Stereo Vision-based Autonomous Vehicle Navigation

by

Guilherme Tebaldi Meira

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

by

April 2016

APPROVED:

Professor Alexander M. Wyglinski, Major Thesis Advisor

Professor Jie Fu

Professor Michael A. Gennert

Abstract

Research efforts on the development of autonomous vehicles date back to the 1920s and recent announcements indicate that those cars are close to becoming commercially available. However, the most successful prototypes that are currently being demonstrated rely on an expensive set of sensors. This study investigates the use of an affordable vision system as a planner for the Robocart, an autonomous golf cart prototype developed by the Wireless Innovation Laboratory at WPI.

The proposed approach relies on a stereo vision system composed of a pair of Raspberry Pi computers, each one equipped with a Camera Module. They are connected to a server and their clocks are synchronized using the Precision Time Protocol (PTP). The server uses timestamps to obtain a pair of simultaneously captured images. Images are processed to generate a disparity map using stereo matching and points in this map are reprojected to the 3D world as a point cloud. Then, an occupancy grid is built and used as input for an A* graph search that finds a collision-free path for the robot. Due to the non-holonomic constraints of a car-like robot, a Pure Pursuit algorithm is used as the control method to guide the robot along the computed path. The cameras are also used by a Visual Odometry algorithm that tracks points on a sequence of images to estimate the position and orientation of the vehicle.

The algorithms were implemented using the C++ language and the open source library OpenCV. Tests in a controlled environment show promising results and the interfaces between the server and the Robocart have been defined, so that the proposed method can be used on the golf cart as soon as the mechanical systems are fully functional.

Acknowledgements

I would like to express my gratitude to my advisor Professor Alexander Wyglinski for his guidance and specially for the weekly doses of excitement. I would also like to thank the Coordination for the Improvement of Higher Education Personnel (CAPES) for sponsoring my scholarship. Furthermore, I would like to thank my friends at the WiLab for the good advice and for the random conversations as well as all the other friends that made me feel a little bit closer to home. Finally, I am deeply grateful for my parents to whom I owe everything I am or hope to be, my fiancée for her support and continuous encouragement and God for loving me unconditionally.

Contents

List of Figures

List of Abbreviations

3D Three-dimensional

ALV Autonomous Land Vehicle

API Application Programming Interface

BSD Berkeley Software Distribution

CPU Central Processing Unit

DARPA Defense Advanced Research Projects Agency

FAST Features from Accelerated Segment Test

GFTT Good Features to Track

GNU GNU is Not Unix

GPIO General Purpose Input and Output

GPL General Public License

GPS Global Positioning System

GPSd GPS Daemon

GPU Graphics Processing Unit

HDMI High-Definition Multimedia Interface

HDOP Horizontal Dilution of Precision

I2C Inter-Integrated Circuit

IO Input and Output

IP Internet Protocol

KLT Kanade-Lucas-Tomasi

LiDAR Light Radar

MMAL Multi-Media Abstraction Layer

MQP Major Qualifying Project

NOAA National Oceanic and Atmospheric Administration

NTP Network Time Protocol

PDOP Position Dilution of Precision

PID Proportional-Integral-Derivative

PRM Probabilistic Roadmap

PTP Precision Time Protocol

PWM Pulse Width Modulation

RAM Random Access Memory

RANSAC Random Sample Consensus

RRT Rapidly-exploring Random Tree

SCL Serial Clock Line

SDA Serial Data Line

SIFT Scale Invariant Feature Transform

SIMD Single Instruction Multiple Data

SSH Secure Shell

SURF Speeded Up Robust Features

TBB Threading Building Blocks

TCP Transmission Control Protocol

UDP User Datagram Protocol

USB Universal Serial Bus

V4L2 Video4Linux 2

VDOP Vertical Dilution of Precision

WMM World Magnetic Model

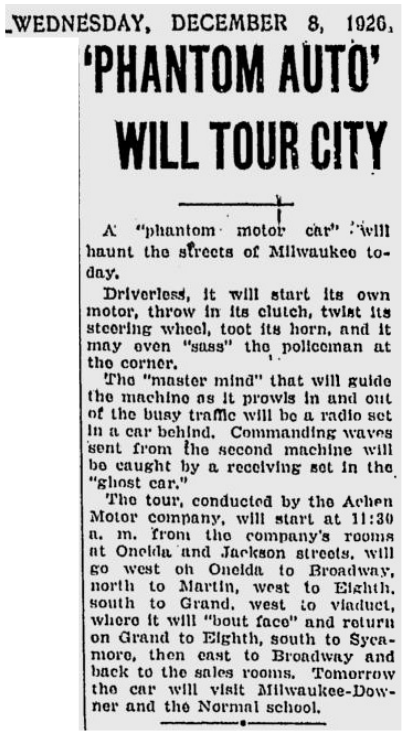
Chapter 1

Introduction

1.1 Autonomous Vehicles

Autonomous vehicles have been studied for decades, with the first experiments in this area dating back to the 1920s. One of the first attempts at a driverless car was developed in 1925 by Houdina Auto Control and consisted of a 1926 Chandler equipped with transmitting and receiving antennas. The car was operated by a second car that followed it and sent out radio signals that were used to command a set of electric motors. Achen Motor, a car distributor in Milwaukee, demonstrated the invention under the name of "Phantom Auto" on the streets of Milwaukee in December 1926 [1], as reported by The Milwaukee Sentinel (Figure 1.1).

Since then, a substantial amount of research has been done both by companies and universities, leading to several prototypes and experimental vehicles throughout the years. In the late 1950s, RCA Labs successfully demonstrated a full-sized car that could use sensors buried on the pavement of a highway in order to drive autonomously [2]. Various other prototypes were developed based on that concept of a "smart highway", such as a driverless Citroen DS developed by the United



WEDNESDAY, DECEMBER 8, 1926.

'PHANTOM AUTO' WILL TOUR CITY

A "phantom motor car" will haunt the streets of Milwaukee today.

Driverless, it will start its own motor, throw in its clutch, twist its steering wheel, toot its horn, and it may even "sass" the policeman at the corner.

The "master mind" that will guide the machine as it prowls in and out of the busy traffic will be a radio set in a car behind. Commanding waves sent from the second machine will be caught by a receiving set in the "ghost car."

The tour, conducted by the Achen Motor company, will start at 11:30 a. m. from the company's rooms at Oneida and Jackson streets, will go west on Oneida to Broadway, north to Martin, west to Eighth, south to Grand, west to viaduct, where it will "bout face" and return on Grand to Eighth, south to Sycamore, then east to Broadway and back to the sales rooms. Tomorrow the car will visit Milwaukee-Downer and the Normal school.

(a) The Milwaukee Sentinel

FREDERICKSBURG, VA., SATURDAY, JUNE 18, 1932.

"Phantom Auto" to Be Operated Here

Driver-less Car to be Demonstrated About City Streets Next Saturday — Controlled Entirely by Radio.

One of the most amazing products of modern science will be demonstrated in Fredericksburg next Saturday, June 25, when the "Phantom Auto" will be piloted through the streets of the city without a driver or occupant, with no one touching it and with no wires or strings attached to it.

It sounds unbelievable but it is true that the driverless car will travel about the city through the heaviest traffic, stopping, starting, turning, sounding its horn, and proceeding just as though there were an invisible driver at the wheel.

Spectacular Event

The effect is uncanny and mystifying and the complete demonstration is one of the most spectacular street events possible. Wherever the "Phantom Auto" has been shown it has attracted huge crowds eager to witness the startling performance of an automobile whirling about the streets without a driver.

The "Phantom Auto" is operated entirely by remote-controlled radio, the equipment used being the latest development of scientific engineering. Elaborate equipment is installed in a new Chevrolet Sedan furnished for the event by Virginia Sales & Service Corp., local Chevrolet dealers. A control car follows in the rear of the "Phantom Auto" and the operator sends out radio impulses which direct every movement of the driverless vehicle.

Use Business Streets

The route of travel of the "Phantom Auto" will be through the principal business streets of the city and every person may have ample opportunity to witness the remarkable demonstration of inventive genius. The time of the performance and the exact route to be followed will be published next week.

The "Phantom Auto" is being brought to Fredericksburg under the sponsorship of The Free Lance-Star in cooperation with the following business firms:

Virginia Sales & Service Corp., Nehi Bottling Co., Dillon Motor Service, Montgomery Ward & Co., Clem Goodman, American Oil Co., Exide Service Station, Boston Variety Store, The Busy Corner, and Chichester-Dickson Co.

(b) The Free Lance-Star

Figure 1.1: Newspapers announcing demonstrations of the Phantom Auto in 1926 and 1932.

Kingdom's Transport and Road Research Laboratory. It used magnetic cables embedded on the road to guide itself and was able to drive at 80 miles per hour without deviation of speed in any weather condition [3].

The first self-sufficient and truly autonomous cars appeared in the 1980s. The Autonomous Land Vehicle (ALV), funded by the Defense Advanced Research Projects Agency Defense Advanced Research Projects Agency (DARPA), used technologies developed by the University of Maryland and Carnegie Mellon University, among other research laboratories. It achieved the first road-following demonstration using LiDAR, computer vision and autonomous robotic control to achieve speeds of up to 19 miles per hour [4, 5].

Nowadays, many major automotive manufacturers such as Ford, BMW and Tesla Motors, as well as technology companies such as Google, Apple and Uber, are making large investments in research and development of autonomous vehicles. Public demonstrations of the currently existing prototypes show that the technology has evolved substantially over the years and is more robust and reliable than ever. Several important names in the industry predict that those cars will be available to the general public in the near future. Elon Musk, Tesla's founder, said in an interview that fully autonomous Teslas are expected to be ready by 2018 and the regulatory approval may take 1 to 3 more years [6].

For years, autonomous vehicles have been depicted as the future of transportation, leading to a world in which riding a car is a much safer and more enjoyable experience and predictions like Musk's indicate that this is becoming a reality very soon.

1.2 Related Work

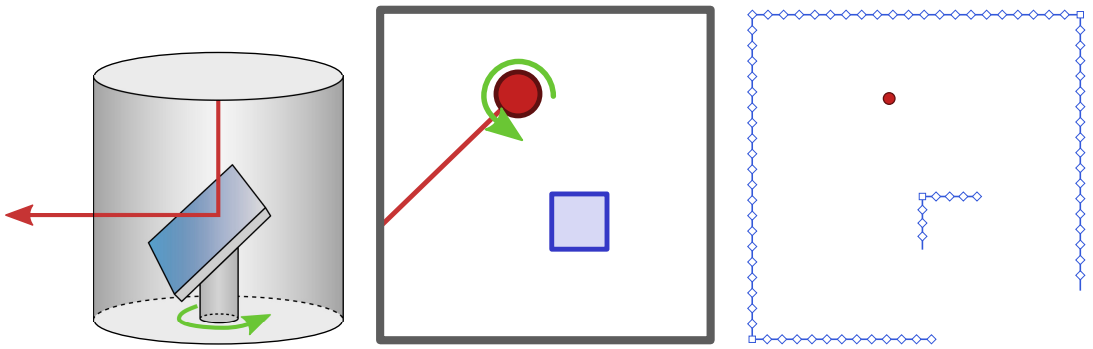
Vehicle autonomy is a complex problem that involves the use of sensors to capture information about the environment around the robot, computer algorithms to process this information and mechanical actuators to execute the path computed by the system. Each one of those subsystems has been explored by researchers and several different approaches are documented in the literature.

1.2.1 Environment Sensing

Autonomous cars are expected to safely drive on roads that it may potentially share with other fast-moving driverless cars, human-driven vehicles, trucks and motorcycles, as well as pedestrians and animals. Being able to quickly obtain detailed information about the environment is critical for the safe operation of a self-driving car.

Spatial data, containing information about obstacles around the vehicle, is commonly obtained using LiDAR, a sensor that measures distance by illuminating the target with a laser light, as illustrated by Figure 1.2. It provides a detailed 3D map of the environment and are primarily used to detect obstacles [7, 8], but other information, like road shape and edges, can be inferred from the data [9, 10]. One of the most well known self-driving cars, the Google car, combines LiDAR information with high resolution maps of the world in order to produce data models that allow it to drive safely and respecting the traffic laws [11].

With the advances in technology, good quality cameras became widely available and computers became fast enough to process images and extract information in real time. This lead to the use of Computer Vision as an additional source of spatial information or even as a replacement for LiDAR. The Navlab, one of the first



(a) Schematic of a LiDAR. (b) Example environment. (c) Example data.

Figure 1.2: Schematic representation of a LiDAR. A rotating mirror moves the laser around the scene to measure distances. In (b), the top view of an example environment is scanned by the LiDAR, that returns the example data shown in (c).

autonomous vehicles, was developed by the Carnegie-Mellon University and used vision sensors to guide the robot, relying on the presence of some characteristics on the road such as color and shape [12]. In less structured environments, other characteristics can also be used as clues to the robot, such as pedestrian footprints and tracks left by other vehicles [13].

Cameras can also be used to obtain 3D information. Stereoscopic vision systems use a pair of cameras pointed at the same scene, and the differences between the images are used to extract depth data. This kind of setup has been successfully used to perform exploration and path planning in an unknown environment [14]. One limitation of cameras, when compared to LiDAR, is the smaller field of view. A LiDAR can provide a 360 degrees view of the environment, while cameras are usually limited to less than 180, but combining the data from the cameras with rotation information from the robot, it is possible to estimate a complete model of the environment around the robot [15]. Other clever uses of cameras include visual odometry, that consists of the estimation of robot movement by tracking points as they move in a sequence of images [16, 17].

Other techniques for obstacle avoidance include the use of ultrasonic sensors.

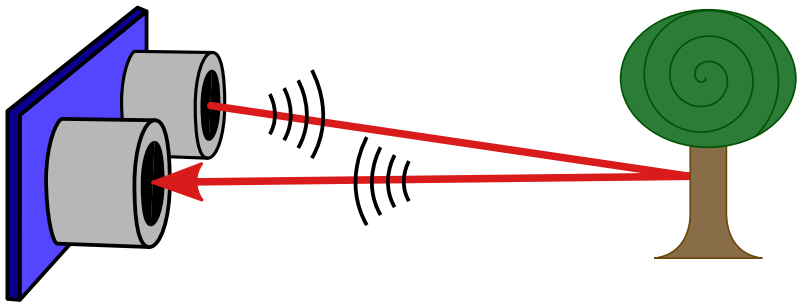


Figure 1.3: Ultrasonic sensor measuring the distance to an obstacle. A sound wave is emitted by the sensor and the reflected wave is captured by it. The time interval between sending the wave and receiving the echo is used to measure the distance.

They can measure distance to an obstacle by emitting high-frequency sound waves and measuring the time interval between sending the signal and receiving the echo, as illustrated by Figure 1.3. Those sensors usually have a smaller range when compared to LiDAR and cameras and are commonly used in conjunction with a vision system to enhance obstacle avoidance [18], but can also be the only sensor for obstacle avoidance in simpler environments [19].

Obstacle detection is essential to make sure that the robot can move without causing damage to the environment or to itself, but in order to travel long distances, it needs positional information. GPS sensors are an affordable and convenient solution to this problem. A receiver can determine its location anywhere on the planet, as long as there is an unobstructed line of sight to at least four GPS satellites. However, the accuracy of the position information given by a GPS may range from 2 to 8 meters [20]. Also, the receiver may not work on indoor environments or inside tunnels, where there is no line of sight to the satellites. GPS information can be fused with other data such as odometry sensors [21] and maps [22] in order to compensate for inaccuracies and outages.

1.2.2 Path Planning

Motion planning is the process of computing a sequence of discrete motions in order to perform a desired movement. It is commonly referred to as the *Piano Mover's Problem* [23], in which a piano must be moved to a goal position, while avoiding obstacles on its way. However, motion planning algorithms have a wide range of applications, from finding a path for a game character in a virtual world [24] to controlling a humanoid robot [25].

Planning for a complex robot with multiple joints is usually done in a high-dimensional configuration space, in which conventional search algorithms are impractical. In these cases, sampling-based algorithms such as the Probabilistic Roadmap (PRM) [26] and the Rapidly-exploring Random Tree (RRT) [27] are well known approaches, with several variations documented in the literature.

On the other hand, car-like robots only navigate in two dimensions, which makes the planning problem more tractable. Planning in a low dimensional space usually is done by discretizing the space and performing a graph search. Graph search is a well known problem and some of the most popular algorithms, such as Dijkstra's [28] and A* [29] are proven to give optimal solutions.

Other common approaches are based on the concept of potential fields [30, 31], that are created by obstacles and repel the robot. Once the potential field is known, a path can be found using some optimization algorithm such as Gradient Descend.

When planning for a car-like robot, even the search space can be created in various different ways. Occupancy grids [32] represent the environment around the robot in a cartesian plane and this approach has also been extended to three dimensions [33]. For vision-based planners, it has been proposed that the conversion of the data obtained by the cameras to an occupancy grid is not necessary and the planning can be done in image space [15].

A planner for a car also must consider the *nonholonomic constraints* of the robot. All the algorithms mentioned so far assume that the robot can move between two arbitrary configurations in a straight line. A car does not have that kind of freedom, it can move forward and backward, in a straight or curved line, but it cannot move sideways.

Existing algorithms can be adapted in order to incorporate those constraints into the planning process. Instead of considering movements in every direction, the possible movements of the robot can be modeled by a small set of motion primitives [34]. Another possible solution is to perform the planning without considering the constraints and then use a *steering algorithm* such as Pure Pursuit [35] or the Stanley method [36] to guide the robot along the trajectory.

1.3 The Robocart

The Robocart is a self-driving golf cart prototype developed at WPI's Wireless Innovation Laboratory. Figure 1.4 shows a concept diagram of the robot. It contains a pair of Raspberry Pi 2[1] single-board computers, each one equipped with a Camera Module [2]. They are connected via ethernet cables to a Linux server on the back of the cart. The server is also connected to a GPS receiver that provides the current coordinates of the golf cart. The server runs the planning software that processes the data from the sensors and computes a path to the goal. Commands are sent to three Arduino[3] boards connected to the steering, throttle and braking systems of the cart. The Arduinos control a set of motors installed on the robot in order to guide it along the desired path.

[1] https://www.raspberrypi.org/
[2] https://www.raspberrypi.org/help/camera-module-setup/
[3] http://arduino.cc/

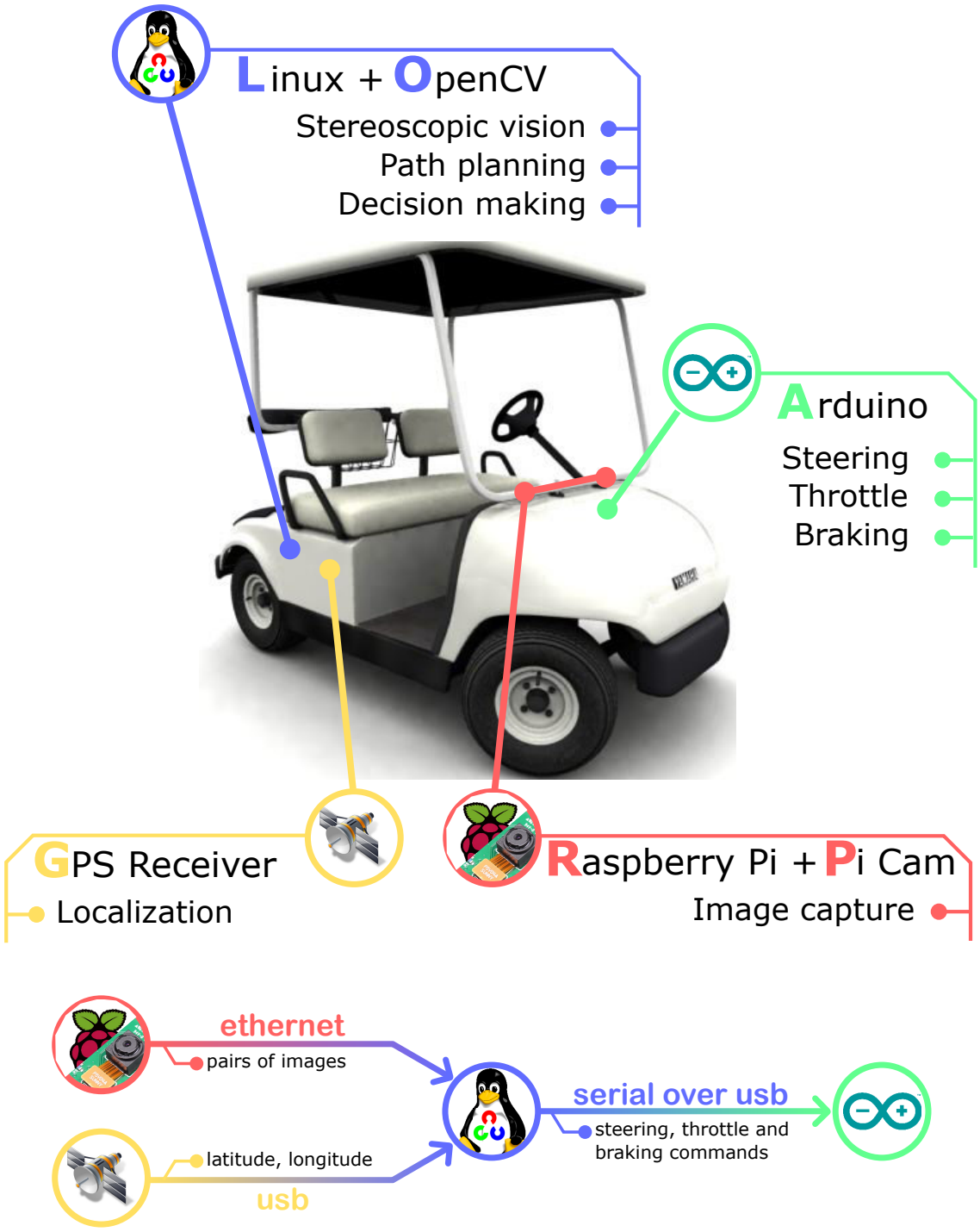


Figure 1.4: Conceptual illustration of the Robocart. Two front-facing cameras form a stereo pair and send images to the server. A GPS Receiver provides localization data. The Linux server uses the data to plan an obstacle-free path to the goal and sends commands to the Arduino boards that drive the motors.

Over the last few years, different teams worked on the Robocart with the objective of adding *drive by wire* capabilities to it, *i.e.* to make the mechanical steering, throttle and braking controllable by a computer.

From 2014 to 2015, the team composed by Prateek Sahay [37], Elizabeth Miller [38] and Gabriel Isko [39] proposed the use of a pair of Raspberry Pi cameras as a stereo pair. They also built the mechanical systems to control the break and steering. From 2015 to 2016, a new team, composed by Robert Crimmins and Raymond Wang, integrated the mechanical systems with the on-board Linux server. This project was developed in parallel with the efforts from both teams and aimed to build the first version of the Robocart's autonomy software.

1.4 Thesis Contributions

The proposed system uses a pair of Raspberry Pi cameras as its main source of information, instead of a more expensive LiDAR sensor, since one of the goals of the Robocart project is to be affordable. Being a golf cart, the Robocart is not expected to drive at high speeds nor on roads. The goal of this project is the development of a system that can guide the robot at low speed along a route defined by a set of GPS coordinates, in an unstructured environment with arbitrary obstacles. We propose the use of a combination of techniques described in the literature:

- An occupancy grid to model the environment in front of the robot;
- A graph search algorithm to find a collision-free path;
- A steering method to guide the robot along the trajectory;
- A visual odometry method that can estimate the location of the robot based on the images from the vision system.

This thesis describes in details the implementation of such system, as well as the results we obtained testing the proposed approach.

1.5 Thesis Organization

This thesis is organized as follows: Chapter 2 provides background information about Computer Vision and Path Planning. Chapter 3 describes the sensors, motors and other systems available on the Robocart. Chapter 4 presents the path planning method we used. Chapter 5 introduces the visual odometry algorithm implemented on the golf cart. Chapter 6 describes the steering method used to follow the computed trajectory. Chapter 7 presents the results we obtained. Finally, Chapter 8 concludes this thesis and discusses future work on the Robocart.

Chapter 2

Computer Vision and Path Planning

This chapter presents background information that is important for the understanding of the remaining chapters. Section 2.1 briefly introduces the mathematics behind the operations on points in a 3D space, such as translation, rotation and scaling, as well as the concept of homogeneous coordinates. Section 2.2 describes the process of computing disparity data from a pair of images of the same scene. Section 2.3 introduces some basic image operations that are used throughout this project. In Section 2.4, we discuss the path planning problem and explain search algorithms that can be used to solve it in a low dimensional case.

2.1 Transformation Matrices and Homogeneous Coordinates

Points in a 3D space can be represented by a set of three coordinates, x, y and z, with respect to a coordinate frame, as illustrated in Figure 2.1. Mathematically, a

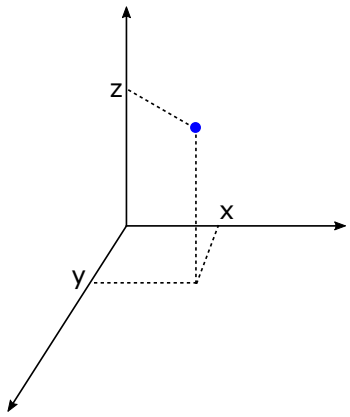
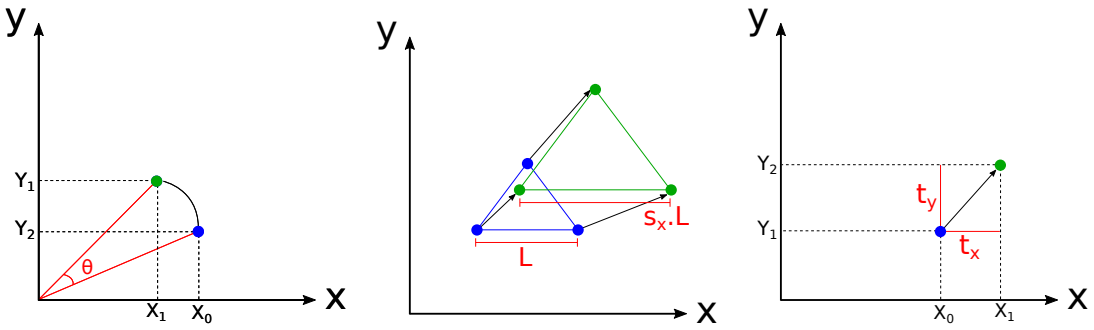


Figure 2.1: A point in a 3D world represented by its three coordinates, one for each one of the axes of the coordinate frame.



(a) Rotation around the z axis.

(b) Scaling on the x and y axes.

(c) Translation on the x and y directions.

Figure 2.2: A 2D view of the three basic operations on a point. The z axis is perpendicular to the paper.

point can be represented by a column vector with three values, as shown in Equation (2.1). Three of the most basic operations that can be performed on that point are rotation, scaling and translation. They are illustrated in Figure 2.2.

$$\vec{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.1}$$

A *rotation* around one of the axes of the coordinate frame changes two of the

three coordinates of a point. For example, a rotation around the z axis changes x and y according to equations (2.2) and (2.3). This operation can be performed by a matrix multiplication, as in Equation (2.4). Rotations around the x and y axes can also be performed by matrix multiplications, as shown in equations (2.5) and (2.6). An arbitrary number of rotations can be combined into a single matrix by multiplying all the rotation matrices, as in Equation (2.7).

$$x_r = x\ \cos(\theta) - y\ \sin(\theta), \tag{2.2}$$

$$y_r = x\ \sin(\theta) + y\ \cos(\theta), \tag{2.3}$$

$$\vec{p}_{rz} = \mathbf{R_z}(\theta) \cdot \vec{p} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x\ \cos(\theta) - y\ \sin(\theta) \\ x\ \sin(\theta) + y\ \cos(\theta) \\ z \end{bmatrix} \tag{2.4}$$

$$\vec{p}_{rx} = \mathbf{R_x}(\theta) \cdot \vec{p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y\ \cos(\theta) - z\ \sin(\theta) \\ y\ \sin(\theta) + z\ \cos(\theta) \end{bmatrix}, \tag{2.5}$$

$$\vec{p}_{ry} = \mathbf{R_y}(\theta) \cdot \vec{p} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x\ \cos(\theta) + z\ \sin(\theta) \\ y \\ -x\ \sin(\theta) + z\ \cos(\theta) \end{bmatrix}, \tag{2.6}$$

$$\vec{p}_r = \mathbf{R_n}(\theta_n) \cdot \mathbf{R_{n-1}}(\theta_{n-1}) \cdot \cdots \cdot \mathbf{R_1}(\theta_1) \cdot \vec{p} = \mathbf{R} \cdot \vec{p}. \tag{2.7}$$

The *scaling* operation does not make much sense for a single point, but when applied to an object represented as a set of points, it changes the scale of the object. Scaling is done by multiplying the coordinates of each point by a scaling factor. It can also be implemented as a matrix multiplication, as in Equation (2.8). An arbitrary number of scalings can be also combined into a single matrix by multiplying all the scaling matrices, as in Equation (2.9).

$$\vec{p}_s = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} s_x \cdot x \\ s_y \cdot y \\ s_z \cdot z \end{bmatrix}, \tag{2.8}$$

$$\vec{p}_s = \mathbf{S_n} \cdot \mathbf{S_{n-1}} \cdot \cdots \cdot \mathbf{S_1} \cdot \vec{p} = \mathbf{S} \cdot \vec{p}. \tag{2.9}$$

Translations are done using addition, as in Equation (2.10). They can be combined by adding all the translation vectors, as shown in Equation (2.11). However, translations are done with addition while rotations and scalings are done with multiplication, so a sequence of those operations cannot be combined into a single matrix. To overcome this limitation, we use *homogeneous coordinates*.

$$\vec{p}_t = \vec{T} + \vec{p} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \end{bmatrix}, \tag{2.10}$$

$$\vec{p}_t = \vec{T}_n + \vec{T}_{n-1} + \cdots + T_1 + \vec{p} = \vec{T} + \vec{p}. \tag{2.11}$$

2.1.1 Homogeneous Coordinates

Homogeneous coordinates are used in a *projective space*. In order to have an intuition about how homogeneous coordinates work, we can consider an example in 2D.

