state error by driving it to zero, and the Differential Control increases damping in order to ensure a continuous performance.

The PIDs are most frequently used to implement path following robots by minimizing the error towards the actual goal. Since we are motivated to build an autonomous wheeled robot that will navigate to any desired goal location while avoiding obstacles crossing its path, we are going to use the PID controller to steer the robot towards the goal coordinates by minimizing the angle between the robot's heading and goal location orientation.

The project relies on the MATLAB based simulator 'Sim.I.am' which allows the testing of controllers and bridge the gap between theory and practice in 'Control Theory'.

Project Objectives

The purpose of our project is to design a smart wheeled robot with an autonomous motion provided that it avoids any obstacle (static or dynamic) in front and navigates properly towards any desired and known coordinates of the goal. Our robot should be able to negotiate different environments and reach successfully our desired goal location. The Wheeled Robot will be able to:

- Displace autonomously.
- Avoid all kinds of obstacles.
- Reach the desired location.

The control design objectives are:

- Stability.
- Tracking.
- Robustness.

Report Organization

This report is divided into two chapters. The first chapter gives a general overview about the models used and the linear algebra applied to develop the mathematical equations. The second chapter introduces the 'Sim.I.am' simulator on which we will test our controllers and ends with the implementation of our controllers. Finally, our report finishes with a general conclusion and suggestions for future works.

CHAPTER 1

Theory Description

In robotics, one of the most important tasks is describing the system and the way it functions, so how can we describe a mobile robot? And how does it achieve its goals? What are the main challenges and limitations? And how can we go around them?

1.1 Mathematical Model:

In order to design behaviors of controllers for Mobile Robots, we inevitably need models to decide how the robots will behave while navigating in an environment. For our Mobile Robot, we will use the **Differential Drive Model**. For instance, to successfully implement this model on the Mobile Robot, we are going to move with this model to some other model called the **Unicycle Model** which will allow us to overcome using complex variables such as wheel velocities.

1.1.1 Differential Drive Model:

A lot of mobile robots use a drive mechanism known as differential drive. It consists of two wheels mounted on a common axis, and each wheel can independently be driven either forward or backward. While we can vary the velocity of each wheel, for the robot to perform rolling motion, the robot must rotate about a point that lies along their common left and right wheel axis. **Figure 1.1** illustrates the differential drive model where the circle represents the actual robot and the black rectangles are supposed to be the wheels **[1].**

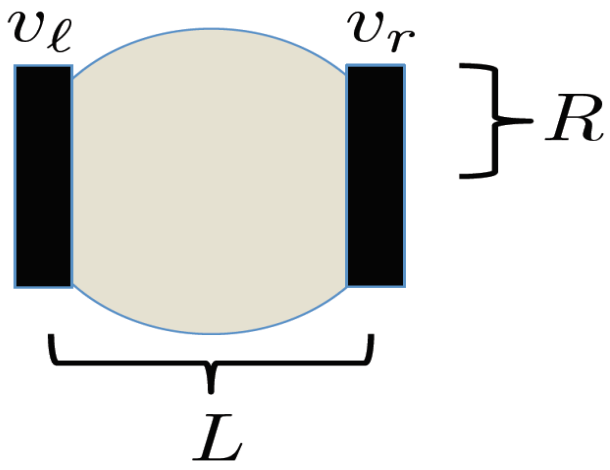


Figure 1.1: The representation of the Differential Drive.

For instance, if we are turning our wheels at same rate, the robot will be moving straight ahead. Also, if one wheel is turning slower than another, then the robot is turning towards the direction in which the slower wheel is mounted. We have seen before that a good controller shouldn't have to take in consideration the particular parameters of each robot in order to neglect the friction coefficient, however we need to consider two parameters which are: the distance that separates the wheel base which is represented by L, and the radius of the wheel represented by R. These two parameters are actually easy to measure which facilitates the use of this model.

In order to control the way our robot is moving; we use two control signals which are the velocities of our two wheels, where v_r and v_l are the speed at which the right and left wheels are turning respectively. These two velocities are the input signals of our system.

The mathematical model relating the two input signals (the velocities of the wheels) directly to the output signals (the position and orientation) of the mobile robot, based on these observations, the configuration transition equation is:

$$\dot{x} = \frac{R}{2}(v_r + v_l)\cos\theta$$

$$\dot{y} = \frac{R}{2}(v_r + v_l)\sin\theta$$

$$\dot{\theta} = \frac{R}{L}(v_r - v_l)$$

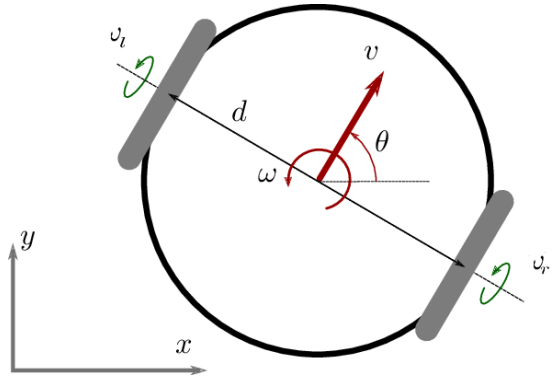


Figure 1.2: The representation of the Mobile Robot in a 2D plane.

The equations above contain $\cos\theta$ and $\sin\theta$ parts because the differential drive moves in the direction that its drive wheels are pointing. The translation speed depends on the average of the angular wheel velocities. To see this, consider the case in which one wheel is fixed and the other rotates. This initially causes the robot to translate at $^1/_2$ of the speed in comparison to bothwheels rotating. The rotational speed $\dot{\theta}$ is proportional to the change in angular wheel speeds. The robot's rotation rate grows linearly with the wheel radius but reduces linearly with respect to the distance between the wheels.

1.1.2 Unicycle Model:

Dealing with the displacement and velocities of the two wheels of a differential drive robot is messy. A preferred model is that of a unicycle (**Figure 1.3)**, where we can think of the robot as having one wheel that can move with a desired $velocity\ (v)$ at a specified heading $theta\ (\theta)$. The unicycle models are selected for their simplicity and good maneuverability. At same time, research is conducted on controllability, feedback, linearization and stabilization raises many research and development challenges in the control of unicycle type robots. Since our robot is designed to navigate to a certain goal location while avoiding obstacles, the unicycle model satisfies the tasks in a stable and smooth manner **[2].**

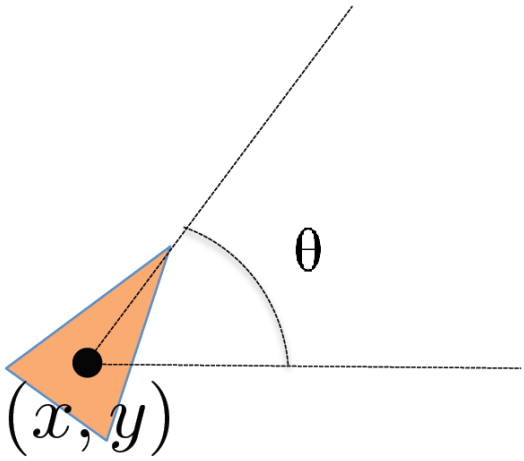


Figure 1.3: The physical description of the Unicycle.

The unicycle type robot is in general a robot moving in a 2D world which is represented by an $x - y\ plane$. It has some forward speed but zero instantaneous lateral motion.

The equations to translate between the unicycle model and our wheel velocities allows us to simplify the differential drive model with the unicycle model. We have seen how to take measured wheel displacements to calculate the new robot pose. The kinematics of the unicycle model is usually described by a simple non-linear model:

$$\dot{x} = v\cos\theta$$

$$\dot{y} = v\sin\theta$$

$$\dot{\theta} = w$$

where $theta(\theta)$ is the orientation of the robot and (x, y) are the coordinates representing the actual position of the mobile robot in the $x - y\ plane$. v and w are the inputs and they represent the linear and angular velocities of the robot respectively.

1.1.3 Mapping of Models:

Since we have our two models, we need to combine them to have a final model which we can implement and design. We will be using the unicycle for analysis and control of its inputs v and ω then map them to the inputs of the differential drive v_r and v_l.

The mapping we are going to use is based on the kinematics of the two models:

$$\begin{cases} \dot{x} = \frac{R}{2}(v_r + v_l)\cos\theta \\ \dot{y} = \frac{R}{2}(v_r + v_l)\sin\theta \\ \dot{\theta} = \frac{R}{L}(v_r - v_l) \end{cases} \dots\dots\dots\dots\dots\dots (1)$$

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = w \end{cases} \dots\dots\dots\dots\dots\dots\dots\dots\dots (2)$$

From (1) and (2) we get the following linear equations:

$$v = \frac{R}{2}(v_r + v_l) \quad \Rightarrow \quad \frac{2v}{R} = v_r + v_l$$

$$w = \frac{R}{L}(v_r - v_l) \quad \Rightarrow \quad \frac{wL}{R} = v_r - v_l$$

Since we are going to map our designed inputs (v, w) onto the actual inputs (v_r, v_l) that are indeed running on the robot, we derive these final linear equations which we are going to use on the mobile robot:

$$v_r = \frac{2v + wL}{2R}$$
$$v_l = \frac{2v - wL}{2R}$$

1.2 Odometry:

Most robotics problems are ultimately reduced to the ability of localization in the environment of navigation. A basic method of navigation is odometry, using knowledge of your wheel's motion to estimate your vehicle's motion and actual location.

We'll assume that the vehicle is differentially driven: it has a motor on the left side of the robot, and another motor on the right side. If both motors rotate forward, the robot goes (roughly) straight. If the right motor turns faster than the left motor, the robot will move left.

Our goal is to measure how fast our left and right motors are turning. From this, we can measure our velocity and rate of turn, and then integrate these quantities to obtain our position.

In order to achieve odometry, we can use:

- **External Sensors:** an external sensor would be a sensor that's measuring something in the environment such as ultrasound, infrared, camera and laser scanners.
- **Internal Sensors:** are sensors that are included in the robot and are measuring the position of the robots such as accelerometers, gyroscopes and wheel encoders.

In this project, we will be using both external (Infrared) and internal (Wheel Encoder). Since the odometry is concerned with the self-localization of the robot in the environment, we will be interested in Wheel Encoders that iterates the number ticks of each wheel **[3].**

Suppose the left wheel has moved by a distance of d_{left} and the right wheel has moved d_{right}. For some small period of time (such that d_{left} and d_{right} are short), we can reasonably assume that the robot trajectory was an arc (see **Figure 1.4**).

The initial state (x, y, θ) defines our starting point, with θ representing the robot's heading. After our vehicle has moved by d_{left} and d_{right}, we want to compute the new position, (x', y', θ').

The center of the robot (the spot immediately between the two wheels that defines the robot's location), travels along an arc as well. Remembering that arc length is equal to the radius times the inner angle, the length of this arc is:

$$d_{center} = \frac{d_{left} + d_{right}}{2}$$

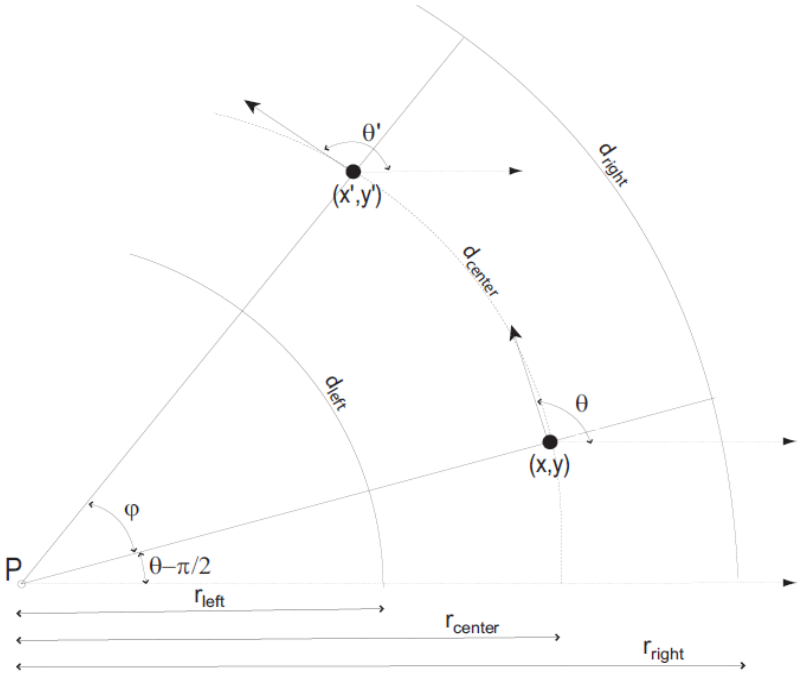


Figure 1.4: The robot is moving counter-clockwise over a small time period.

Given basic geometry, we know that:

$$\phi r_{left} = d_{left} \dots\dots\dots\dots\dots\dots\dots (1)$$

$$\phi r_{right} = d_{right} \dots\dots\dots\dots\dots\dots\dots . (2)$$

If $d_{baseline}$ is the distance between the left and right wheels, we can write:

$$r_{left} + d_{baseline} = r_{right}$$

Subtracting (1) from (2), we see:

$$\phi r_{right} - \phi r_{left} = d_{right} - d_{left}$$

$$\phi(r_{right} - r_{left}) = d_{right} - d_{left}$$

$$\phi d_{baseline} = d_{right} - d_{left}$$

$$\phi = \frac{d_{right} - d_{left}}{d_{baseline}}$$

All of our arcs have a common origin at point P . Note that the angle of the robot's baseline with respect to the $x - axis$ is $\theta - {}^{\pi}/_{2}$. We now compute the coordinates of P :

$$P_x = x - r_{center}\cos(\theta - {}^{\pi}/_{2})$$
$$= x - r_{center}\sin(\theta)$$
$$P_y = y - r_{center}\sin(\theta - {}^{\pi}/_{2})$$
$$= y + r_{center}\cos(\theta)$$

Now we can compute x'andy':

$$x' = P_x + r_{center}\cos(\phi + \theta - {}^{\pi}/_{2})$$
$$= x - r_{center}\sin(\theta) + r_{center}\sin(\phi + \theta)$$
$$= x + r_{center}[-\sin(\theta) + \sin(\phi)\cos(\theta) + \sin(\theta)\cos(\phi)$$

And

$$y' = P_y + r_{center}\sin(\phi + \theta - {}^{\pi}/_{2})$$
$$= y + r_{center}\cos(\theta) - r_{center}\cos(\phi + \theta)$$
$$= y + r_{center}[\cos(\theta) - \cos(\phi)\cos(\theta) + \sin(\theta)\sin(\phi)]$$

If ϕ is small (as is usually the case for small time steps), we can approximate $\sin(\phi) = \phi$ and $\cos(\phi) = 1$. This now gives us:

$$x' = x + r_{center}[-\sin(\theta) + \phi\cos(\theta) + \sin(\theta)]$$
$$= x + r_{center}\phi\cos(\theta)$$
$$= x + d_{center}\cos(\theta)$$

and

$$y' = y + r_{center}\,[\cos(\theta) - \cos(\theta) + \phi\sin(\theta)]$$
$$= y + r_{center}\,\phi\,\sin(\theta)$$
$$= y + d_{center}\,\sin(\theta)$$

In summary, our odometry equations for (x', y', θ') reduce to:

$$x' = x + d_{center}\cos(\theta)$$
$$y' = y + d_{center}\sin(\theta)$$
$$\theta' = \theta + \phi$$

where:

$$d_{center} = \frac{d_{left} + d_{right}}{2}$$
$$\phi = \frac{d_{right} - d_{left}}{d_{baseline}}$$

We still have to measure the distance travelled by each wheel (d_{left} and d_{right}) which implies the use of Wheel Encoders. The working principle of wheel encoders is counting the ticks in order to compute the number of revolutions made by the wheels in a certain amount of time. So, a wheel encoder gives the distance moved by each wheel. The speeds of our motors give us two quantities: the rate at which the vehicle is turning, and the rate at which the vehicle is moving forward. Given the amount of rotation of the motor and the diameter of the wheel, we can compute the actual distance that the wheel has covered. In order to simplify the calculations, we will work on the previous assumption that approximates the distance traveled by each wheel as an arc which is valid for a short time scale (**Figure 1.5**). In our mobile robot, we consider the distance of the baseline as L, the distance traveled by the left wheel as D_l, the distance traveled by the right wheel as D_r and the distance turned by the center of the robot is referred to as D_c.

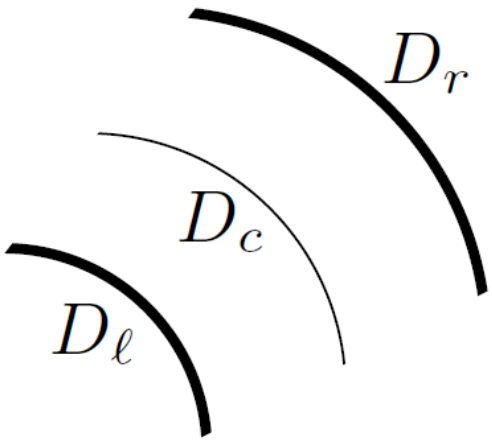


Figure 1.5: The distances traveled by the wheels and robot.

In order to measure the distances D_r and D_l, we assume that each wheel has its own number of ticks per revolution, then let N be the number of ticks accomplished by a wheel per revolution. Since most of wheel encoders give the total tick count since the beginning, we need to count the number of ticks made by each wheel since the last position. We compute the number of ticks using the following relation:

$$\Delta tick = tick' - tick$$

where $tick'$ is the number of ticks accumulated at the actual position(x', y', θ'), $tick$ is the number of ticks saved from the previous position(x, y, θ) and $\Delta tick$ is the number of ticks realized by the wheel. We assume the number of ticks done by the left wheel as $\Delta tick_{left}$ and the number of ticks done by the right wheel as $\Delta tick_{right}$. Then we use the following equations to calculate the distances:

$$D_l = 2\pi \frac{\Delta tick_{left}}{N}$$

$$D_r = 2\pi \frac{\Delta tick_{right}}{N}$$

Then the actual kinematics of the robot is given by:

$$x' = x + D_c \cos(\theta)$$
$$y' = y + D_c \sin(\theta)$$
$$\theta' = \theta + \frac{D_r - D_l}{2}$$

where D_c is computed as the following:

$$D_c = \frac{D_r + D_l}{2}$$

1.3 Controllers Design:

1.3.1 Go-To-Goal Controller:

The main objective of our robot is navigating to a specified location. The go-to-goal behavior will make our robot move from its actual position (x, y, θ) to a new position which is described as the goal location (x_g, y_g, θ_g)as it is represented in the **Figure 1.6.**

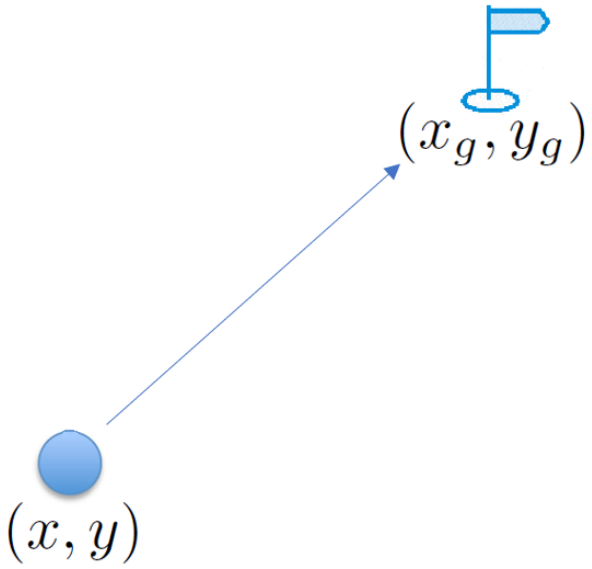


Figure 1.6: An illustration of a point robot and its goal location.

