Universität Heidelberg
Zentrales Institut für Technische Informatik
Lehrstuhl Automation

Bachelor-Arbeit

Geometric Feature Extraction for Indoor Navigation

Name:	Charles Barbret
Matrikelnummer:	3443570
Betreuer:	Prof. Dr. sc. techn. Essameddin Badreddin
Datum der Abgabe:	11. April 2021

Abstract

The intent is to create a system that can detect Walls, Doors and Corners. We want to be able to detect these features to be able to add a semantic aspect to the wheelchairs driving ability. Concretely, this means we wish to eventually be able to tell the wheelchair to go down the hallway, turn left at the intersection and go in the second door on the right and have the wheelchair know exactly what we mean and execute this.

Contents

List of Symbols

α	angle
P_n	nth Point made up of the tuple (p_{nx}, p_{ny}). The reference frame has the scanner at (0,0). Along positive y axis is the direction the scanner is facing.
d	distance between two Points
r	radius of a circle
scanner	a laser scanner, optionally attached to a device that can move

1 Introduction

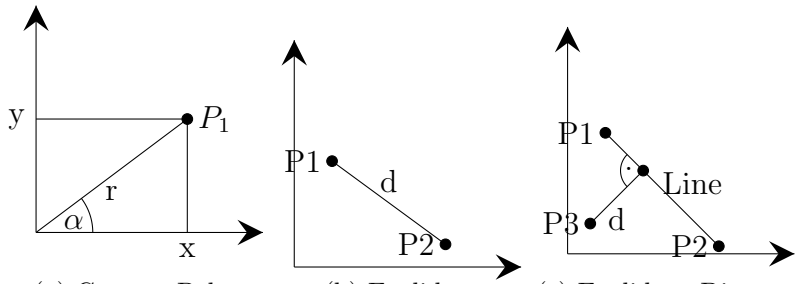
We want to classify navigational situations by extracting contextual features. 854

1.1 Motivation

This work aims to set the foundation of assisted indoor navigation for electric wheelchair users. It does this by scanning the environment with a laser scanner and converting the scan points into semantic terms that a human can work with. These semantic terms include a *Wall*, a *Door* and a *Corner*. By having semantic terms be something a computer can understand it is much easier for a human to tell the machine. With these defined, future works will be able to take these terms and provide end user friendly navigation options. To include the option to tell the wheelchair where to go and have it understand that, as if speaking to a human. While the thesis aims to aid electric wheelchair navigation any example and clarification of situation will use the term Robot, as nothing mentioned in this requires an electric wheelchair.

1.2 Basic Formulas

In this chapter we introduce formulas used throughout this work. These formulas should provide the reader with the tools to fully understand and be able to work with this work.



(a) Convert Polar coordinates to Cartesian coordinates

(b) Euclidean Distance between two Points

(c) Euclidean Distance between a Point and a line

Figure 1.1: Basics set 1

The first formula is that of converting between polar and Cartesian coordinates. The reason we want to do this is that a laser scanner returns a set of distances. Each index of the list is the angle and the value is the distance, so we have α and r. Figure 1.1a shows an example of how the angle and distance can represent a point in a coordinate

system. By using the formula 1.1 we can extract the X and Y coordinates, which can be used easier to apply various algorithms to.

$$\begin{aligned} x &= r * cos(\alpha) \\ y &= r * sin(\alpha) \end{aligned} \tag{1.1}$$

Figure 1.1b shows an example of the euclidean distance between two points. We determine this distance with the equation 1.2. An example for where we use this distance function is in the Breakpoint Detection, which is covered in Chapter 2.

$$d = \sqrt{(p_{2x} - p_{1x})^2 + (p_{2y} - p_{1y})^2} \tag{1.2}$$

Figure 1.1c offers a visualization for the formula 1.3. The idea is that we want to find the shortest distance from a point to a Wall. The shortest distance will be perpendicular to the Wall.

$$d = \frac{|(p_{2y} - p_{1y}) * p_{3x} - (p_{2x} - p_{1x}) * p_{3y} + p_{2x} * p_{1y} - p_{2y} * p_{1x}|}{\sqrt{(p_{2y} - p_{1y})^2 + (p_{2x} - p_{1x})^2}} \tag{1.3}$$

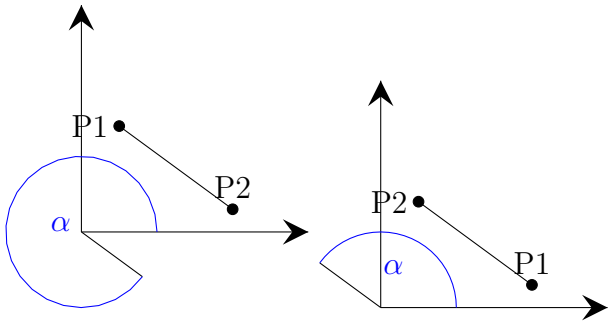


Figure 1.2: Angle between two Point (aka Angle of a line)