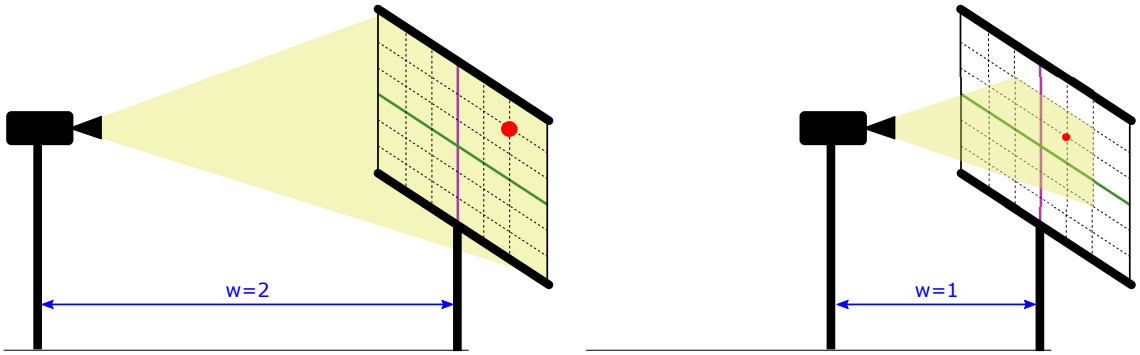


Figure 2.3: Intuition about homogeneous coordinates. As we move the projector towards the plane, w decreases as well as x and y.

Consider a cartesian plan like the one shown on Figure 2.3. The green line is the x axis and the purple line is the y axis. A point is *projected* on that plane by a projector that is at a distance $w = 2$ to that plane. In order to represent that point, we need not only its x and y coordinates, but also the value of w.

Since we are using three coordinates to represent a point in a 2D plane, there are infinite ways to represent the same point. As we move the projector away from the plane (increase w), the x and y coordinates of the point increase. As we move the projector towards to the plane (decrease w, as shown on the right on Figure 2.3), the x and y coordinates of the point decrease.

Mathematically, any value of $w \neq 0$ is acceptable, but when dealing with computer graphics and vision, usually w is kept equals to 1 at all times. Given a point $\vec{p} = (x, y, w)$, Equation (2.12) shows how it can be converted to the "standard" representation by dividing all the coordinates by w.

$$\vec{p}_{std} = \left(\frac{x}{w}, \frac{y}{w}, 1\right). \tag{2.12}$$

In 3D, homogeneous coordinates work the same way. A vector $\vec{p} = (x, y, z)$ is represented using four coordinates (x, y, z and w) and any value of $w \neq 1$ is

converted to the "standard" representation using Equation (2.13), that represents the same process we used in 2D.

$$\vec{p}_{std} = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right). \tag{2.13}$$

One of the main benefits of adding this new dimension is that now we are able to perform rotations and translations using matrix multiplication.

Rotation matrices are created by augmenting our previous rotation matrices with a new row and a new column, as in Equation (2.14). Scaling matrices work the same way and is shown in Equation (2.15). Finally, translation matrices are represented as shown in Equation (2.16).

$$\tilde{\mathbf{R}}_{\mathbf{x}}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{\mathbf{x}}(\theta) & \vec{0} \\ \vec{0}^T & 1 \end{bmatrix}, \tag{2.14}$$

$$\tilde{\mathbf{S}}(\mathbf{s_x}, \mathbf{s_y}, \mathbf{s_z}) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{S}(s_x, s_y, s_z) & \vec{0} \\ \vec{0}^T & 1 \end{bmatrix}, \tag{2.15}$$

$$\vec{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.16}$$

It is easy to notice that multiplying the translation matrix by a vector adds t_x,

t_y and t_z to the respective coordinates, as expected from a translation operation, as shown in Equation (2.17).

$$\vec{T} \cdot \vec{p} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ w \end{bmatrix}. \tag{2.17}$$

With homogeneous coordinates, an arbitrary sequence of rotations and translations can be combined into a single matrix, which is convenient and efficient.

2.2 Stereoscopic Vision

Humans and many other animals have *binocular vision*, *i.e.* two eyes are used simultaneously to obtain visual information about the environment. This kind of vision system has advantages over a *monocular* one, such as a wider field of view and a spare eye in case one of them is damaged. However, not every binocular vision system is the same. While some creatures, like horses, have one eye on each side of the face, others, like humans, have both eyes aligned side-by-side, which allows each eye to capture a view of the same scene from a slightly different angle. The two images are processed by the brain, that uses the positional differences, or *disparities*, between the images to infer depth information.

Being able to estimate a three-dimensional model of the world based on a pair of two-dimensional images is an important part of the way humans interact with the world. Activities as simple as stepping off a curb or as complex as performing a surgery rely on that ability. Using a pair of cameras and appropriate algorithms, a robot can use that same principle to sense the world around it and perform tasks

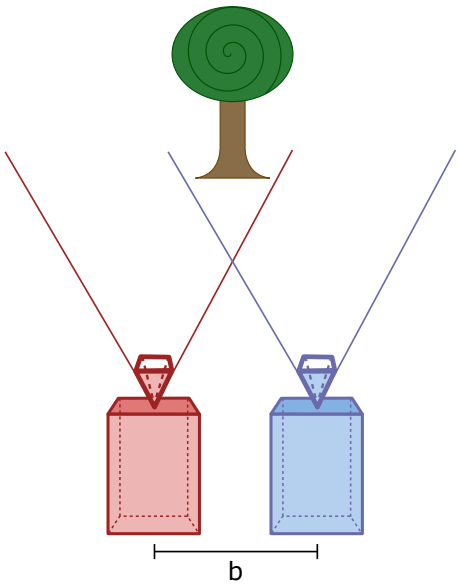


Figure 2.4: The camera setup considered in this chapter. Cameras are horizontally aligned and separated by a baseline distance b.

such as mapping, navigation and obstacle avoidance.

The most common stereoscopic vision setups use two cameras horizontally or vertically aligned, but there are other approaches described in the literature using a single moving camera [40] or even still images [41]. For this project, we use a pair of cameras aligned horizontally, separated by a distance b, called *baseline distance*, as illustrated in Figure 2.4. In the next subsections we describe an approach to compute depth information from a pair of images.

2.2.1 Camera model

The process of projecting a 3D scene onto a 2D image can be understood using the *pinhole camera model*, where the camera aperture is considered to be a point. Figure 2.5 represents one of the cameras. Let (x_w, y_w, z_w) be a point in a coordinate frame fixed somewhere on the scene (the *world* coordinate frame). That point can be represented as (x_c, y_c, z_c) on a coordinate frame fixed on the camera (the *camera*

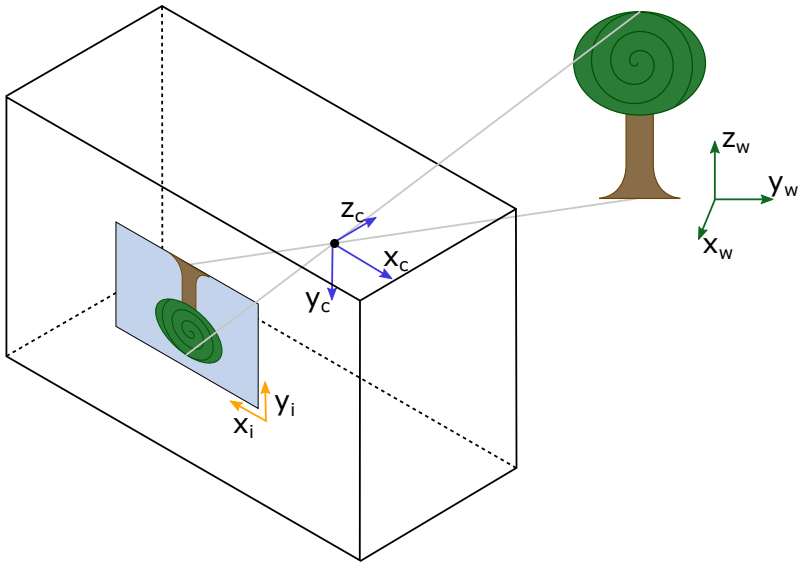


Figure 2.5: The three coordinate frames: world (in green), camera (in blue) and image (in yellow). The object location in the world coordinates can be converted to the camera coordinates and then projected to the image plane.

coordinate frame). Points in the world coordinate frame can be represented in the camera coordinate frame by performing a rotation followed by a translation. Using homogeneous coordinates, as described in Section 2.1, we represent this transformation in Equation (2.18).

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}. \tag{2.18}$$

The rotations and translations that specify the position of the camera relative to the external world are called the *extrinsic parameters*.

The points on the camera coordinate frame can, then, be projected onto the 2D

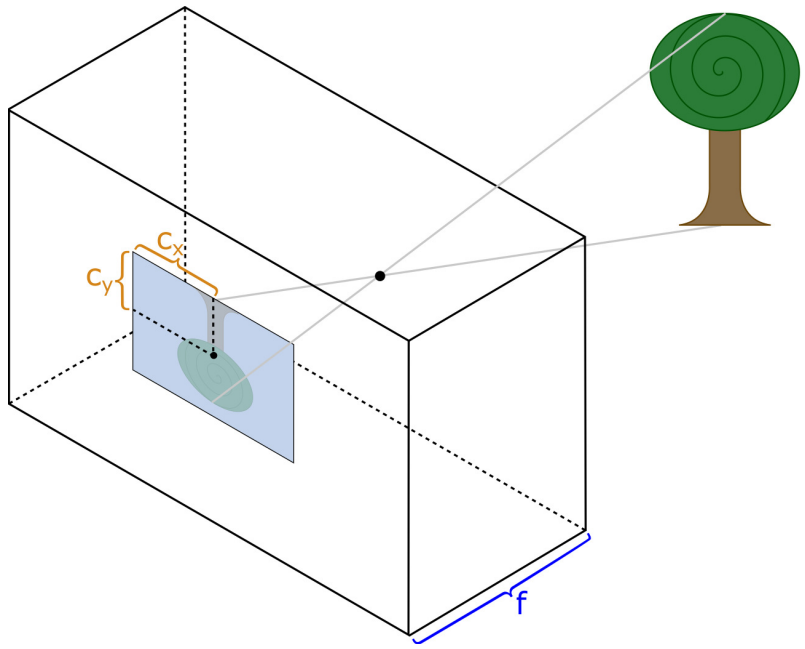


Figure 2.6: Intrinsic parameters of the camera. In blue, the focal distance f. Due to imperfections, it can eventually be different in the x and y directions. In orange, the principal point coordinates. It is usually located at the center of the image.

image plane by Equation (2.19).

$$s \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix}. \tag{2.19}$$

The parameters c_x and c_y are the coordinates of the principal point, that is usually at the center of the image and f_x and f_y are the focal lengths expressed in pixel units and represent the distance between the image plane and the pinhole. Figure 2.6 illustrates the physical meaning of those parameters. The s is simply a scaling factor that keeps our new 2D point in the "standard" representation ($w = 1$).

The values of c_x, c_y, f_x and f_y are specific to the camera and do not change, as long as the focal length is fixed. They are called the *intrinsic parameters*.

Equation (2.20) shows how the two matrices can be combined in a single opera-

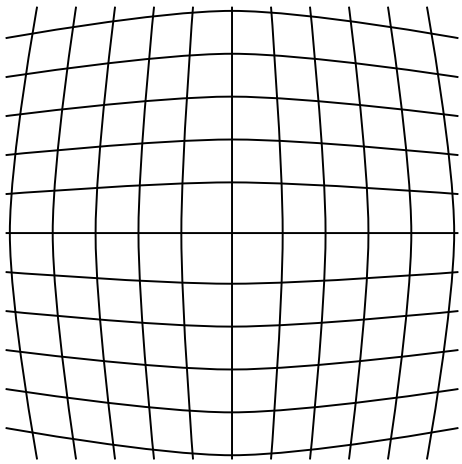


Figure 2.7: "Barrel effect" caused by radial distortion due to camera imperfections. It can be modeled and corrected by software.

tion.

$$s \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \mathbf{P} \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix}, \tag{2.20}$$

where P is called the *projection matrix*.

Finally, due to imperfections, real cameras can produce significantly distorted images. But, since those distortions are usually constant, they can be modeled and corrected by software, using the Brown-Conrady model [42, 43].

Two common forms of distortion are the *radial* and the *tangential* distortions. The radial distortion manifests in form of a "barrel effect", illustrated by Figure 2.7.

The radial distortion is corrected by equations (2.21) and (2.22).

$$x_{corrected} = x_i(1 + k_1 r^2 + k_2 r^4 + k_3 r^6), \tag{2.21}$$

$$y_{corrected} = y_i(1 + k_1 r^2 + k_2 r^4 + k_3 r^6), \tag{2.22}$$

where k_1, k_2 and k_3 are the distortion coefficients and r is the Euclidean distance from the pixel (x_i, y_i) to the center of the image (c_x, c_y), given by Equation (2.23).

$$r = \sqrt{(x_i - c_x)^2 + (y_i - c_y)^2}. \tag{2.23}$$

The tangential distortion occurs because the camera lenses are not perfectly parallel to the image plane. It can be corrected by equations (2.24) and (2.25).

$$x_{corrected} = x_i + 2\,p_1\,x_i\,y_i + p_2\,(r^2 + 2x_i^2), \tag{2.24}$$

$$y_{corrected} = y_i + p_1\,(r^2 + 2\,y_i^2) + 2\,p_2\,x_i\,y_i, \tag{2.25}$$

where p_1 and p_2 are distortion coefficients.

2.2.2 Camera calibration

In order to use images from the cameras to compute disparity, we need to know the extrinsic, intrinsic and distortion parameters. With the extrinsic and intrinsic parameters, we can relate points on both 2D images to points in the 3D world and the distortion parameters are used to correct problems caused by camera imperfections.

The process of computing those parameters is called *calibration*. The usual process consists of taking pictures of a known calibration pattern in several different positions and orientations. Calibration patterns must be easily and precisely recognizable. Figure 2.8 shows two commonly used patterns: the chessboard and the circle grid. Figure 2.9 shows a pair of pictures of a chessboard pattern, captured simultaneously by two cameras and the chessboard corners detected by the calibration software.

Once the images have been captured and the patterns have been detected, initial

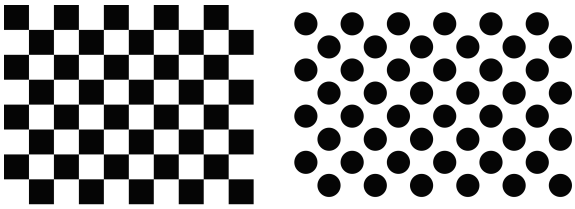
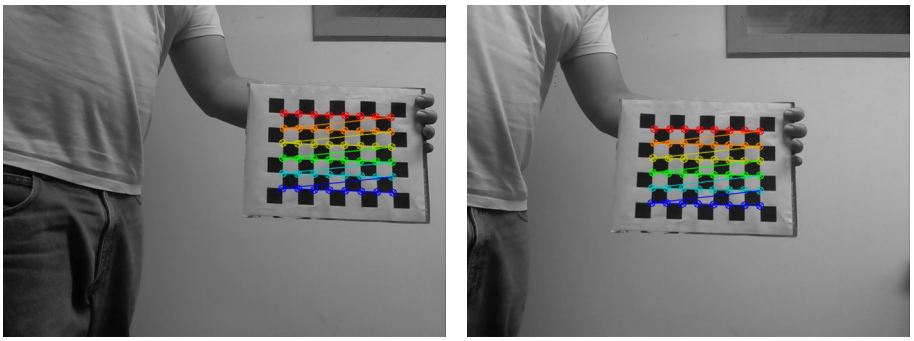


Figure 2.8: Commonly used calibration patterns. On the left, the chessboard, on the right, the circle grid. Both can be easily detected by calibration software.

(a) Left and right images of a chessboard pattern taken from a pair of cameras.



(b) Chessboard corners identified by the calibration program.

Figure 2.9: Capturing images for calibration and detecting chessboard corners.

values for the intrinsic and extrinsic parameters can be computed analytically. Those parameters are then refined using the Levenberg-Marquardt Algorithm to minimize the reprojection error. For a detailed explanation of the method, please refer to [44].

2.2.3 Image rectification

The disparity computation consists of finding corresponding points on the left and right camera images. Without any constraints, each pixel on one of the images requires a search in both dimensions of the other image. The process is significantly sped up if we assume that the cameras are perfectly aligned, because in that case corresponding pixels are always on the same line in both images, reducing the search to a single dimension.

However, even with high precision equipment, it can be impractical to maintain that level of alignment. To solve that problem, a process of *rectification* can be done in software. Using the parameters computed during the calibration process, it is possible to compute rotation matrices R_1 and R_2 and new projection matrices P_1 and P_2 that make both camera image planes the same. For details of this computation, please refer to [45].

Finally, with R_1, R_2, P_1, P_2 and the distortion coefficients, we can compute two *distortion and rectification maps* for each camera. Each map is a 2D matrix the same size of the image. One of the maps contains the new x position of the pixel at that location on the rectified and undistorted image. The other map contains the y position. Using those maps, we can undistort and rectify an image efficiently.

Figure 2.10 shows a pair of images after rectification. The distortion is removed from the images and the Region of Interest (ROI), where there are no artifacts generated by the undistortion process, is indicated by the red rectangle. The green lines are epipolar lines. When the cameras are properly calibrated, each epipolar

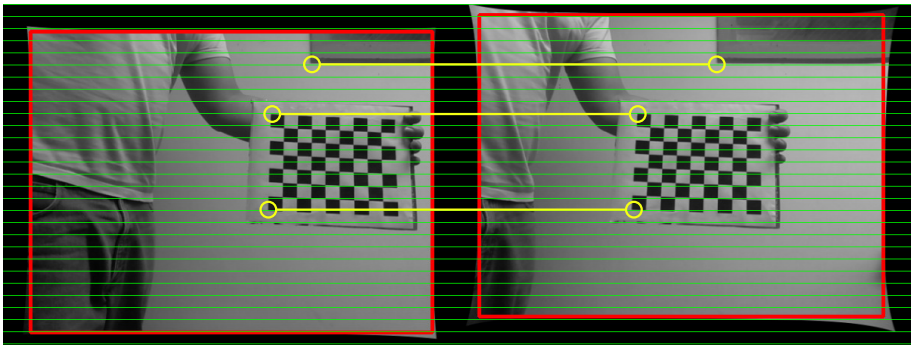


Figure 2.10: The same pictures from Figure 2.9 after rectification. The red square is the Region of Interest (ROI) and the green lines are epipolar lines. The yellow markers indicate points where the same object appears on both images and show that they are on the same epipolar line.

line intersects the same regions on both images. We indicated in yellow some points where this is easily verifiable. The corner of the window frame and the corners of the chessboard are on the same epipolar lines in both images.

2.2.4 Disparity computation

Once the images are rectified and undistorted, the depth information can be calculated. Given a point (x_l, y) on the left image and the corresponding point (x_r, y) on the right image, the *disparity* between the two points is defined by Equation (2.26). The 3D point that corresponds to both 2D points can, then, be computed by Equation (2.27). Equations (2.28), (2.29) and (2.30) convert the 3D point to the "standard" representation in homogeneous coordinates.

$$disparity(x_l, x_r) = x_l - x_r, \tag{2.26}$$

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -c_{xl} \\ 0 & 1 & 0 & -c_{yl} \\ 0 & 0 & 0 & f \\ 0 & 0 & -\frac{1}{b} & \frac{c_{xl}-c_{xr}}{b} \end{bmatrix} \begin{bmatrix} x_l \\ y \\ disparity(x_l, x_r) \\ 1 \end{bmatrix}, \tag{2.27}$$

$$x_{3d} = \frac{x_p}{w}, \tag{2.28}$$

$$y_{3d} = \frac{y_p}{w}, \tag{2.29}$$

$$z_{3d} = \frac{z_p}{w}, \tag{2.30}$$

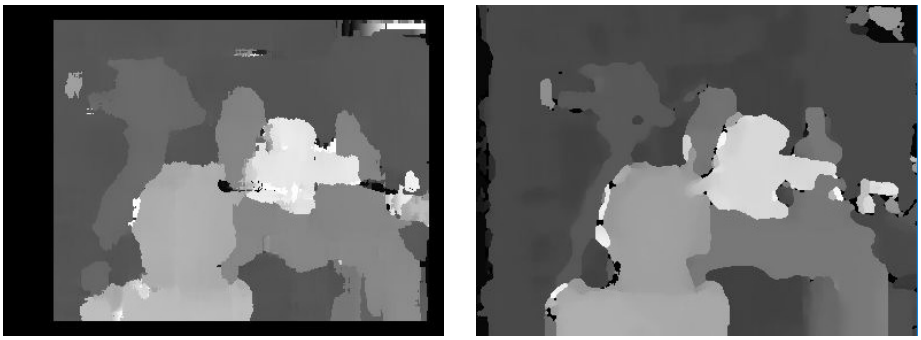
where c_{xl} and c_{yl} are the c_x and c_y intrinsic parameters of the left camera, c_{xr} is the c_x intrinsic parameter of the right camera and b is the baseline distance (the distance between the two cameras).

Despite the simplicity of the math involved in the process of reprojecting matching points to 3D, finding the pairs of matching points can be challenging. A very large number of approaches can be found in the literature. In fact, there are stereo vision benchmarks such as the KITTI Vision Benchmark Suite [46] that maintain a ranking of the algorithms that perform best on a particular dataset.

Some of the most widespread algorithms are the Block Matching Algorithm [47] and the Semi-Global Block Matching Algorithm [48], that are available in OpenCV. A qualitative analysis of the algorithms shows that the former is faster, while the latter generates better results. Those observations are confirmed by the ranking available at the KITTI Vision Benckmark Suite website[1]: the Block Matching Algorithm runs in 0.1s and produces 25.27% of outliers while the Semi-Global Block Matching Algorithm runs in 1.1s and produces 10.86% of outliers. For detailed information about the algorithms, please refer to [47] and [48].

[1] http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo

(a) Left and right images from the Tsukuba Dataset.



(b) Block Matching Algorithm.

(c) Semi-Global Block Matching Algorithm.

Figure 2.11: Disparity map generation from a pair of images of the Tsukuba Dataset.

Figure 2.11 shows the disparity image computed by both algorithms using images from the Tsukuba Dataset [49, 50]. Algorithm parameters were manually adjusted using the Stereo Tuner application created during the development of this project[2]. Brighter colors indicate points with higher disparity, where the objects are closer to the camera.

2.3 Image Operations

Images are usually represented in a computer as two-dimensional arrays of pixels. Operations can be executed on those pixels in order to achieve a desired effect. In the following subsections, we introduce some basic image operations that are commonly

[2]https://github.com/guimeira/stereo-tuner

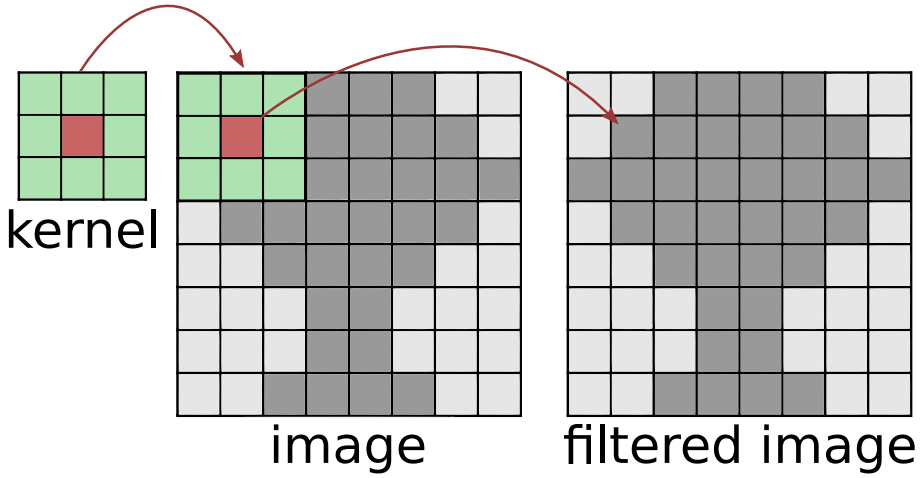


Figure 2.12: Filtering an image using a 3x3 kernel. The filtered image is generated by sliding the kernel over the original image. The pixel at the origin of the kernel (indicated in red) is generated by some operation involving itself and the pixels around it (indicated in green).

used in Computer Vision and how they are implemented.

2.3.1 Filtering

Filtering operations modify each pixel of an image based on its current value as well as the value of other pixels around it. Each pixel of the filtered image uses information from a neighborhood of pixels in the original image and the contribution of each pixel is defined by a *kernel*. Linear filters compute weighted sums of pixel values while morphological operations are implemented using minimum and maximum values.

Figure 2.12 illustrates how the filtering process works. To generate the filtered image, we slide the kernel through the original image. The pixels covered by the kernel are used for the computation of the new value of the pixel at the kernel origin. The origin is usually located at the center of the kernel, but that is not mandatory.

When part of the kernel is outside of the image, the values for the non-existing pixels need to be extrapolated. Some commonly used extrapolation methods include

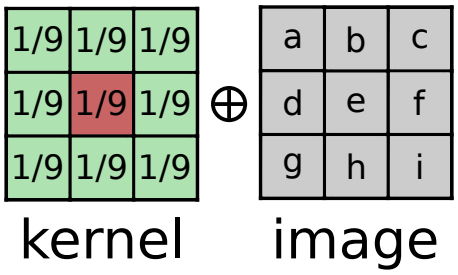


Figure 2.13: Application of a normalized box filter to a portion of the image. Each cell in the kernel determines the weight assigned to a pixel in the computation.

assuming the non-existing pixels are zero, or repeat the values of the pixels at the border. For some applications, it is preferable to use no extrapolation and make the filtered image smaller than the original.

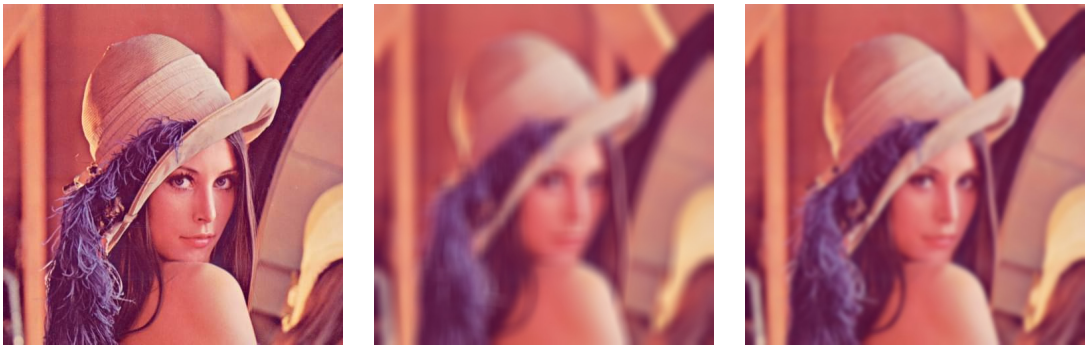
2.3.2 Blur and Gaussian Blur

Blur is the simplest filter that can be applied to an image. It is a linear filter and its kernel is a *normalized box filter*, shown in Equation (2.31). In other words, each pixel is the average of the pixels covered by the kernel. Figure 2.13 shows a 3x3 kernel applied to a portion of an image. In this example, the new value for the pixel at the center of the kernel given by Equation (2.32). This operation is known as a *Box Blur* and is used as a simple method to reduce noise.

$$K = \frac{1}{w_{kernel} \cdot h_{kernel}} \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 1 \end{bmatrix}, \tag{2.31}$$

$$e_{new} = \frac{1}{9}a + \frac{1}{9}b + \frac{1}{9}c + \frac{1}{9}d + \frac{1}{9}e + \frac{1}{9}f + \frac{1}{9}g + \frac{1}{9}h + \frac{1}{9}i. \tag{2.32}$$

Instead of a simple box filter, we can generate a kernel using a two-dimensional



(a) Original Image. (b) Box Blur. (c) Gaussian Blur.