In order to reach the goal location using a differential drive robot that we can model as a unicycle, we set the linear velocity v as constant. Then, we need to control the heading which is directly related to the angular velocity w that is controlled using the reference tracking.

The reference tracking is exerting a control action on a system in order to manipulate the process output to be the same as the reference input. The reference tracking is based on closed loop controllers that are also called feedback controllers.

A closed-loop control system is a system in which the value of some output quantity is measured using sensors. Feeding back the value of the controlled quantity, allows the manipulation of an input quantity so as to bring the value of the controlled quantity closer to a desired value. The difference between the reference value and the measured output is described as the error e.

Since our main objective is to steer the robot towards a desired location, we will be dealing with angles.

Firstly, we set the reference point to be the desired angle θ_d. We already have the actual heading of our robot as *theta* θ. The tracking error e for this kind of problem is:

$$e = \theta_d - \theta$$

The desired angle θ_d can be calculated using the actual coordinates of the mobile robot (x, y) and the desired location coordinates (x_g, y_g). The following arc tangent formula can be used to compute the desired angle:

$$\theta_d = \tan^{-1}\left(\frac{y_g - y}{x_g - x}\right)$$

Since we have our tracking error, we can plug this error in a controller which will be acting on it to correct the desired heading.

1.3.2 PID controller:

We assume that our mobile robot is driving at a constant velocity v_0, which implies the following design model:

$$\begin{cases} \dot{x} = v_o \cos\theta \\ \dot{y} = v_o \sin\theta \\ \dot{\theta} = w \end{cases}$$

Then the objective is to make the mobile robot drive in the desired heading, which implies controlling the angular velocity ω. In order to achieve the desired heading control, we need to implement a controller.

The PID controller compares the measured output with the input reference.

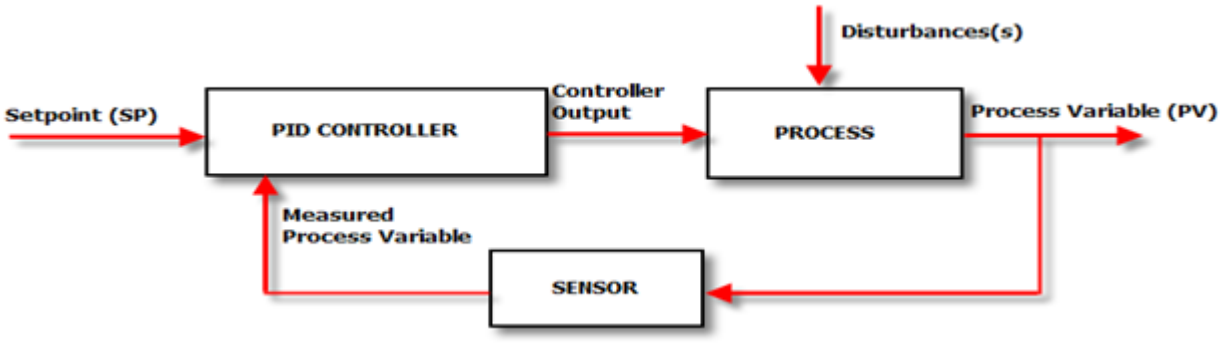
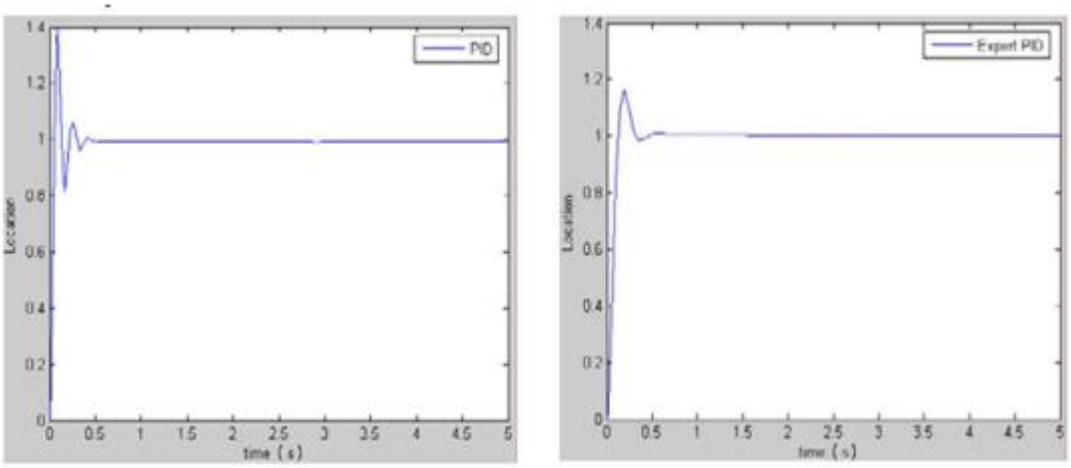


Figure 1.7: Block diagram illustration of a simple PID controlled system.

The PID controller combines the position error and error of change output to correct the PID parameters for mobile robot. The mobile robot uses infrared sensors to avoid obstacles around it while heading to reach the desired position.

The simple output response curve using a simple PID control is shown in **Figure 1.8a**. By using control theory on the PID, the output response curve is presented in **Figure 1.8b**. As a result, the PID controller has a better performance.



(a) (b)

Figure 1.8: (a) A simple PID output response, (b) A well-tuned PID output response.

The control of the mobile robot implies controlling the heading which is a control loop feedback mechanism. In PID control, the current output is based on the feedback of the previous output, which is computed so as to keep the error small. The error is calculated as the difference between the desired and the measured value, which should be as small as possible. A correction of ω is applied by summing three terms, known as the proportional term, integral term, and derivative term **[4].**

Since we have a model and a controller, we can control the steering of our mobile robot by governing the angular velocity w driving the robot towards the desired angle by implementing the following PID regulator:

$$w = K_P e + K_I \int e \, d\tau + K_D \dot{e}$$

Where w is the angular speed or the steering control input, K_P is the proportional gain constant, K_I is the integral gain constant and K_D is the differential gain constant.

The Proportional term is used in calculating current errors, the Integral term provides information about the amount of previous errors and the derivative term predicts the future errors.

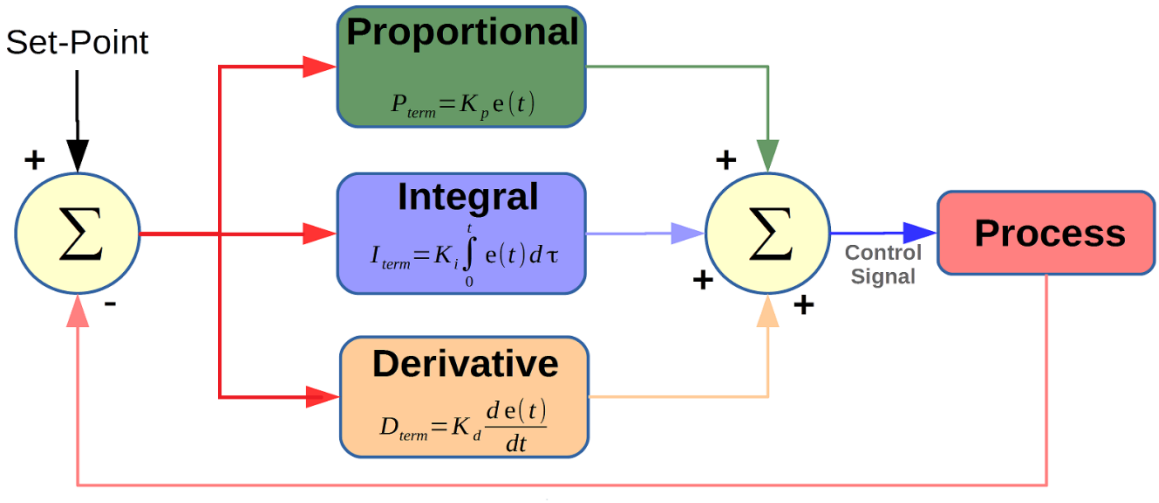


Figure 1.9: Block diagram illustration of each term used in a PID regulator.

1.3.3 Tricky angles:

Since dealing with angles is one of the most complicated tasks, we need to implement our controller carefully by taking into consideration every possible case.

The main issue we could face is having significant errors while the actual errors are too small. Suppose we have the following actual and desired angles:

$$\theta = 100\pi \; rad$$

and

$$\theta_d = 0 \; rad$$

this yields to the following error:

$$e = \theta_d - \theta = -100\pi$$

which seems to be a huge error and can affect the actual behavior of the mobile robot by making it spinning around without achieving its tasks. Whereas, we also notice that:

$$e = -100\pi = 0 \; rad$$

which implies that the robot is already on the desired angle and there is no need for correction.

We handle this issue by ensuring that the error always belongs to the range $[-\pi, \pi]$ which is described by the following notation:

$$e \in [-\pi, \pi]$$

In order to apply the above solution, we use the inverse of the tangent function or simply: the *arctangent* function. Then the error is calculated using the bellow mathematical relationship:

$$e = \tan^{-1}(\frac{\sin(e)}{\cos(e)})$$

1.3.4 Obstacle Avoidance Controller:

Now that we have designed a controller which has the ability to take our robot to a goal location, we have to implement another controller which is responsible of avoiding obstacles while navigating to this goal location.

Actually, our second main objective is to drive the robot safely without colliding with the different kind of obstacles that exist in an unknown navigation environment. The go-to-goal behavior alongside with the Obstacle Avoidance behavior are the basic dynamic duo of mobile robots.

For the obstacle avoidance controller, we are going to use the same concept used for the go-to-goal controller by a defining a desired heading while sensing an obstacle that is close to the robot. We assume that we have the situation described in the **Figure 1.10** where the obstacle is represented with a wall that is situated between the robot and its defined goal location.

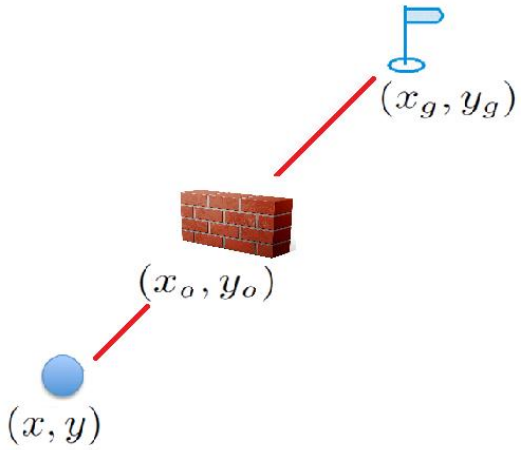


Figure 1.10: An illustration of a point robot facing an obstacle in its path to goal location.

The location of the obstacle (x_o, y_o) is identified using the sensors mounted on the mobile robot which is referred to as a Range-Sensor Skirt. A Range-Sensor Skirt is an array of sensors distributed evenly on the outside side of the robot with the objective of detecting any obstacle around it within a defined range specified in the sensor characteristics.

If we were building a pure obstacle avoidance controller, we can simply steer our robot to a direction opposite to the obstacle location using the following equation:

$$\theta_d = \theta_o + \pi$$

where θ_o is the steering angle towards the obstacle location and can be calculated using the arc tangent function:

$$\theta_o = \tan^{-1}\left(\frac{y_o - y}{x_o - x}\right)$$

However, our main objective is to drive the robot to the desired location which implies taking the goal location into consideration while avoiding the obstacle. The second approach that can be used is going perpendicularly to the obstacle direction which can be implemented using the equation bellow:

$$\theta_d = \theta_o \pm \frac{\pi}{2}$$

where the sign of $\frac{\pi}{2}$ depends on which direction makes the robot closer to the desired goal location. Even if the second approach gives a much better result than the first one, it still needs adjustments by blending the two controllers using behavior-based control.

Behavior-based control implies switching between different modes of behaviors or operations depending on the actual condition of navigation. If the robot detects no obstacles around it, it switches to the go-to-goal mode. Whereas, if it detects an obstacle on his path to the goal location, it switches to the obstacle avoidance controller. We refer to these kind of systems as switched or hybrid systems.

1.4 Hybrid Automata:

Hybrid Automata is a modeling formalism for hybrid systems that results from an extension of finite-state machines by associating with each discrete state a continuous-state model. Conditions on the continuous evolution of the system invoke discrete state transitions. A broad set of analysis methods is available for hybrid automata including methods for the reachability analysis and stability analysis. A hybrid automaton is a transition system that is extended with continuous dynamics. It consists of locations, transitions, invariants, guards, *n*-dimensional continuous functions, jump functions, and synchronization labels **[5].**

Since we are looking for a switch logic to combine our two dynamic behaviors, the hybrid automata provide the means to describe both continuous dynamics with a discrete switch logic. This implies that the discrete logic will be modeled as a finite state machine that moves between different discrete states. Inside each state we have continuous dynamics that describe our desired behavior.

Let the continuous state of the system to be x. As we will be switching between different modes of operations, we add an additional discrete state q which will indicate the actual mode in which the system is. The dynamics now become:

$$\dot{x} = f_q(x, u)$$

where $f_q(x, u)$ depends on the mode we are operating in. The transition between different discrete modes can be encoded in a state machine as:

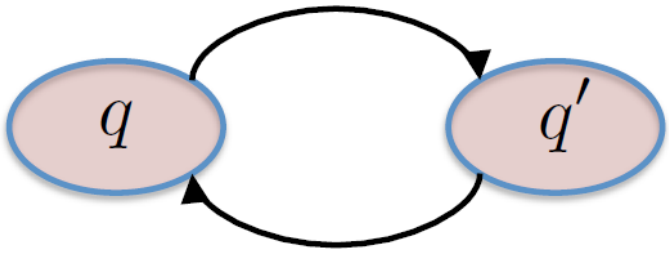


Figure 1.11: An illustration of the transition between two different modes.

When we jump between the different discrete modes q and q', we say that transitions between different states in the finite state machine are being made.

The conditions under which a transition occurs are called guard conditions, i.e., a transition occurs from q to q' if:

$$x \in G_{q,q'}$$

The guards are used to check whether some conditions are satisfied in order to make a jump from one mode to another.

As a final component, we would like to allow abrupt changes in the continuous state as the transitions occur. These abrupt changes are called resets and represented as the following:

$$x := R_{q,q'}$$

The resets are used to set the states to specific values after that a transition is made from one mode to another. Putting all of this together yields to a very rich model known as a hybrid automata (HA) model (see **Figure1.12**).

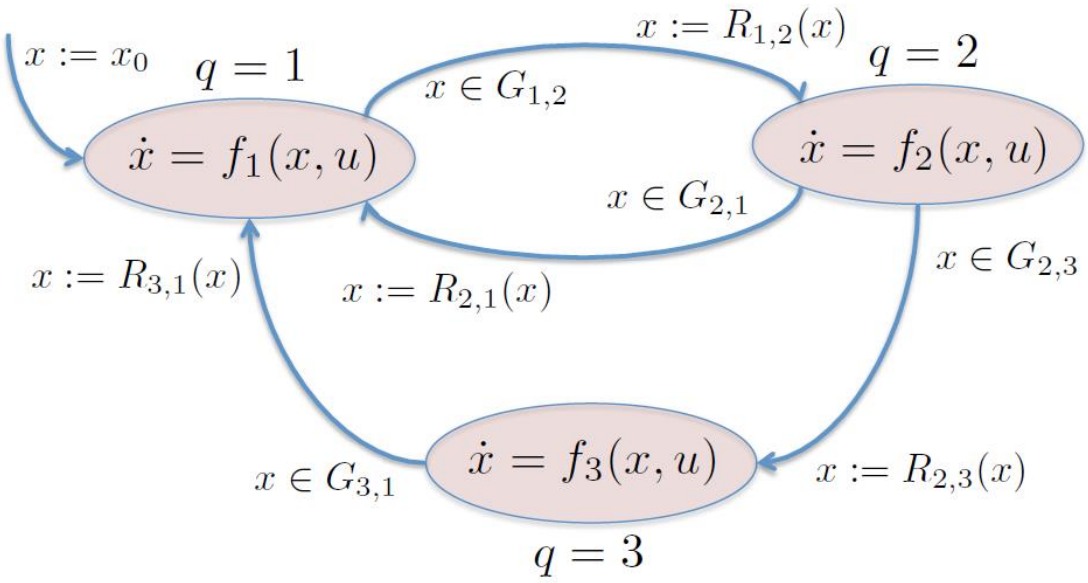


Figure 1.12: An example of a hybrid automata model.

An important point to take into consideration, is that a hybrid system can be destabilized by switching between different modes even if the different subsystems or modes were asymptotically stable themselves. If we ignore the resets, there will be no abrupt changes in the states when making transitions. The system becomes a switch system where we have:

$$\dot{x} = f_\sigma(x, u)$$

where σ is a switch signal and it indicates in which mode the system is running. Assume there are p discrete modes in our system then:

$$\sigma(t) \in \Sigma = \{1, \ldots., p\}$$

Given a switching system $\dot{x} = f_\sigma(x)$, we can define three different kind of stability:

1. *Universal Asymptotic Stability:* it implies that there is nothing that can destabilize the system, which means that x will always go to *zero* for any value of *sigma*σ:

$$x \to 0, \qquad \forall \sigma$$

2. *Existential Asymptotic Stability:* it implies that there exists a switch signal σ that makes the state x go to *zero* such as:

$$\exists\ \sigma\ s.t. x \to 0$$

3. *Hybrid Asymptotic Stability:* it implies that we have a hybrid system that is itself generating the switch signal, which means that the switch signal is generated by an underlying hybrid automation and x goes to *zero* not for any or for all sigma σ, but for the one that happens to be the one that we have in our hybrid system:

$$x \to 0$$

As a conclusion, we have to design stable controllers as subsystems and always make sure that the resulted Hybrid Automata model is stable for each possible switching signal.

Finally, we need to consider an important phenomenon in the hybrid automata model. Suppose we have the following system:

$$\dot{x} = \begin{cases} -1 & ; x \geq 0 \\ +1 & ; x < 0 \end{cases}$$

The above system is represented as a Hybrid Automata model in the **Figure 1.13**.