The formula 1.4 uses the two parameter arctan, to determine the angle between two points. The normal atan2 function determines the angle between 0 and a single point. To compensate for this we translate one point to the origin and the second relative to that. Figure 1.2 shows this process in action.

$$\alpha = atan2(p_{2y} - p_{1y}, p_{2x} - p_{1x}) \tag{1.4}$$

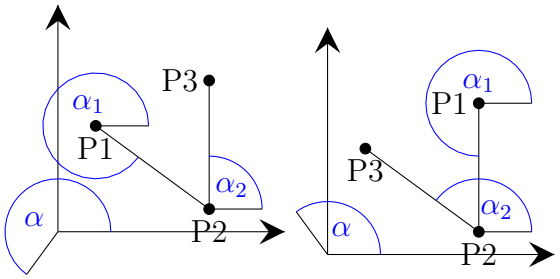


Figure 1.3: Angle between three Points (aka Angle between two lines)

The last formula involves applying equation 1.4 twice, once with $\alpha_1 = (P_2, P_1)$ and once with $\alpha_2 = (P_3, P_2)$. Finally we subtract $\alpha = \alpha_1 - \alpha_2$ and receive the angle between the two lines, as shown in figure 1.3. Once again the order of points matters.

2 State of the art

2.1 Line Extraction in 2D Range Images for Mobile Robotics overview

To get to the point of determining what a wall, door, corner and corridor is, the work of BORGES and ALDON 2004 is used. The idea here is to find certain points, called rupture and break points, which are explained in 2.1.1 and 2.1.2. We then use these to group segments of connected wall, that we then split at the corners. This provides us with the walls.

2.1.1 Rupture Detection

The idea of the Rupture Detection is quite simple. If there are Points of the laser scan that are too far from the Robot we know there is a discontinuity in our surrounding area. This can occur for example, when looking down a long hallway or when looking through a doorway. If this occurs we want to ensure that the system recognizes that any object it is looking at starts or ends with the rupture points. Examples of Rupture Points can be seen in figure 2.1

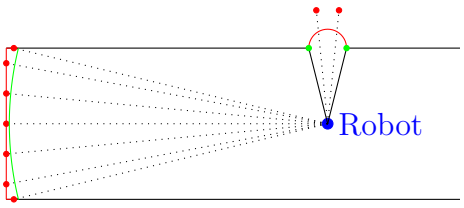


Figure 2.1: Rupture Points shown in red. Detectable points shown in green

2.1.2 Breakpoint Detection

The idea of the Breakpoint Detection closely resembles that of the Rupture Detection. While looking at each Point, the distances between two are measured. This length is now compared to a D_max, which is determined by equation 2.1. Here P_{n-1} is the previous point, $\Delta\phi$ is the angle increment, λ corresponds to the worst case of incidence angle of the laser scan ray with respect to a line for which the scan points are still reliable and σ is <the max margin of error?>. The distance of D_max is exceeded for example, when we are dealing with two seperate walls, or when an object, such as a pillar, is obstructing a wall, but is not directly connected to that wall. Examples of break points are shown in 2.2

$$D\_max = P_{n-1} * \frac{sin(\Delta\phi)}{sin(\lambda - \Delta\phi)} + (3 * \sigma) \tag{2.1}$$

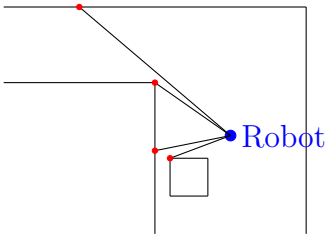


Figure 2.2: Breakpoints shown in red

2.1.3 Line Extraction

After having found the rupture and breakpoints we can group the points into continuous wall segments. These segments are not necessarily individual walls as the breakpoint detection is not good at detecting corners. The first step of this is to group the first Points together until a Point either has a Rupture or Breakpoint associated with it. Once this happens this segment will be considered a continuous segment and sent into the Iterative Endpoint Fit, which is explained next. Now that the first segment is taken care of we continue where we left off, working our way in the same way through all the available Points to group the Points into connected segments that tend to be Walls.

2.1.4 Iterative Endpoint Fit

The Iterative Endpoint Fit process takes one Wall segment and breaks is up at the Corners, should these be in the segment. The way it does this is by taking the first and last Point and connecting these with a line. Now it goes through all Points in between the first and last Point and checks which Point is the farthest away to this line. Should the Point that this process finds exceed a certain length we split the segment at this Point and rerun the process on both segments, until we end up with only straight lines. Figure 2.3 shows a step by step process of this process.