Figure 2.14: Application of a Box Blur and a Gaussian Blur on an example image. Both of them use a 21x21 kernel and the Gaussian Blur uses a standard deviation $\sigma = 3.5$ (the default value for this kernel size on OpenCV)

Gaussian function in Equation (2.33).

$$K(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.33}$$

where σ is the standard deviation of the Gaussian distribution. The filtering operation using that kernel is known as a *Gaussian Blur* and is also commonly used for noise reduction.

Figure 2.14 shows the application of both kinds of blur on an example image using a 21x21 kernel.

2.3.3 Erosion and Dilation

Erosion is a morphological operation. Instead of computing a weighted average of the pixels covered by the kernel, we simply choose the pixel with minimum value and assign that value to the pixel at the kernel origin.

Figure 2.15 shows the application of a 3x3 kernel to erode a small part of an image. The new value of the pixel at the center of the kernel is given by Equation

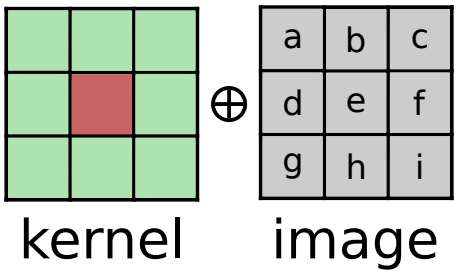


Figure 2.15: Application of Erosion/Dilation using a 3x3 kernel. In this case, we don't need values on the kernel cells, the kernel only determines the pixels that are going to be compared in the operation.

(2.34).

$$e_{new} = min(a, b, c, d, e, f, g, h, i). \tag{2.34}$$

The Dilation operation works in the same way, but instead of taking the minimum value covered by the kernel, we take the maximum. In the example shown in Figure 2.15, the new value of the pixel at the origin is given by Equation (4.8).

$$e_{new} = max(a, b, c, d, e, f, g, h, i). \tag{2.35}$$

Kernels for Dilation and Erosion do not have weights on their cells, but they are not necessarily square or rectangular. Cross and ellipse shaped kernels are also commonly used for those operations.

Figure 2.16 shows the application of the Erosion and Dilation operations on an example image.

2.3.4 Opening and Closing

Opening and Closing are morphological operations built from the combination of Erosion and Dilation. Opening of an image I is defined bu Equation (2.36).

$$opening = dilate(erode(I)). \tag{2.36}$$

(a) Original image. (b) Dilation. (c) Erosion.

Figure 2.16: Eroding and dilating an image using a square kernel.

Opening can be used to remove small objects from the foreground, placing them in the background.

Closing works in the opposite way and is defined by Equation (2.37).

$$closing = erode(dilate(I)). \tag{2.37}$$

It is used to remove small holes in the foreground, turning small islands of background into foreground.

Figure 2.17 shows the steps of the Opening and Closing operations.

2.4 Path Planning

As mentioned before in Section 1.2.2, path planning can be applied to different problems with a varying level of complexity, potentially dealing with a large number of dimensions and constraints. In this project, the world is modeled as a 2D grid, so in this section we will limit the discussion to this scenario.

Our goal is to find the shortest path in a 2D grid while avoiding obstacles. A visual representation of that problem is given by Figure 2.18. In the next subsections, we describe in details the algorithms that are commonly used for this kind of

(a) Opening.

(b) Closing.

Figure 2.17: Demonstration of the Opening and Closing operations. Opening can remove noise from the background while Closing removes it from the foreground.

problem.

2.4.1 Problem representation

A common approach to the 2D path planning problem is to represent it as a graph. Using that representation, we can make use of well known graph search techniques that are guaranteed to give optimal results under certain conditions.

Usually, the graph representation of a 2D grid uses nodes to represent the grid cells and edges to represent the movement from any given cell to its neighbors. An edge is created between a node and each one of its neighbors that does not contain an obstacle. Two grid cells are considered neighbors according to the connectivity of the grid. The grid is said to be *4-connected* if each cell (x, y) is connected to the four cells with coordinates $(x \pm 1, y)$ or $(x, y \pm 1)$. In other words, only horizontal and vertical movement is allowed. If diagonal movement is allowed, the grid is said to be *8-connected*. In this case, each cell (x, y) is connected to the four cells that it would be connected to in a 4-connected grid and also to the ones with coordinates

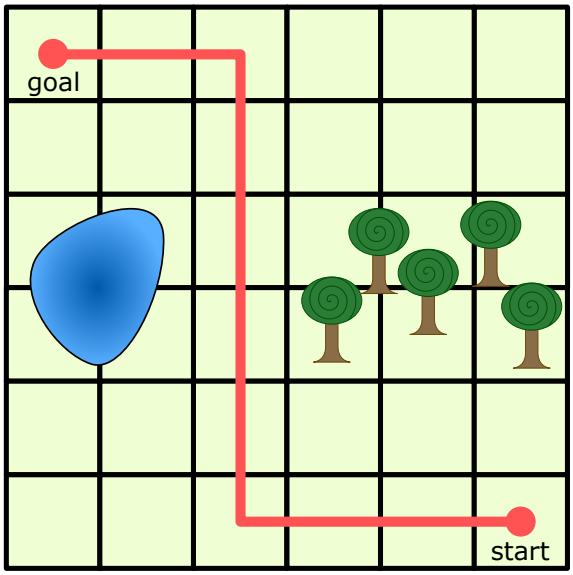


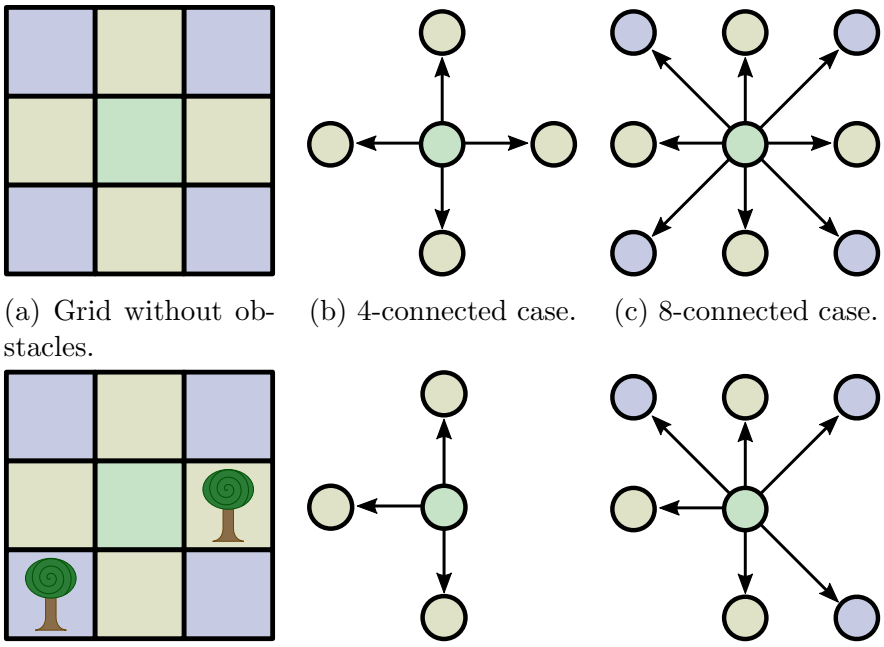
Figure 2.18: Representation of our path planning problem. Given a 2D grid, a start and an end positions, we need to find the shortest path between the two points that doesn't collide with obstacles.

$(x \pm 1, y \pm 1)$ and $(x \pm 1, y \mp 1)$.

Figure 2.19 illustrates the conversion from a grid to a graph. Each cell is represented by a node on the graph. On the first row, there are no obstacles, so in the 4-connected case (second column) the center cell is connected to four neighbors and in the 8-connected case (third column) the center cell is connected to eight neighbors. On the second row, 2 obstacles were added, to illustrate how they are translated to the graph representation. The graph is not aware of the existence of obstacles, they are encoded in the graph as the lack of a connection between the occupied cell and its neighbors.

2.4.2 Breadth First Search

Breadth First Search (BFS) is the simplest graph search algorithm, but the more sophisticated ones use a similar structure, with improvements, so we will start by looking into it. The algorithm uses a queue of nodes that represent an expanding



(a) Grid without obstacles. (b) 4-connected case. (c) 8-connected case.

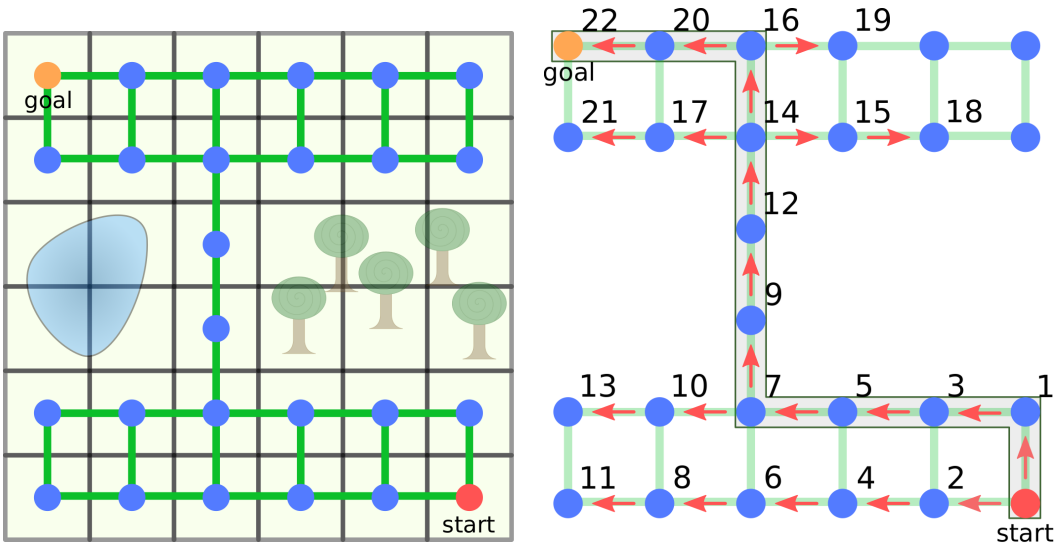
(d) Grid with obstacles. (e) 4-connected case. (f) 8-connected case.

Figure 2.19: Conversion between a grid and a graph.

frontier. The frontier initially contains the start node. On each step of the algorithm, nodes are removed from the frontier and its neighbors are analyzed. Each one of the neighbors that have not been visited yet are added to the back of the queue. The algorithm ends when the goal node is taken from the frontier.

In order to return a path from start to goal, the algorithm stores for each visited node a reference to the node that added it to the queue. When the goal is found, the path is generated by following those references back to the start position.

Figure 2.20 illustrates the use of BFS on the grid from Figure 2.18. The numbers next to the nodes indicate the order in which they are visited in the process. Initially, the start node is added to the queue. When it is removed from the queue, it adds its neighbors (2 and 3) to it. After that, node 1 is removed and inserts 3 in the queue. Then node 2 is removed and adds 4. The process continues until node 20 adds node 22, that is the goal node. The path is, then, generated by following the arrows from



(a) Example world. (b) BFS execution.

Figure 2.20: An execution of the BFS algorithm. Initially, the queue contains the *start* node. Numbers next to the nodes indicate the order that they are added to the queue. Arrows are references to the "parent" of each node during the execution of the algorithm. The shaded box shows the path found by the algorithm.

the goal (22-20-16-14-12-9-7-5-3-1). A description of the implementation is given in Algorithm 1.

Breadth First Search is able to find and optimal path from the starting node to the goal, *i.e.*, a path with a minimum number of nodes. However, it assumes that any movement from a node to its neighbor has the same cost, which may not be true even in simple cases. Figure 2.21 illustrates one of those cases, in which the cost of traveling between nodes is given by the distance between the center of the cells they represent. In this case, traveling horizontally or vertically is cheaper than diagonally.

2.4.3 Dijkstra's Algorithm

Dijkstra's Algorithm is similar to the Breadth First Search, but incorporates the notion of *cost* to each edge. Depending on how the costs are arranged on the graph,

Algorithm 1 Breadth First Search

```
1: function BreadthFirstSearch(start, goal)
2:     frontier ← empty queue
3:     cameFrom ← empty map
4:     Insert start into frontier
5:     cameFrom[start] ← none
6:     while frontier is not empty do
7:         current ← frontier.first() ▷ Removes the first element from the queue
8:         if current = goal then
9:             return GeneratePath(cameFrom, start, goal)
10:        end if
11:
12:        for all neighbors of current as next do
13:            if next is not in cameFrom then
14:                Insert next into frontier
15:                cameFrom[next] ← current
16:            end if
17:        end for
18:    end while
19: end function
20: function GeneratePath(cameFrom, start, goal)
21:    path ← empty list
22:    Insert goal into path
23:    current ← goal
24:    while current ≠ start do
25:        current ← cameFrom[current]
26:        Insert current into path
27:    end while
28:    return path
29: end function
```

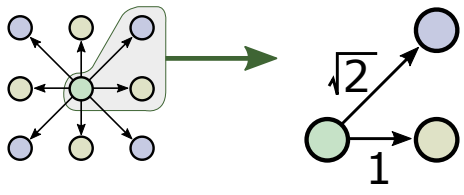


Figure 2.21: Graph with different costs for different edges. Traveling horizontally or vertically is cheaper than traveling diagonally. Breadth First Search cannot be applied in such graph.

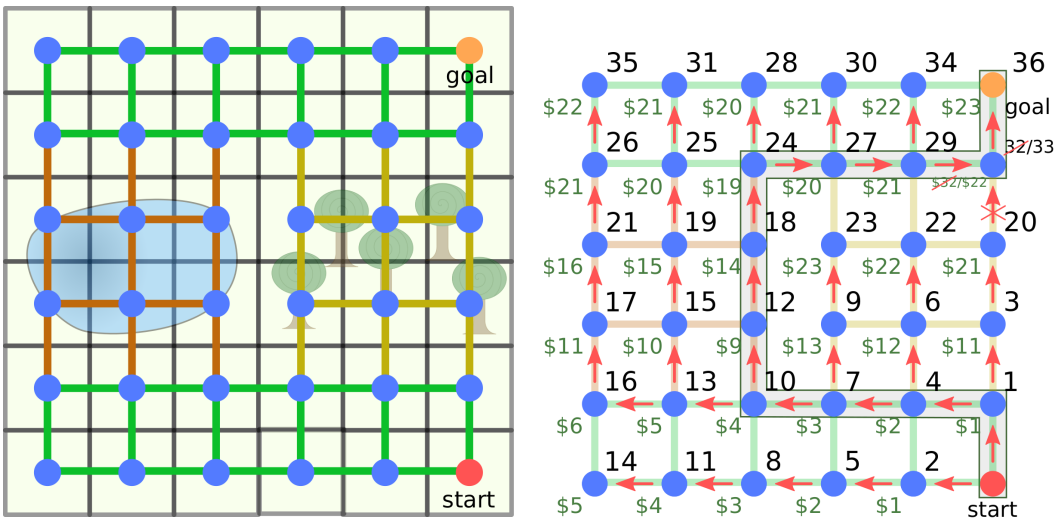
it might be cheaper to visit a higher number of nodes.

The difference from Dikstra's Algorithm to BFS is that it uses a priority queue instead of a regular queue. The nodes in the queue are sorted according to their *priorities*. The priority of each node is the total movement costs from the start node. This change makes it possible to visit a node more than once with different costs. In that case, the node will be added to the frontier again if we reach it with a smaller cost than before. Algorithm 2 describes the implementation of the Dijkstra's Algorithm.

Algorithm 2 Dijkstra's Algorithm

```
 1: function DIJKSTRA(start, goal)
 2:     frontier ← empty priority queue
 3:     cameFrom ← empty map
 4:     costSoFar ← empty map
 5:     Insert start into frontier with priority = 0
 6:     cameFrom[start] ← none
 7:     costSoFar[start] ← 0
 8:     while frontier is not empty do
 9:         current ← frontier.first() ▷ Removes the first element from the queue
10:         if current = goal then
11:             return GENERATEPATH(cameFrom, start, goal) ▷ Same as in BFS
12:         end if
13:
14:         for all neighbors of current as next do
15:             newCost ← costSoFar[current] + COST(current, next)
16:             if next is not in cameFrom and newCost < costSoFar[next] then
17:                 costSoFar ← newCost
18:                 Insert next into frontier with priority = newCost
19:                 cameFrom[next] ← current
20:             end if
21:         end for
22:     end while
23: end function
```

The $Cost(from, to)$ function used in the algorithm returns the cost to move from node $from$ to node to and is dependent on the problem. Figure 2.22 shows an



(a) Example world. (b) Dijkstra's Algorithm execution.

Figure 2.22: An execution of Dijkstra's Algorithm. Now we assume it's possible to walk on the lake with cost 5 and on the forest with cost 10. Walking on the land has cost 1. The black numbers near the nodes indicate the order in which they were visited. The green numbers indicate the smallest cost to move from the start to that node.

execution of the algorithm. This grid is similar to the one in Figure 2.18, but the goal position and the sizes of the lake and the forest are changed. Also we consider that it is possible to walk through the lake with cost 5 and through the forest with cost 10. Walking on the land has cost 1. The algorithm finds the path with the lowest cost, going through the lake, instead of the one with the least number of nodes, that would require going through the forest.

2.4.4 A*

After an execution of Dijkstra's Algorithm, we have the smallest cost to go from the start node to every node that is visited during the execution. However, when we are interested in only one goal point, we can speed up execution by visiting less nodes. The A* (pronounced *A-star*) algorithm accomplishes that by using a *heuristic*.

The heuristic is used during the computation of the priority of a node in the

priority queue. It estimates the cost to move from the current node to the goal. Instead of using only the current cost from start to the node as the priority, we add to it the estimated cost from the node to the goal. This prioritizes nodes that are closer to the goal and penalizes nodes that are moving away from it.

Algorithm 3 shows the implementation of A*. It is identical to Dijkstra's Algorithm, except for the priority computation when the node is inserted in the priority queue.

Algorithm 3 A*

```
1: function ASTAR(start, goal)
2:     frontier ← empty priority queue
3:     cameFrom ← empty map
4:     costSoFar ← empty map
5:     Insert start into frontier with priority = 0
6:     cameFrom[start] ← none
7:     costSoFar[start] ← 0
8:     while frontier is not empty do
9:         current ← frontier.first() ▷ Removes the first element from the queue
10:        if current = goal then
11:            return GENERATEPATH(cameFrom, start, goal) ▷ Same as in BFS
12:        end if
13:
14:        for all neighbors of current as next do
15:            newCost ← costSoFar[current] + COST(current, next)
16:            if next is not in cameFrom and newCost < costSoFar[next] then
17:                costSoFar ← newCost
18:                Insert next into frontier with priority = newCost +
   HEURISTIC(goal, next)                    ▷ The difference from Dijkstra's
19:                cameFrom[next] ← current
20:            end if
21:        end for
22:    end while
23: end function
```

The function $Heuristic(from, to)$ estimates the distance between two nodes. When using grids, two commonly used distance metrics are the Euclidean Distance and the Manhattan Distance, illustrated by Figure 2.23.

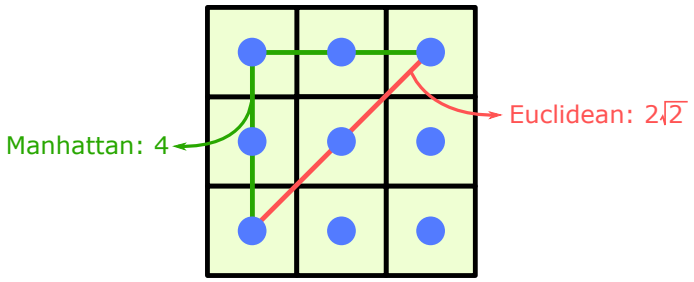


Figure 2.23: Two commonly used distance metrics. The green line is the Manhattan Distance and the red line is the Euclidean Distance.

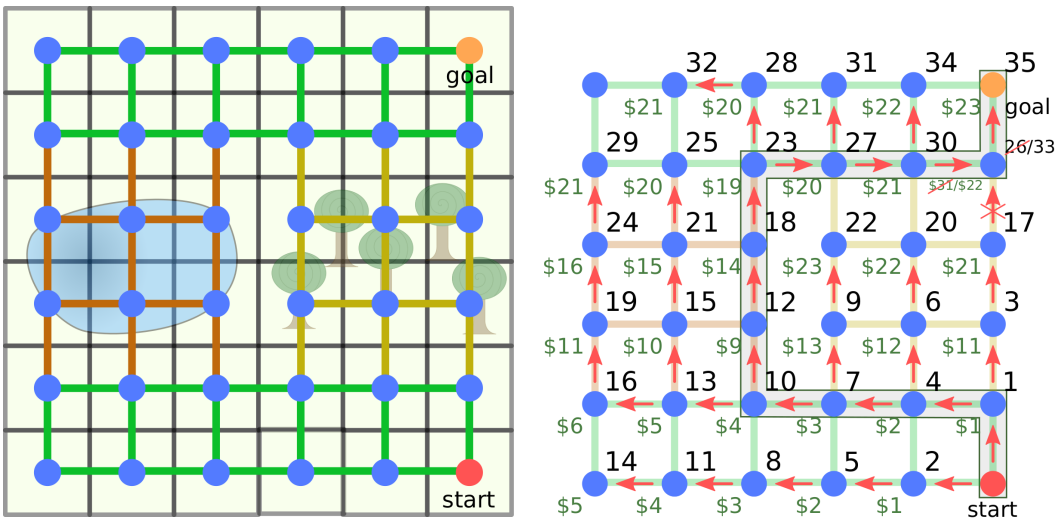
The Euclidean Distance is the length of a straight line between two points, given by Equation (2.38). The Manhattan Distance is the sum of the absolute differences of the cartesian coordinates of two points, given by Equation (2.39).

$$euclidean(p1, p2) = \sqrt{(p1.x - p2.x)^2 + (p1.y - p2.y)^2}, \tag{2.38}$$

$$manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|. \tag{2.39}$$

Even though A* uses a heuristic during its execution, it will find an optimal path, like Dijkstra's Algorithm does, as long as the heuristic is *admissible*. An admissible heuristic never overestimates the cost to the goal. The Euclidean Distance is admissible, because the shortest distance between two points is a straight line. The Manhattan Distance can be used as a admissible heuristic on 4-connected grids, but it is not admissible on 8-connected grids. It is also possible to give up optimality and search more aggressively towards the goal by applying weights to the heuristic and the cost so far when computing the priority of the nodes.

Figure 2.24 shows an execution of the A* algorithm. The scenario is the same as the one on Figure 2.22. The algorithm visits one less node in this execution when compared to Dijkstra's. This is a simple and small test case and the improvement



(a) Example world. (b) A* execution.

Figure 2.24: An execution of the A* algorithm. The upper left node is not visited during the search. The highlighted path is the same obtained by Dijkstra's and has optimal cost.

looks irrelevant, but depending on the size and the configuration of the graph, A* can greatly outperform Dijkstra's and in a bad scenario, it will visit at most the same number of nodes. This makes A* the preferred choice for this kind of problem.

2.5 Summary

This chapter presented the background information that will be used in the next chapters of this thesis. Section 2.1 presented transformations that can be applied to points in a 3D space, as well as their mathematical representation. It also introduced homogeneous coordinates as a solution to the problem of not being able to represent translations as a matrix multiplication. Using homogeneous coordinates, an arbitrary sequence of translations, rotations and scalings can be encoded in a single matrix, obtained by the multiplication of the matrices that represent each one of the transformations. Section 2.2 explained the details of a stereoscopic vision

system and how to process a pair of images in order to obtain depth information. The camera model was introduced as well as equations to project a point into an image and to reproject the point to a 3D world using a pair of images. Popular matching algorithms were briefly presented as well. Section 2.3 introduced basic image operations and how they can be implemented using filters. We presented the Gaussian Blur operation, that can be used to reduce noise, as well as the Dilation and Erosion operations, that can be combined to build the morphological opening and closing. Finally, Section 2.4 discussed the path planning problem in two dimensions. It demonstrated how the problem can be reduced to a graph search and how graph search algorithms can be applied to it. Then, graph search algorithms were presented. We started with the simplest one, the Breadth First Search. In order to support paths with different costs, it was modified into the Dijkstra's Algorithm. Then, an heuristic was added to the algorithm in order to speed it up, leading to the A*.

Chapter 3 presents our autonomous vehicle testbed, the Robocart. It describes all the hardware that is available on the vehicle, as well as the communication protocols that connect them.

Chapter 3

Autonomous Vehicle Testbed

The Robocart is the 1995 Club Car[1] golf cart shown on Figure 3.1 that is being modified to become a research platform for autonomous vehicles. Several teams have worked on it in order to implement the systems that turn the golf cart into a computer-controllable vehicle, while this project focuses on its first path planning and decision making capabilities. This chapter describes each one of the systems available on the Robocart, as well as the communication protocols between them and the server. Section 3.1 describes the vision system composed by a pair of Raspberry Pi cameras. Section 3.2 introduces the compass sensor that allows the robot to be aware of the direction it's facing. The GPS sensor is presented in Section 3.3. Sections 3.4, 3.5 and 3.6 present the mechanical systems for throttle, braking and steering respectively. Finally, Section 3.7 describes the computer that controls the other systems.

[1] http://www.clubcar.com/

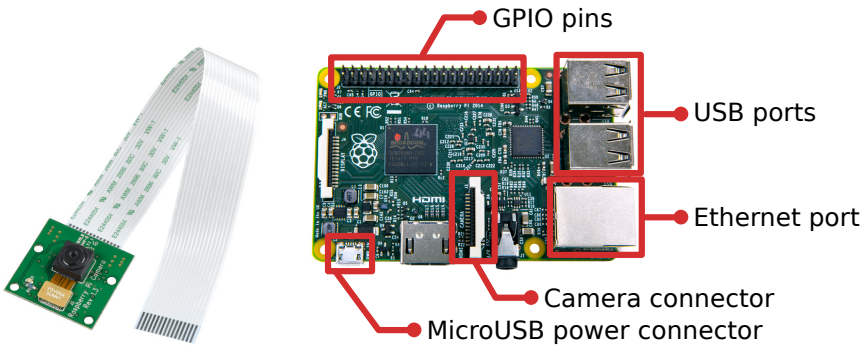
Figure 3.1: The Robocart research platform.

3.1 Cameras

In the original project of the Robocart, the heavy processing and decision making would be done in a fixed server that would communicate with the vehicle via WiFi. As a result, the previous team proposed a vision system composed by a pair of Raspberry Pi computers, each one of them connected to a Raspberry Pi Camera Module. The Raspberry Pi is a credit card-sized single board computer, developed by the Raspberry Pi Foundation with the objective of promoting the teaching of basic computer science in schools and developing countries [51].

On the original design, the computers would send the images to the server using an Universal Serial Bus (USB) WiFi adapter and receive back the commands to be executed using the General Purpose Input and Output (GPIO) pins available on the device. The current team, however, decided to move the server to the Robocart, so that it doesn't depend on a wireless connection. Since we already had the two Raspberry Pis and the cameras, we decided to keep using them as part of the vision system, but connected them to the server via ethernet.

The boards used in this project are Raspberry Pi 2 model B. They have a 900MHz quad-core ARM Cortex-A7 Central Processing Unit (CPU), 1GB of Random Access



(a) The Camera Module.

(b) The Raspberry Pi 2 with important parts of the board indicated.

Figure 3.2: Camera Module and the Raspberry Pi.

Memory (RAM), 4 USB ports, 40 GPIO pins, High-Definition Multimedia Interface (HDMI) and ethernet ports [52] and a VideoCore IV 3D graphics core. The Camera Module is also developed by the Raspberry Pi Foundation, specifically for the Raspberry Pi. It is a 5 megapixel camera, with support for video modes 1080p30, 720p60 and VGA90, as well as still captures [53]. Figure 3.2 shows the board with important components indicated as well as the Camera Module.

The decision of using the Raspberry Pi for the image capture added complexity to the project. While capturing images from an USB camera connected to the computer is very simple with the appropriate libraries, the capture using the Raspberry Pi required the development of customized software to interface with the cameras and transmit the data over the network. This software was developed as part of this project.

The access to the Raspberry Pi Camera Module is done using the Multi-Media Abstraction Layer (MMAL) framework. It is a low level interface to multimedia components running on the VideoCore. When this project started to be developed, there was very little documentation about MMAL available, the best sources of information about the framework being the example applications `raspistill` [54]

and `raspivid` [55], two text-based applications that can respectively capture a still image and record a video.

Some third-party applications that interface with the Raspberry Pi Camera invoke one of those applications to obtain the data they need and then process it. Others use MMAL to communicate with the camera directly. The solution we chose for our capture application is the official Video4Linux 2 (V4L2) driver, `bcm2835-v4l2`, developed and maintained by the Raspberry Pi Foundation. V4L2 is an Application Programming Interface (API) and a collection of device drivers for supporting video capture devices on Linux systems. This driver handles the communication using MMAL and exposes the video capture capabilities via the V4L2 API. Using this approach, we can interface with a well-known and documented API and we avoid the inter-process communication overhead of invoking another application to obtain the data.