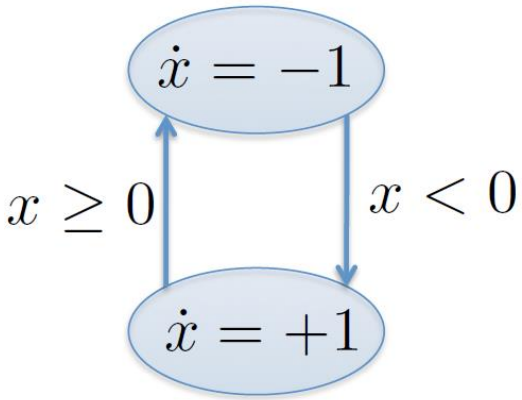


Figure 1.13: The Hybrid Automata model of the above example.

Since we have only two modes, we notice that when the system reaches *zero* 0, it starts switching infinitely many times in a single time-instant which causes what is called as the *Zeno Phenomenon*.

1.4.1 The Zeno Phenomenon:

The Zeno Phenomenon leads to unnatural loss of stability of equilibriums and the emergence of unexpected and meaningless solutions in case of interconnected systems. Many works are devoted to the question of prolongation of such solutions, however there exist no unified or commonly accepted prolongation method **[6].**

One solution to the Zeno Phenomenon is using Sliding Mode Control. Let us be more general and assume that we have the following system:

$$\dot{x} = \begin{cases} f_1(x) & ; g(x) \geq 0 \\ f_2(x) & ; g(x) < 0 \end{cases}$$

The above general system has the Hybrid Automata Model represented in the **Figure 1.14**.

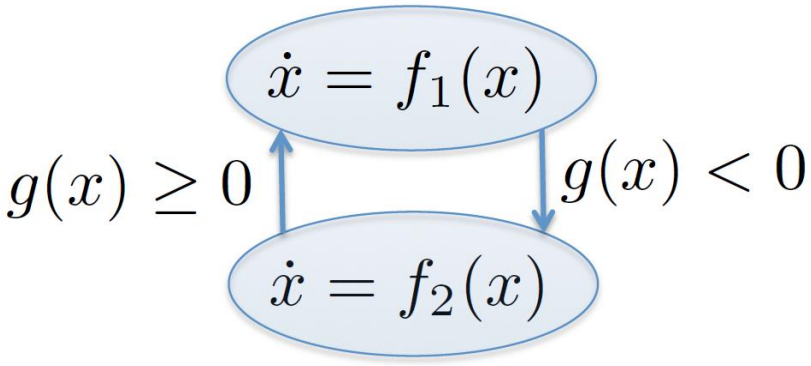


Figure 1.14: The Hybrid Automata model of a general system.

Consider the switching surface $g(x) = 0$ represented in the **Figure 1.15,** decides in which mode the system will be operating. If $g(x) > 0$ then the system will be using the $f_1(x)$, otherwise if $g(x) < 0$ the system will be using the $f_2(x)$.The most important point in our system is the $g(x) = 0$ where there is a possibility to have a sliding.

The sliding along the switching surface when occurs because $f_1(x)$ and $f_2(x)$are pulling in different directions. The sliding mode happens when $f_1(x)$ tries to drive the system towards the surface where $g(x) < 0$ and $f_2(x)$ tries to drive the system towards the surface where $g(x) > 0$(see **Figure 1.15**).

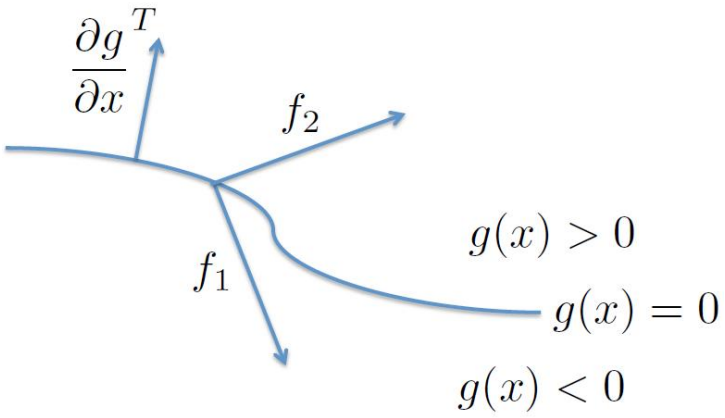


Figure 1.15: The illustration of the switching surface $g(x)$.

In order to check is the sliding occurs, we need to analyze the vector that is normal to the switching surface which is called the *gradient*. The sliding mode happens if the following conditions are satisfied:

$$\frac{\partial g}{\partial x} f_1 < 0 \ \ and \ \ \frac{\partial g}{\partial x} f_2 > 0 \ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

The term $\frac{\partial g}{\partial x} f$ is simply the derivative of g in the direction of f and is called the *Lie Derivative*. The *Lie Derivative* is denoted by the term $L_f g$. The conditions above (1) which should be satisfied to have sliding become:

$$L_{f_1} g < 0 \ and \ L_{f_2} g > 0$$

In the case where the above conditions are satisfied, we apply *Regularizations* to our system in order to solve the *Zeno Phenomenon* problem. At $g(x) = 0$, the change in $g(x)$will be *zero* which implies that $\frac{dg}{dt} = 0.$

The *Regularizations* implies introducing a new mode called the **Induced Mode** which defined by the following equation:

$$\dot{x} = \sigma_1 f_1 + \sigma_2 f_2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Consider now the relation bellow:

$$\frac{dg}{dt} = \frac{dg}{dx}\dot{x} = \frac{\partial g}{\partial x}(\sigma_1 f_1 + \sigma_2 f_2) = \sigma_1 L_{f_1} g + \sigma_2 L_{f_2} g$$

since $\frac{dg}{dt} = 0$, then:

$$\sigma_1 L_{f_1} g + \sigma_2 L_{f_2} g = 0 \quad \Rightarrow \quad \sigma_2 = -\sigma_1 \frac{L_{f_1} g}{L_{f_2} g}$$

Since we are not allowed to flow backward, then σ_1 and σ_2 must be positive. We also want the sum of σ_1 and σ_2 to be one in order to respect the dynamics of the system. So, we have the following additional constraints about σ_1 and σ_2:

$$\sigma_1, \sigma_2 \geq 0 \; and \; \sigma_1 + \sigma_2 = 1$$

now we can compute the *induced mode* using the above relation:

$$\sigma_2 = -\sigma_1 \frac{L_{f_1} g}{L_{f_2} g}$$

since we also have:

$$\sigma_1 + \sigma_2 = \sigma_1 \left(1 - \frac{L_{f_1} g}{L_{f_2} g}\right) = 1$$

then we get:

$$\sigma_1 = \frac{1}{1 - \frac{L_{f_1} g}{L_{f_2} g}} = \frac{L_{f_2} g}{L_{f_2} g - L_{f_1} g} \ldots\ldots\ldots\ldots\ldots\ldots (3)$$

which yields to:

$$\sigma_2 = -\frac{L_{f_1} g}{L_{f_2} g - L_{f_1} g} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (4)$$

we substitute the equations (3) and (4) into the relation (2) in order to get the final expression representing the induced mode:

$$\dot{x} = \frac{1}{L_{f_2} g - L_{f_1} g}(L_{f_2} g f_1 - L_{f_1} g f_2)$$

After the Regularizations, the Hybrid Automata of the general system represented in the **Figure 1.14** becomes more stable because of the additional sliding mode that we have introduced (see **Figure1.16**).

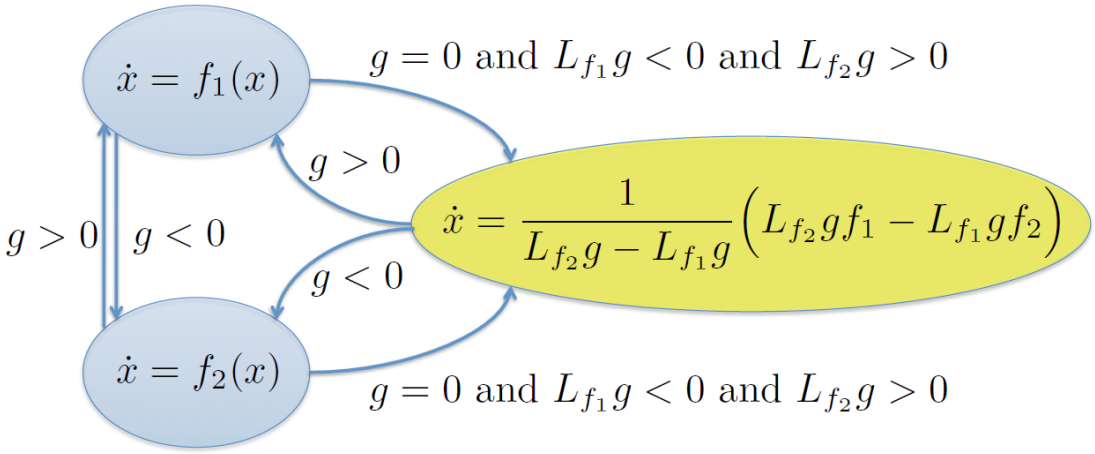


Figure 1.16: The Hybrid Automata model of a general system after Regularizations.

1.4.2 Type of obstacles:

In the avoidance obstacle behavior, we have considered a standard simple type of obstacles which is not sufficient for a well-designed mobile robot. There are different types of obstacles that goes from the simplest to the most complicated one **[7].**

1. Point-Obstacles:

For this type of obstacles (see **Figure 1.17**), the two previous behaviors are sufficient unless we need to add an induced mode due to *Zeno Phenomenon*. One way to deal with this type of obstacles, is by adding some noise which is not really needed in practice since the world is already noisy.

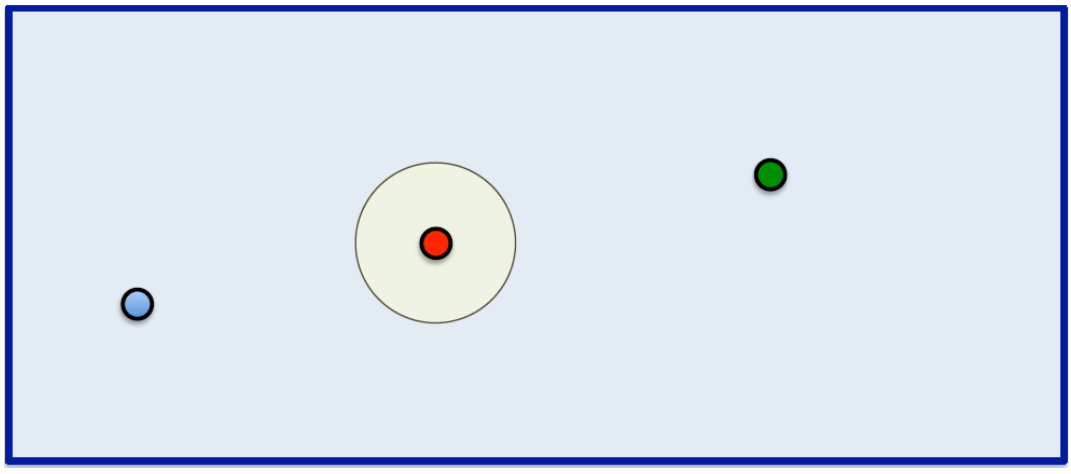


Figure 1.17: An illustration of a Point-Obstacle.

2. **Circular Obstacles:**
   Basically, circular obstacles are point obstacles that are just larger (see **Figure1.18**) which means that we can deal with them using only the two previous controllers.

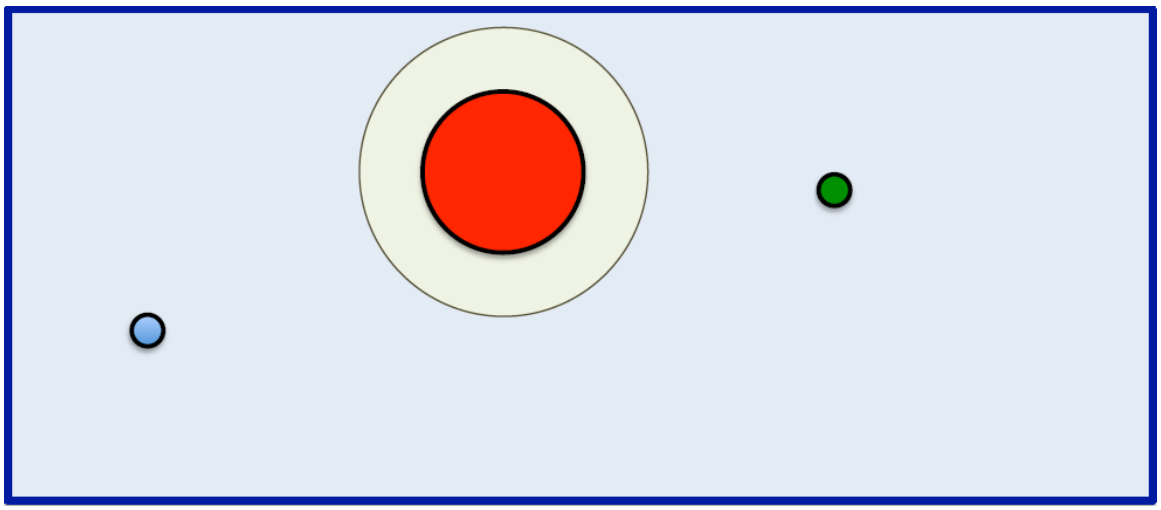


Figure 1.18: An illustration of a Circular Obstacle.

3. **Convex Obstacles:**
   A convex obstacle is more complexed obstacle that can look like a circle or rectangle (see **Figure 1.19**). Convexity implies that every two point in the obstacle can be joined with a straight line that completely lies inside the obstacle. The two behaviors that we have seen before will not be enough in order to avoid that kind of obstacles.

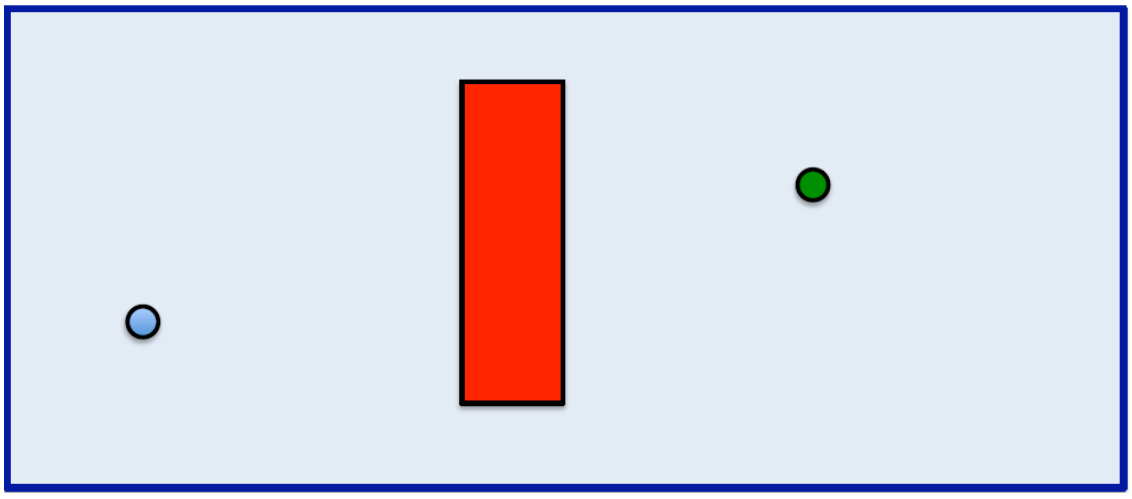


Figure 1.19: An illustration of a Convex Obstacle.

4. **Non-Convex Obstacles:**
   A convex obstacle is even more complexed than convex obstacle. The non-convex obstacle can look like any shape because it simply means that there exist two points that we cannot connect with a straight line (see **Figure 1.20**). This type of obstacles requires a more sophisticated behavior in order to reach the goal.

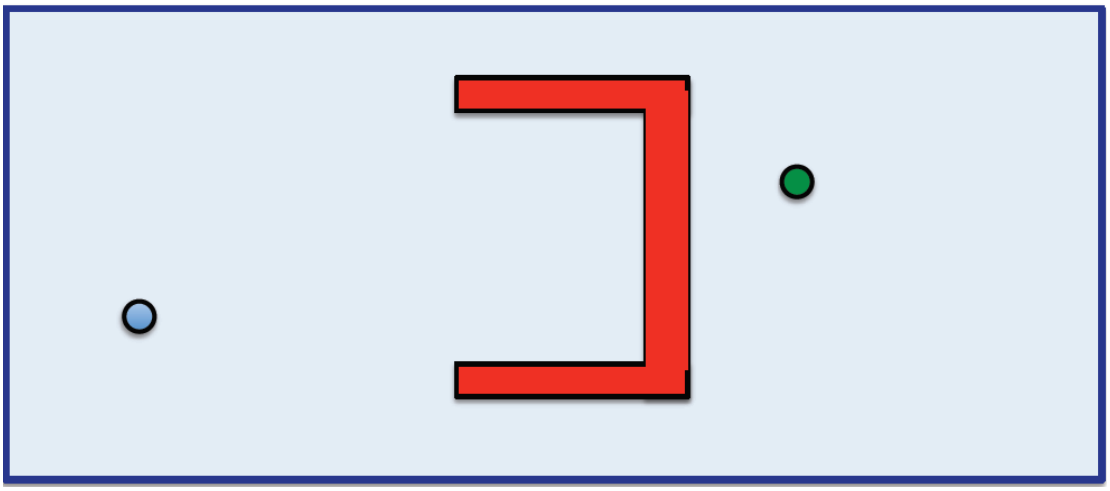


Figure 1.20: An illustration of a Non-Convex Obstacle.

5. **Labyrinth Obstacles:**
   The labyrinth obstacle is the most complexed obstacle that could face the robot while navigating towards the goal location. This type of obstacle which is also called a maze obstacle, is considered as one of the most challenging problems to face when designing robots. This type of obstacles (see **Figure 1.21**). requires a more advanced behavior to pass it and reach the desired location.