Regarding the transmission of the data over an Internet Protocol (IP) network, the two available options are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is a connection-oriented protocol and offers guaranteed and in-order delivery of the data. UDP is connectionless and does not provide any kind of guarantee regarding the delivery of packets or the order in which they are delivered. Despite all the advantages offered by TCP, real time internet applications often opt for UDP because of the problems TCP introduces in order to provide its reliability guarantees. One of the main problems happens when a packet is lost. In that case, the receiver must wait until the sender notices the loss and retransmits the packet. In real time applications, it might be preferable to simply ignore the lost packet and continue instead of requesting a retransmission, because when the packet is finally received, it might already be irrelevant [56].

Based on that argument, our first attempt was based on UDP. We built a simple

application that reads JPEG frames using the V4L2 API and transmits them using UDP. Two problems arose from this approach. First, because UDP offers no flow control, the amount of data sent by the application flooded the network connection, interrupting the other connections, including our Secure Shell (SSH) session. The second problem is that the size limit of a UDP packet, imposed by the underlying IPv4, is approximately 64kB. Due to the compression method used (JPEG), different images have different sizes, even if their dimensions are the same, so some of them fit in a single packet and some do not. In order to use UDP, we would need to implement some kind of flow control to avoid flooding the connection, as well as some mechanism to send images in multiple packets and guarantee that they are delivered in order.

At this point, we started to investigate TCP. It offers the features we need in a solid and well tested implementation and, since our applications run on a wired local network containing only the server and the two Raspberry Pi computers, packet loss should not be a problem. The TCP-based application behaved better than the UDP-based one, but we still needed to address another problem: synchronization.

In a stereovision system, specially a moving one, it is important that the images are captured at the same time. In professional stereovision equipment, dedicated hardware is used to ensure that both cameras capture exactly at the same time. In our setup, it is not possible to obtain that kind of synchronization, so we use software to synchronize the images as much as we can.

Our synchronization is based on the clocks of the server and of the Raspberry Pi computers. To synchronize the clocks, we use the Precision Time Protocol (PTP), defined by the IEEE standard 1588. PTP is designed for computers connected via a local network that require accuracy beyond the attainable using the Network Time Protocol (NTP) [57]. PTP implementations can achieve higher accuracy by using

the network card to perform timestamping in hardware, but even though our server does have hardware timestaping capabilities, they are not present on the Raspberry Pi. In that case, software timestamping is supported by the Linux kernel and can also be used.

There are several implementations of PTP available on the internet. In this project, we used the one developed by the Linux PTP Project [58], which is open-source, released under the GNU is Not Unix (GNU) General Public License (GPL), and supports both hardware and software timestamping. In our setup, the server acts as the master and each Raspberry Pi is a slave. In our tests, the `ptp4l` program reports clock offsets in the order of tens of thousands of nanoseconds.

The capture application running on the Raspberry Pi continually captures images from the camera, but instead of sending every image to the server over the network, it stores the images on a circular buffer, along with the timestamp that represents the time in which the image was captured. When the server needs a pair of images, it sends the current value of its clock to the two computers. The computers, then, compare this timestamp with each one in the buffer and return the image whose timestamp is closest to the one sent by the server. Figure 3.3 illustrates this process.

With this approach, we can capture as many frames from the camera as possible, because we donot have to wait for a frame to be transmitted before we can capture a new one. And with a higher number of captured frames, we have more chances of obtaining two frames that were captured almost simultaneously.

Figure 3.4 shows a simple test to determine if the images are being captured at the same time. We show a running stopwatch to both cameras and use the method described above to capture a pair of pictures. In our tests, we are able to consistently obtain results as the one shown on Figure 3.4, in which the images are synchronized

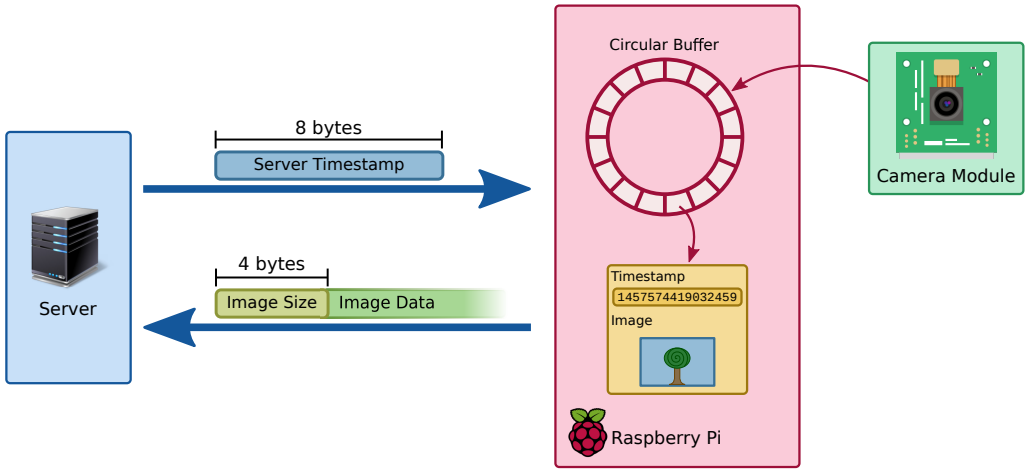


Figure 3.3: Illustration of the capture protocol. The circular buffer is continuously fed with images from the camera and their respective timestamps. When the server needs an image, it sends its current timestamp to the Raspberry Pi. The image with the closest timestamp is chosen from the buffer and sent back to the server.

to the hundredth of a second.

3.2 Compass

In order to drive towards a goal point, the robot needs to know in which direction it is facing. Since, at its current state, the Robocart does not have odometry sensors in place, we added a digital compass, or magnetometer, to it.

Magnetometers are instruments capable of measuring the strength and the direction of a magnetic field. Since the planet has a magnetic field that is roughly aligned with its rotation axis, a magnetometer can be used to determine the direction in which it's facing.

The integrated circuit we chose as our magnetometer is the HMC5883L [59], a 3-axis digital compass. It is a popular chip, because it is simple and affordable. Breakout boards containing the chip as well as the other required circuitry such as capacitors, resistors and voltage regulators, can be easily found online.

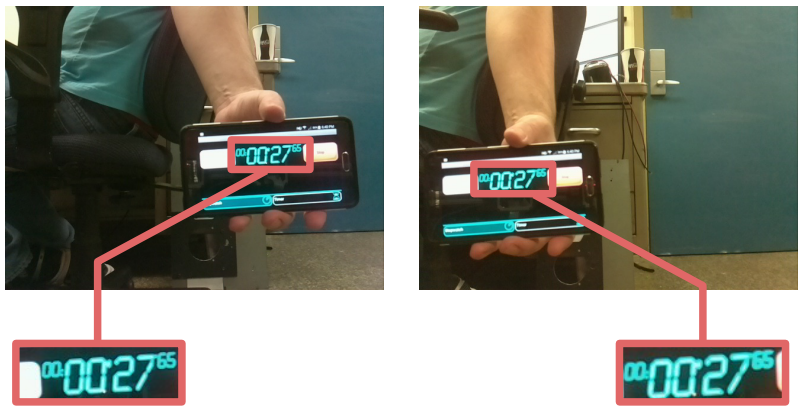


Figure 3.4: Testing the synchronization of the images captured by our cameras. A running stopwatch is shown to the cameras and the images are captured. The value read in both images is the same, indicating that the captures are synchronized to the hundredth of a second.

The communication with the chip is done via a Inter-Integrated Circuit (I2C) bus. An I2C bus is composed by 2 lines, the Serial Data Line (SDA) and the Serial Clock Line (SCL). A master node generates the clock and initiates the communication with slave nodes connected to the bus. Slave nodes receive the clock and respond to the master when requested. Each slave on the bus has an address that the master uses when it wants to communicate with a node.

I2C is a widespread standard and there are implementations of its message protocol for most platforms, specially embedded ones. The Raspberry Pi has built in support for I2C, with dedicated SDA and SCL pins on its GPIO header. Kernel support must be enabled using the tool `raspi-config` [60]. Once the kernel support is enabled, the I2C bus is detected as a device in the `/dev` directory and the communication is done using the `open`, `ioctl`, `write` and `read` calls. Figure 3.5 shows the HMC5889L integrated circuit and how it is connected to the GPIO pins of a Raspberry Pi 2.

The communication with the magnetometer is simple. It has 13 8-bit registers that can be accessed by the master. To interact with the magnetometer, the master

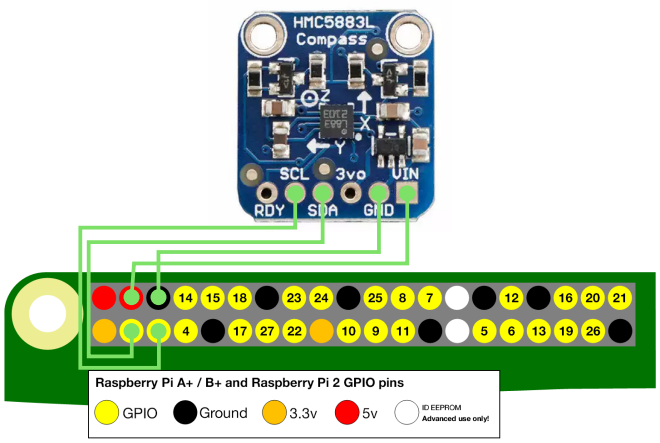


Figure 3.5: The HMC5883L breakout board and the connections to the Raspberry Pi GPIO header. SDA and SCL are connected to the dedicated I2C pins on the Raspberry Pi. VIN can be connected to any 3.3V or 5V pin. GND can be connected to any ground pin. RDY signal indicates when a measurement is ready. It's not necessary as long as the master device does not violate the timing constraints of the chip.

writes a byte on the bus that contains the address of the slave node and a bit indicating if it is a read or write operation. It is followed by another byte indicating the register that needs to be read or written. On a write operation, the master then writes the new value of the register. On a read operation, the device responds writing the current value of the register on the bus.

Registers 0 to 2 are configuration and mode registers and specify how the device must operate. Those registers are set at the beginning of the execution. Register 9 is a status register and registers 10 to 12 are identification registers. The registers that are actually used during the execution of the program are registers 3 to 8. They contain a 2-byte representation of the magnetic field strength in the directions X, Y and Z. For a detailed description of each one of the registers, please refer to the circuit's datasheet [59].

The HMC5883L offers several levels of sensitivity. The intensity of Earth's magnetic field varies from 22 to 67 gauss (G) [61], so the sensitivity of ± 1.3 G is enough

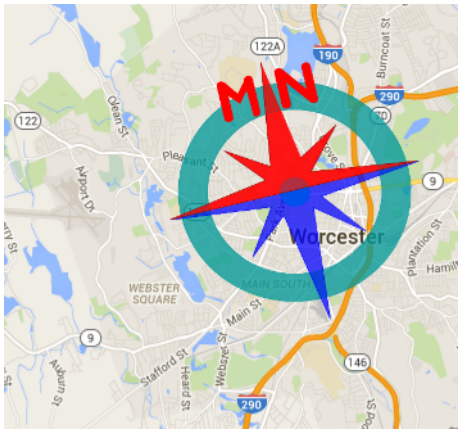


Figure 3.6: Declination value in the region of Worcester according to the NOAA website. The magnetic north is approximately 14 degrees to the west of the actual geographic north.

for our purposes. According to the datasheet, for that sensitivity level, the values read from the chip must be multiplied by 0.92 to obtain the actual magnetic field values.

The X, Y and Z values describe a vector that points towards the magnetic north. Assuming that the magnetometer is level to the Earth's surface, we can ignore the Z dimension and use X and Y to compute the heading of the robot. The `atan2` function can be used to convert the X and Y values into the heading angle.

Finally, since Earth's magnetic north is not aligned with the geographic north, we need to compensate for that difference. The difference between the magnetic north and the true north is called *declination* and its value can be computed using a model such as the World Magnetic Model (WMM) [62]. The NOAA offers a website [63] in which it's possible to calculate the declination value for any part of the world. For the region of Worcester, the current[2] value is 14°24' W, as illustrated in Figure 3.6. For an example code for I2C communication with the HMC5883L chip in the Raspberry Pi using the C++ language, please refer to Appendix A.

[2]This value changes over time. For Worcester, it changes 0°3' E per year.

In the current setup, the magnetometer is connected to one of the Raspberry Pi computers that also handles the image capture. The heading information is sent to the server every time it requests an image. The three 2-byte integers provided by the compass are sent to the server after the image data.

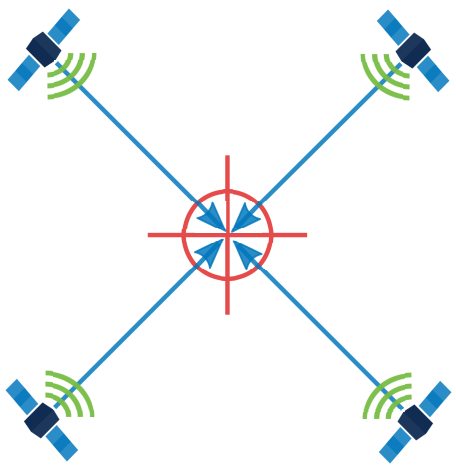
3.3 GPS

The localization data is provided by the BU-353 [64] GPS module connected directly to the server via USB. The manufacturer provides Linux drivers for the device and the communication is done using the GPS Daemon (GPSd) [65].

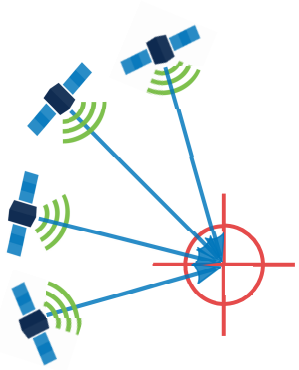
GPSd is a service that monitors GPS devices on a computer and handles the communication with the device. Using GPSd simplifies the process of acquiring GPS data and increases compatibility. It provides data in a easy to parse format that is independent of the protocol used by the underlying device, so an application that uses GPSd to access the data is compatible with every device supported by it.

The GPSd package includes a C library and a C++ wrapper class, that is used in this project for the GPS communication.

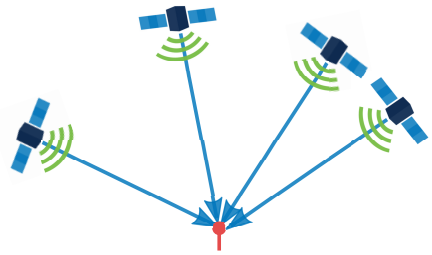
In order to calculate its current position, the GPS device must see at least 4 satellites in the sky, but more satellites can be used to increase precision if they are available. The arrangement of the satellites also affects precision. The quality of the arrangement is measured by two factors: the Horizontal Dilution of Precision (HDOP) and the Vertical Dilution of Precision (VDOP). HDOP is related to the distribution of the satellites in the 4 compass quadrants. Having a satellite in each one of the quadrants provides better precision when computing the horizontal position. acVDOP is related to how the satellites are spread in the sky. Well spread out satellites provide better precision when computing the vertical position than



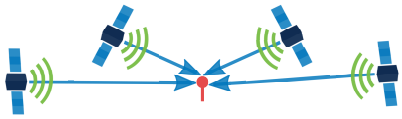
(a) Low HDOP (good resolution).



(b) High HDOP (bad resolution).



(c) Low VDOP (good resolution).



(d) High VDOP (bad resolution).

Figure 3.7: On the top row, how the distribution of the satellites affects the horizontal precision. On the bottom row, the effects of the distribution on the vertical precision.

satellites located only near the horizon. This is illustrated by Figure 3.7.

HDOP and VDOP can be combined in a single value called Position Dilution of Precision (PDOP). Smaller PDOP values indicate a better precision and values higher than 10 are considered very bad. PDOP values are provided by GPSd along with the position data and can be used by the application to determine if the information is precise enough for it to continue execution.

3.4 Throttle

The throttle on the golf cart is controlled by a resistance. The accelerator pedal controls a 5kΩ potentiometer that is connected to the motor controller. When the driver pushes the pedal, the resistance becomes smaller, indicating to the controller that the speed of the motor should increase.

To control this system from the computer, we used a digital potentiometer, an integrated circuit that can adjust a resistance value according to a digital signal sent by some other component. The chip we used for this system is the DS1803-010 [66]. It contains two 10kΩ potentiometers that can be adjusted independently.

Each potentiometer is accessed through three pins on the integrated circuit, representing the low, high and wiper pins of a mechanical potentiometer. The position of the wiper pin is controlled by a digital signal.

This chip also uses an I2C bus to receive commands from the master and the communication protocol is even simpler than the one used by the compass, since the chip never sends data to the master. To adjust the wiper position, the master first sends the address of the chip, followed by a command byte that indicates if we want to set the first, the second of both potentiometers and then one or two value bytes that represent the desired wiper position. Zero sets the wiper as close to the low pin as possible while 255 puts the wiper as close to the high pin as possible.

When the chip is powered on, its initial state is zero (wiper close to the low pin). So, the chip is wired in such a way that the motor controller reads the resistance between the high and the wiper pins so that, when the chip is powered on, the motor controller sees a 10kΩ resistance and doesn't move, instead of seeing a 0Ω resistance and moving at full speed.

To control this integrated circuit, instead of using the Raspberry Pi, we use

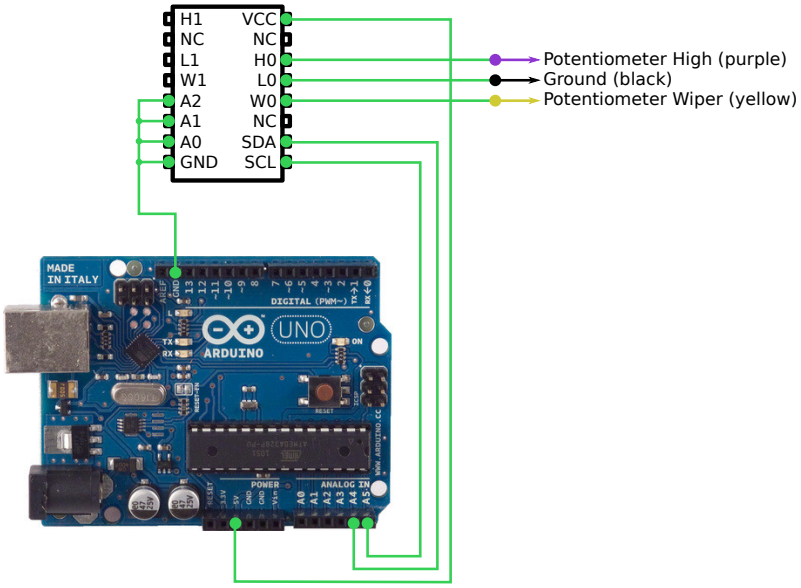


Figure 3.8: The DS1803-010 potentiometer and its connections to the Arduino and to the golf cart. In out setup, all the address pins (A0, A1 and A2) are connected to the ground (0V). The NC (no connection) pins do not have a function. The H1, W1 and L1 pins correspond to the second potentiometer and are not used in this project. The chip is powered by the 5V output from the Arduino. The SDA and SCL pins are connected to the corresponding pins on the Arduino (A4 and A5 respectivelly) and the H0, W0 and L0 are connected to the throttle system of the golf cart.

an Arduino Uno board. The Arduino Uno is a microcontroller board based on the ATmega328P and provides 14 digital Input and Output (IO) pins, 6 analog inputs and a USB port that can be used for powering, programming and serial communication with the board [67].

The Arduino can also communicate on a I2C bus easily. The I2C communication is implemented by the Wire library [68], that is present on the Arduino IDE by default. For an example code demonstrating the communication with the digital potentiometer, please refer to Appendix B. Figure 3.8 shows how the potentiometer is connected to the Arduino and to the Robocart.

The Arduino is connected to the server via USB, that provides power to the board and allows communication between them using serial over USB. Linux natively

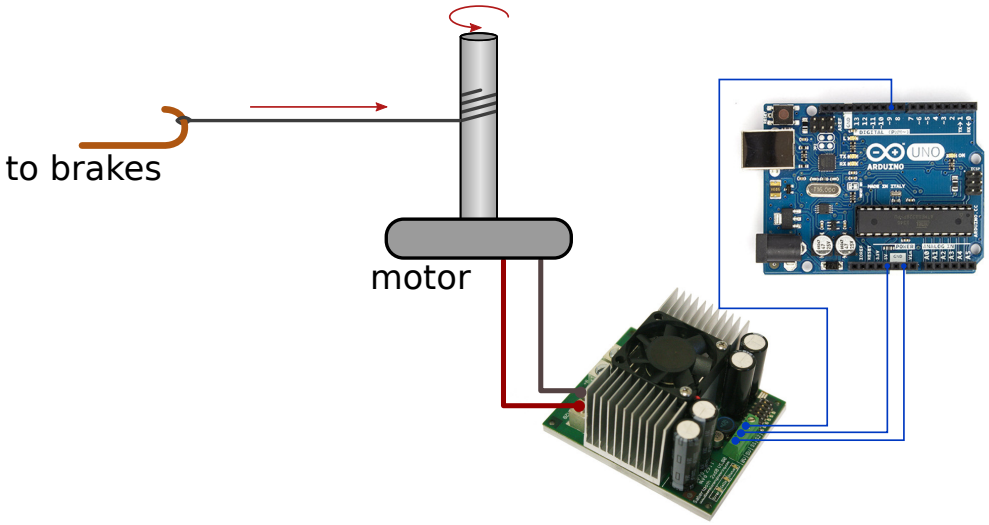
supports serial over USB devices. When the Arduino is connected to the computer, it is mapped to a device on the `/dev` folder and can be accessed by other programs using the usual read and write functions. For an example code demonstrating the communication between the server and an Arduino board, please refer to Appendix C.

3.5 Brakes

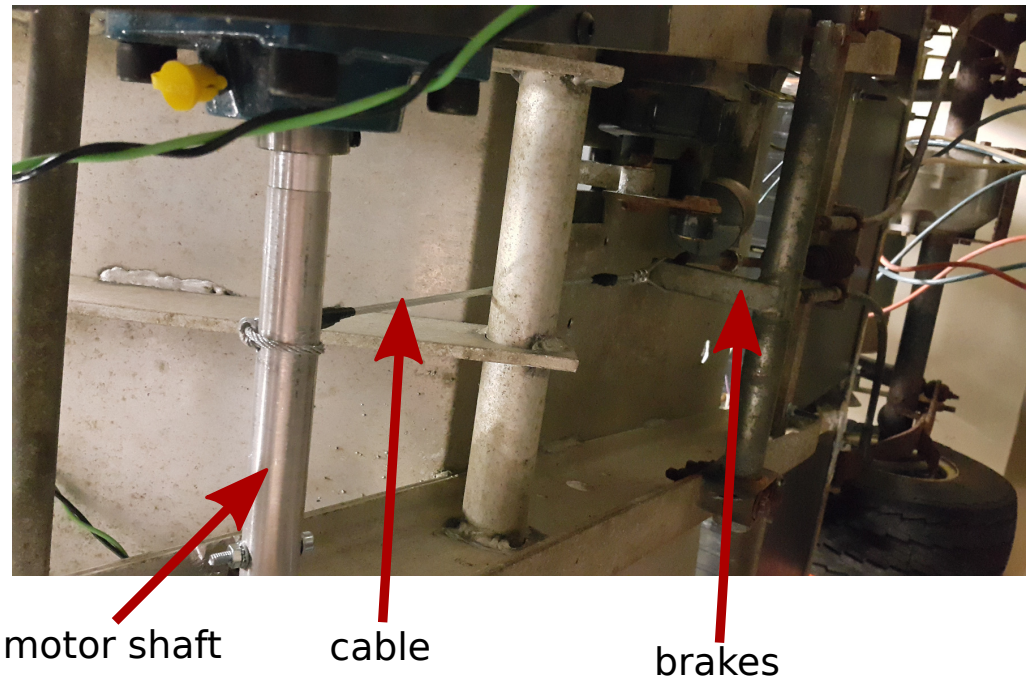
The brakes are implemented using a windshield wiper motor attached to a cable that pulls the actual brakes of the golf cart. In this design, the autonomous braking system to not interfere with the manual one, so the user can manually stop the cart if it is necessary. The motor is controlled by a Sabertooth 2x60 motor controller [69]. This is a two-channel controller and it is used to control both the brake motor and the steering motor presented in the next section. The controller accepts several different kinds of input signals such as serial and Pulse Width Modulation (PWM). An Arduino is the interface between the server and the controller. It receives a command from the server via serial over USB and generates the corresponding PWM signal to the controller. Figure 3.9 shows a diagram of the braking system as well as a picture of it implemented on the Robocart. The implementation of the brakes was not part of this project, so it is not described in details here. For more information, please refer to the report from Robert Crimmins and Raymon Wang.

3.6 Steering

Automatic steering is implemented using a gear attached to the golf cart's steering column. This gear is connected to a motor using a chain. This chain is also attached to a potentiometer whose resistance is used as the feedback signal representing the

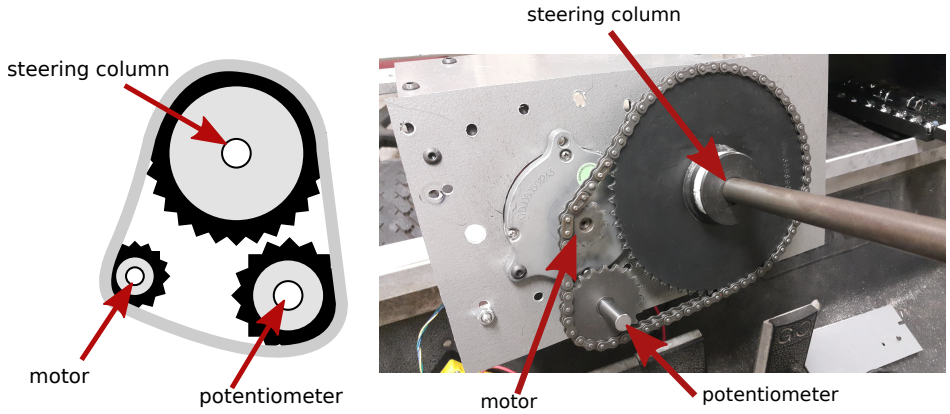


(a) Diagram of the autonomous braking system.



(b) Autonomous brakes implemented on the golf cart.

Figure 3.9: Diagram of the autonomous braking system and the actual implementation on the Robocart.



(a) Diagram of the steering system.

(b) Autonomous steering implemented on the golf cart.

Figure 3.10: Diagram of the autonomous steering system and the actual implementation on the Robocart.

current position of the steering column. Currently, the system is controlled by an Arduino that receives a command from the server via serial over USB and moves the steering column to the desired position using a simple Proportional-Integral-Derivative (PID) controller. The output of the PID is converted to a PWM signal that is sent to the Sabertooth motor controller. Figure 3.10 shows a diagram of the steering system and the actual implementation on the Robocart. The implementation of the steering system is not a part of this project and is mentioned here for completeness. For more information, please refer to the report from Robert Crimmins and Raymond Wang.

3.7 Server

The server is responsible for interfacing with the vision system and the sensors to acquire information about the environment around the robot, run the planning algorithms and send commands to the mechanical systems that drive the Robocart along the desired path. It has a Intel Core i7-4770K processor [70] and 32GB

of DDR3 RAM. The processor has 4 physical cores and 8 threads, and its base frequency is 3.5GHz. The server also contains a NVIDIA GeForce GTX 760 [71] graphics card with 2GB of memory and support for CUDA.

On the software side, it runs Ubuntu 14.04.4 LTS "Trusty Tahr" [72] and some of the vision processing is done by OpenCV 3.0 [73].

The server is powered by a Cen-tech power inverter that is connected to the batteries of the Robocart. It is capable of providing 750 watt continuous and 1500 watt peak power.

Finally, the ethernet network is created using a Rosewill RNX-N150RT wireless router running DD-WRT v24-sp2 [74].

3.8 Summary

This chapter introduced the Robocart, a research platform developed by the Wireless Innovation Laboratory at WPI. Section 3.1 described in details the vision system that is available on the golf cart, as well as the communication protocol and the synchronization algorithm developed during this project for the image capture. Section 3.2 explained how we use a magnetometer to determine the current heading of the robot. Section 3.3 presented the GPS sensor used for this project and introduced some concepts related to the precision of the data provided by it. In Section 3.4 we discussed the throttle system, that was also implemented as part of this project. Sections 3.5 and 3.6 described the brake and steering systems and how to interface with them from the computer. Finally, Section 3.7 briefly listed the hardware that is available on the Robocart server.

In the next chapter, we begin the discussion of the planning algorithms implemented on the Robocart, starting by the obstacle avoidance method.

Chapter 4

Obstacle Avoidance

Obstacle avoidance is the first of the three modules implemented in this project. Its main sources of information are the images from a pair of cameras. The methods described in Section 2.2 can be used to estimate a 3D model of what is ahead of the robot, however, noisy image data and other imperfections can lead to inaccurate disparity information. To address this problem, our proposed obstacle avoidance method is based on a probabilistic method known as *Occupancy Grid*, explained in details in Section 4.1. Section 4.2 addresses a problem with the planning algorithm, that considers the robot as a point. Section 4.3 describes the implementation of the A* algorithm on the occupancy grid and how GPS coordinates are converted to goal points for the planner. Section 4.4 discusses the implementation details and presents some results.

4.1 Occupancy grid

Originally proposed in [32], occupancy grids became a popular tool for representing the surroundings of mobile robots. They are commonly used with range sensors such as a LiDAR, but in this project we investigate the use of stereoscopic images

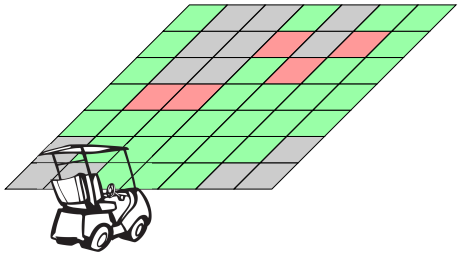
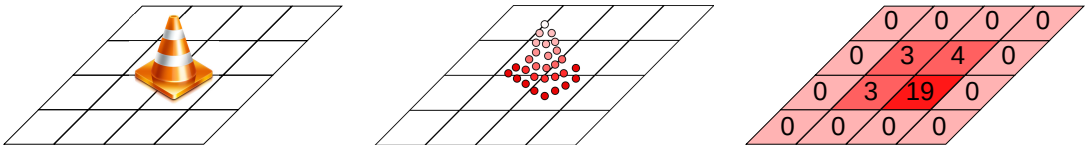


Figure 4.1: Occupancy grid representation (not to scale). Gray cells are unknown, green cells are free and red cells are occupied.



(a) Obstacle.

(b) Point cloud for this area (point on the ground omitted).

(c) Each cell contains the number of points above it.

Figure 4.2: Converting the representation of an obstace from a point cloud to a grid.

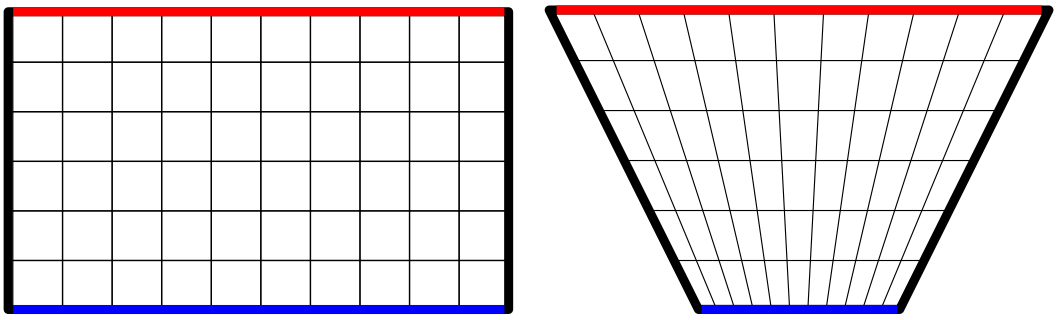
to populate the occupancy grid.

Our occupancy grid is defined as an array of evenly distributed rectangular cells covering a predefined area. After the grid is populated, each cell contains one of three possible values: *unknown*, *occupied* or *free*, as illustrated by Figure 4.1.

The process of computing the occupancy grid is similar to the one proposed in [75]. It starts with the generation of the disparity map using a pair of images from the cameras, as explained in Section 2.2. Using Equation (2.27), the disparity image is converted to a *point cloud*, *i.e.*, a set of 3D points.

Each point in the set is assigned to one of the cells in the occupancy grid, according to its position, as illustrated by Figure 4.2. Each cell also stores the average height of the points in it.

Figure 4.3 illustrates a problem that arises from the way we aggregate points on the grid. On the left, we see an image taken from a camera facing forward and on the right we see a bird's eye view of the region contained in that picture. Regions



(a) Pixels in a picture.

(b) Bird's eye view of the region contained in the picture.

Figure 4.3: Uneven number of pixels on different regions. Regions closer to the camera contain more pixels than regions that are farther.

closer to the camera, like the one indicated in blue, contain more pixels than regions that are farther, like the one indicated in red. So, the grid cells representing regions closer to the camera tend to have more points on them.

In [75] this problem is addressed by adjusting the number of points in each cell using the *sigmoid function* in Equation (4.1).

$$n'_{i,j} = n_{i,j} \cdot S(d_{i,j}) = n_{i,j} \cdot \frac{r}{1 + e^{-d_{i,j} \cdot c}}, \tag{4.1}$$

where $n_{i,j}$ is the number of points in cell (i, j), $n'_{i,j}$ is the new number of points in cell (i, j) after the adjustment, $d_{i,j}$ is the distance from cell (i, j) to the camera and r and c are control coefficients.

Figure 4.4 shows a plot of the sigmoid function for $r = 1$ and $c = 0.5$. The function can smoothly restrain the number of points in the cells closer to the camera and amplify it on cells that are farther.

Once the number of points in each cell is corrected, we can compute the probability of a cell being occupied. Intuitively, the probability of a cell being occupied is higher in cells that contain more points. We can represent that information using a probability model similar to the one in [76], shown in Equation (4.2), where

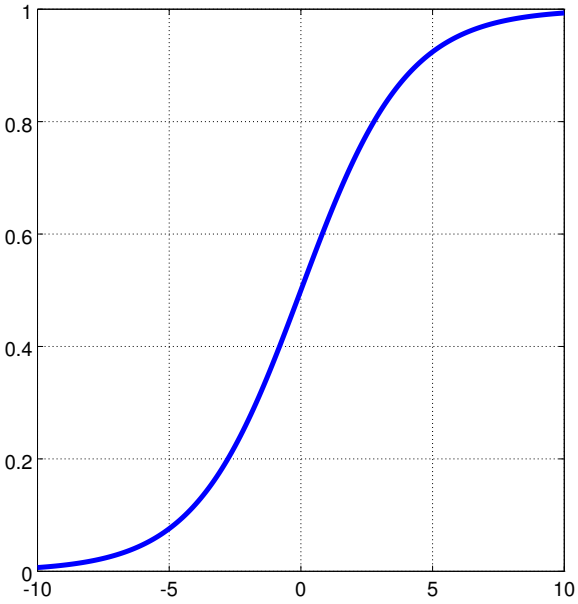


Figure 4.4: Sigmoid function plotted with parameters $r = 1$ and $c = 0.5$.

the probability of a cell being occupied is modeled using an exponential function. For decision making, a more convenient way of representing probabilities is using *log-odds* where the odds of a cell being occupied are represented in *dB*, as shown in Equation (4.3).

$$P_{i,j}(O|num) = 1 - e^{-\frac{n'_{i,j}}{\delta_n}}, \tag{4.2}$$

$$l_{i,j}(O|num) = 10 \cdot log\left(\frac{P_{i,j}(O|num)}{1 - P_{i,j}(O|num)}\right). \tag{4.3}$$

In [77] there is a discussion about some advantages of this representation. It captures some behaviors of probabilities that are not obvious on the usual representation. For example, while it is not intuitive that the difference between 50% and 50.01% is trivial compared to the difference between 99.98% and 99.99%, it becomes clearer when they are represented as 0 dB, 0.0017dB, 37 dB and 40 dB respectively.

Finally, we process the average point height information, $h_{i,j}$, for each cell in the same way (equations (4.4) and (4.5)), then, we define the final log-odds of occupancy for a cell as a weighted average of the two log-odds, as in Equation (4.6).

$$P_{i,j}(O|\text{height}) = 1 - e^{-\frac{h_{i,j}}{\delta_h}}, \tag{4.4}$$

$$l_{i,j}(O|\text{height}) = log\left(\frac{P_{i,j}(O|\text{height})}{1 - P_{i,j}(O|\text{height})}\right), \tag{4.5}$$

$$l_{i,j}(O) = w_n\, l_{i,j}(O|num) + w_h\, l_{i,j}(O|\text{height}), \tag{4.6}$$

where w_n and w_h are weights and $w_n + w_h = 1$.

Once we have the log-odds of the occupancy of each cell, we use two thresholds, n_t and l_t to decide on the status of a cell according to Equation (4.7).

$$O_{i,j} = \begin{cases} unknown & \text{if } n'_{i,j} < n_t \\ occupied & \text{if } l_{i,j}(O) \leq l_t \\ free & \text{if } l_{i,j}(O) > l_t \end{cases} \tag{4.7}$$

4.2 Obstacle Dilation

Once we have an occupancy grid, it can be used as input for the search algorithm that will generate a path that does not collide with obstacles. However, the search algorithm represents the robot as a point. This can be a problem, because the optimal paths often are the ones that are very close to the obstacles.

A technique that is commonly used to overcome this problem is *obstacle dilation* [78, 79]. It solves the problem by enlarging the obstacles by an amount corresponding to half of the width of the robot plus some clearance distance c. This dilation ensures that when the point that represents the robot is touching an obstacle in

the occupancy grid, the actual robot will be at a distance c of the real obstacle, as shown in Figure 4.5.

In our occupancy grid, obstacles are dilated only horizontally. Each non-occupied cell has its state changed to *occupied* if any of the n cells to its left or to its right is occupied, where n is the number of cells that cover a region corresponding to half the width of the robot w_{robot} plus a clearance distance c, as in Equation (4.8). This is implemented by a Dilation operation using a $(2n + 1)$x1 kernel.

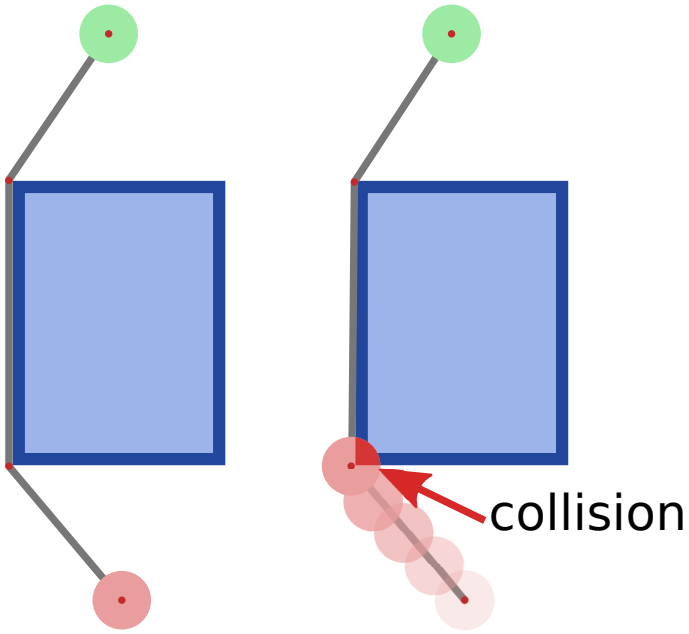
$$n = \frac{\frac{w_{robot}}{2} + c}{w_{cell}} \tag{4.8}$$

However, a problem arises from the obstacle dilation. Isolated pixels that are considered occupied due to noise and imprecision are dilated and interfere with the planning algorithm in a negative way, potentially making it impossible to find a path to the goal. We overcome this problem by simply executing an Opening operation using a 3x3 kernel before the Dilation, to turn very small occupied cells into free cells.

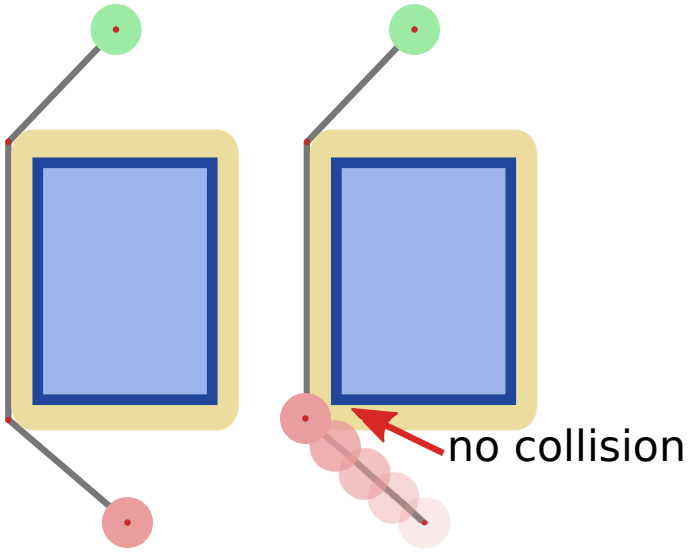
4.3 Path Planning

After the occupancy grid is populated, it can be used to find a path that makes the robot move towards the goal while avoiding obstacles. However, before running any planning algorithm, we must define our goal.

The input to our planner is a set of GPS coordinates representing a path that the robot must follow. The planner will guide the robot in a straight line to the first point in the path and when the robot is close enough to that point, it will proceed to the next. This means that if there is a curve between the start and the goal points, as shown in Figure 4.6, the input to the planner must contain an intermediary point

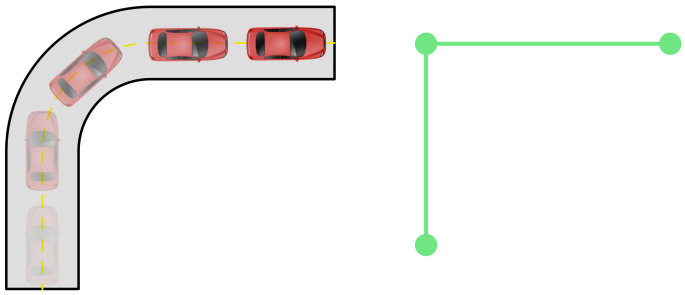


(a) The optimal path to move the robot from the red position to the green position touches the obstacle. When the robot follows that path, it collides with the obstacle.

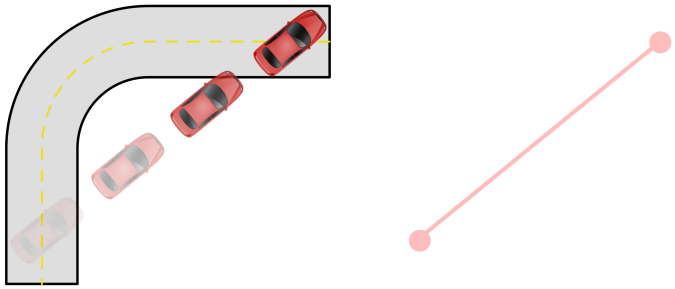


(b) The obstacle is dilated, so the robot does not collide with it anymore.

Figure 4.5: Planning with and without obstacle dilation.



(a) A trajectory with three points and the car executing the curve correctly.



(b) A trajectory with only two points. In this case the car goes straight to the final point and ignores the curve.

Figure 4.6: The robot always tries to follow a straight line from its current position to the next goal coordinates.

to indicate that. The generation of this set of coordinates is beyond the scope of this project and can be done manually, but in the future it could, for example, be generated by an application that uses maps to compute the best path between two addresses.

The planning algorithm runs on the occupancy grid that was described in the previous section. In order to run the algorithm, the goal point must be converted from GPS coordinates to a point in the occupancy grid. To handle this problem, we use three different coordinate frames.

The *world frame* defines a plane that is parallel to the surface of the Earth in the vicinity of our trajectory points. Projecting the GPS coordinates on that plane makes them easier to deal with, because we can work on a Cartesian coordinate

system, instead of a spherical one. Since the distances we are dealing with are much smaller than the radius of the Earth, the error introduced by this approximation is negligible.

The origin of the world coordinate frame is the first GPS coordinate of the path (lat_0, lon_0), corresponding to the initial position of the robot. The Y axis points north and the X axis points east.

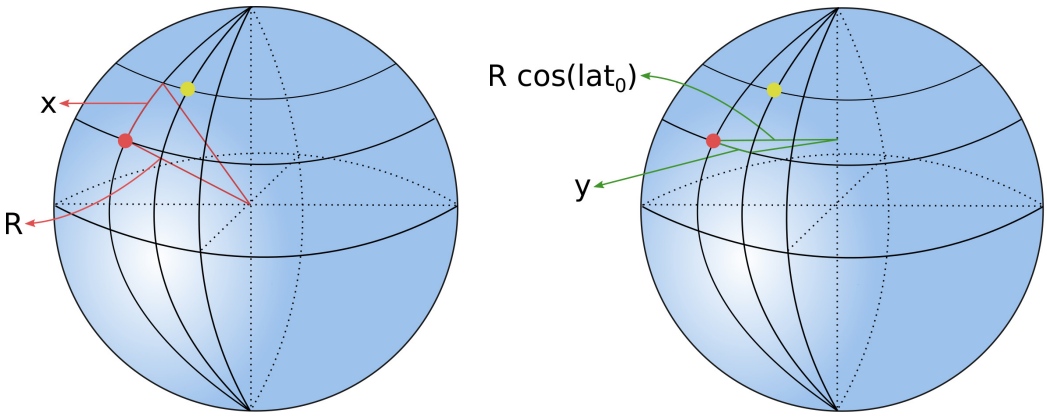
A GPS coordinate (lat, lon) can be projected in that plane using equations (4.9) and (4.10) [80].

$$y_W = R \cdot (lat - lat_0) \cdot \frac{\pi}{180} \tag{4.9}$$

$$x_W = R \cdot (lon - lon_0) \cdot \frac{\pi}{180} \cdot cos(lat_0) \tag{4.10}$$

Figure 4.7 shows a visual representation of those equations. The Y coordinate is computed by multiplying the latitude difference in radians by the radius R of the Earth, which gives as a result the arc length in the same unit as the radius. The X coordinate is a distance along a line of latitude. Since the latitude lines are circles that become smaller as the latitude increases, the $cos(lat_0)$ factor is added to the equation. Figure 4.8 illustrates the world coordinate frame.

The second coordinate frame is the *robot frame*. It is on the same plane as the world frame, but centered on the camera pair. The X axis points to the right side of the robot while the Y axis points to the front. The conversion from the world frame to the robot frame done by a translation that depends on the position of the cart and a rotation that depends on its heading. It can be represented as a matrix multiplication in homogeneous coordinates as shown in Equation (4.11).



(a) Y coordinate (b) X coordinate

Figure 4.7: Computing the coordinates of the yellow point on the world frame centered at the red point. The Y coordinate is the arc length of the latitude distance between the two points. The X coordinate is the distance along a latitude line.

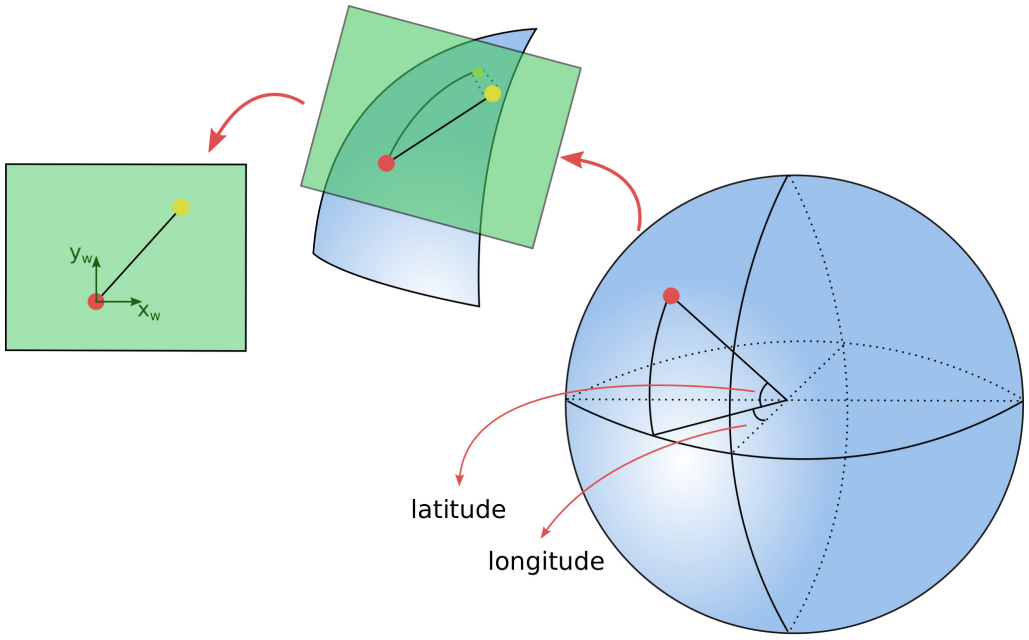


Figure 4.8: Representation of the world coordinate frame. It is a plane parallel to the surface of the Earth near the starting position of the robot (red point). Other points of the trajectory are projected onto this plane.

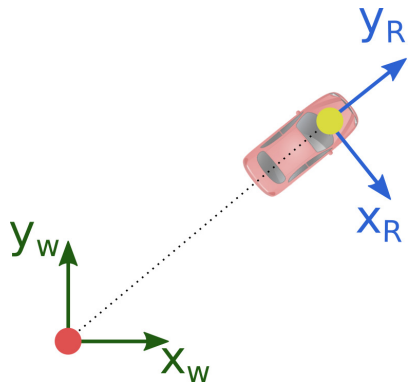


Figure 4.9: Representation of the robot coordinate frame. It is on the same plane as the world frame, but moves along with the robot.

$$\begin{bmatrix} x_R \\ y_R \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} cos(-\theta_h) & -sin(-\theta_h) & 0 \\ sin(-\theta_h) & cos(-\theta_h) & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{rotation}} \cdot \underbrace{\begin{bmatrix} 1 & 0 & -p_x \\ 0 & 1 & -p_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translation}} \cdot \begin{bmatrix} x_W \\ y_W \\ 1 \end{bmatrix} \tag{4.11}$$

where (p_x, p_y) are the coordinates of the robot in the world frame, obtained by the conversion of the location data from the GPS and θ_h is the current heading of the robot, obtained from the compass. As explained in Section 2.1, the two operations can be converted into one by multiplying the matrices that represent them, as in Equation (4.12).

$$\begin{bmatrix} x_R \\ y_R \\ 1 \end{bmatrix} = \begin{bmatrix} cos(-\theta_h) & -sin(-\theta_h) & -p_x\, cos(-\theta_h) + p_y\, sin(-\theta_h) \\ sin(-\theta_h) & cos(-\theta_h) & -p_x\, sin(-\theta_h) - p_y\, cos(-\theta_h) \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_W \\ y_W \\ 1 \end{bmatrix} \tag{4.12}$$

Finally, the *occupancy grid frame* is located at the upper left corner of the occupancy grid. As shown in Figure 4.10, each cell in the occupancy grid corresponds to a region in the world and mapping between those two coordinate frames is done using the size of each grid cell.

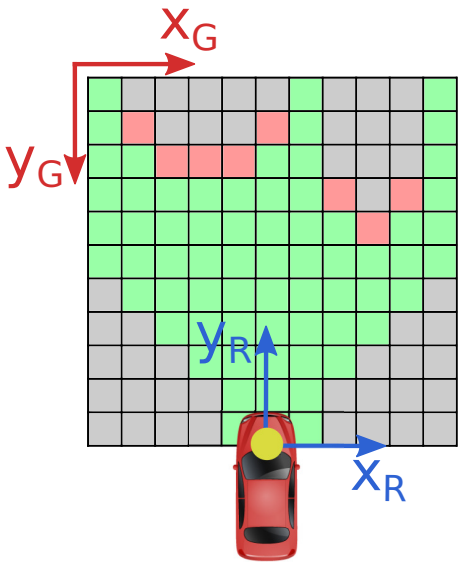


Figure 4.10: Representation of the occupancy grid frame. Each cell of the grid covers a region of the world. Points in the world frame are represented in the grid frame by the cell that contains it.

Points inside the area covered by the occupancy grid are mapped to the cells in which they are located. Points outside of the grid are mapped to the borders of the grid, for path planning purposes. Figure 4.11 shows several points in the world frame and the cells of the occupancy grid that they are mapped to.

The start position for the planning algorithm is always at the center of the bottom row of the grid, where, according to our convention, the robot always is. The goal position is the first entry in a list of non-visited GPS coordinates that corresponds to the desired path that the robot should follow. The GPS coordinates are converted to the world frame, then to the robot frame and finally mapped to a cell in the occupancy grid that is the goal for the search algorithm.

The path is obtained by running the A* algorithm on the occupancy grid. Each grid cell is a node in the graph and edges are created between each node and its neighbors, as long as they are not marked as *occupied*. The weights assigned to each edge are given by the Euclidean distance between the nodes and the Euclidean

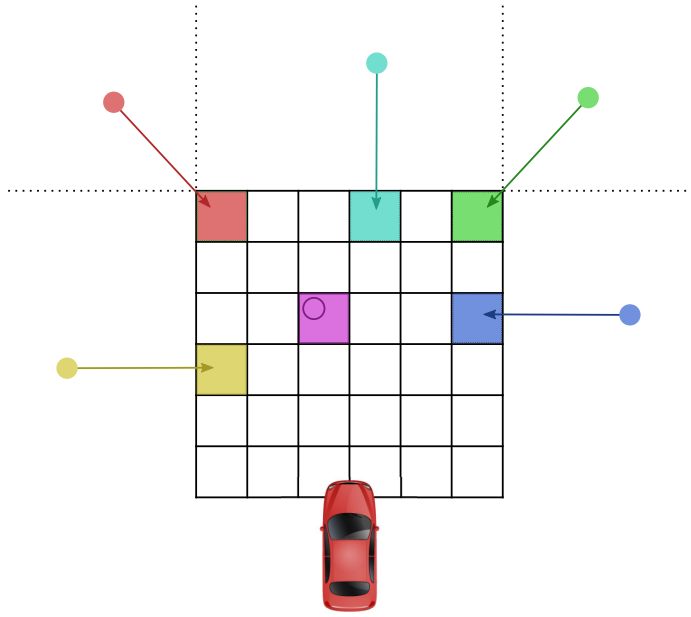


Figure 4.11: Points in the world frame and their mappings to the grid frame. Points inside the grid (pink) are mapped to the cells that contain them. Points outside the grid are mapped to the borders of the grid.

distance is also used as the heuristic. The algorithm returns a list of cells in the occupancy grid that compose the path. This list is sent to the Steering Module, described in Chapter 6, to be executed.

4.4 Implementation and Results

The algorithm was implemented using the C++ language, since it tends to be more efficient than higher level languages such as Python or Java. The OpenCV library was used for some of the image processing tasks, namely, image rectification, blurring, stereo disparity computation and reprojection to 3D. OpenCV is a library of functions aimed at computer vision. It was originally developed by Intel, but currently it's maintained by Itseez. OpenCV is cross-platform and distributed under the open-source Berkeley Software Distribution (BSD) license.

In order to make better use of the multiple cores available in our server and

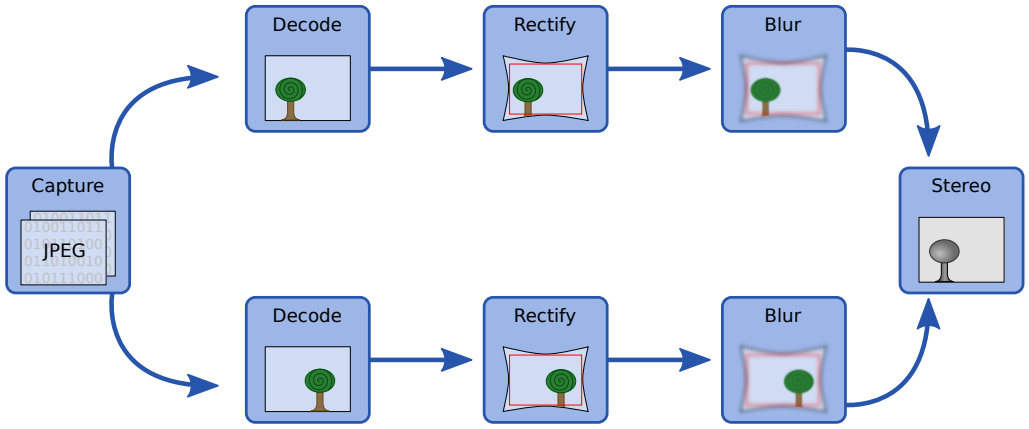
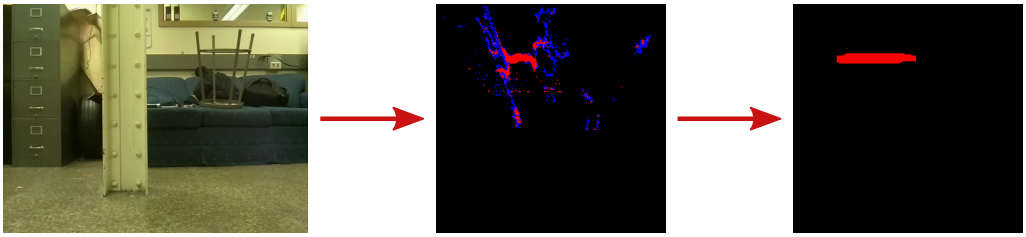


Figure 4.12: Execution flow of the first image processing steps. The decoding, rectification and blurring steps can be executed in parallel for the left and right images. The results can, then, be combined by the stereo computation node.