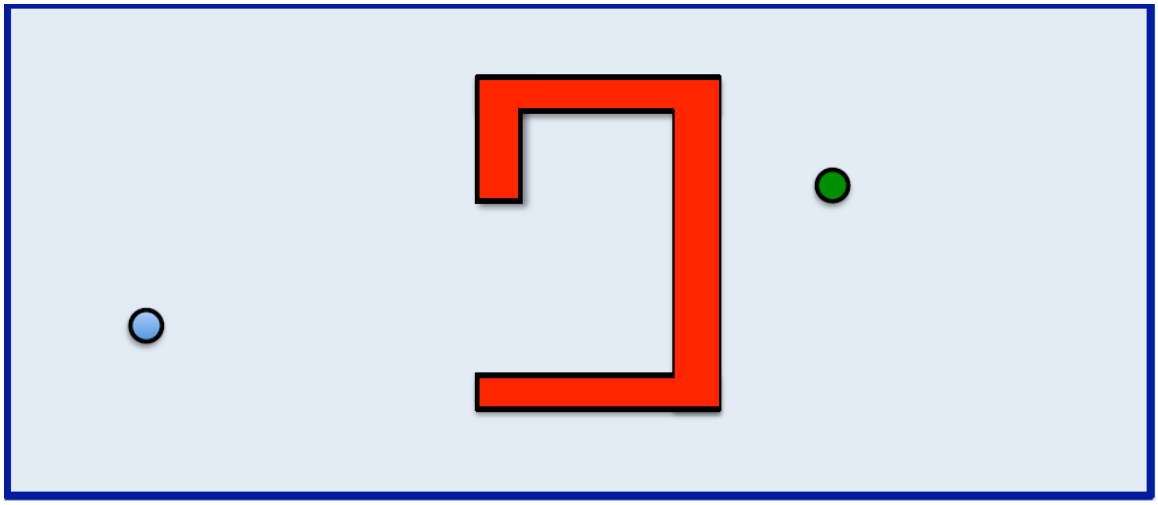


Figure 1.21: An illustration of a Labyrinth Obstacle.

1.4.3 Wall-Following Behavior:

In order to negotiate complex environment, the previous dynamic behaviors: Go-to-Goal and Avoid Obstacles are now longer sufficient. The need of an additional behavior becomes essential because the robot needs a controller to follow the boundary of an obstacle/wall in order to be able to get around in the world. The missing behavior is the wall following which needs to be designed.

The follow wall controller should maintain a constant distance to the obstacle/wall which is called the safety distance and is represented by a disc around the circular obstacle in **Figure 1.22.** We can clearly move in two different direction along a wall using either the *clockwise follow wall* vector u_{FW}^{c} or the *counterclockwise follow wall* vector u_{FW}^{cc}.

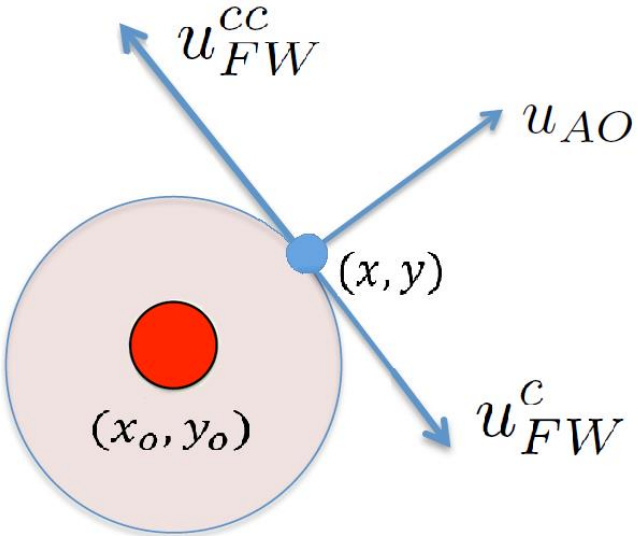


Figure 1.22: A representation of the follow wall vectors.

The follow wall vector u_{FW} is simply the obstacle avoidance vector u_{AO} rotated by either $-\pi/2$ for u_{FW}^{cc} or $\pi/2$ for u_{FW}^{c} and scaled using by the scalar α.

The rotation is made using the following rotation matrix $R(\varphi)$:

$$R(\varphi) = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix}$$

Then the final expressions to calculate both u_{FW}^{c} and u_{FW}^{cc} are:

$$u_{FW}^{c} = \alpha R(-\pi/2) u_{AO} = \alpha \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} u_{AO}$$

$$u_{FW}^{cc} = \alpha R(\pi/2) u_{AO} = \alpha \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} u_{AO}$$

Since we have two possible vectors: *clockwise follow wall* vector u_{FW}^{c} or the *counterclockwise follow wall* vector u_{FW}^{cc}, we need a test to choose which is more appropriate to use at each possible obstacle. Consider using the inner product of two vectors:

$$\langle v, w \rangle = v^T w = \|v\| \|w\| cos(\angle(v, w))$$

The best approach is to consider the *go-to-goal vector* u_{GTG} using the following mathematical relations:

$$if\ \langle u_{GTG}, u_{FW}^{c} \rangle > 0 \ \Rightarrow u_{FW}^{c}$$
$$if\ \langle u_{GTG}, u_{FW}^{cc} \rangle > 0 \ \Rightarrow u_{FW}^{cc}$$

The above tests allow us to determine which direction to take when following a wall. We check the go-to-goal inner product with follow wall clockwise, if the result is positive the robot should go clockwise which implies that the angle between u_{GTG} and u_{FW}^{c} is less than $\pi/2$. This also means that angle between u_{GTG} and u_{FW}^{cc} is greater than $\pi/2$. the same logic is followed when using the second test that is based on the go-to-goal inner product with follow wall counter-clockwise.

Now we need to relate the wall following behavior to the go-to-goal and obstacle avoidance behaviors. In order to link our three behaviors, we are going to use the induced mode presented as a solution to the *Zeno Phenomenon* in hybrid systems. Suppose we have the situation represented in the **Figure 1.23** where x is the actual position of the robot, x_o is the obstacle position and x_g is the goal position.

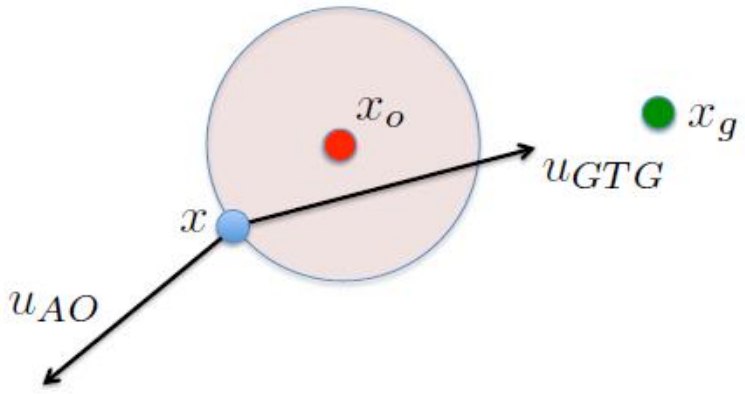


Figure 1.23: An example of navigation situation.

Let us consider *delta* Δ as the safety distance that the robot need to keep from the obstacle when following it such as:

$$\Delta = \|x - x_o\|$$

Since we are going to deal with derivatives, it is much easier to use the square of a norm than the norm itself. We define the switching surface $g(x)$as the following:

$$g(x) = \frac{1}{2}(\|x - x_o\|^2 - \Delta^2) = 0$$

In order to compute the induced mode, we have to introduce the following functionssuch as:

$$f_1(x) = C_{GTG}(x_g - x)$$

$$f_2(x) = C_{AO}(x - x_o)$$

where C_{GTG} is the go-to-goal component and C_{AO} is the avoid obstacle component. When $g(x) > 0$,

the system will be using the function $f_1(x)$ to reach the goal position. Otherwise if $g(x) < 0$, the system switches to the function $f_1(x)$ to avoid the obstacle which is represented bellow:

$$\dot{x} = \begin{cases} f_1(x) \ ; \ g(x) > 0 \\ f_2(x) \ ; \ g(x) < 0 \end{cases}$$

we have the following expression of the induced mode:

$$\dot{x} = \frac{1}{L_{f_2}g - L_{f_1}g}(L_{f_2}gf_1 - L_{f_1}gf_2)$$

we also have:

$$\frac{\partial g}{\partial x} = (x - x_o)^T$$

where we can define the bellow relations:

$$L_1 g = \frac{\partial g}{\partial x} f_1 = C_{GTG}(x - x_o)^T(x_g - x)$$

$$L_{f_2}g = \frac{\partial g}{\partial x} f_2 = (x - x_o)^T C_{AO}(x - x_o) = C_{AO}\|x - x_o\|^2$$

If we switch to the wall following behavior, we need to set conditions in order to stop following the wall otherwise the robot will keep following the wall. We want our robot to switch from the wall following behavior if enough progress has been made and if it has a clear path towards the goal.

We define τ to be the time at which we switched to the follow wall behavior. Then we can define the progress as:

$$\|x_g - x\| < \|x_g - x(\tau)\|$$

and the clear shot towards the goal condition as:

$$\langle u_{AO}, u_{GTG} \rangle > 0$$

1.4.4 Hybrid Automata of the Mobile Robot:

Finally, since we have all the necessary behaviors that makes our robot navigate from its actual position to the goal location without colliding with the different kinds of obstacles that can exist in his environment. By connecting all the previous modes and setting necessary guards, we get the final Hybrid Automata represented in the **Figure 1.24** which will be followed while implementing out mobile robot.

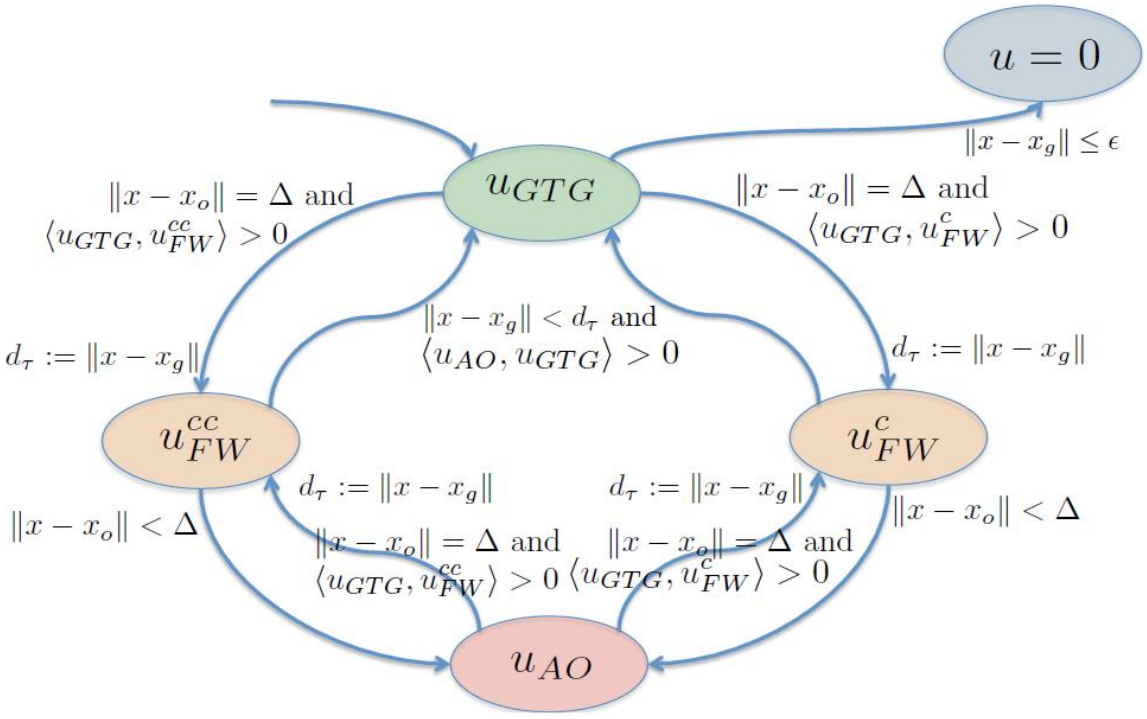


Figure 1.24: The final Hybrid Automata of the mobile robot.

Conclusion 1

In this chapter, we looked at the mathematical model of the differential drive and mapped it to the Unicycle one. After that we tackled the topic of odometry and explained the different behaviors of navigation. Finally, we considered the problem of hybrid automata and infinite switching.

CHAPTER 2

Simulation

This chapter will be about simulation where we put all of the previously discussed theory through the test in the Sim.i.am simulator. What is this simulator? How does it represent the real robot, its sensors and physical limitations? Can we achieve desired behaviors? How to solve the problems introduced before? Was it useful?

2.1 Sim.I.am: A Robot Simulator

Sim.I.am is an open-source mobile robot simulator based on MATLAB and Simulink that facilitates the implementation and design of controllers and algorithms that can be deployed on both simulated and actual mobile robots.

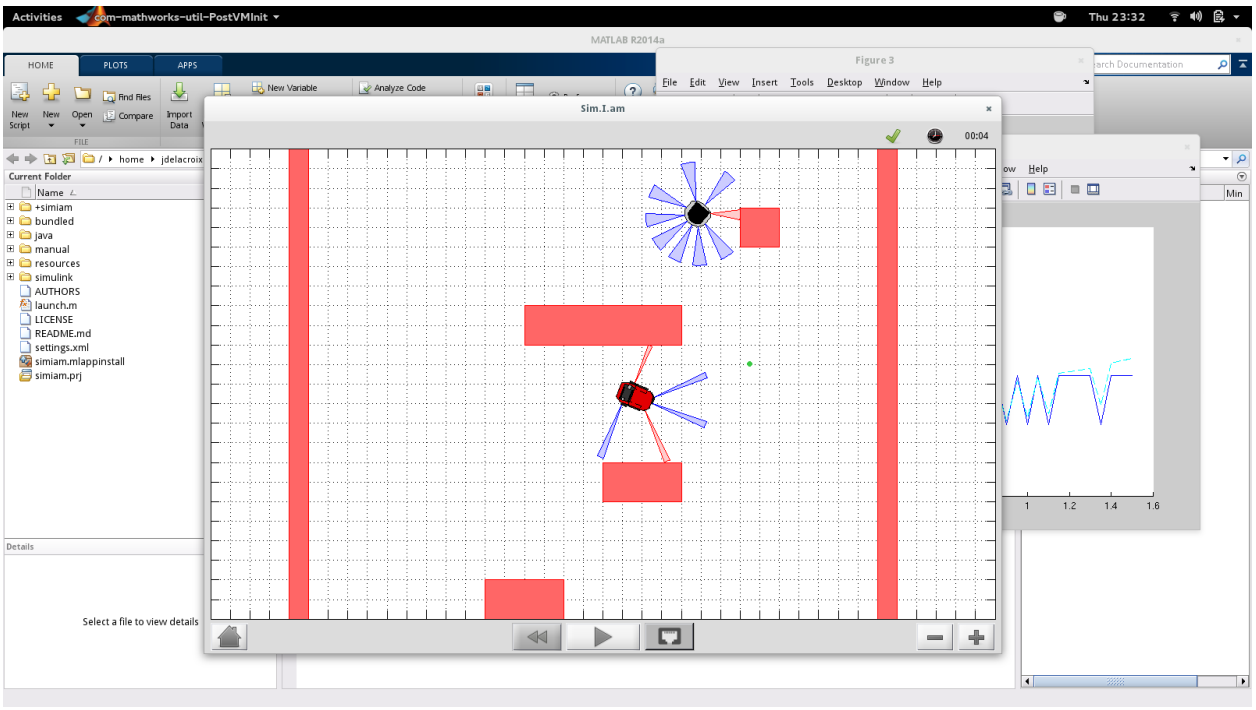
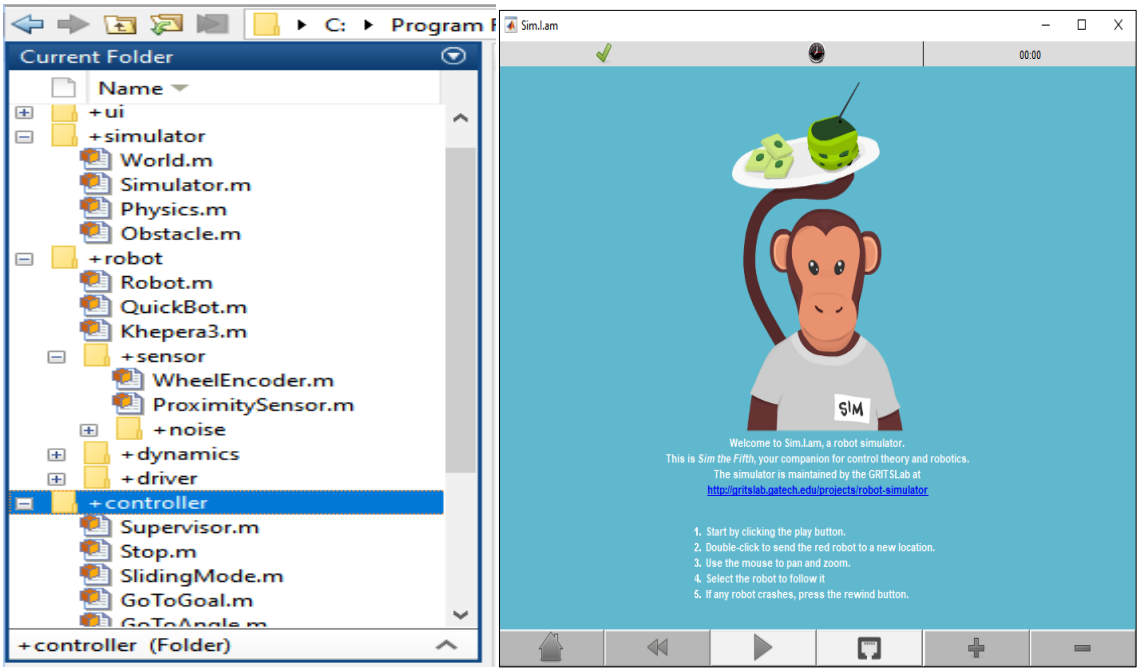


Figure 2.1: A simulation of a mobile robot and a Khepera III in the Sim.I.am simulator.

Sim.I.am is a mobile robot simulator designed to allow students to bridge the gap between theory and practice in control theory by enabling them to design and implement controllers for a mobile robot then test them in the simulator, finally deploy the code on an actual robotic Hardware such as the Khepera III mobile robot (and others) without ever having to implement code outside of MATLAB so that focus stays on the design of the controllers instead of implementation details that often derail the learning experience **[8].**

The simulated robot for this project is a differential-drive mobile robot with IR (infra-red) obstacle sensing unit, wheel encoders, and Wi-Fi connectivity. The simulator allows students to use IR sensors and wheel encoders as feedback in their controllers, and control the mobile robot via input signals to the left and right wheels of the robot.



(a) **(b)**

Figure 2.2: (**a)** File that makes up the simulator, **(b)** The user interface of the simulator.

The classes of the envirement shown in **Figure 2.1** (2 dimentional grid, border and abstacles) can be found in the **+simulator** (**Figure 2.2a**) folder whereas the code of the differential drive (red robot) and the Khepera III robots that are navigating there is implemented in the **+robot**. The wheel IR sensors encoder and the programes are in the directory **+robot/+sensor**. The work done in this project is implemented mostly in the **+controller** folder where the robot have different controllers, each designed and tuned individually and has its own file with this form Name_of_controller.m (example: GoToGoal.m).

For the sake of organizing the work, the simulator has the QBSupervisor.m file in **+controller/+quickbot** where an object of every controller is created then the logic to switch between them is maintained according to the events happening to the robot to successfully and safely navigate the envirement to the gaol location. More information about the QBSupervisor is available in **Appendix A.**

In order to text the design, the play button shown in **Figure 2.2b** that pops up after launching should be pressed to start simulation. The button at its right is used to connect the simulator to a real robot. The plus and minus bottons are used for zooming in and out the envirement. The first button in the left is used to stop simulation and rest, the one next to it is for restarting the simulation in case the robot crashes.

While the simulator is a somewhat idealized version of the real world, it provides the students with a sufficient tool to test whether their controllers are behaving correctly. If a controller did not work in the simulator, it almost assuredly would not work on the real robot. Rather than port their controller from MATLAB to C, the simulator provides a network interface (TCP/IP) that simply links the inputs/outputs from the student's controllers to the real robot instead of the simulated robot. This approach allows students to focus their attention on adapting their control design to the real robot, rather than worry about porting their controller to C. The **Sim.I.am** simulator is maintained by the **Georgia Robotics and InTelligent (GRITS) Laboratory** at the **Georgia Institute of Technology**.

2.1.1 Mobile Robot Simulator:

The simulated **Mobile Robot** equipped with five infrared (IR) range sensors, of which three are located in the front and two are located on its sides. The simulated **Mobile Robot** has a two-wheel differential drive system (two wheels, two motors) with a wheel encoder for each wheel.

Figure 2.3 shows the simulated **Mobile Robot**. The robot simulator recreates the **Mobile Robot** as faithfully as possible. For example, the range, output, and field of view of the simulated IR range sensors match the specifications in the datasheet for the actual Sharp GP2D120XJ00F infrared proximity sensors on the **Mobile Robot [9]**.

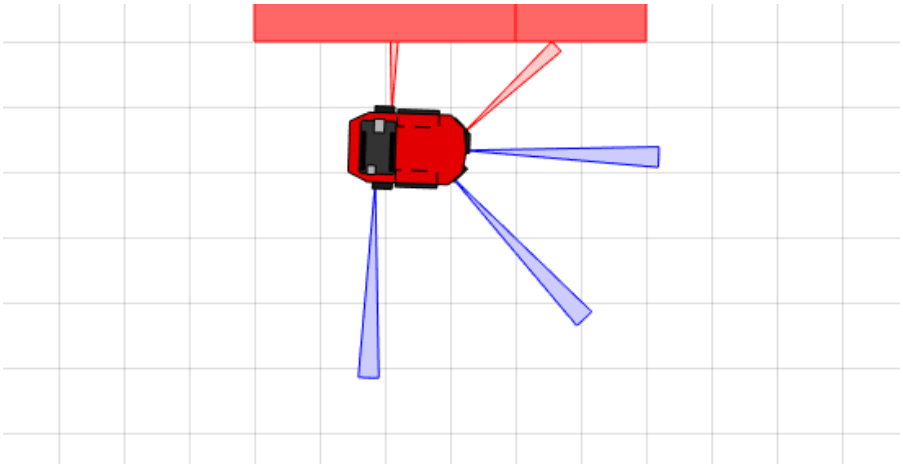


Figure 2.3: The simulated Mobile Robot.

2.1.2 IR Range Sensors Characteristics:

In this section we cover some of the details pertaining to the five simulated IR sensors onboard the simulated **Mobile Robot**. The orientations (relative to the body of the **Mobile Robot**, as shown in **Figure 2.3**) of IR sensors 1 through 5 are 90°, 45°, 0°, 45° and 90°, respectively. IR range sensors are effective in the range from 0.04 m to 0.3 m only. However, the IR sensors return raw values in the range of **[0.4, 2.75]** V instead of the measured distances. **Figure 2.4a** demonstrates the function that maps these sensors values to distances. To complicate matters slightly, the controller onboard the physical **Mobile Robot** digitizes the analog output voltage using a voltage divider and a 12-bit, 1.8V analog-to-digital converter (ADC). To faithfully recreate the **Mobile Robot** in simulation, we simulate the effect of this digitization. **Figure 2.4b** is a look-up table to demonstrate the relationship between the ADC output, the analog voltage from the IR proximity sensor, and the approximate distance that corresponds to this voltage.

Any controller can access the IR array through the robot object that is passed into its execute function. For example,

```
ir_distances = robot.get_ir_distances( );

for i=1:numel(robot.ir_array)

fprintf('IR #%d has a value of %d', i,
robot.ir_array(i).get_range());

fprintf('or %0.3f meters.\n', ir_distances(i));

end
```

It is assumed that the function `get_ir_distances` properly converts from the ADC output to an analog output voltage, and then from the analog output voltage to a distance in meters. Based on the look-up table in **Figure 2.4b,** then the conversion from analog output voltage to ADC output can be described using the following mathematical equation:

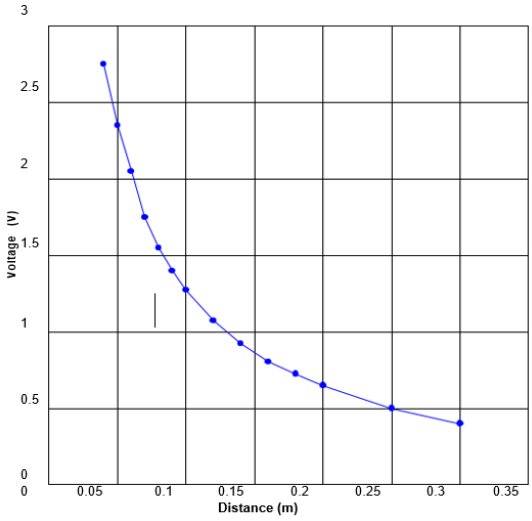
$$V_{ADC} = \frac{1000 \cdot V_{analog}}{3}$$

The simulator uses a different voltage divider on the ADC, therefore:

$$V_{ADC} = \frac{1000 \cdot V_{analog}}{2}$$

Converting from the analog output voltage to a distance is a little bit more complicated, because the relationships between analog output voltage and distance is not linear as it is demonstrated in the **Figure 2.4a.** The look-up table provides a coarse sample of points on the curve in **Figure 2.4a**.

MATLAB has a `polyfit` function to t a curve to the values in the look-up table, and a `polyval` function to interpolate a point on that fitted curve. The combination of these two functions can be used to approximate a distance based on the analog output voltage.



(a) Analog voltage output when an object is be-tween 0.04m and 0.3m in the IR proximity sensor's field of view.

Distance (m)	Voltage (V)	ADC Out
0.04	2.750	917
0.05	2.350	783
0.06	2.050	683
0.07	1.750	583
0.08	1.550	517
0.09	1.400	467
0.10	1.275	425
0.12	1.075	358
0.14	0.925	308
0.16	0.805	268
0.18	0.725	242
0.20	0.650	217
0.25	0.500	167
0.30	0.400	133

(b) A look-up table for interpolating a distance(m) from the analog (and digital) output voltages.

Figure 2.4: (a)A graph and a (b)table illustrating the relationship between the distance and output voltage of the sensor.

2.1.3 Differential Wheel Drive:

Since the simulated Mobile Robot has a differential wheel drive (i.e., is not a unicycle), it has to be controlled by specifying the angular velocities of the right and left wheel (v_r, v_l), instead of the linear and angular velocities of a unicycle (v, w).These velocities are computed by a transformation from (v, w) to (v_r, v_l). Recall that the kinematics of the unicycle are defined as:

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

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

$$\dot{\theta} = w$$

The kinematics of the differential drive are defined as:

$$\dot{x} = \frac{R}{2}(v_r + v_l)\cos(\theta)$$

$$\dot{y} = \frac{R}{2}(v_r + v_l)\sin(\theta)$$

$$\dot{\theta} = \frac{R}{L}(v_r - v_l)$$

where R is the radius of the wheels and L is the distance between the wheels.

The speed of the simulated **Mobile Robot** can be set in the following way assuming that the `uni_to_diff` function has been implemented, which transforms (v, w) to (v_r, v_l):

```
v = 0.15; % m/s
w = pi/4; % rad/s

%   Transform from v,w to v_r,v_l and set the speed of the robot
[vel_r,  vel_l]  =  obj.robot.dynamics.uni_to_diff(robot,v,w);
obj.robot.set_speeds(vel_r, vel_l);
```

The maximum angular wheel velocity for the physical **Mobile Robot** is approximately 80 RPM or 8.37 rad/s and this value is reflected in the simulator. It is therefore important to note that if the simulated **Mobile Robot** is controlled to move at maximum linear velocity, it is not possible to achieve any angular velocity, because the angular velocity of the wheel will have been maximized. Therefore, there exists a tradeoff between the linear and angular velocity of the **Mobile Robot**: the faster the robot should turn, the slower it has to move forward.

2.1.4 Wheel Encoders:

Each of the wheels is outfitted with a wheel encoder that increments or decrements a tick counter depending on whether the wheel is moving forward or backwards, respectively. Wheel encoders may be used to infer the relative pose of the simulated robot. This inference is called **odometry**. The relevant information needed for odometry is the radius of the wheel (32.5mm), the distance between the wheels (99.25mm), and the number of ticks per revolution of the wheel (16 ticks/rev). For example,

```
R = robot.wheel_radius; % radius of the wheel
L = robot.wheel_base_length; % distance between the wheels

tpr = robot. encoders(1).ticks_per_rev; % ticks per revolution
for the right wheel
fprintf('The right wheel has a tick count of %d\n',
robot.encoders(1).state);
fprintf('The left wheel has a tick count of %d\n',
robot.encoders(2).state);
```

2.2 Differential Drive

We start by Implementing the transformation from unicycle kinematics to differential drive kinematics, i.e. convert from (v,w) to the right and left **angular** wheel speeds (v_r,v_l).

In the simulator, (v,w) corresponds to the variables v and w, while (v_r,v_l) correspond to the variables `vel_r` and `vel_l`. The function used by the controllers to convert from unicycle kinematics to differential drive kinematics is named uni_to_diff, and inside of this function you will need to de ne `vel_r`(v_r) and `vel_l`(v_l) in terms of `v, w, R`, and `L`. `R` is the radius of a wheel, and `L` is the distance separating the two wheels.

```
function [vel_r,vel_l] = uni_to_diff(obj,v,w)
  R = obj.wheel_radius;
  L = obj.wheel_base_length;

  vel_r = (2*v+w*L)/(2*R);
  vel_l = (2*v-w*L)/(2*R);

end
```

2.3 Odometry

We Implement now the odometry for the robot, such that as the robot moves around, its pose (x,y,θ) is estimated based on how far each of the wheels have turned. We Assume that the robot starts at $(0,0,0)$.

As seen in the first chapter, the general idea behind odometry is to use wheel encoders to measure the distance the wheels have turned over a small period of time, and use this information to approximate the change in pose of the robot.

The pose of the robot is composed of its position (x,y) and its orientation θ on a 2-dimensional plane. The currently estimated pose is stored in the variable `state_estimate`, which bundles `x`, `y`, and `theta`. The robot updates the estimate of its pose by calling the `update_odometry` function which is called every dt seconds, where dt is 0.033s (or a little more if the simulation is running slower).

```
% Get wheel encoder ticks from the robot
        right_ticks = obj.robot.encoders(1).ticks;
        left_ticks = obj.robot.encoders(2).ticks;

% Recall the previous wheel encoder ticks
        prev_right_ticks = obj.prev_ticks.right;
        prev_left_ticks = obj.prev_ticks.left;

% Previous estimate
[x, y, theta] = obj.state_estimate.unpack();
```

```
% Compute odometry here
        R = obj.robot.wheel_radius;
        L = obj.robot.wheel_base_length;
        m_per_tick = (2*pi*R)/obj.robot.encoders(1).ticks_per_rev;
```

where `right_ticks` and `left_ticks` are the accumulated wheel encoder ticks of the right and left wheel. `prev_right_ticks` and `prev_left_ticks` are the wheel encoder ticks of the right and left wheel saved during the last call to update odometry. `R` is the radius of each wheel, and `L` is the distance separating the two wheels. `m_per_tick` is a constant that tells you how many meters a wheel covers with each tick of the wheel encoder. So, we multiply `m_per_tick` by `(right_ticks-prev_right_ticks)` to get the distance travelled by the right wheel since the last estimate.

```
% Calculate the distance travelled by the robot wheels
  d_right = (right_ticks - prev_right_ticks)* m_per_tick;
  d_left = (left_ticks - prev_left_ticks)* m_per_tick;

  d_center = (d_right + d_left)/2;
  phi = (d_right - d_left)/L;

  x_dt = d_center*cos(theta);
  y_dt = d_center*sin(theta);
  theta_dt = phi;
```

Once we have computed the change in(x, y, θ) (let us denote the changes as `x_dt`, `y_dt`, and `theta_dt`), you need to update the estimate of the pose:

```
% Update the estimate of the pose
  theta_new = theta + theta_dt;
  x_new = x + x_dt;
  y_new = y + y_dt;
fprintf('Estimated(x,y,theta):(%0.3g,%0.3g,%0.3g)\n',x_new,y_ne
w,theta_new);
% Save the wheel encoder ticks for the next estimate
      obj.prev_ticks.right = right_ticks;
      obj.prev_ticks.left = left_ticks;

% Update your estimate of (x,y,theta)
      obj.state_estimate.set_pose([x_new,
      y_new,atan2(sin(theta_new),cos(theta_new))]);
```

2.4 IR Distance Sensors

We use the table in **Figure 2.4b** in the "IR Range Sensors" section of the second chapter, which maps distances (in meters) to raw IR values. Then, we implement code that converts raw IR values to distances (in meters).

To retrieve the distances (in meters) measured by the IR proximity sensor, we need to implement a conversion from the raw IR values to distances in the `get_ir_distances` function.

```
function ir_distances = get_ir_distances(obj)
    ir_array_values = obj.ir_array.get_range();
    ir_voltages = ir_array_values*3/1000;
    coeff = [-0.0182 0.1690 -0.6264 1.1853 -1.2104 0.6293];
    ir_distances = polyval(coeff, ir_voltages);
end
```

The variable `ir_array_values` is an array of the IR raw values. The `coeff` variable contains the coefficients returned by:

```
polyfit(ir_voltages_from_table,ir_distances_from_table,5);
```

where the first input argument is an array of IR voltages from the table in **Figure 2.4b** and the second argument is an array of the corresponding distances from the table in **Figure 2.4b**. The third argument specifies that we will use a fifth-order polynomial to fit to the data. Instead of running this fit every time, we execute the polyfit once in the MATLAB command line, and enter them manually on the third line, i.e. `coeff = [ ... ];`.

2.5 Motor Limitations

We have two limitations of the motors on the physical **Mobile Robot** (which are simulated on the **Mobile Robot** we use in simulation). The first limitation is that the robot's motors have a maximum angular velocity, and the second limitation is that the motors stall at low speeds. Suppose that we pick a linear velocity v that requires the motors to spin at 90% power. Then, we want to change w from 0 to some value that requires 20% more power from the right motor, and 20% less power from the left motor. This is not an issue for the left motor, but the right motor cannot turn at a capacity greater than 100%. The results are that the robot cannot turn with the w specified by our controller.

Since our PID controllers focus more on steering than on controlling the linear velocity, we want to prioritize w over v in situations where we cannot satisfy w with the motors. In fact, we will simply reduce v until we have sufficient headroom to achieve w with the robot. The function `ensure_w` is designed to ensure that w is achieved even if the original combination of v and w exceeds the maximum v_r and v_l. However, it is also true that the motors have a minimum speed before the robot starts moving. If no enough power is applied to the motors,

the angular velocity of a wheel remains at 0. Once enough power is applied, the wheels spin at a speed vel_{min}. The `ensure_w` function will also take this limitation into account. For example, small (v,w) may not be achievable on the **Mobile Robot**, so `ensure_w` function scales up v to make w possible. Similarly, if (v,w) are both large, `ensure_w` scales down v to ensure w.

Suppose $v_{r,d}$ and $v_{l,d}$ are the angular wheel velocities needed to achieve w. Then `vel_rl_max` is $max(v_{r,d}, v_{l,d})$ and `vel_rl_min` is $min(v_{r,d}, v_{l,d})$. A motor's maximum forward angular velocity is `obj.robot.max_vel`(or vel_{max}). So, for example, the equation that represents the if/else statement for the right motors is:

$$v_r = \begin{cases} v_{r,d} - \left(max\left(v_{r,d}, v_{l,d}\right) - vel_{max}\right) & , if\ max\left(v_{r,d}, v_{l,d}\right) > vel_{max} \\ v_{r,d} - \left(min\left(v_{r,d}, v_{l,d}\right) + vel_{max}\right) & , if\ min\left(v_{r,d}, v_{l,d}\right) < -vel_{max} \\ v_{r,d}, & , \quad otherwise \end{cases}$$

which defines the appropriate v_r (or `vel_r`) needed to achieve w. This equation also applies to computing a new v_l. The results of ensure w is that if v and w are so large that v_r and/or v_l exceed vel_{max}, then v is scaled back to ensure w is achieved.

```
% Find the max and min vel_r/vel_l
    vel_rl_max = max(vel_r_d, vel_l_d);
    vel_rl_min = min(vel_r_d, vel_l_d);

%Shift vel_r and vel_l if they exceed max/min vel
    if (vel_rl_max > vel_max)
        vel_r = vel_r_d - (vel_rl_max-vel_max);
        vel_l = vel_l_d - (vel_rl_max-vel_max);
    elseif (vel_rl_min < vel_min)
        vel_r = vel_r_d + (vel_min-vel_rl_min);
        vel_l = vel_l_d + (vel_min-vel_rl_min);
    else
        vel_r = vel_r_d;
        vel_l = vel_l_d;
```

2.6 Controllers

2.6.1 Go-To-Goal Controller:

We implement the Go-To-Goal Controller using the different parts of a PID regulator that steers the robot successfully to some goal location. This is known as the go-to-goal behavior.

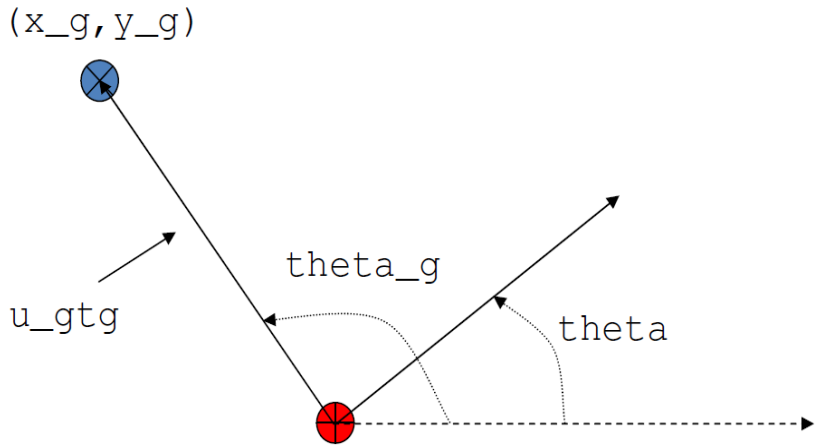


Figure 2.5: Steering the Mobile Robot to the goal location (x_g, y_g) with heading angle θ_g.