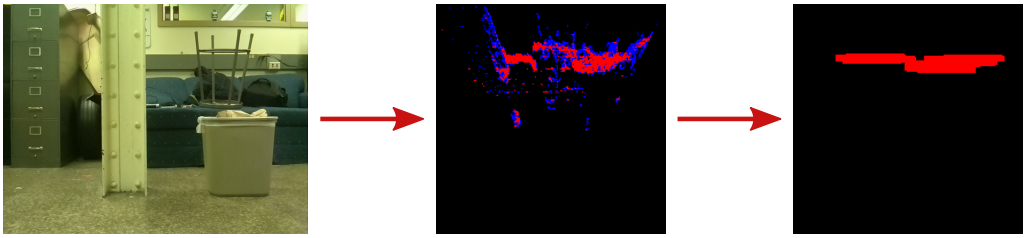
speedup execution, we use Intel Threading Building Blocks (TBB) [81]. TBB's *flow graph* allows us to describe the execution of the program as a graph. Once the graph is built, parts of the program that can be executed in parallel are sent to different threads by TBB.

In this case, all the processing that needs to be done on each image separately can be done in parallel for the left and right images. This includes the decoding of the JPEG images sent by the Raspberry Pi computers, the undistortion and rectification, as well as a Gaussian Blur to reduce noise as illustrated by Figure 4.12. Building the application as a set of TBB nodes also ensures that in the future other functionality can be easily added and executed in parallel with other tasks that it doesn't depend on.

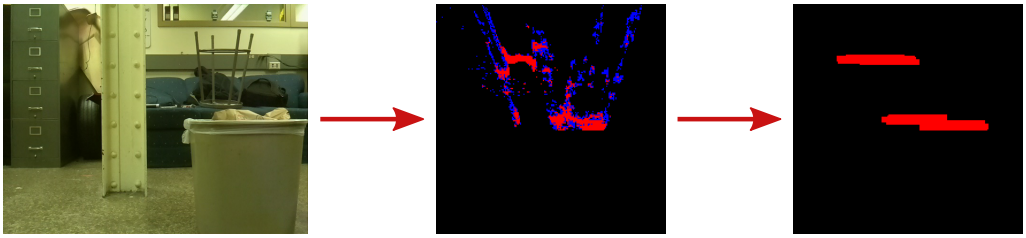
Figure 4.13 shows the occupancy grids generated by the algorithm from some pictures captured in the laboratory. The first column shows the picture captured by the left camera, the middle column shows the occupancy grid and the right column shows the resulting grid after the obstacle dilation. On the first picture, the only obstacle detected is the pillar. On the second picture, we add a trash bin at same



(a) Pilar as the only obstacle.



(b) Pilar and trash bin at the same distance from the robot.



(c) Trash bin closer to the robot than the pilar.

Figure 4.13: Occupancy grids generated for obstacles in the laboratory. On the first column, the image captured by the left camera. In the middle, the occupancy grid. On the left, the occupancy grid after the filtering and obstacle dilation steps.

distance from the cameras as the pillar, and this is what is shown in the occupancy grid. On the third picture, the trash bin is moved closer to the cameras and this is reflected on the occupancy grid.

In order to test the conversion from the GPS coordinates to goal points in the occupancy grid, our application can use mock GPS coordinates and compass readings as the position and orientation of the robot instead of the data from the actual sensors. Figure 4.14 shows some coordinates obtained from Google Maps [82] and the corresponding goal points computed by the application. The first row illustrates the case in which the robot is facing the goal point, but it is outside of the range of the occupancy grid. In this case, the goal is mapped to the first row of the grid.

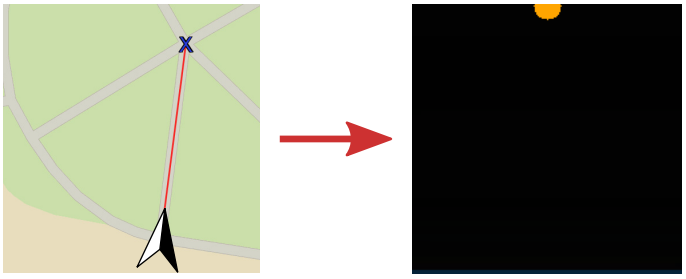
In the second row, the robot is closer to the goal, so it is mapped to a point inside the grid. On the third row the robot is not facing the goal, so it is mapped to the rightmost column of the grid. These results show that the mapping from GPS coordinates to points in the occupancy grid is consistent.

Finally, the path planning can also be tested using mock sensor data and pictures captured in the laboratory. Figure 4.15 shows the image captured by the left camera and the path that was computed by A* using the occupancy grid with dilated obstacles. To better visualize the path, Figure 4.15 also shows a 3D rendering of the point cloud generated from the pair of images. The path generated by the planning algorithm is also plotted on the point cloud in red. As expected, it goes around the pillar and the obstacle dilation ensures that the path does not touch the obstacle.

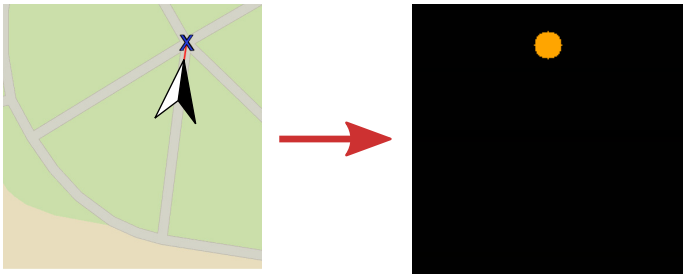
4.5 Summary

In this chapter, we discussed the obstacle avoidance method implemented in this project. Section 4.1 introduced the concept of an occupancy grid and explained the method we use to generate it from a pair of images. Section 4.2 addressed the problem of optimal paths that are very close to obstacles. We introduced the obstacle dilation technique as a solution to this problem. Section 4.3 discussed the path planning algorithm, focusing on the conversion of the data from sensors to a format that is easy for A* to handle. Section 4.4 listed the tools used to implement the proposed approach and presented some of the results obtained from this implementation using mock GPS and compass data and pictures taken in the laboratory.

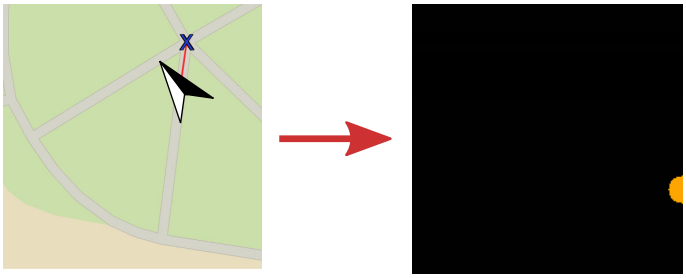
In the next chapter, we discuss visual odometry, a technique that uses stereo images to estimate the position and orientation of the robot.



(a) Goal in front of the robot, outside of the occupancy grid.

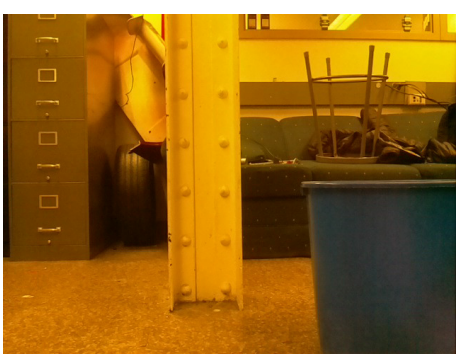


(b) Goal in front of the robot, inside the occupancy grid.

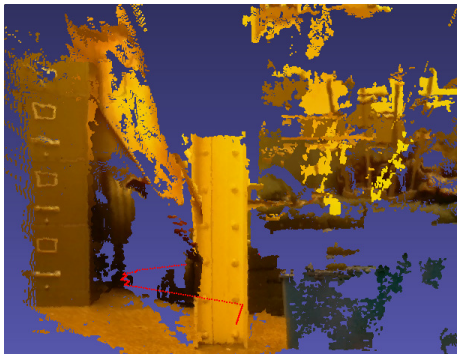


(c) Goal to the right of the robot, outsisde of the occupancy grid.

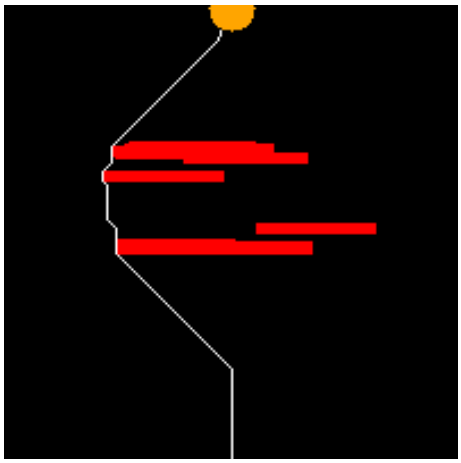
Figure 4.14: On the left column, the robot at a simulated position and rotation is indicated by the arrow. The X indicates the goal point. On the right column, the goal point on the grid frame is indicated by the orange circle.



(a) Picture from the left camera.



(b) Point cloud generated from the images and the path illustrated in red.



(c) Occupancy grid with dilated obstacles.

Figure 4.15: Path found around the pillar and its representation in a 3D point cloud.

Chapter 5

Visual Odometry

Odometry techniques are used to estimate a robot's position and orientation. There are multiple ways of obtaining that information and it usually involves using dedicated sensors that continuously monitor certain parts of the robot such as its motors. On the Robocart, the only sources of information about the position and orientation of the robot are the GPS and the magnetometer, however, those sensors only work reliably under certain conditions, *e.g.*, the GPS usually cannot obtain a fix unless it is outdoors and magnetometer readings can be affected by pieces of metal near the sensor. In this project, we investigate the use of the images from the cameras as another source of odometry data, that could potentially be used to make up for GPS outages and incorrect compass readings.

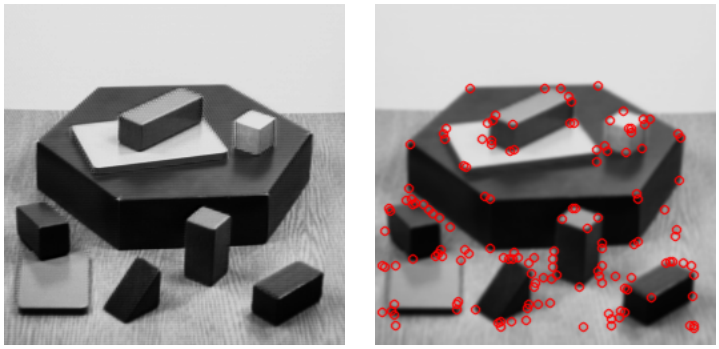
There are several visual odometry techniques available in the literature, and they have been used even on exploration rovers in Mars [83]. They can be classified as *monocular*, when it uses only one camera, or *stereo*, when it uses two or more cameras. The biggest limitation of Monocular Odometry is the fact that it can only estimate a robot's trajectory up to a scale factor, *i.e.*, it's possible to tell that a robot moved one unit in the x direction and 2 units in the y direction,

but it is not possible to convert those measurements to a meaningful distance unit without additional information [84]. Therefore, in this project, we use a Stereo Visual Odometry method, similar to the one described in [85]. Section 5.1 presents some of the available algorithms for the feature detection step of the odometry method. Section 5.2 explains how to track those features frame by frame. In Section 5.3 we present a method to select a subset of the features that is consistent across the frames, so that we can estimate the odometry information more reliably. Section 5.4 presents a method to compute odometry data using an minimization algorithm. Finally, Section 5.5 presents some implementation details and results obtained from the algorithm.

5.1 Feature Detection

The proposed method works by tracking a set of *features* across a sequence of images from the cameras. By measuring how much the features moved, we can estimate the movement of the camera. The first step of this process is to detect features on the image. Features are points in the image, however, not every pixel in the image is a good feature. Since we need to track them across a sequence two or more images, features must be easily distinguishable from other points. Those features are usually located at the corners of the objects in a scene, so feature detection can be done by a *corner detection algorithm*.

There are several corner detection methods described in the literature such as Good Features to Track (GFTT) [86], Scale Invariant Feature Transform (SIFT) [87], Speeded Up Robust Features (SURF) [88] and FAST [89]. In this project, we use FAST as our corner detector, because it outperforms the other algorithms in terms of speed [75], which is important for our use case.



(a) Original image.

(b) Corner detection output.

Figure 5.1: Detecting corners on an example image. A Gaussian Blur with a 5x5 kernel is applied to the image before the corner detection.

Figure 5.1 shows the FAST detector applied to an example image.

A problem of running the corner detector algorithm over the entire image is that there is a chance that most features will be concentrated in certain regions of the image. This problem can be addressed by the use of *bucketing* [84]. Bucketing works by dividing the image into a grid and running the algorithm on each one of the cells and extracting at most a certain number of features from each grid cell. This process gives us a better distributed set of features.

5.2 Feature Tracking

Once we have a set of features on one of the images, we need to track them across the other frames. It is possible to run the feature detection process on the next frame and try to match the descriptors returned by the algorithm on both frames to determine the corresponding features. This approach is useful for finding a known object within an image, but obtaining good results depends on having good descriptors that are invariant to changes of orientation, scale and viewpoint and such descriptors, such as SIFT, can be expensive to compute.

Figure 5.2: Tracking 30 features detected with FAST using KLT. Frames 1, 50 and 100 of the video are shown in the image.

The approach used in this project exploits the fact that the changes in the feature positions are small through subsequent frames, so we can track them using *optical flow*. Optical flow algorithms can compare a sequence of images and determine the direction in which a point is moving. One of the most popular optical flow algorithm is the KLT [90, 91] feature tracker. KLT assumes that neighboring pixels have similar motion and uses that assumption to direct the search towards the best match. It works well for for small motions, but bigger motions can be tracked using *image pyramids*. Pyramids are multi-scale representations of an image and they are built by blurring and subsampling the image. By making the image smaller, big motions become small and KLT can, then, be used to track it. By running KLT on all levels of the pyramid, we can track smaller and bigger motions. Figure 5.2 shows KLT tracking points on a video.

In order to estimate the motion of the robot, the tracked features must be re-projected to 3D using Equation (2.27), but to perform the reprojection, we need the disparity information, *i.e.*, the location of the feature on the left and right images on both frames. This information can also be computed using KLT by a *circular tracking* [75]. Circular tracking is illustrated in Figure 5.3. It starts by detecting the features on the left image of frame t using, for example, FAST. Then, we search for the equivalent features on the right image of frame t using KLT. The features detected in this step are used as input for another execution of KLT that will search

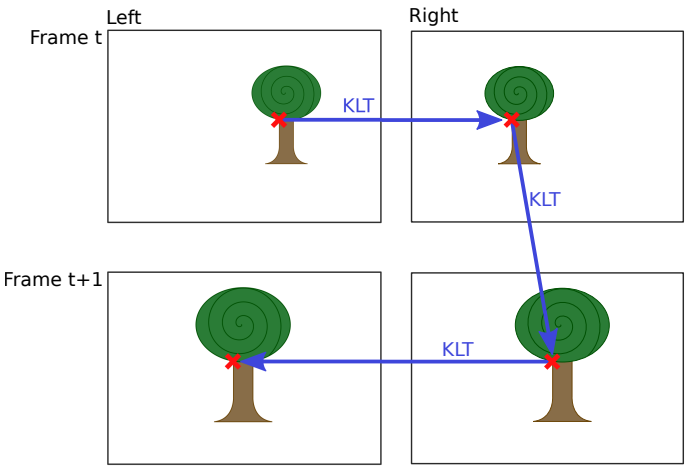


Figure 5.3: Features are detected by a corner detector on the left image of frame t, then, KLT tracks that feature across the other images.

for features on the right image of frame $t+1$. Finally, we use this set of features to search for the features on the left image of frame $t+1$. At the end of this process, we have a set of features detected on the left and right image of each frame. Matches that are obviously wrong (equivalent points on the left and right images with a big difference in their y coordinate, violating the epipolar constraint) are discarded and, then, we can compute the disparity and reproject the points to 3D.

5.3 Inlier Detection

The basic assumption of this visual odometry algorithm is that the scene being observed is static and only the robot is moving. If there are moving objects on the scene and we track features of these objects, they will interfere with the results. Some methods deal with this problem by using a robust estimator such as Random Sample Consensus (RANSAC) that can tolerate a certain number of false matches and reject them. In this algorithm, instead, the author proposes the use of an *inlier detection* step, as opposed to an *outlier rejection* step.

The inlier detection is based on a *rigidity constraint*. Since the scene is rigid, the

distance between any two features must be the same on the two frames. If this is not the case, either there was an error on the feature matching, or we are tracking a moving feature that should not be used for the odometry estimation.

The process consists of building a *consistency matrix* $\mathbf{M}$ whose entries are either 1 for features that are consistent or 0 for features that are not, as in Equation (5.1).

$$\mathbf{M}_{i\,j} = \begin{cases} 1, & \text{if } |w_t^i - w_t^j| - |w_{t+1}^i - w_{t+1}^j| < \delta \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

where w_t^i are the world coordinates of feature i on frame t.

Once the $\mathbf{M}$ matrix is built, we need to find the largest subset of features that are consistent with each other, *i.e.*, a set of features whose consistency matrix is composed only by ones. This problem is equivalent to the *Maximum Clique Problem*. A *clique* is a subset of a graph in which every node is connected to all the other nodes. This problem is known to be NP-complete and, therefore, there is no efficient solution for it.

A sub-optimal solution can be found using the greedy approach shown in Algorithm.

1. Initialize the clique to contain the match with the largest number of consistent matches.

2. Find a set v of matches compatible with all the matches already in the clique.

3. Select the match connected to the largest number of other nodes in v and add it to the clique. Repeat from step 2 until no more nodes can be added.

When the algorithm returns, we have a set of matches that are consistent with each other. This set can, then, be used to estimate the movement of the robot.

5.4 Odometry Estimation

Since we are tracking only the points that belong to static objects on the scene, any movement of those points is actually caused by the movement of the robot. We can represent the movement of the robot as a rotation followed by a translation and the two operations can be combined into a 4x4 matrix $\mathbf{T}$ in homogeneous coordinates. If the robot performs a movement represented by $\mathbf{T}$, this is reflected on the tracked points as the opposite movement represented by $\mathbf{T}^{-1}$. Intuitively, if a point moved a distance d towards the robot, in fact, the robot is the one that moved a distance d in the opposite direction towards the point. This gives us a relationship between points w_t in frame t and points w_{t+1} in frame $t+1$, as shown in Equation (5.2).

As explained in Section 2.2.1, a point in the 3D world can be projected to a 2D image using a *projection matrix* (see Equation (2.20)). Multiplying Equation (5.2) by the projection matrix $\mathbf{P}$, we obtain Equation (5.3). The term $\mathbf{P}w_{t+1}$ is the pixel j_{t+1} on the frame $t+1$. Substituting that into Equation (5.3) gives us Equation (5.4). Using the same reasoning, but using $\mathbf{T}$ to convert w_{t+1} to j_t, we also obtain Equation (5.5).

$$w_{t+1} = \mathbf{T}^{-1}\, w_t \tag{5.2}$$

$$\mathbf{P}\, w_{t+1} = \mathbf{P}\, \mathbf{T}^{-1}\, w_t \tag{5.3}$$

$$j_{t+1} = \mathbf{P}\, \mathbf{T}^{-1}\, w_t \tag{5.4}$$

$$j_t = \mathbf{P}\, \mathbf{T}\, w_{t+1} \tag{5.5}$$

Ideally, it $\mathbf{T}$ perfectly represents the movement of the robot, Equation (5.4) must be true. However, if it does not correctly represent the movement of the robot, applying the transformation $\mathbf{T}^{-1}$ to w_t and then projecting it to 2D generates a *reprojection error*. The reprojection error is quantified as the difference between

where a point should be and where it actually is. It is computed for all the features F_t in frame t and for the features F_{t+1} in frame $t+1$, as in Equation (5.6).

$$\epsilon = \sum_{F_t, F_{t+1}} (j_t - \mathbf{P}\,\mathbf{T}\,w_{t+1})^2 + (j_{t+1} - \mathbf{P}\,\mathbf{T}^{-1}\,w_t)^2 \tag{5.6}$$

Therefore, the matrix $\mathbf{T}$ that best represents the movement of the robot is the one that minimizes the reprojection error ϵ. This is a minimization problem known as *least-squares curve fitting* and can be solved by the *Levenberg-Marquardt Algorithm* [92, 93].

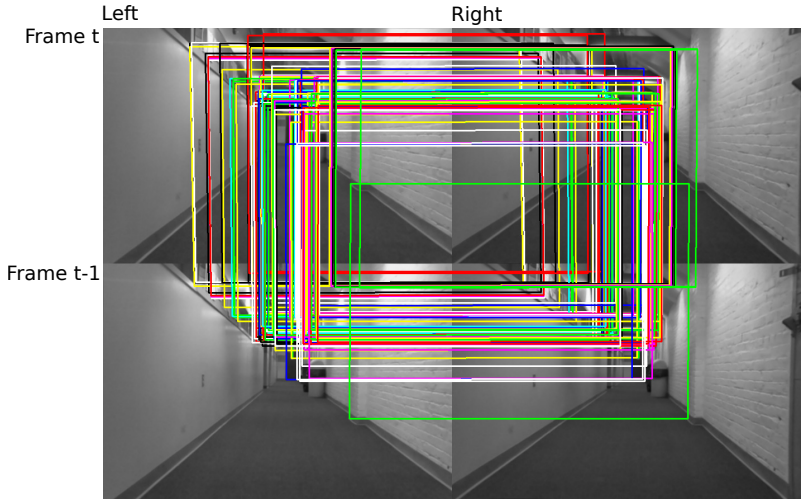
The matrix found by the algorithm is considered valid if the number of elements in the clique is above a threshold l_c and the reprojection error is below a threshold l_e.

5.5 Implementation and Results

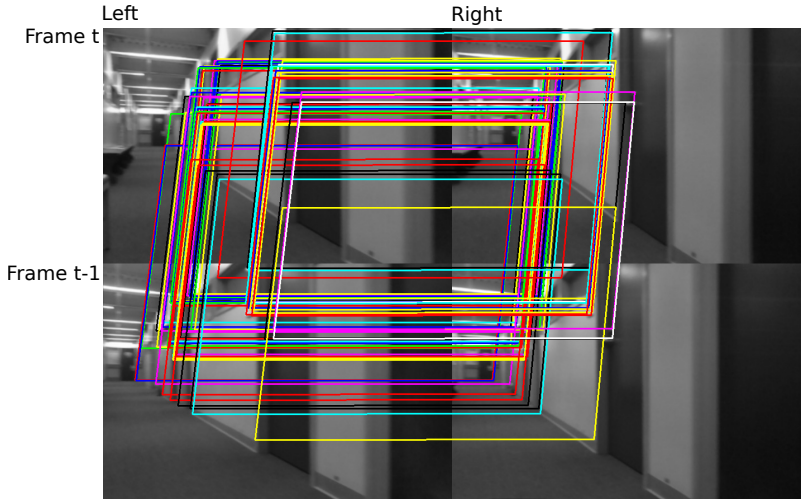
The visual odometry algorithm is also implemented in C++ using OpenCV for some of the image processing, namely, the feature detection using FAST and the tracking using the KLT. For the least-squares curve fitting, we used the implementation of the Levenberg-Marquardt method for the C language known as Levmar [94].

Figure 5.4 shows the output of the tracking process. The corners of each quadrilateral touch corresponding points on each of the four images. When the robot is moving forward, the polygons look like rectangles, whereas when the robot is turning, the shapes are skewed. The image that shows a left turn is a good illustration of the fact that the movement of the points is the inverse of the robot's movement: while the robot is turning left, the tracked points are moving to the right.

In Chapter 7 we will discuss in more details the accuracy and performance of the visual odometry algorithm.



(a) Robot moving forward.



(b) Robot turning left.

Figure 5.4: Tracking points across frames. Corners of a quadrilateral touch corresponding points on the four images. When the robot is moving forward, the quadrilaterals are more rectangle-shaped while when the robot is turning, they become skewed.

5.6 Summary

This chapter presented the visual odometry algorithm implemented in this project. Section 5.1 discussed the feature detection problem and briefly introduced some of the algorithms that can solve it, including FAST, the one that was chosen for this project. Section 5.2 explained how we used KLT to track the features frame by frame, as well as how to use the same method to derive disparity information for a set of points. In Section 5.3, we described a method to select a set of consistent points to compute the odometry information, by building a consistency matrix and reducing the problem to the Maximum Clique Problem. Section 5.4 introduced the concept of reprojection error and showed how to represent the odometry problem as a least-squares curve fitting problem, that can be solved by the Levenberg-Marquardt Algorithm. Finally, Section 5.5 presented some of the implementation details and some of the results obtained by the algorithm. In Chapter 7, we will present the results obtained by this method.

In the next chapter, we discuss Pure Pursuit, a steering method that can compute steering angles in order to make the robot follow the desired path.

Chapter 6

Steering Algorithm

The planner described in Chapter 4 assumes that the robot is able to move in any direction, however, that is not the case for the Robocart. By controlling the steering wheel and the throttle, we can make the robot drive forwards and make curves, but it cannot move sideways. In order to guide the robot along the path computed by the planner, we need a method that takes into account the limitations of a car-like robot.

In this chapter, we describe the *Pure Pursuit* method. Pure Pursuit is a simple path tracking method proposed in the early 80s and implemented on the Carnegie-Mellon Navlab. Coulter [35] describes its implementation and points out a curious fact: not only the algorithm proved to be more robust than the other ones tried on the Navlab, but after reviewing the code, the team found out that two completely different values for the look-ahead distance parameter were being used in different parts of the code and the tracker was working properly in spite of that major mistake. Section 6.1 describes the mapping of the path from the image to the robot frame. Section 6.2 introduces a simplified model for the car. In Section 6.3 we derive the control law that generates the steering angles to track the desired path. Finally,

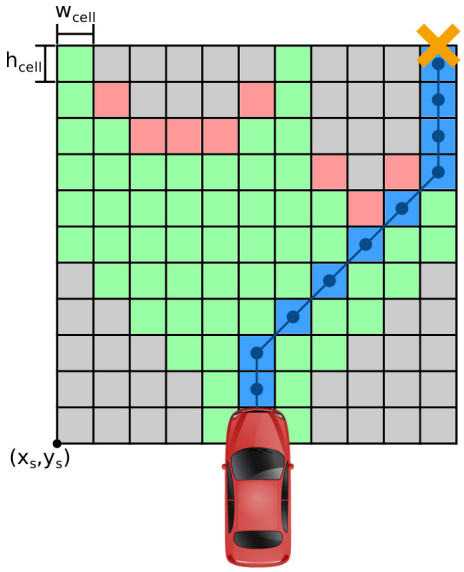


Figure 6.1: Converting a path from the grid frame (blue) to the cart frame (dark blue). Each cell in the grid frame is represented by the point in the cart frame corresponding to the center of the cell.

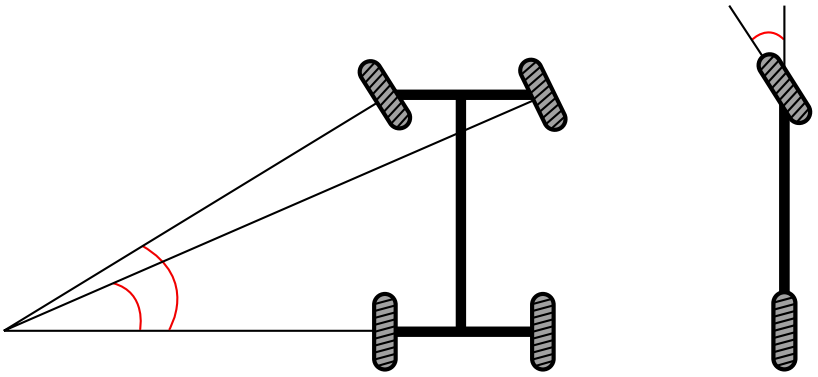
Section 6.4 shows the obtained results and briefly discusses the effects of the look-ahead distance parameter on the behavior of the algorithm.

6.1 Coordinate System

The A* algorithm runs on the occupancy grid and, therefore, the path generated by it is on the grid frame. However, the best frame for the steering algorithm to work on is the robot frame. The conversion from the grid frame is simple, as shown in Figure 6.1. Each point in the grid frame is converted to the point in the cart frame that is in the center of the cell.

We can use the cart frame coordinates of the corner of the occupancy grid, (x_s, y_s), and the size of a cell, w_{cell} x h_{cell} to convert a grid point (x_g, y_g) to a cart point (x_c, y_c) as follows:

$$x_c = x_s + (x_g + 0.5) \cdot w_{cell}$$



(a) Ackermann Steering. (b) Bicycle model.