We calculate the heading angle θ_g , to the goal location (x_g, y_g). Let u be the vector from the robot located at (x, y) to the goal located at (x_g, y_g), then θ_g is the angle u makes with the $x - axis$ (positive $\theta_g$is in the counterclockwise direction).

The vector u can be expressed in terms of its components along the x and y axis (u_x, u_y). In the code they represent `u_x` and `u_y` . We use these two components and the `atan2` function (to make sure θ_g stays in $[-\pi, \pi]$) to compute the angle to the goal θ_g (`theta_g` in the code).

The `atan2` function returns the four-quadrant inverse tangent (tan^{-1}) of `Y` and `X`, which must be real. The `atan2` function follows the convention that `atan2(x,x)` returns 0.

```
% distance between goal and robot in x-direction
      u_x = x_g - x;
% distance between goal and robot in y-direction
      u_y = y_g - y;
% angle from robot to goal. Hint: use ATAN2, u_x, u_y here.
      theta_g = atan2(u_y,u_x);
```

We calculate the error between the heading to the goal `theta_g` θ_g and the current heading of the robot `theta`θ which is represented by the error `e_k`.

```
% error between the goal angle and robot's angle

e_k = theta_g-theta;
e_k = atan2(sin(e_k), cos(e_k));
```

We finally calculate the proportional, integral, and derivative terms for the PID regulator that steers the robot to the goal.

The PID regulator will steer the robot to the goal, i.e. compute the correct angular velocity w. The PID regulator needs three parts implemented:

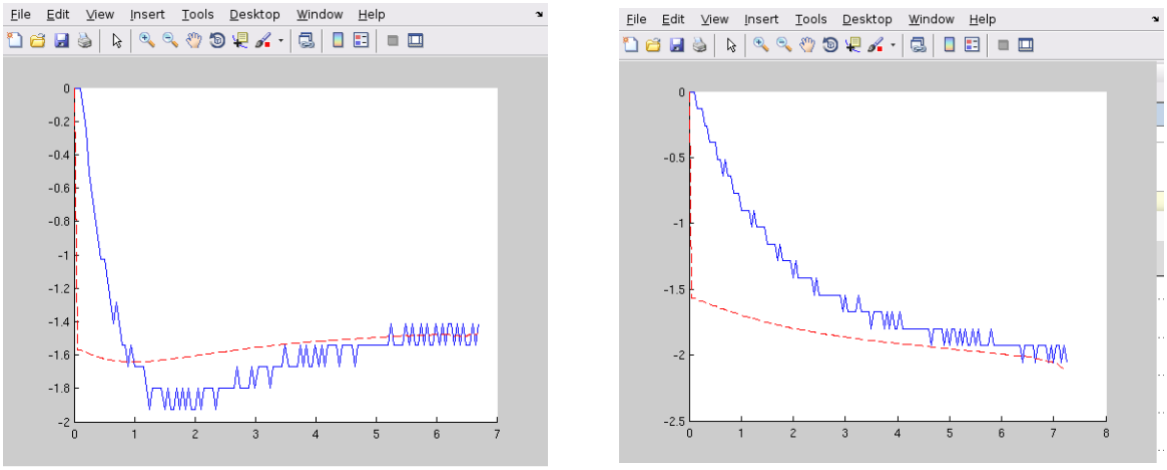
(i) The first part is the proportional term `e_P` which is simply the current error `e_k`. `e_P` is multiplied by the proportional gain `obj_Kp` when computing `w`.

(ii) The second part is the integral term `e_I`. The integral needs to be approximated in discrete time using the total accumulated error `obj.E_k`, the current error `e_k`, and the time step `dt`. `e_I` is multiplied by the integral gain `obj_Ki` when computing `w`, and is also saved as `obj.E_k` for the next time step.

(iii) The third part is the derivative term `e_D`. The derivative needs to be approximated in discrete time using the current error `e_k`, the previous error `obj.e_k_1`, and the time step `dt`. `e_D` is multiplied by the derivative gain `obj.Kd` when computing w, and the current error e k is saved as the previous error `obj.e_k_1` for the next time step.

We need to tune our PID gains to get a fast settle time ($\theta_g$matches θ within 10% in three seconds or less) and there should be little overshoot (maximum θ should not increase beyond 10% of the reference value θ_g). What you don't want to see are the following two graphs when the robot tries to reach goal location (x_g, y_g) = $(0, 1)$:

Figure 2.6b demonstrates undershoot, which could be fixed by increasing the proportional gain or adding some integral gain for better tracking. Picking better gains leads to the graph in **Figure 2.7.**



(a)Overshoot (b) Undershoot (slow settle time)

Figure 2.6: PID gains were picked poorly, which lead to (a)Overshoot and (b) Undershoot.

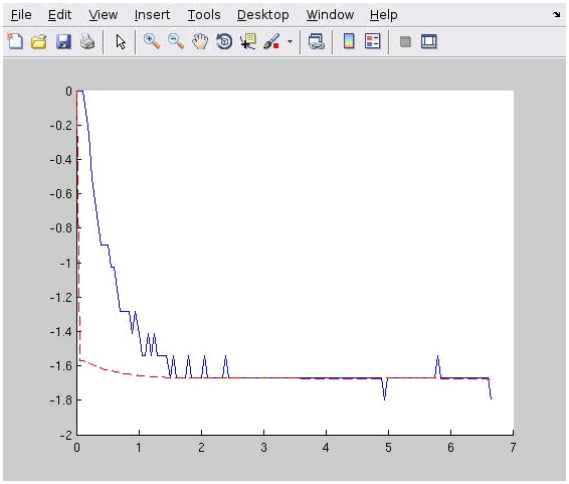


Figure 2.7: Faster settle time and good tracking with little overshoot.

We write the following code to implement the Go-To-Goal controller using the PID regulator:

```
% error for the proportional term
    e_P = e_k;
% error for the integral term.
    e_I = obj.E_k + e_k*dt;
% error for the derivative term.
    e_D = (e_k-obj.e_k_1)/dt;
    w = obj.Kp*e_P + obj.Ki*e_I + obj.Kd*e_D;
% Save errors for next time iteration
   obj.E_k = e_I;
   obj.e_k_1 = e_k;
```

2.6.2 Obstacle Avoidance Controller

We will be implementing the different parts of a controller that steers the robot successfully away from obstacles to avoid a collision. This is known as the avoid-obstacles behavior. The IR sensors allow us to measure the distance to obstacles in the environment, but we need to compute the points in the world to which these distances correspond**. Figure 3.8** illustrates these points with a black cross.

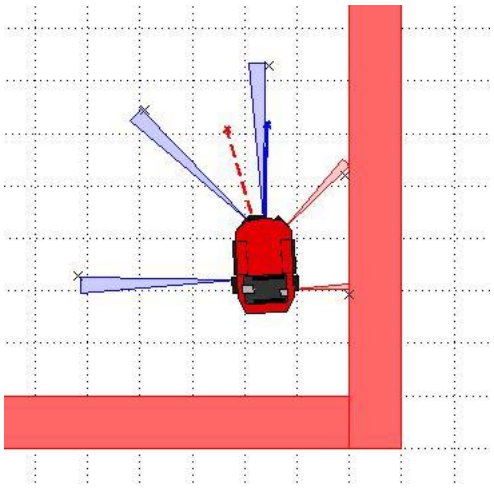


Figure 2.8: IR range to point transformation.

The strategy for obstacle avoidance that we will use is as follows:

1. Transform the IR distances to points in the world.
2. Compute a vector to each point from the robot, $u_1, u_2, \dots, u_5$.
3. Weigh each vector according to their importance, $\alpha_1 u_1, \alpha_2 u_2, \dots, \alpha_5 u_5$. For example, the front and side sensors are typically more important for obstacle avoidance while moving forward.
4. Sum the weighted vectors to form a single vector,

$$u_{ao} = \alpha_1 u_1 + \alpha_2 u_2 + \cdots + \alpha_5 u_5$$

5. Use this vector to compute a heading and steer the robot to this angle.

This strategy will steer the robot in a direction with the freest space (i.e., it is a direction away from obstacles). For this strategy to work, we will need to implement three crucial parts of the strategy for the obstacle avoidance behavior:

Firstly, we will transform the IR distance (which we have converted from the raw IR values in IR Distance Sensors) measured by each sensor to a point in the reference frame of the robot:

A point p_i that is measured to be d_i meters away by sensor i can be written as the vector (coordinate) $v_i = \begin{bmatrix} d_i \\ 0 \end{bmatrix}$ in the reference frame of sensor i. We first need to transform this point to be in the reference frame of the robot. To do this transformation, we need to use the pose (location and orientation) of the sensor in the reference frame of the robot: $(x_{s_i}, y_{s_i}, \theta_{s_i})$ or in code, `(x_s, y_s, theta_s)`. The transformation is defined as:

$$v_i' = R(x_{s_i}, y_{s_i}, \theta_{s_i}) \begin{bmatrix} v_i \\ 0 \end{bmatrix},$$

where R is known as the transformation matrix that applies a translation by (x, y) and a rotation θ by:

$$R(x, y, \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}$$

which we will implement in the function `obj.get_transformation_matrix`.

We will also need to implement the transformation in the `apply_sensor_geometry` function. The objective is to store the transformed points in `ir_distances_rf`, such that this matrix has v_1' as its rst column, v_2' as its second column, and so on.

```
function ir_distances_wf = apply_sensor_geometry(obj, ir_distances,
state_estimate)
% Apply the transformation to robot frame.

            ir_distances_rf = zeros(3,5);
    for i=1:5
                x_s = obj.sensor_placement(1,i);
                y_s = obj.sensor_placement(2,i);
                theta_s = obj.sensor_placement(3,i);

                R = obj.get_transformation_matrix(x_s,y_s,theta_s);
                ir_distances_rf(:,i) = R*[ir_distances(i); 0; 1];
    end
end
```

Secondly, we transform the point in the robot's reference frame to the world's reference frame:

A second transformation is needed to determine where a point p_i is located in the world that is measured by sensor i. We need to use the pose of the robot, (x, y, θ), to transform the robot from the robot's reference frame to the world's reference frame. This transformation is defined as:

$$v_i'' = R(x, y, \theta)v_i'$$

We need also to implement this transformation in the `apply_sensor_geometry` function. The objective here is to store the transformed points in `ir_distances_wf`, such that this matrix has v_1'' as its first column, v_2'' as its second column, and so on. This matrix now contains the coordinates of the points illustrated in **Figure 2.8** by the black crosses these points approximately correspond to the distances measured by each sensor approximately (because of how we converted from raw IR values to meters).

```
% Apply the transformation to world frame.

      [x,y,theta] = state_estimate.unpack();

      R = obj.get_transformation_matrix(x,y,theta);
      ir_distances_wf = R*ir_distances_rf;

      ir_distances_wf = ir_distances_wf(1:2,:);
```

Finally, we use the set of transformed points to compute a vector that points away from the obstacle. The robot will steer in the direction of this vector and successfully avoid the obstacle:

In the function `execute`, we implement the following strategy:

(i) We compute a vector u_i to each point (corresponding to a particular sensor) from the robot. Use a point's coordinate from `ir_distances_wf` and the robot's location (x, y) for this computation.

(ii) We pick a weight α_i for each vector according to how important the particular sensor is for obstacle avoidance. For example, if you were to multiply the vector from the robot to point i (corresponding to sensor i) by a small value (e.g., 0.1), then sensor i will not impact obstacle avoidance significantly. We need to make sure that the weights are symmetric with respect to the left and right sides of the robot. Without any obstacles around, the robot should only steer slightly right (due to a small asymmetry in the how the IR sensors are mounted on the robot).

(iii) We sum up the weighted vectors, $\alpha_i u_i$, into a single vector u_{ao}.

(iv) We use u_{ao} and the pose of the robot to compute a heading that steers the robot away from obstacles (i.e., in a direction with free space, because the vectors that correspond to directions with large IR distances will contribute the most to u_{ao}).

```
% Compute the heading vector for obstacle avoidance

    sensor_gains = [.7 1.5 0.5 1.5 .7];
    u_i = (ir_distances_wf-repmat ([x; y],1,5))*diag(sensor_gains);
    u_ao = sum(u_i,2);
```

2.6.3 AOandGTG (Blending) Controller:

We will combine the two previous controllers into a single controller: The Goal-to-Goal and Avoid obstacles Controllers. The AOandGTG controller will allow the robot to drive to a goal, while not colliding with any obstacles on the way

To implement our blending controller, we need to combine two vectors: u_{gtg}(the vector pointing to the goal from the robot) and u_{ao}(the vector pointing from the robot to a point in space away from obstacles). These two vectors need to be combined (blended) in some way into the vector $u_{ao,gtg}$, which is the vector that points the robot both away from obstacles and towards the goal.

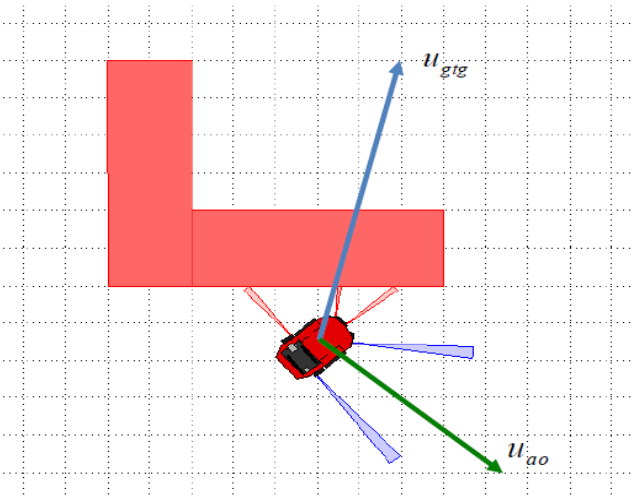


Figure 2.9: The u_{gtg} and u_{ao}vectors pointing out of the mobile robot.

The linear combination of the two vectors u_{gtg} and u_{ao} will yield to the vector $u_{ao,gtg}$ which result in the robot driving to a goal without colliding with any obstacles in the way is computed our two vectors We need to weigh each vector according to their importance:

$$u_{ao,gtg} = \alpha u_{gtg} + (1 - \alpha)u_{ao} \qquad ; where\ 0 < \alpha < 1$$

The following code implement the previous equations:

```
% Blending the two vectors
      alpha = 0.25;
      u_ao_gtg = alpha*u_gtg+(1-alpha) *u_ao;
```

By setting the goal location $(1, 1)$, the robot will navigate successfully to the goal without colliding with the obstacle that is in the way. The output plot will likely look something similar to:

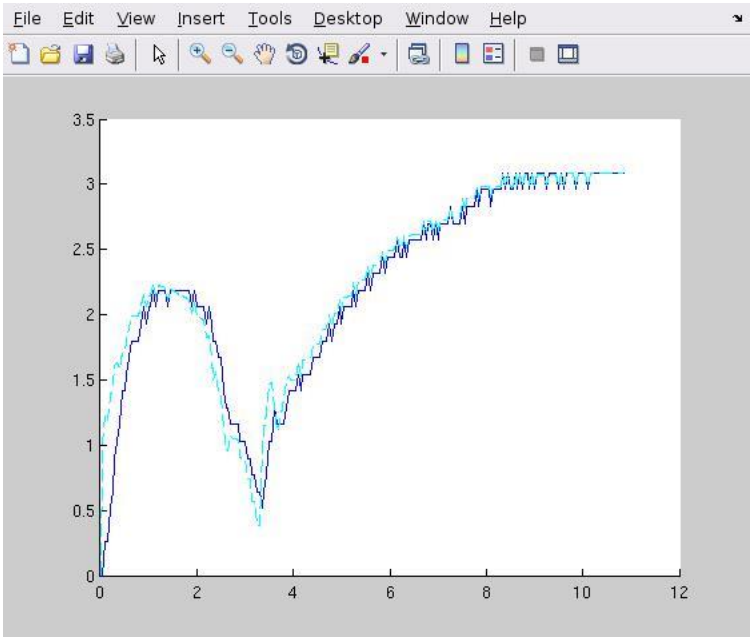


Figure 2.10: The AOandGTG controller output resulting from the specified goal location.

2.6.4 Wall-Following Controller:

We will be implementing a wall following behavior that will aid the robot in navigating around different kind of obstacles that we have stated in the first chapter.

We start by computing a vector $u_{fw,t}$, that estimates a section of the obstacle next to the robot using the robot's right (or left) IR sensors. In the **Figure 3.7**, this vector, $u_{fw,t}$ (`u_fw_t`), is illustrated in red.

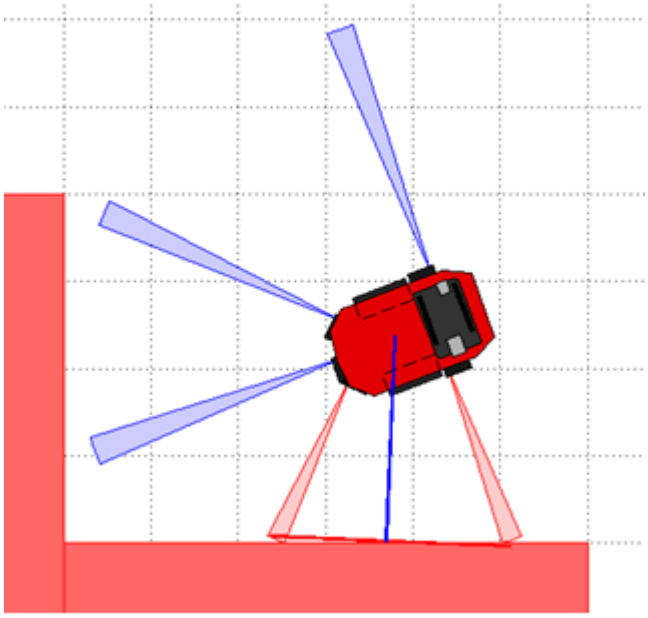
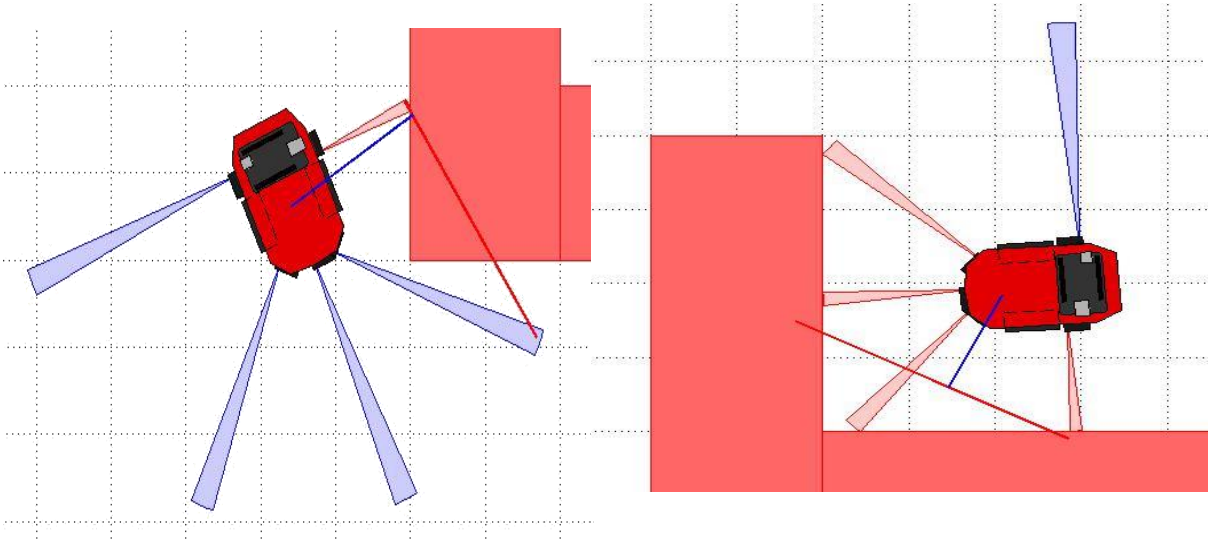


Figure 2.11: The illustration of the $u_{fw,t}$ vector.