Figure 6.2: Car with Ackermann Steering and the angles of the inside and outsize front wheels indicated and the corresponding Bicycle Model.

$$y_c = y_s + (y_g + 0.5) \cdot h_{cell}$$

6.2 Vehicle Model

When a four-wheeled vehicle is driven on a circle, the inside front tire must make a smaller turning radius than the outside front tire in order to avoid making the tires slip sideways, which reduces its lifespan. That behavior is typically implemented by a geometric arrangement of linkages known as Ackermann Steering [95]. For the purposes of path tracking, cars with Ackermann Steering are commonly modeled by the simplified bicycle model [96]. This model combines the two front wheels into one and does the same to the rear wheels. Figure 6.2 shows a representation of a vehicle with Ackermann Steering and the bicycle model.

Figure 6.3 shows the bicycle model in more details.

In order to drive on a circle of radius R, a vehicle of length L must turn the front wheel to an angle δ. The right triangle on the picture gives us a simple relationship between the three variables:

$$tan(\delta) = \frac{L}{R} \tag{6.1}$$

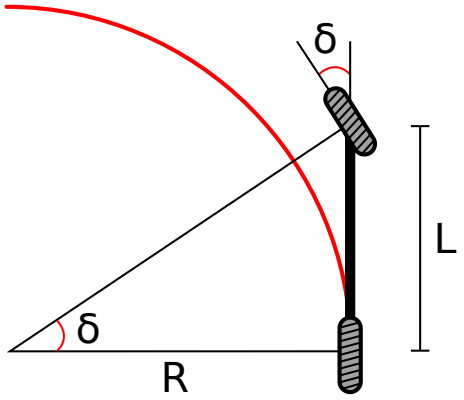


Figure 6.3: Detailed bicycle model. A vehicle of length L must turn the front wheel to an angle δ in order to drive on a circle of radius R.

6.3 Pure Pursuit

Using Equation (6.1), we can make the bicycle drive on a circle of radius R by adjusting the angle of the front wheel to δ. In order to track a path, Pure Pursuit chooses a goal point (x_g, y_g) that is a determined *look-ahead distance* l_d away from the rear axle. Then, it geometrically calculates the curvature of a circular arc that connects the rear axle to that goal point. Finally, it computes a steering angle that makes the robot drive on the computed circle. We will use Figure 6.4 to derive the equation for the steering angle.

The green line connects the rear axle to the goal point (x_g, yg) and its length is the look-ahead distance l_d. The angle between the green line and the vehicle is α. The angle $\angle ACB$ is complementary to α, therefore:

$$\angle ACB = 90 - \alpha$$

The triangle $\triangle ABC$ is isosceles, so:

$$\angle ACB = \angle ABC$$

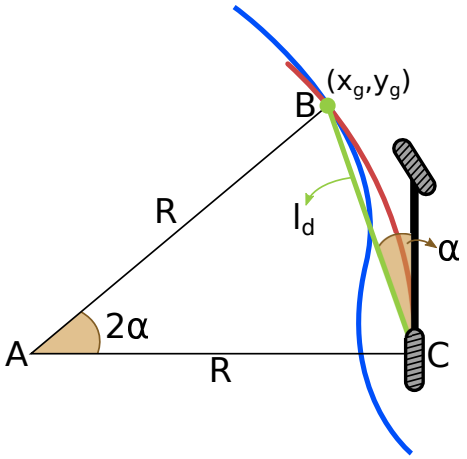


Figure 6.4: Diagram used for the derivation of the control law for the Bicycle Model. The path to be tracked is represented in blue, the look-ahead distance in green and the desired curve in red.

The sum of the angles in a triangle is π radians, so the angle $\angle BAC$ is:

$$\angle BAC + \angle ACB + \angle ABC = \pi$$

$$\angle BAC + 2 \cdot \left(\frac{\pi}{2} - \alpha\right) = \pi$$

$$\boxed{\angle BAC = 2\alpha}$$

This result is already written on Figure 6.4.

Applying the Law of Sines to $\triangle ABC$:

$$\frac{l_d}{sin(2\alpha)} = \frac{R}{sin\left(\frac{\pi}{2} - \alpha\right)}$$

Using the trigonometric identities $sin(2\theta) = 2\ sin(\theta)\ cos(\theta)$ and $sin\left(\frac{\pi}{2} - \theta\right) = cos(\theta)$:

$$\frac{l_d}{2sin(\alpha)\cancel{cos(\alpha)}} = \frac{R}{\cancel{cos(\alpha)}}$$

$$\frac{l_d}{sin(\alpha)} = 2R$$

Substituting Equation (6.1), we get:

$$\frac{l_d}{sin(\alpha)} = \frac{2\,L}{tan(\delta)}$$

Solving for δ, we obtain the Pure Pursuit control law:

$$\boxed{\delta(t) = tan^{-1}\left(\frac{2\,L\,sin(\alpha(t))}{l_d}\right)}$$

Since the length of the vehicle L and the look-ahead distance l_d are fixed, the steering angle δ can be computed by choosing the goal point and computing α, then substituting it in the control law.

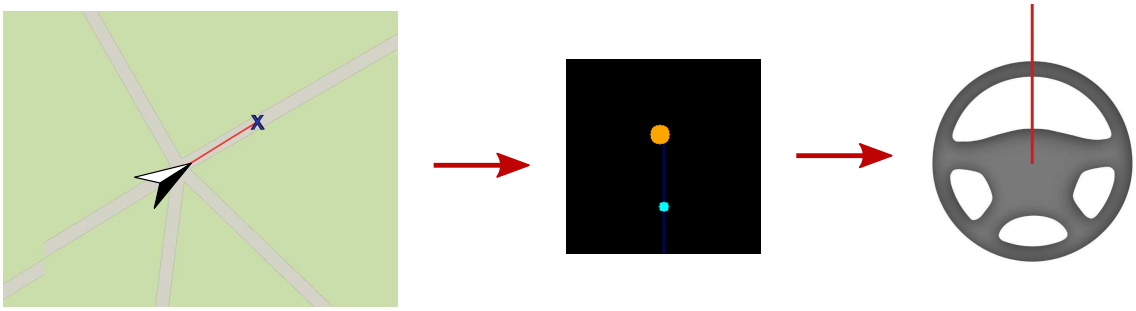
6.4 Implementation and Results

The implementation of the Pure Pursuit is straightforward and was done using the C++ language, without the use of third party libraries.

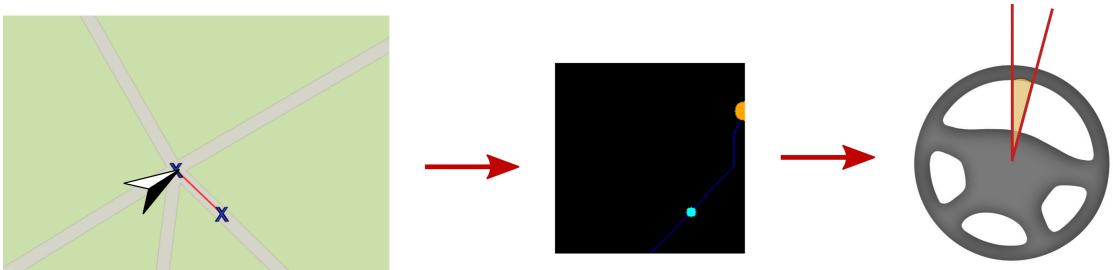
Figure 6.5 shows the output of the algorithm for a goal in front of the robot, in which case the robot is expected to drive straight ahead, as well as a goal to the right, where the robot is expected to turn the steering wheel to the right.

Pure Pursuit has only one parameter to be tuned: the look-ahead distance. A smaller look-ahead distance results in a more precise tracking while increasing it makes the path smoother. Figure 6.5 also shows the output of the planner for the same goal to the right of the robot with a bigger look-ahead distance. The steering angle becomes smaller, which means that the robot will make a smoother curve, instead of trying to precisely track the path.

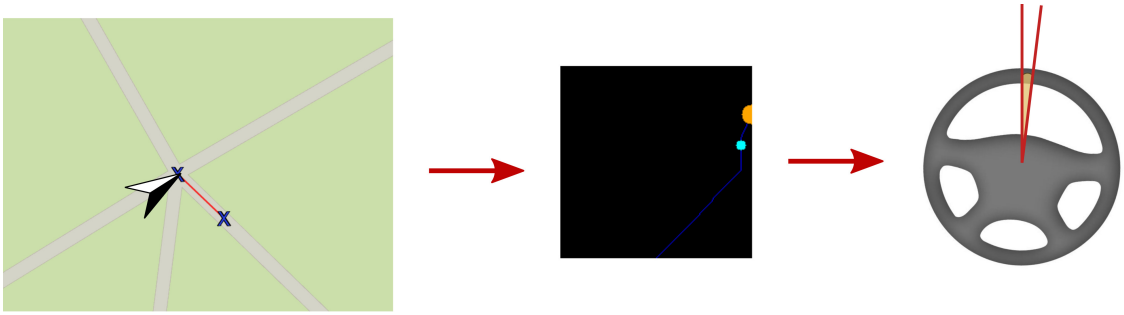
More results regarding the output of Pure Pursuit are shown and discussed in Chapter 7.



(a) Goal in front of the robot. Steering angle is 0°.



(b) Goal to the right of the robot. Steering angle is 15°.



(c) Goal to the right of the robot and bigger look-ahead distance. Steering angle is 7°.

Figure 6.5: Simulated robot and goal locations and the corresponding output of the planner. The orange dot is the goal, the blue line is the path computed with A* and the light blue dot is the point of the path that is one look-ahead distance away from the robot.

6.5 Summary

This chapter described the algorithm used in this project in order to generate steering angles that make the robot track the path computed by A*. Section 6.1 explained the process of converting the path from the image coordinate frame to the robot frame, by mapping each point in the path to the center of the cell that covers the point on the occupancy grid. Section 6.2 described the bicycle model, that makes some assumptions in order to simplify the derivation of the control law for the path tracking. Section 6.3 derived the equations for the Pure Pursuit algorithm. Finally, Section 6.4 showed some outputs from the algorithm when the goal is ahead of the robot and when it needs to make a curve. It also briefly discussed how the look-ahead distance influences the algorithm.

The next chapter describes the testing procedure we used to validate the algorithms implemented during this project as well as evaluates the performance of the individual parts of it.

Chapter 7

Evaluation and Results

In the previous chapters, we presented results for each one of the algorithms executed in isolation. In this chapter, the algorithms are combined in order to generate a path for the robot. The algorithms are implemented in the C++ language, using the OpenCV library for some of the vision-related tasks. Section 7.1 introduces the experimental platform used during the tests. In Section 7.2, the results are analyzed qualitatively to determine if the instructions generated by the algorithm are reasonable. In Section 7.3 we analyze the results quantitatively in terms of execution time, since a fast response is crucial when dealing with a moving vehicle. Finally, Section 7.4 presents the results obtained from the visual odometry algorithm.

7.1 Experimental Platform

Integrating the planner developed in this project with the mechanical systems on the Robocart is a challenging task that requires the use of elaborate control techniques as well as a considerable amount of fine tuning on both the planner and the physical components of the robot. This is a project by itself and will be done by the team working on the next iteration of the Robocart. In order to test and validate the

approach proposed here, we built an experimental platform that allows us to arrange the sensors in a similar way as they are going to be mounted on the golf cart and safely collect data for analysis.

The platform was mounted on a service cart, as shown in Figure 7.1. The cameras are mounted on the Raspberry Pi computers using a 3D-printed support and the computers themselves are attached to a 80/20 bar using another 3D-printed part. The bar is, then, attached to the cart. Both printed parts were designed by the team that is currently working on the mechanical systems of the golf cart, composed by Robert Crimmins and Raymond Wang. The compass is connected to the GPIO pins of the Raspberry Pi that controls the left camera and fixed on the cart by a zip tie. The computers are connected to a laptop via ethernet using a Rosewill RNX-N150RT wireless router and the laptop is also connected to a GPS receiver. Finally, the system is powered by a car battery connected to a power inverter.

In this setup, we can use a laptop instead of the Robocart server because no heavy processing is done online. During the experiment, the cart is pushed by a person and the laptop records all the data produced by the sensors. After the capture, the data is fed to the algorithms for processing. This methodology allows us to experiment with different parameters for the algorithms and do a fair comparison since we can use the exactly same sensor data in each execution.

7.2 Qualitative Analysis

In order to evaluate the behavior of the algorithm, we moved the platform along the path shown on Figure 7.2, on the third floor of Atwater Kent Laboratories. This path is mostly obstacle-free, but has walls close to the robot, which allow us to investigate how the algorithms react to them. There are also two sharp curves to

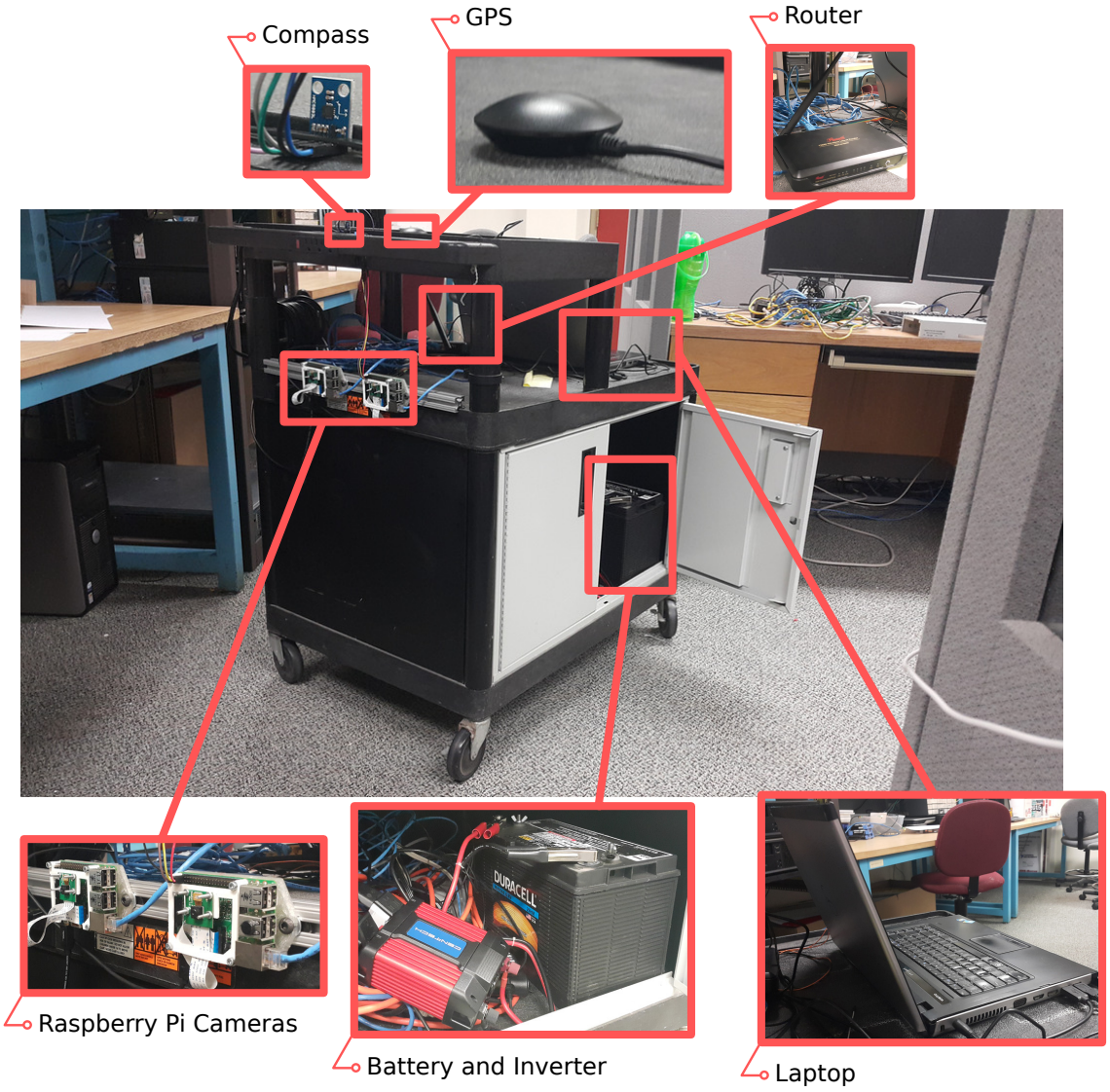


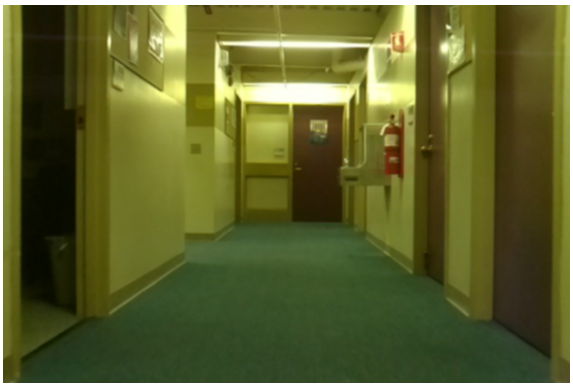
Figure 7.1: Experimental platform with all its components indicated.

Figure 7.2: Path executed by the experimental platform for the tests. It started on the blue dot and moved along the red line, stopping on the green dot.

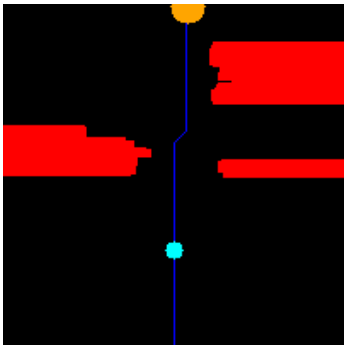
represent the conditions in which the robot must change its direction of movement.

The output of the planner on a place with walls is shown on Figure 7.3. The walls are detected as obstacles and represented on the occupancy grid. We can also verify if the obstacles on the occupancy grid are represented on a correct scale by measuring the distance between the two walls on the grid. On Figure 7.3, that distance is approximately 47 pixels. The occupancy grid is 200x200 and the X axis covers an area of 320 centimeters. This means that the distance between the two walls indicated on the grid is 75.2 centimeters. Adding to that the width of the robot (61cm) and the clearance (15cm on each side), the real distance between the walls according to the grid is 166.2 centimeters or approximately 65.4 inches. The actual distance obtained with a tape measure is approximately 70 inches, as shown in Figure 7.4.

Figure 7.5 shows the output of the planner when there is a wall in front of the robot. The wall is clearly shown in the picture as an obstacle. We can also inspect



(a) Image seen by the left camera.



(b) Planner output.

(c) Robot location.

Figure 7.3: Planner output on a free path with walls on the sides. The dilated obstacles are represented in red. The orange dot is the goal and the blue dot is the look-ahead distance for Pure Pursuit. The blue line is the path found by the algorithm.

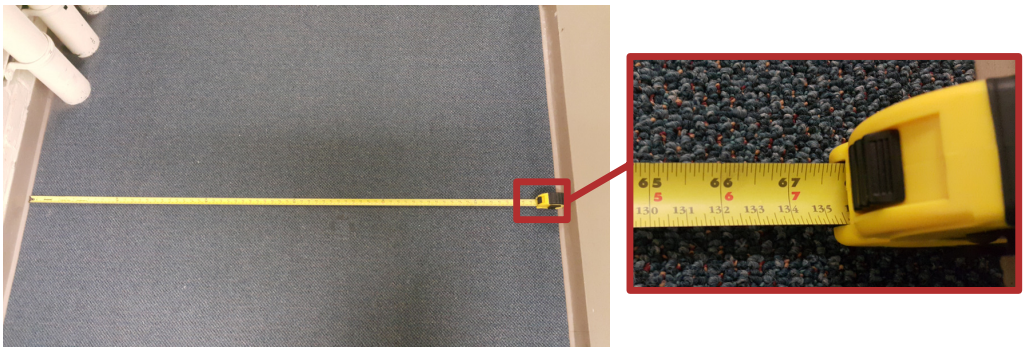
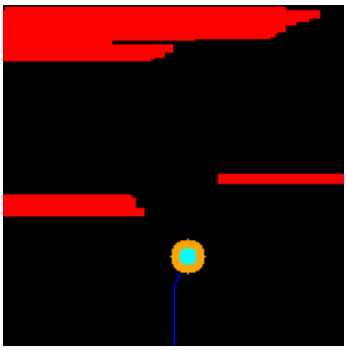
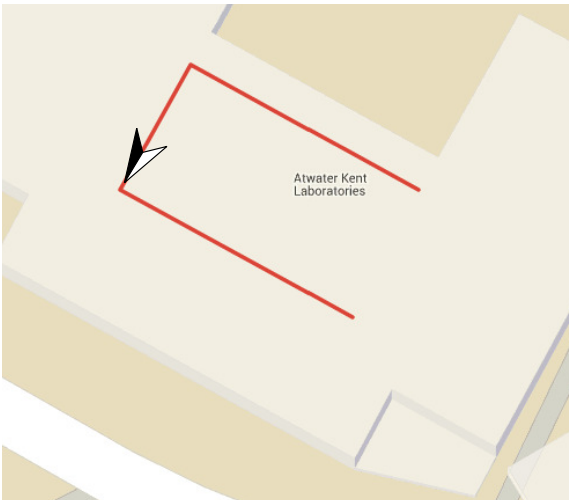


Figure 7.4: Distance between the two walls.

(a) Image seen by the left camera.



(b) Planner output.

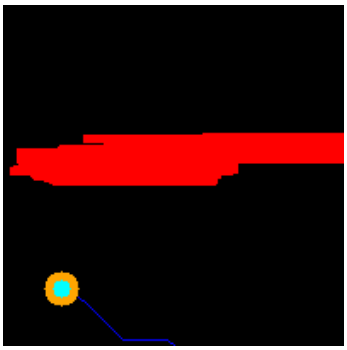


(c) Robot location.

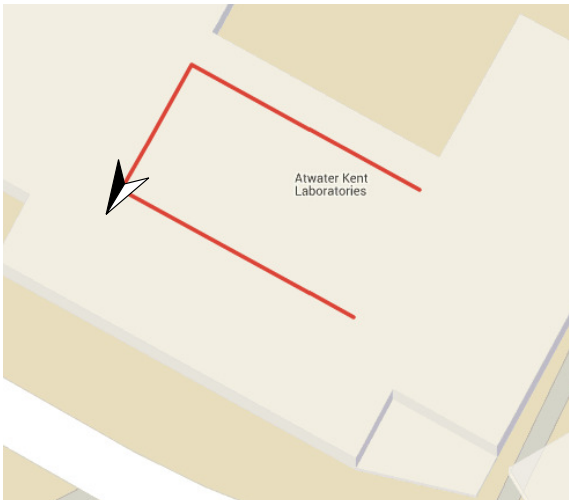
Figure 7.5: Planner output on a free path with walls on the sides and in front of the robot.

the distance between the walls on the grid to verify if it is consistent with the real world. The vertical distance between the wall on the left and the wall in front of the robot is 78 pixels and the distance covered by the occupancy grid in the Z direction is 500 centimeters. Since obstacle dilation is only applied in the horizontal direction, we can convert the pixels to centimeters directly, without considering the robot width and the clearance distance. Doing the conversion, we obtain 195 centimeters or 76.8 inches. As already shown in Figure 7.4, the actual distance is approximately 70 inches.

(a) Image seen by the left camera.



(b) Planner output.



(c) Robot location.

(d) Steering angle (23°)

Figure 7.6: Planner output on a curve.

Another important aspect of the planner is its ability to guide the robot on curves. Figure 7.6 shows the output of the planner on a curve. As expected, it outputs a steering angle that makes the robot curve to the left. In the future, this angle can be passed to the steering system that will generate the appropriate control signals for the steering motor to make the robot turn as desired.

7.3 Execution Time Analysis

Since the planner proposed here is meant to be used on a moving vehicle, the algorithms must run in real time so that the robot can quickly react to changes in the environment around it. In order to investigate the execution time, we ran the planner on one of the datasets captured during the experiments. Each run processes the first 400 frames and the processing time for each frame is recorded. During this experiment, most of the I/O and all the graphic output of the program were disabled, since those can be time-consuming operations.

Figure 7.7 shows the execution time when using Semi-Global Block Matching algorithm for the stereo disparity computation. The figure contains results of two executions, one of them with visual odometry running and the other without it. The purpose of this experiment was to investigate if running the odometry algorithm has a significant impact on the performance. The graph shows that running the visual odometry causes a slight increase in execution time. Since the odometry algorithm relies on computationally expensive operations, its use should cause a considerable increase in the execution time, but the use of TBB to run it in parallel with the stereo vision algorithm ensures that the multiple cores of the server are used and the execution time is less affected.

Figure 7.8 shows a similar experiment, but the Semi-Global Block Matching was replaced by the Block Matching algorithm. With this change, the planner becomes more than 5 times faster. Even though the slower disparity method gives better results, with proper parameter tuning the faster algorithm can replace it and provide a considerable speedup.

Using the faster method for disparity computation, the algorithm is able to process approximately 33 frames per second. Assuming that for a robust navigation,

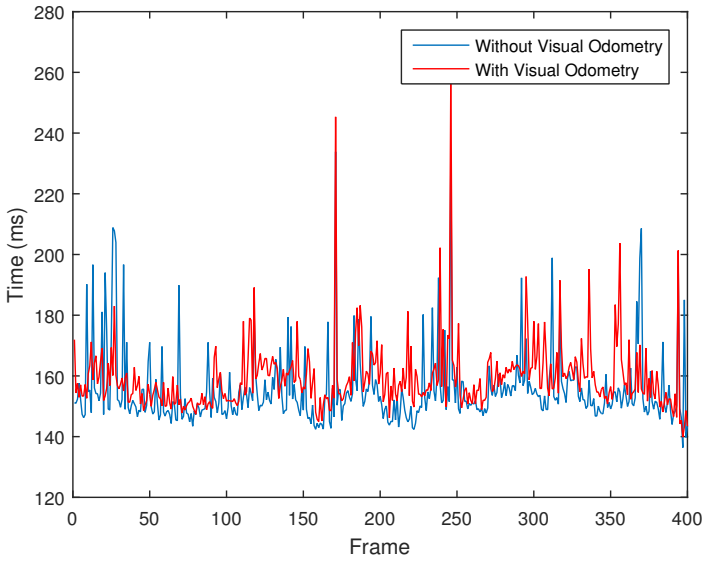


Figure 7.7: Execution of the planner using the Semi-Global Block Matching algorithm with and without the visual odometry running in parallel. Without visual odometry, the average execution time is 155ms with standard deviation of 12ms. With the visual odometry, the average execution time is 159ms with standard deviation of 11ms.

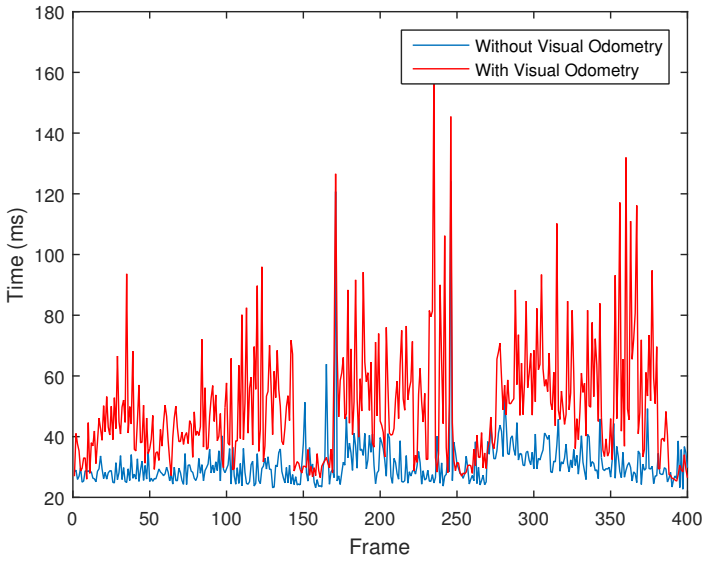


Figure 7.8: Execution of the planner using the Block Matching algorithm with and without the visual odometry running in parallel. Without visual odometry, the average execution time is 30ms with standard deviation of 8ms. With the visual odometry, the average execution time is 49ms with standard deviation of 19ms.

the robot must react at least once every quarter meter [15], the current implementation could, in theory, drive the Robocart at approximately 8.3 meters per second or about 18 miles per hour. This maximum speed value can certainly be increased by optimizing the code using techniques that were not considered in this project, such as offloading some of the processing to the Graphics Processing Unit (GPU), using TBB more aggressively to increase parallelism on the CPU or even lower level approaches such as the use of Single Instruction Multiple Data (SIMD) assembly instructions, *e.g.* Intel SSE, to operate on multiple pieces of data at the same time, however, the current theoretical maximum speed is more than the average speed of a golf cart (12 to 14 miles per hour [97]). It is also worth mentioning that the proposed approach is not suitable nor intended for use on high speed scenarios such as highway driving.

7.4 Visual Odometry

The visual odometry method was evaluated by logging the estimated position of the robot on each frame and then plotting the results after the execution of the algorithm. Figure 7.9 shows the obtained result.

The algorithm was able to track the position of the robot with accuracy using only the images from the cameras. These results are very promising, and indicate that this method can potentially be used to complement the GPS and compass sensors in order to make the localization of the robot more reliable.

7.5 Summary

This chapter presented the results obtained in this project with respect to the planning and odometry algorithms. Section 7.1 presented the experimental platform

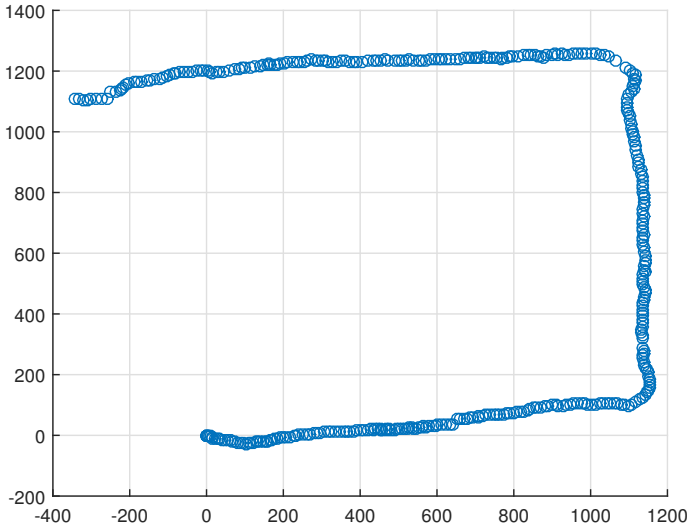


Figure 7.9: Path tracked by the visual odometry algorithm. The robot starts at coordinates $(0, 0)$. The proposed method is able to estimate with good precision the movement of the robot.

used to capture data for the experiments. Section 7.2 analyzed the output of the planner qualitatively with respect to obstacle and wall detection and the behavior of the planner on curves. Section 7.3 discussed the performance of the algorithm as well as the impact of the visual odometry and the stereo disparity algorithms on the execution time. Finally, Section 7.4 showed the output of the visual odometry algorithm.

The next chapter concludes this document summarizing the results obtained in this project.

Chapter 8

Conclusions and Future Work

The Robocart project aims to develop a fully autonomous golf cart and this thesis is part of that effort. We proposed and implemented some of the systems on the robot that are necessary for its autonomy, as well as developed a first prototype of a planner that can guide the robot along a desired path. This chapter summarizes the achievements of this project with respect to the autonomous systems implemented on the Robocart in Section 8.1 and the proposed path planner in Section 8.2. In Section 8.3 we conclude this document by discussing possible next steps for the Robocart project as well as making some recommendations for future teams working on it.

8.1 Robocart Systems

The goal of this iteration of the Robocart was the implementation of five systems on the robot, namely, cameras, compass, throttle, braking and steering. Braking and steering were implemented by Robert Crimmins and Raymond Wang as a Major Qualifying Project (MQP). At the time of this writing, brakes are fully functional and steering is in final stages of implementation. The details of their implementation are described in their final report.

The throttle, compass and the cameras were implemented as part of this thesis. Throttle was very simple to implement, since the Arduino offers an easy way to communicate with devices using I2C. Interfacing with the compass is also easy, since the Raspberry Pi offers I2C communication capabilities as well. On the other hand, capturing images from the cameras was a more challenging task.

Interfacing with the cameras via MMAL is not straightforward and good documentation is not available, however, the V4L2 driver makes the task much simpler. The driver proved to be very stable and worked flawlessly during the entire project. We also proposed a method to synchronize the image capture on both cameras. During the experiments, our method worked surprisingly well, achieving synchronization down to the hundredth of a second and producing good quality disparity maps, even when the cameras are moving. Moreover, this method does not require any additional hardware and the PTP implementation maintained by the Linux PTP project is open-source, can be obtained for free and compiles without issues on the Raspberry Pi and on modern Ubuntu distributions.

Overall, the Raspberry Pi is a good platform to work with. In recent years, several single-board computers became available on the market, but the strong community around the Pi makes it stand out from the crowd. On their forums it is possible to interact and exchange information with other people, from enthusiasts to experienced engineers, and discuss solutions to common problems. However, it is worth mentioning that, even though we obtained good results with this camera setup, it is not optimal. OpenCV has a standard interface for capturing images that can easily extract frames from video files or read data from cameras connected to the computer. Since we use a completely different method to capture images, OpenCV applications that are available online have to be modified before we can test them with our setup, which slows down the development process. This setup

makes sense on the original concept of the Robocart, in which the server runs on the cloud and the images are sent over the network, however, now that the server is mounted on the robot there is no good reason for using that kind of setup. Substituting the Raspberry Pi cameras by affordable USB webcams will make the system more reliable and easier to maintain, as well as speed up the development process.

8.2 Path Planner

The planner proposed in this project has three main components: the obstacle avoidance, the visual odometry and the steering algorithm. Each one of them was implemented based on methods described on the literature with some modifications.

The obstacle avoidance method based on an occupancy grid is able to correctly identify obstacles and to plan a path around them. Our implementation also performs well and, in theory, is fast enough to guide a golf cart at full speed. Being a purely vision-based algorithm, the proposed method does not require expensive hardware and can be implemented using open-source tools and libraries available on the internet, making it a very affordable approach to vehicle autonomy. On the other hand, there are some drawbacks of relying solely on computer vision. Vision algorithms in general and specially those intended for disparity computation tend to have a large number of parameters. Combining the parameters for the disparity algorithm and the occupancy grid computation, our method has over 30 parameters that can be adjusted. Those parameters allow the user to fine tune the algorithm for a specific scenario, but the tuning process can be tedious and involves some amount of guesswork, even with proper knowledge of each parameter. In order to ensure the safety of the vehicle and its passengers, this method should be complemented by other techniques that are reliable with little or no tuning, such as using ultrasonic

sensors to stop the car when it is close to an obstacle that was not detected by the vision system.

The Pure Pursuit method was used to generate steering angles that guide the car along the path computed by the obstacle avoidance algorithm. Even though we could not test it on an actual vehicle during this project, it produced consistent results on the tests using our experimental platform.

Finally, the visual odometry algorithm was able to track the movement of the robot with satisfactory precision. A reliable visual odometry module is probably the biggest contribution of this project to the Robocart, since it can be used by other systems as a standalone module to improve the quality of the localization data.

In summary, the results obtained during this project indicate that a vision-based approach for vehicle autonomy might be feasible. The algorithms described and implemented in this thesis can be used as a foundation for more elaborate methods that can, in the near future, turn the Robocart into a fully autonomous golf cart.

8.3 Future Work

Building a self-driving vehicle is a challenging task, even for giant companies like Google, and even though the Robocart is taking its first steps towards full autonomy, there is room for improvement in every aspect of the project. In this section, we list some of the work that can be done in the near future on the Robocart.

Replace vision system: As already mentioned before, the camera setup is not optimal and slows down the development. A future team should consider replacing it by a pair of USB webcams. There are also inexpensive "3D webcams" available on the market, such as the Minoru 3D [98], that are basically two cameras sharing the same USB port.

Integrate algorithms with the Robocart: Integrating the proposed algorithms with the physical golf cart involves communicating with various sensors and interfacing with mechanical systems.

Add more sensors to the system: For increased reliability, more sensors can be added to the cart and integrated into the planner. Ultrasonic sensors, for example, are not expensive and would significantly improve safety for the robot and its passengers by stopping it from colliding with obstacles not detected by the planner.

Improve existing algorithms: There is a substantial amount of algorithms described in the literature that could be used to improve the ones implemented in this project. The occupancy grid construction can be modified to detect moving objects [75] and Control theory can be used to build a more sophisticated path tracking method [96].

Fuse visual odometry with other sensors: Data from the compass can be noisy and is affected by metallic objects in the environment and the data from the GPS has accuracy in the order of meters. Combining that data with the more fine-grained information provided by the visual odometry can potentially provide very precise localization for the Robocart, as well as allow it to operate on environments where sensor data is not available, such as indoor locations.

These are projects that can be executed in the short term and can help the Robocart to achieve full autonomy. Once it becomes a full-fledged self-driving car, it can be used in various creative ways, such as a shuttle for the campus or even a snow plow. The possibilities are endless.

Appendix A

Example Communication with the Compass

```
1  #include <cstdio>
2  #include <cstdlib>
3  #include <cmath>
4  #include <fcntl.h>
5  #include <unistd.h>
6  #include <cstring>
7  #include <sys/ioctl.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <linux/i2c-dev.h>
11
12 /* Writes the 8-bit value 'val' to register 'reg'
13  * on the I2C device represented by 'fd'.
14  */
```

```
15 int writeToDevice(int fd, int reg, int val) {
16   char buf[2];
17   buf[0] = reg;
18   buf[1] = val;
19
20   if(write(fd,buf,2) != 2) {
21     printf("Error writing to device.\n");
22     exit(1);
23   }
24 }
25
26 /* Main Function */
27 int main() {
28   int fd;
29
30   //Try to open the I2C device:
31   if((fd = open("/dev/i2c-1", O_RDWR)) < 0) {
32     printf("Failed to open I2C device.\n");
33     exit(1);
34   }
35
36   //Set the I2C device address (0x1e for our compass):
37   if(ioctl(fd, I2C_SLAVE, 0x1e) < 0) {
38     printf("Device not connected.\n");
39     exit(1);
40   }
```

```
41
42  //Configure device:
43  //Configuration register A:
44  //8 samples per measurement, data output rate 15Hz,
       normal measurement mode
45  writeToDevice(fd,0x00,0x70);
46
47  //Configuration register B:
48  //Gain = 390
49  writeToDevice(fd,0x01,0xA0);
50
51  //Mode register:
52  //Continuous measurement mode, normal I2C speed
53  writeToDevice(fd,0x02,0x00);
54
55  //Wait for the device to be ready:
56  usleep(6*1000);
57
58  unsigned char buf[16];
59
60  while(true) {
61    //Send read command:
62    buf[0] = 0x03;
63
64    if(write(fd,buf,1) != 1) {
65      printf("Error writing to device.\n");
```

```
66        exit(1);
67      }
68
69      //The device sends back the values of the X, Y and Z
           registers:
70      if(read(fd,buf,6) != 6) {
71        printf("Error reading from device.\n");
72        exit(1);
73      }
74
75      //Convert to the 2-byte representation:
76      short x = (buf[0] << 8) | buf[1];
77      short y = (buf[4] << 8) | buf[5];
78      short z = (buf[2] << 8) | buf[3];
79
80      //Compute the heading direction:
81      float direction = 0;
82
83      if(y > 0) {
84        direction = M_PI/2 - atan2(x,y);
85      } else if(y < 0) {
86        direction = 3*M_PI/2 - atan2(x,y);
87      } else if(x < 0) {
88        direction = M_PI;
89      } else {
90        direction = 0;
```

```
91      }
92
93      //Adjustment to point to the geographic north in
           Worcester:
94      direction -= (14+24/60)*M_PI/180;
95
96      //Ourput data:
97      printf("x=%d, y=%d, z=%d, direction rad = %f,
           direction deg = %f\n",x,y,z,direction,direction
           *180/M_PI);
98
99      //Wait for the next measurement:
100     usleep(67*1000);
101   }
102 }
```

Appendix B

Example Communication with the Digital Potentiometer

```
1  #include <Wire.h>
2
3  byte incomingByte = 0;
4  byte val = 0;
5
6  /* Setup function, executed once at the beggining. */
7  void setup() {
8    //Initialize Wire:
9   Wire.begin();
10
11  //Initialize serial:
12  Serial.begin(9600);
13 }
14
```

```
15 /* Loop function, executed indefinitely. */
16 void loop() {
17  //If there is data available at the serial port:
18  if(Serial.available() > 0) {
19   //Read byte from the serial port:
20   incomingByte = Serial.read();
21
22   //The server sends a number between 0 and 100. Our
      potentiometer
23   //varies between 0 and 10k ohms, but we only need half
      of it.
24   //So, we map the received value from the computer to
      the appropriate
25   //range:
26   val = map(incomingByte,0,100,128,255);
27
28   //Begin transmission to our potentiometer (address = 0
      x28);
29     Wire.beginTransmission(0x28);
30
31     //Send write command:
32     Wire.write(B10101001);
33
34     //Send the desired resistance value:
35     Wire.write(val);
36
```

```
37      //Finish transmission:
38      Wire.endTransmission();
39   }
40 }
```

Appendix C

Example Communication with the Arduino

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <errno.h>
6  #include <termios.h>
7  #include <string.h>
8  #include <sys/ioctl.h>
9
10 int main() {
11   struct termios options;
12   int fd = open("/dev/ttyACM0", O_RDWR | O_NONBLOCK);
13
14   if(fd == -1) {
```

```
15      printf("Could not connect to Arduino.\n");
16      return 0;
17    }
18
19    //Open serial device:
20    if(tcgetattr(fd,&options) < 0) {
21      printf("Could not get attributes of the serial port\n
           ");
22      close(fd);
23      return 0;
24    }
25
26    //Set baud rate:
27    speed_t baudRate = B9600;
28    cfsetispeed(&options, baudRate);
29    cfsetospeed(&options, baudRate);
30
31    //Set 8N1:
32    options.c_cflag &= ~PARENB;
33    options.c_cflag &= ~CSTOPB;
34    options.c_cflag &= ~CSIZE;
35    options.c_cflag |= CS8;
36
37    //No flow control:
38    options.c_cflag &= ~CRTSCTS;
39
```

```
40   //Enable read and ignore control lines:
41   options.c_cflag |= CREAD | CLOCAL;
42
43   //Turn off software flow control:
44   options.c_iflag &= ~(IXON | IXOFF | IXANY);
45
46   //Make raw:
47   options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
48   options.c_oflag &= ~OPOST;
49
50   options.c_cc[VMIN] = 0;
51   options.c_cc[VTIME] = 0;
52
53   tcsetattr(fd, TCSANOW, &options);
54   if(tcsetattr(fd,TCSAFLUSH, &options) < 0) {
55     printf("Could not set attributes of the serial port.\
        n");
56     close(fd);
57     return 0;
58   }
59
60   //Here is where we send the actual data.
61   uint8_t byte;
62   write(fd,&byte,1);
63
64   //After we are done:
```

```
65    close(fd);
66 }
```

Bibliography

[1] T. M. Sentinel, "'phantom auto' will tour the city," https://news.google.com/newspapers?id=unBQAAAAIBAJ\&sjid=QQ8EAAAAIBAJ\&pg=7304,3766749\&hl=en, 1926, [Online; accessed 08-February-2016].

[2] T. Press-Courier, "Reporter rides driverless car," https://news.google.com/newspapers?id=vUpeAAAAIBAJ\&sjid=3WANAAAAIBAJ\&pg=6885,3667738\&hl=en, 1960, [Online; accessed 08-February-2016].

[3] T. Telegraph, "Cruising into the future," http://www.telegraph.co.uk/motoring/4750544/Cruising-into-the-future.html, 2001, [Online; accessed 08-February-2016].

[4] T. Kanade, C. Thorpe, and W. Whittaker, "Autonomous land vehicle project at cmu," in *Proceedings of the 1986 ACM fourteenth annual conference on Computer science*. ACM, 1986, pp. 71–80.

[5] R. Wallace, A. Stentz, C. E. Thorpe, H. Maravec, W. Whittaker, and T. Kanade, "First results in robot road-following." in *IJCAI*. Citeseer, 1985, pp. 1089–1095.

[6] D. Borsen, "Visions for tesla, the auto industry and self-driving teslas," https://www.youtube.com/watch?v=bl5vLC3Xlgc, 2015, [Online; accessed 08-February-2016].

[7] J. Larson and M. Trivedi, "Lidar based off-road negative obstacle detection and analysis," in *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*. IEEE, 2011, pp. 192–197.

[8] A. Vadlamani, M. Smearcheck, D. Haag, and M. Uijt, "Preliminary design and analysis of a lidar based obstacle detection system," in *Digital Avionics Systems Conference, 2005. DASC 2005. The 24th*, vol. 1. IEEE, 2005, pp. 6–B.

[9] W. Zhang, "Lidar-based road and road-edge detection," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*. IEEE, 2010, pp. 845–848.

[10] K. Peterson, J. Ziglar, and P. E. Rybski, "Fast feature detection and stochastic parameter estimation of road shape using multiple lidar," in *Intelligent*

Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on. IEEE, 2008, pp. 612–619.

[11] E. Guizzo, "How googles self-driving car works," *IEEE Spectrum Online, October*, vol. 18, 2011.

[12] C. Thorpe, M. H. Hebert, T. Kanade, and S. A. Shafer, "Vision and navigation for the carnegie-mellon navlab," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 10, no. 3, pp. 362–373, 1988.

[13] D. Song, H. N. Lee, J. Yi, and A. Levandowski, "Vision-based motion planning for an autonomous motorcycle on ill-structured roads," *Autonomous Robots*, vol. 23, no. 3, pp. 197–212, 2007.

[14] D. Murray and J. J. Little, "Using real-time stereo vision for mobile robot navigation," *Autonomous Robots*, vol. 8, no. 2, pp. 161–171, 2000.

[15] M. W. Otte, S. G. Richardson, J. Mulligan, and G. Grudic, "Local path planning in image space for autonomous robot navigation in unstructured environments," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on.* IEEE, 2007, pp. 2819–2826.

[16] D. Nistér, O. Naroditsky, and J. Bergen, "Visual odometry," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, vol. 1. IEEE, 2004, pp. I–652.

[17] ——, "Visual odometry for ground vehicle applications," *Journal of Field Robotics*, vol. 23, no. 1, pp. 3–20, 2006.

[18] A. Ohya, A. Kosaka, and A. Kak, "Vision-based navigation by a mobile robot with obstacle avoidance using single-camera vision and ultrasonic sensing," *Robotics and Automation, IEEE Transactions on*, vol. 14, no. 6, pp. 969–978, 1998.

[19] J. Borenstein and Y. Koren, "Obstacle avoidance with ultrasonic sensors," *Robotics and Automation, IEEE Journal of*, vol. 4, no. 2, pp. 213–218, 1988.

[20] N. National Coordination Office for Space-Based Positioning and Timing, "Gps accuracy," http://www.gps.gov/systems/gps/performance/accuracy/, [Online; accessed 14-February-2016].

[21] P. Bonnifait, P. Bouron, P. Crubillé, and D. Meizel, "Data fusion of four abs sensors and gps for an enhanced localization of car-like vehicles," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 2. IEEE, 2001, pp. 1597–1602.

[22] H. Li, F. Nashashibi, and G. Toulminet, "Localization for intelligent vehicle by fusing mono-camera, low-cost gps and map data," in *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*. IEEE, 2010, pp. 1657–1662.

[23] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[24] A. Patel, "Introduction to a*," http://www.redblobgames.com/pathfinding/a-star/introduction.html, [Online; accessed 18-February-2016].

[25] J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue, "Motion planning for humanoid robots," in *Robotics Research. The Eleventh International Symposium*. Springer, 2005, pp. 365–374.

[26] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 4, pp. 566–580, 1996.

[27] S. M. LaValle, "Rapidly-exploring random trees a ew tool for path planning," 1998.

[28] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[29] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[30] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The international journal of robotics research*, vol. 5, no. 1, pp. 90–98, 1986.

[31] Y. Koren and J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*. IEEE, 1991, pp. 1398–1404.

[32] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer*, vol. 22, no. 6, pp. 46–57, 1989.

[33] M. Herman, "Fast, three-dimensional, collision-free motion planning," in *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*, vol. 3. IEEE, 1986, pp. 1056–1063.

[34] J. Barraquand and J.-C. Latombe, "Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles," *Algorithmica*, vol. 10, no. 2-4, pp. 121–155, 1993.

[35] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," DTIC Document, Tech. Rep., 1992.

[36] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann *et al.*, "Stanley: The robot that won the darpa grand challenge," *Journal of field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.

[37] P. Sahay, "Robocart: Autonomous ground vehicle - electromechanical foundations design," https://www.wpi.edu/Pubs/E-project/Available/E-project-043015-034638/unrestricted/Final_Report.pdf, Worcester Polytechnic Institute, Tech. Rep., 2015.

[38] E. A. Miller, "Robocart - system design for the first-generation autonomous golf cart," https://www.wpi.edu/Pubs/E-project/Available/E-project-032615-123118/unrestricted/Robocart-EMIller.pdf, Worcester Polytechnic Institute, Tech. Rep., 2015.

[39] G. Isko, "Robocart - system framework for machine vision component of an autonomous vehicle," https://www.wpi.edu/Pubs/E-project/Available/E-project-020716-200415/unrestricted/FINAL_SystemFrameworkforMachineVisionRobocart_GabeIsko_1-28-2015.pdf, Worcester Polytechnic Institute, Tech. Rep., 2015.

[40] R. A. Newcombe and A. J. Davison, "Live dense reconstruction with a single moving camera," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE, 2010, pp. 1498–1505.

[41] J. Michels, A. Saxena, and A. Y. Ng, "High speed obstacle avoidance using monocular vision and reinforcement learning," in *Proceedings of the 22nd international conference on Machine learning*. ACM, 2005, pp. 593–600.

[42] A. E. Conrady, "Decentred lens-systems," *Monthly notices of the royal astronomical society*, vol. 79, no. 5, pp. 384–390, 1919.

[43] D. C. Brown, "Decentering distortion of lenses," *Photometric Engineering*, vol. 32, no. 3, pp. 444–462, 1966.

[44] Z. Zhang, "A flexible new technique for camera calibration," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 11, pp. 1330–1334, 2000.

[45] A. Fusiello, E. Trucco, and A. Verri, "A compact algorithm for rectification of stereo pairs," *Machine Vision and Applications*, vol. 12, no. 1, pp. 16–22, 2000.

[46] M. Menze and A. Geiger, "Object scene flow for autonomous vehicles," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[47] K. Konolige, "Small vision systems: Hardware and implementation," in *Robotics Research*. Springer, 1998, pp. 203–212.

[48] H. Hirschmüller, "Stereo processing by semiglobal matching and mutual information," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 2, pp. 328–341, 2008.

[49] S. Martull, M. Peris, and K. Fukui, "Realistic cg stereo image dataset with ground truth disparity maps," in *ICPR workshop TrakMark2012*, vol. 111, no. 430, 2012, pp. 117–118.

[50] M. Peris, A. Maki, S. Martull, Y. Ohkawa, and K. Fukui, "Towards a simulation driven stereo vision system," in *Pattern Recognition (ICPR), 2012 21st International Conference on*. IEEE, 2012, pp. 1038–1042.

[51] R. Cellan-Jones, "A 15 pound computer to inspire young programmers," http://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2011/05/a_15_computer_to_inspire_young.html, [Online; accessed 07-March-2016].

[52] "Raspberry 2 model b," https://www.raspberrypi.org/products/raspberry-pi-2-model-b/, [Online; accessed 07-March-2016].

[53] "Camera module - raspberry pi," https://www.raspberrypi.org/products/camera-module/, [Online; accessed 07-March-2016].

[54] "Raspistill," https://www.raspberrypi.org/documentation/usage/camera/raspicam/raspistill.md, [Online; accessed 10-March-2016].

[55] "Raspivid," https://www.raspberrypi.org/documentation/usage/camera/raspicam/raspivid.md, [Online; accessed 10-March-2016].

[56] G. Fiedler, "Udp vs. tcp," http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/, [Online; accessed 07-March-2016].

[57] J. C. Eidson, *Measurement, control, and communication using IEEE 1588*. Springer Science & Business Media, 2006.

[58] "The linux ptp project," http://linuxptp.sourceforge.net/, [Online; accessed 09-March-2016].

[59] *3-Axis Digital Compass IC - HMC5883L*, Honeywell, 2 2013, rev. E.

[60] "Configuring i2c," https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c, [Online; accessed 08-March-2016].

[61] A. Chulliat, S. Macmillan, P. Alken, C. Beggan, M. Nair, B. Hamilton, A. Woods, V. Ridley, S. Maus, and A. Thomson, "The us/uk world magnetic model for 2015-2020," 2015.

[62] "World magnetic model," http://www.ngdc.noaa.gov/geomag/WMM/, [Online; accessed 07-March-2016].

[63] "Magnetic field calculators," http://www.ngdc.noaa.gov/geomag-web/\#declination, [Online; accessed 10-March-2016].

[64] "Bu-353 support," http://usglobalsat.com/s-122-bu-353-support.aspx, [Online; accessed 10-March-2016].

[65] "Gpsd," http://www.catb.org/gpsd/, [Online; accessed 10-March-2016].

[66] *DS1803 - Addressable Dual Digital Potentiometer*, Dallas Semiconductor.

[67] "Arduino uno," https://www.arduino.cc/en/Main/ArduinoBoardUno, [Online; accessed 07-March-2016].

[68] "Arduino - wire," https://www.arduino.cc/en/Reference/Wire, [Online; accessed 07-March-2016].

[69] "Sabertooth 2x60 user's guide," https://www.dimensionengineering.com/datasheets/Sabertooth2x60.pdf, [Online; accessed 10-April-2016].

[70] "Intel core i7-4770k processor (8m cache, up to 3.90 ghz)," http://ark.intel.com/products/75123/Intel-Core-i7-4770K-Processor-8M-Cache-up-to-3_90-GHz, [Online; accessed 10-March-2016].

[71] "Geforce gtx 760 gtx graphics card," http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-760, [Online; accessed 10-March-2016].

[72] "Ubuntu 14.04.4 lts (trusty tahr)," http://releases.ubuntu.com/14.04/, [Online; accessed 10-March-2016].

[73] "Opencv," http://opencv.org/, [Online; accessed 10-March-2016].

[74] "Dd-wrt," http://www.dd-wrt.com/site/index, [Online; accessed 10-March-2016].

[75] Y. Li and Y. Ruichek, "Occupancy grid mapping in urban environments from a moving on-board stereo-vision system," *Sensors*, vol. 14, no. 6, pp. 10 454–10 478, 2014.

[76] M. Perrollaz, J.-D. Yoder, A. Spalanzani, and C. Laugier, "Using the disparity space to compute occupancy grids from stereo-vision," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on.* IEEE, 2010, pp. 2721–2726.

[77] B. Lee and J. Sanders, "Log-odds," https://dl.dropboxusercontent.com/u/34547557/log-probability.pdf, 2011, [Online; accessed 26-January-2016].

[78] S. Kolski, D. Ferguson, M. Bellino, and R. Siegwart, "Autonomous driving in structured and unstructured environments," in *Intelligent Vehicles Symposium, 2006 IEEE*. IEEE, 2006, pp. 558–563.

[79] M. Sugiyama, Y. Kawano, M. Niizuma, M. Takagaki, M. Tomizawa, and S. Degawa, "Navigation system for an autonomous vehicle with hierarchical map and planner," in *Intelligent Vehicles' 94 Symposium, Proceedings of the*. IEEE, 1994, pp. 50–55.

[80] "Transformation between (x,y) and (longitude, latitude)," http://mathforum.org/library/drmath/view/51833.html, [Online; accessed 11-March-2016].

[81] "Threading building blocks," https://www.threadingbuildingblocks.org/, [Online; accessed 16-March-2016].

[82] "Google maps," https://www.google.com/maps, [Online; accessed 18-March-2016].

[83] Y. Cheng, M. Maimone, and L. Matthies, "Visual odometry on the mars exploration rovers," in *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, vol. 1. IEEE, 2005, pp. 903–910.

[84] "Visual odmetry from scratch - a tutorial for beginners," http://avisingh599.github.io/vision/visual-odometry-full/, [Online; accessed 04-April-2016].

[85] A. Howard, "Real-time stereo visual odometry for autonomous ground vehicles," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 3946–3952.

[86] J. Shi and C. Tomasi, "Good features to track," in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*. IEEE, 1994, pp. 593–600.

[87] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[88] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer vision–ECCV 2006*. Springer, 2006, pp. 404–417.

[89] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision–ECCV 2006*. Springer, 2006, pp. 430–443.

[90] B. D. Lucas, T. Kanade *et al.*, "An iterative image registration technique with an application to stereo vision." in *IJCAI*, vol. 81, 1981, pp. 674–679.

[91] C. Tomasi and T. Kanade, *Detection and tracking of point features*. School of Computer Science, Carnegie Mellon Univ. Pittsburgh, 1991.

[92] K. Levenberg, "A method for the solution of certain non–linear problems in least squares," 1944.

[93] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[94] M. Lourakis, "levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++," [web page] `http://www.ics.forth.gr/~lourakis/levmar/`, Jul. 2004, [Accessed on 31 Jan. 2005.].

[95] "Ackerman steering principle," http://www.rctek.com/technical/handling/ackerman_steering_principle.html, [Online; accesed 22-March-2016].

[96] J. M. Snider, "Automatic steering methods for autonomous automobile path tracking," *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08*, 2009.

[97] E. Arnold, "How fast do golf carts go (and how to make them faster)?" http://www.golfcartsmidwest.com/index.php?option=com_content&view=article&id = 68 : how-fast-do-golf-carts-go-and-how-to-make-them-faster & catid = 18 : golf-cart-faqs&Itemid=69, [Online; accessed 12-April-2016].

[98] "Minoru 3d webcam," http://www.minoru3d.com/, [Online; accessed 13-April-2016].