The direction of the wall following behavior (whether it is following the obstacle on the left or right) is determined by `inputs.direction`, which can either be equal to `right` or `left`. Suppose we want to follow an obstacle to the left of the robot, then we use the left set of IR sensors (1-3). If we are following the wall, then at all times there should be at least one sensor that can detect the obstacle. So, we need to pick a second sensor and use the points corresponding to the measurements from these two sensors to form a line that estimates a section of the obstacle. In the **Figure 2.11** above, sensors 1 and 2 are used to roughly approximate the edge of the obstacle.

Corners are trickier (see **Figure 2.8**), because typically only a single sensor will be able to detect the wall. The estimate is off as one can see in the **Figure 2.8**, but as long as the robot isn't following the wall too closely, it will be ok.



(a)Outside Corner (b) Inside Corner

Figure 2.12: The illustration of the $u_{fw,t}$ vector near a corner.

If we want to estimate a section of the wall using the right sensors (from IR sensors 1-3), we need to pick the two sensors with the smallest reported measurement in `ir_distances`. Suppose sensor 2 and 3 returned the smallest values, then it is important that the sensor with smaller ID (we assume it is sensor 2) is assigned to p_1 (`p_1`) and the sensor with the larger ID (we assume it is sensor 3) is assigned to p_2 (`p_2`), because we want that the vector points in the direction that the robot should travel, then let:

```
p1 = ir_distances_wf(:,2)
p2 = ir_distances_wf(:,3)
```

Let us assume $u_{fw,t}$ as the vector that estimates a section of the obstacle such as:

$$u_{fw,t} = p2 - p$$

Now that we have the vector $u_{fw,t}$ (represented by the red line in the Figures), we need to compute a vector $u_{fw,p}$ that points from the robot to the closest point on $u_{fw,t}$. This vector isvisualized as blue line in the Figures and can be computed using a little bit of linear algebra:

$$u'_{fw,t} = \frac{u_{fw,t}}{\|u_{fw,t}\|}, \quad u_p = \begin{bmatrix} x \\ y \end{bmatrix}, \quad u_a = p_1$$

$$u_{fw,p} = (u_a - u_p) - ((u_a - u_p) \cdot u'_{fw,t})u'_{fw,t}$$

where $u_{fw,p}$ corresponds to `u_fw_p` and $u'_{fw,t}$ corresponds to `u_fw_tp` in the code. You can notice a small technicality which that we are computing $u_{fw,p}$ as the vector pointing from the robot to the closest point on $u_{fw,t}$, as if $u_{fw,t}$ were infinitely long.

All the vectors used in the previous equations are illustrated in the **Figure 3.13.**

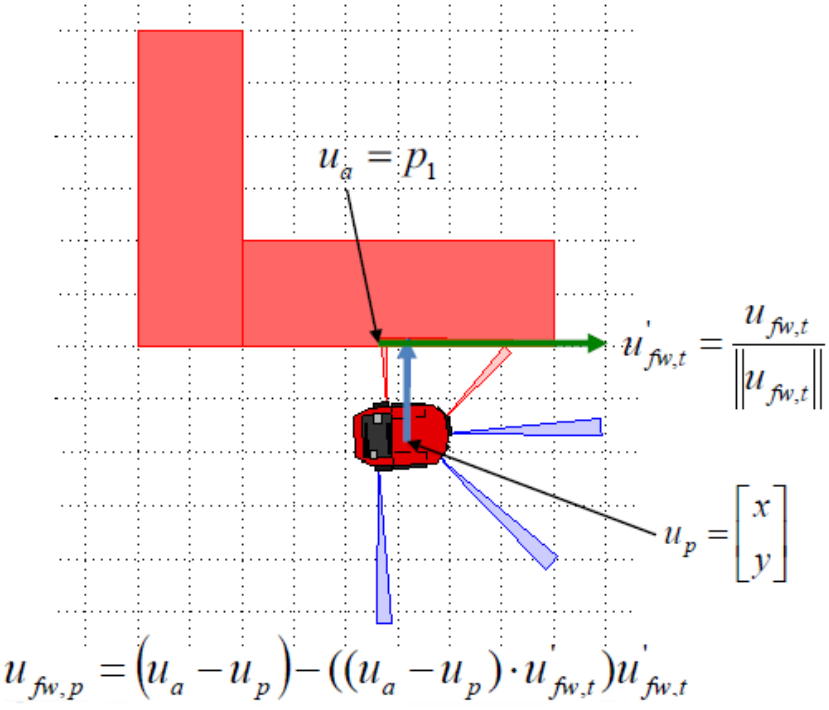


Figure 2.13: The illustration of the $u'_{fw,t}$ and $u_{fw,p}$vectors near.

The last step is to combine $u_{fw,t}$ and $u_{fw,p}$ such that the robot follows the obstacle all the way around at some distance d_{fw} (`d_fw`). $u_{fw,t}$ will ensure that the robot drives in a direction that is parallel to an edge on the obstacle, while $u_{fw,p}$ needs to be used to maintain a distance d_{fw} from the obstacle.

One way to achieve this is,

$$u'_{fw,p} = u_{fw,p} - d_{fw}\frac{u_{fw,p}}{\|u_{fw,p}\|}$$

Suppose the d is the distance between the robot and the obstacle, then the vector $u'_{fw,p}$(u_fw_pp) is:

- Pointing towards the obstacle if: $d > d_{fw}$.
- Near zero if:$d \simeq d_{fw}$.
- Pointing away from the obstacle if: $d < d_{fw}$.

All that is left is to linearly combine $u'_{fw,t}$ and $u'_{fw,p}$ into a single vector u_{fw} (`u_fw`) that can be used with the PID controller to steer the robot along the obstacle at the distance d_{fw}.

$$u_{fw} = \alpha u'_{fw,t} + \beta u'_{fw,p}$$

Figure 2.14 illustrates the u_{fw}, $u'_{fw,t}$ and $u'_{fw,p}$vectors.

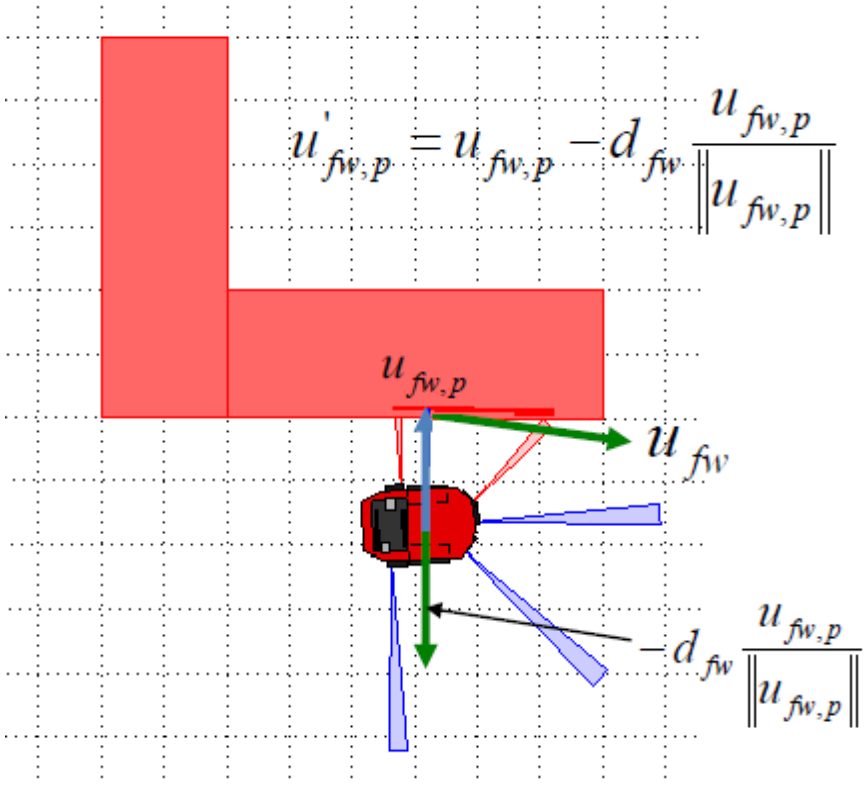


Figure 2.14: The illustration of the u_{fw}, $u'_{fw,t}$ and $u'_{fw,p}$vectors.

Finally, we implement all the steps above in the code bellow:

```
% Selecting p_2 and p_1, then compute u_fw_t
if(strcmp(inputs.direction,'right'))
% Pick two of the right sensors based on ir_distances
             S = [1:3 ; ir_distances(5:-1:3)'];
             [Y,i] = sort(S(2,:));
             S = S(1,i);
             Sp = 5:-1:3;

             S1 = Sp(S(1));
             S2 = Sp(S(2));

if(S1 < S2)
                    p_1 = ir_distances_wf(:,S2);
                    p_2 = ir_distances_wf(:,S1);
else
                    p_1 = ir_distances_wf(:,S1);
                    p_2 = ir_distances_wf(:,S2);
end
else
% Pick two of the left sensors based on ir_distances
             S = [1:3 ; ir_distances(1:3)'];
             [Y,i] = sort(S(2,:));
             S = S(1,i);
             Sp = 1:3;

             S1 = Sp(S(1));
             S2 = Sp(S(2));

if(S(1) > S(2))
                    p_1 = ir_distances_wf(:,S(2));
                    p_2 = ir_distances_wf(:,S(1));
else
                    p_1 = ir_distances_wf(:,S(1));
                    p_2 = ir_distances_wf(:,S(2));
end
end
        u_fw_t = p_2-p_1;

% Computing u_a, u_p, and u_fw_tp to compute u_fw_p
u_fw_tp = u_fw_t/norm(u_fw_t);
u_a = p_1;
         u_p = [x;y];
         u_fw_p = ((u_a-u_p)-((u_a-u_p)'*u_fw_tp)*u_fw_tp);

% Combining u_fw_tp and u_fw_pp into u_fw;
         u_fw_pp = u_fw_p/norm(u_fw_p);
         u_fw = d_fw*u_fw_tp+(u_fw_p-d_fw*u_fw_pp);
```

The above function takes in the direction of wall following as an input when it is called in the QBSupervisor.m file using the `sliding_right(inputs.direction')` and `sliding_left (inputs.direction)` functions, where `inputs.direction` is equal to either `'left'` or `'right'`. Finally, we compute `u_fw` for each side.

Once the follow wall vector is returned, we want to know whether we need to follow the wall or not by knowing if the obstacle is in our way to the goal. We do that by solving for σ_1 and σ_2 in the following equation:

$$[u_{gtg} \quad u_{ao}] \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = u_{fw}$$

We want to write u_fw as a linear combination of `u_gtg` and `u_ao`, we return true if both σ_1 and σ_2 are positive; meaning that `u_fw` lays between the GTG vector and the AO vector where the `u_gtg` is driving the robot to the obstacle that `u_ao` is driving it away from.

```
A=[u_gtg u_ao];
sigma = inv(A)*u_fw;

slide = false;
if sigma(1) > 0 && sigma(2) > 0
slide = true;
end
```

We use `slide` to decide whether we start following the wall and in what direction. Again, we change `e_k` to:

```
theta_fw = atan2(u_fw(2), u_fw(1));
e_k = theta_fw-theta;
e_k = atan2(sin(e_k), cos(e_k));
```

Using Wall-Following controller may ensure following the path at a certain distance from an obstacle but it does not take the robot to the goal location; so, we just need it to get to the other side of an obstacle until we can break away and switch to going to goal.

The conditions that must be satisfied in order to stop following the wall are:

- The position of the robot is closer to the goal than it was when it started following the wall.
- The angle between u_ao and u_gtg is smaller than 60 degrees (Tunable).

To check the first condition we save the distance from the goal to the point that the robot started following the wall `d_prog` via function `set_progress_point()`(called every time before switching to go to goal),

```
function set_progress_point_new(obj)
    [x, y, theta] = obj.state_estimate.unpack();
    obj.d_prog = (norm([x-obj.goal(1);y-obj.goal(2)]));
end
```

then we compare it with the distance from the goal to the current position of the robot at every update via `progress_made()`. If it returns true then the first condition is satisfied.

```
function rc = progress_made(obj, state, robot)

        % Check for any progress
        [x, y, theta] = obj.state_estimate.unpack();
        rc = false;

        distance =[x-obj.goal(1);y-obj.goal(2)];

        if (norm(distance)<(obj.d_prog -0.1))
           rc = true;
        end
end
```

To check the second condition, we make a simple computation:

```
function rc = check_angel_gtg_ao(u_ao, u_gtg)

         th1 =  atan2(u_ao(2),u_ao(1))*180/pi ;
         th_gtg = atan2(u_gtg(2),u_gtg(1)) *180/pi;
         th_diff =  min([abs(th1 - th_gtg),abs(th1 - th_gtg+360),
                                 abs(th1 - th_gtg-360)]) ;
            rc = false;

            if (th_diff < 60)
            rc = true;
            end
   end
```

2.7 Tests and results

Now that we know how each controller work wall, we need to know when to use them. Before starting simulation we initialize the coordinates of starting point of the robot (in the xml file that describes the environment), the goal it wants to reach and set the current controller that it should drive with to the Go_To_Goal in the QBSupervisor.m file which is responsible for counting and updating the current pose of the robot, checking the events happening and switching between controllers accordingly. The robot will try to navigate the environment in **Figure 2.15.**

Where the red rectangles are obstacles and the green circle is the goal point which is now the origin (0,0). Every rectangle represents 0.1 cm.

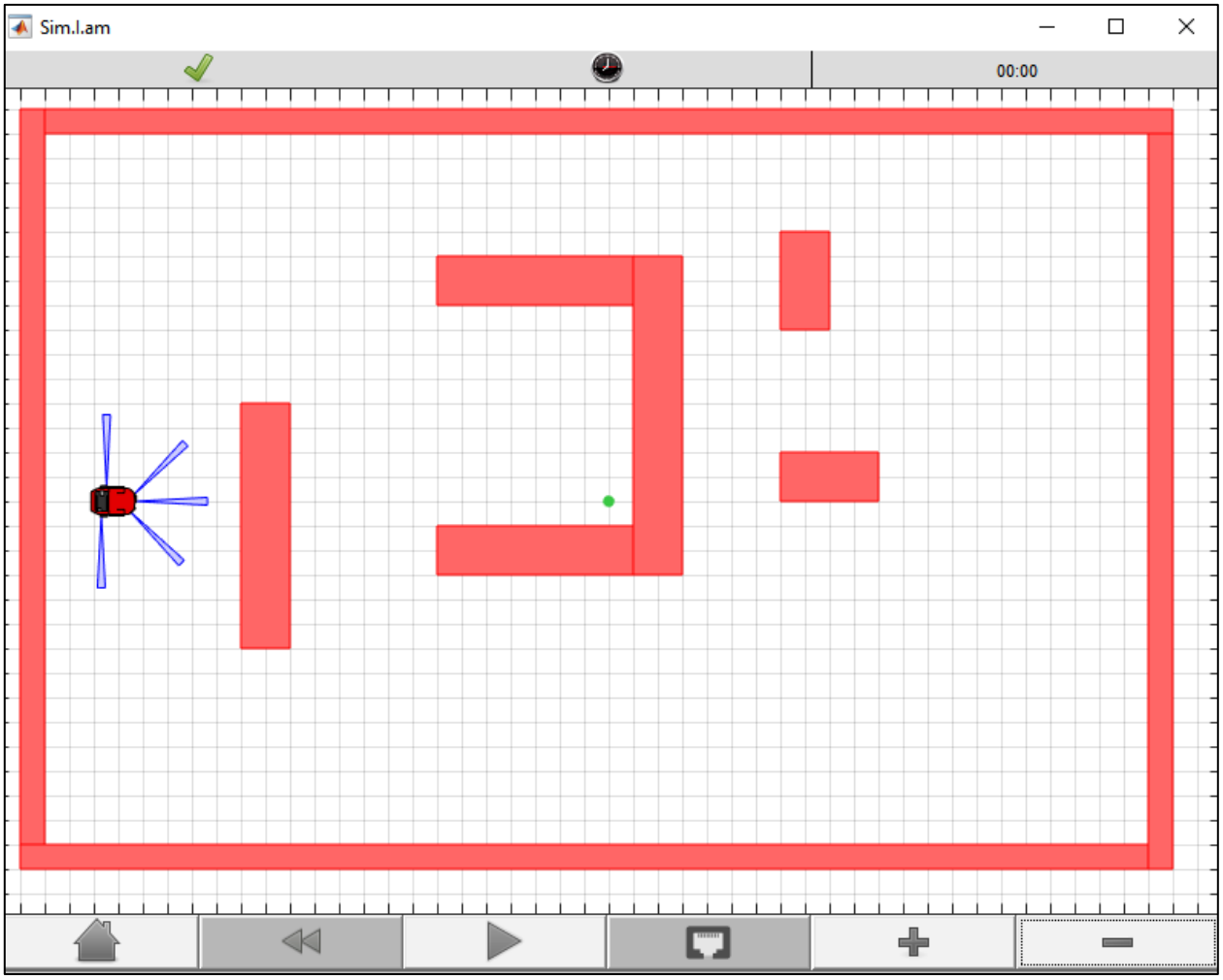


Figure 2.15: Simulation environment.

The following **Figure 2.16** represents a Flowchart about how the decisions are made when switching between controllers.

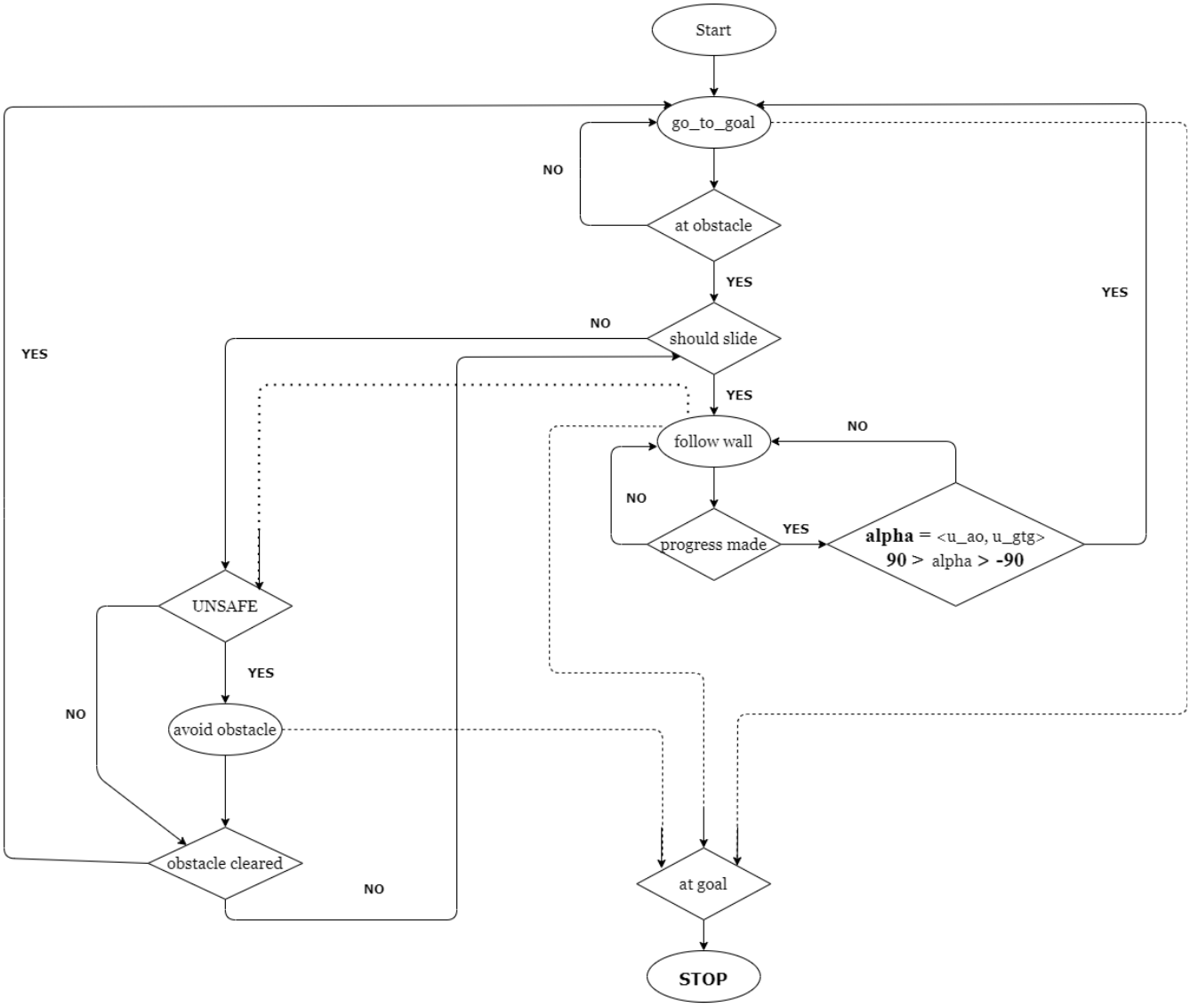
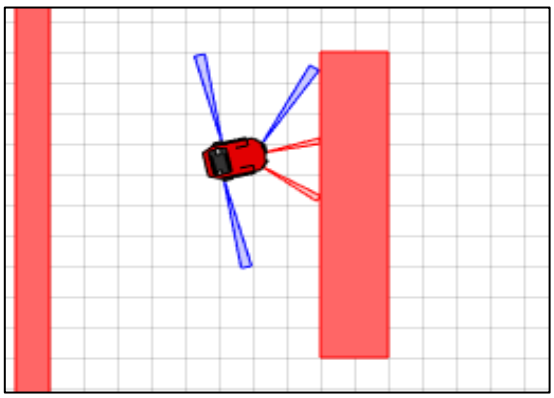
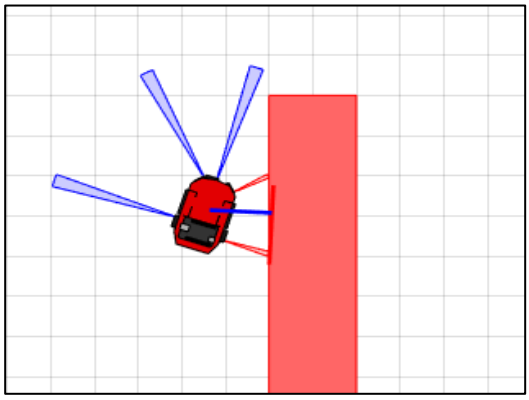


Figure 2.16: Flowchart representation of the robots' switching logic.

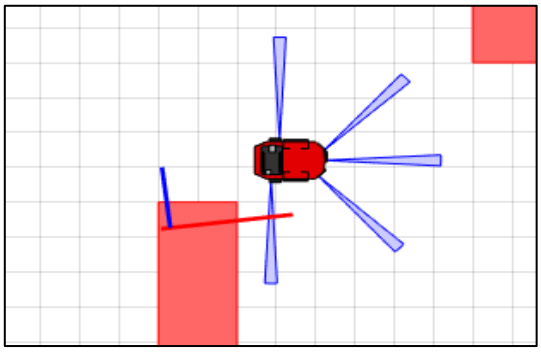
Using the above switching logic, our Mobile Robot reached the desired goal location successfully without collision. The different steps of a navigation example are shown in the **Figure 2.17.**



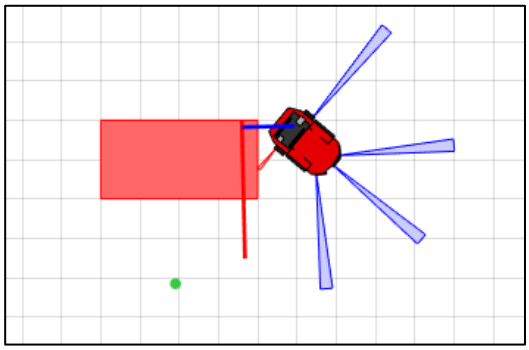
(a) At Obstacle.



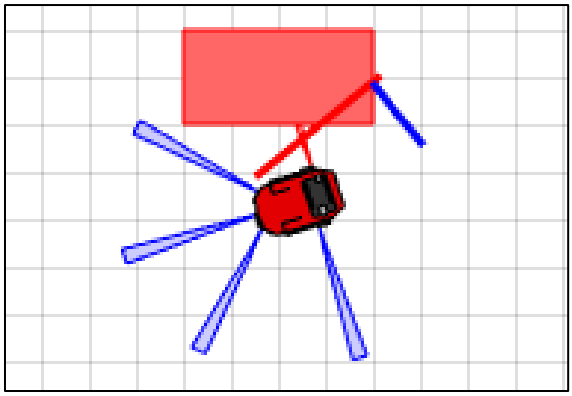
(b) Wall Following.



(d) Obstacle Avoidance.



(c) Break from Wall Following.



(e) At Goal.

Figure 2.17: Example of a complete navigation of the mobile robot.

2.8 Discussion

These results have been obtained by implementing stages of design and theory then testing them on simulation.

First, we implemented the go to goal behavior in an obstacle-free environment adjusting the heading of the robot to the goal. Second, the avoid obstacle behavior is triggered at a certain distance (`d_unsafe`) from an obstacle while going to a goal then once safe the robot switches back to the first controller. Switching could cause multiple problems like the *Zeno Phenomenon* which consists of switching too many times and error in state estimation. As a solution, the blending mode was introduced to solve this complication and having a smoother ride.

As the robot tries to reach its destination, it may encounter a malicious type of obstacles that is described to be non-convex where it fails to get around them using the previous controllers alone. Thus, one more controller is required to follow a path that is decided by the geometry of the obstacle at distance `d_fw` until it is cleared and no longer in the way to the goal location; this is known as the follow wall controller. Finally, we switched between them according to the events encountered by the robot.

Although the simulation has been successful, however it is not very powerful for many reasons; nowadays we have more accurate sensors, advanced AI and path planning algorithms that cover this subject. Furthermore, since the robots depends largely on the values to decide what to do when facing obstacles, the five sensors used by the simulator are defiantly not enough to fully understand the nature of the obstacle thus not enough to deal with all kinds of hostile navigation environment, sensing skirt are used for more efficient behaviors.

Conclusion 2

In this chapter, we introduced the Sim.i.am simulator and showed the way it closely demonstrated real world components, then we implemented the PID controller for the error for each behavior and provided the switching logic that supervised the robot in its journey to the goal location and shown results and discussion.

CONCLUSION

Through this project, we have been able to achieve our goals in bridging the gap between theory and practice by successfully designing and controlling a differential drive robot to navigate a world that is full of obstacles reaching a goal smartly and safely using a PID controller to stabilize the error which is the difference between the desired angle that is obtained from calculation depending on the state and events happening during navigation and the angle the robot that is obtained from sensors.

We used the Unicycle model and the Differential drive model, mapped them together for simplicity and manipulation. We implemented the odometry concept to keep track of position. At any given time, the robot is exactly at one state and can change in response to an input to another state. A finite number of states, an initial state and inputs can define what is called a finite state machine which perfectly describes the logic used to control the robot.

For future work, we consider to deal with Self-driving Cars which is one of the most interesting topics of research around the world. The question is how can we use what we learned in this project to control a four wheeled car? And what are the similarities between car-like robots and the unicycle?

In a four wheeled robot the direction in which the robot is pointing can be obtained from the back wheels that are at angle Φ and the front wheels are used for steering, thus their angle ψ becomes important so it is a state. **Figure 4.1** below shows a diagram of the robot in question where the states are: the position (x, y), the speed v, the heading Φ (in the equation it is θ) and the steering angle ψ.

$$\dot{x} = v\cos(\theta + \psi)$$
$$\dot{y} = v\sin(\theta + \psi)$$
$$\dot{\theta} = \frac{v}{l}\sin(\psi)$$
$$\dot{\psi} = \sigma$$

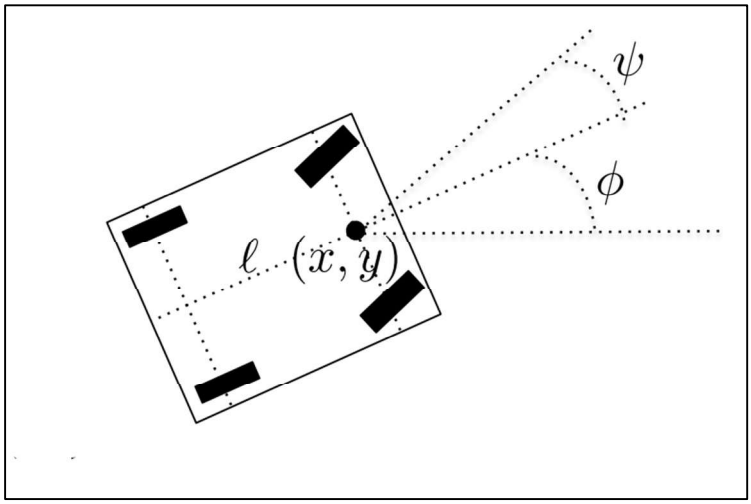


Figure 3.1: Car-like kinematics.

Now we try to find how to make the car-like robot behave like a unicycle.

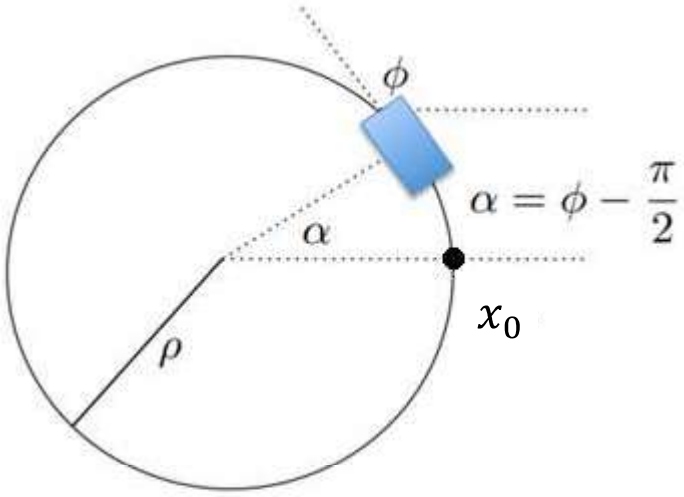


Figure 3.2: Unicycle curvature.

Based on the figure above showing a unicycle curvature, we get the following relations:

$$x = x_0 + \rho\cos(\alpha)$$

$$= x_0 + \sin(\theta)$$

$$\dot{x} = \omega\rho\cos(\theta) = v\cos(\theta)$$

$$\rho = \frac{v}{w} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

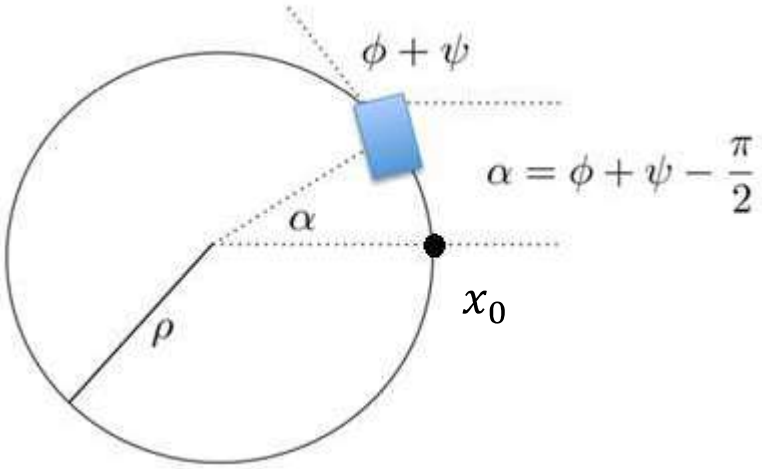


Figure 3.3: Car curvature.

Based on the figure above showing a car curvature, we get the following relations:

$$x = x0 + \rho \sin(\theta + \psi)$$

$$\dot{x} = \rho(\dot{\theta} + \dot{\psi})\cos(\theta + \psi) = v \cos(\theta + \psi)$$

We assume the steering angle is held at a fixed value, so $\dot{\psi} = 0$. We also have

$$\dot{\theta} = \frac{v}{l}\sin(\psi)$$

which yields to the following equation:

$$\rho = \frac{l}{\sin(\psi)} \ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

From equations (1) and (2) we could make the car behave like the unicycle by making

$$\sin(\psi) = \frac{\omega l}{v}$$

for small values of ψ we have:

$$\psi d = \frac{\omega l}{v}$$

we can then, for instance, use a P-regulator to control σ where

$$\sigma = k(\psi d - \psi)$$

Appendix A

Switching Supervisor

QBSupervisor

As seen before the QBSupervisor.m file is responsible for the switching logic between behaviors and updating the position of the robot. The main components of this file are

- Execute function (`execute()`): this function keeps getting called and executed while the simulation is running. It responsible for the switching between controllers and updating the position of the robot.
- State machine support functions: these include the function listed bellow

1) `Switch_to_state(name)`: this function uses takes a name as an input that will be used to choose a controller from an array of controllers initialized earlier to switch to that behavior for example to start following wall we call:

```
switch_to_state('follow_wall');
```

2) `Check_event(name)`: events are the reporters that tells us whether the robot or sensors has satisfied some mathematical clauses that have a physical significance for our robots. For example:

```
if check_event('unsafe')
   switch_to_state('avoid_obstacles');
end
```

3) `Is_in_state(name)`: this function returns true if the robot's state is the one entered as a parameter. We use it to check the in switching logic as follows

```
If is_in_state('avoid_obstacles')&&...
          check_event(obstacle_is_cleared)
        switch_to_state('go_to_goal');
end
```

this simply means that if the robot is avoiding obstacle and the sensors' range is clear of obstacles; then it can switch to go_to_goal behavior.

The code bellow shows the switching logic used for our mobile robot:

```
if (obj.check_event('at_goal'))
                if (~obj.is_in_state('stop'))
                    [x,y,~] = obj.state_estimate.unpack();
                    fprintf('stopped at (%0.3f,%0.3f)\n', x, y);
                end
                obj.switch_to_state('stop');

       elseif obj.check_event('unsafe')
                obj.switch_to_state('avoid_obstacles');

       elseif (obj.is_in_state('go_to_goal')
                  ||obj.is_in_state('ao_and_gtg')) ...
                      &&obj.check_event('at_obstacle')

                If obj.check_event('slidong_left')
                     obj.fw_direction = 'left';
                     obj.set_progress_point_new();
                     obj.switch_to_state('follow_wall');

                elseif obj.check_event('slidong_right')
                     obj.fw_direction = 'right';
                     obj.set_progress_point_new();
                     obj.switch_to_state('follow_wall');
                end

        elseif obj.is_in_state('follow_wall')

                angle = check_angel_gtg_ao(u_ao, u_gtg);

                    if angle&&obj.check_event('progress_made')
                         obj.switch_to_state('go_to_goal')
                    end

        elseif obj.is_in_state('avoid_obstacles')
                if(obj.check_event('obstacle_cleared'))
                    disp('Obstacle is cleared :')
                obj.switch_to_state('go_to_goal');
                end

        else
                  if(~obj.is_in_state('go_to_goal'))
                  obj.switch_to_state('go_to_goal');
                  end
  end
```

References

[1] *G. Dudek, Computational Principles of Mobile Robots (2000).*

[2] *Kansas State University – Polytechnic Campus. LabVIEW Robotics Programming Study Guide, The Unicycle Model. Retrieved from:* http://faculty.salina.k-state.edu/tim/robotics_sg/Control/kinematics/unicycle.html

[3] *Massachusetts Institute of Technology, E. Olson. A Primer Odometry and Motor Control (2004).*

[4] *T. Sokunphal. Velocity Control of a Car-Like Mobile Robot (2017).*

[5] *S. Kowalewski & all. Handbook of Hybrid Systems Control (2009).*

[6] *S. Dashkovskiy1 and P. Feketa, Zeno Phenomenon in Hybrid Dynamical Systems (2017).*

[7] *Georgia Institute of Technology, Dr. M. Egerstedt. Control of Mobile Robots. Retrieved from:* https://www.coursera.org/learn/mobile-robot/lecture/UEKZs/convex-and-non-convex-worlds

[8] *Georgia Robotics and InTelligent Systems Laboratory, J.P. de La Croix (2003). Sim.I.am. Retrieved from:* http://gritslab.gatech.edu/home/2013/10/sim-i-am/

[9] *Georgia Robotics and InTelligent Systems Laboratory, J.P. de La Croix, M. Hale (2016). Coursera: Control of Mobile Robots.*