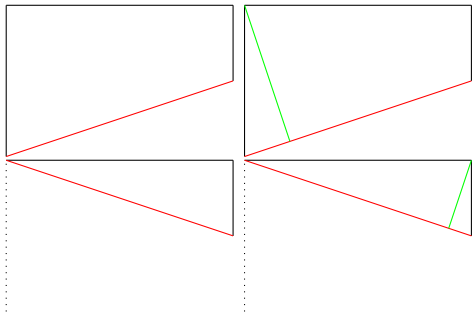


Figure 2.3: Iterative Endpoint Fit process

2.2 Procrustes

The Procrutes algorithm aims to take two consecutive scans and match them to each other. It tries to find a translation and rotation matrix that can be applied to one scan to set all Point in the reference frame of the second scan.

The algorithm does this by first translating the centers of both scan sets to the origin (0,0). From here both sets are normalized. Finally the first set is rotated to best fit the second set. If the difference is within a threshold we have our rotation and translation matrices.

3 Feature Extraction

3.1 Definitions

The following terms will be defined in this part and referenced throughout the rest of the work.

- In this work a *Point* P refers to a simple tuple of two floating point values. With $P = (p_x, p_y)$
- A *Wall*, shown in Figure 3.1, consists of a tuple of two Points. With $W = (P_s, P_e)$. A thing to note here, the Wall (A, B) is not the same as the Wall (B, A). The order in which each Point is added determines the direction of the Wall. The reason for the direction is to be able to determine at a glance which side

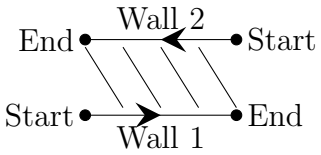


Figure 3.1: Two walls with their direction indicated by an arrow. The shaded area is navigable by the scanner

- Similar to a Wall, a *Door* too is made up of a tuple of two Points as shown in Figure 3.2. A Door has the added restriction that the distance be between 0,5 and 1 Meters.

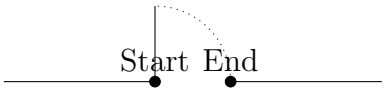


Figure 3.2: Here we have 2 Walls separated by an open Door

- A *Corner* is made up of a tuple containing two Walls, which share a Point, and one integer. $C = (W_1, W_2, i)$ The Point both Walls share is the corner in question. It is always the end Point of the first Wall and the start Point of the second Wall. The integer keeps track of the Corner type. Possible Corner types are inner Corner, outer Corner and potential Corner. The corner type impacts how the Robot can approach the Corner.
- Corridor := {[List[Wall] a, List[Wall] b]}

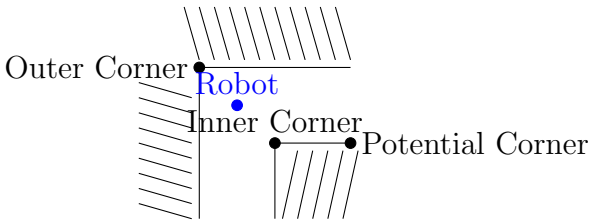


Figure 3.3: Corner Type Examples

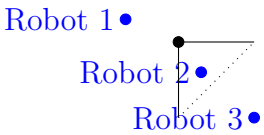


Figure 3.4: Possible Robot Relative Locations

3.2 Corner Detection

3.2.1 Corner Type Detection

When we look at a Corner there are two possibilities for the orientation, an inner Corner and an outer Corner

As humans we can identify the type by standing in front of the Corner and looking at it. But when we are dealing with machines it is not always that simple. As mentioned above a Corner is defined by the two connecting Walls. Unfortunately to determine whether or not the Corner is an inner Corner we cannot calculate the angle between the two Walls, as we always will get the smaller of the two angles. When looking at a Corner one can make the observation, that the construct is simply a triangle with one of its connecting lines missing. With this in mind we can examine the relative position of the robot to this triangle. When considering that the robot needs to see both Walls to properly determine that it is looking at a Corner we end up with 3 distinct possibilities:

1. We have an inner corner
2. We have an outer corner where the robot is within the triangle
3. We have an outer corner where the robot is outside of the triangle

With this in mind we now need to figure out how to convey this to the robot. One way we can do this is by measuring distances. To fully determine if we are dealing with an inner or outer Corner we will need three distances.

- the distance from the robot to the corner
- the shortest distance from the robot to the missing leg of the triangle
- the shortest distance from the corner to the missing leg of the triangle

Once we have these three distances we can start comparing these against each other to determine, whether or not the corner is an inner or an outer. With three variables there are three comparisons to be made:

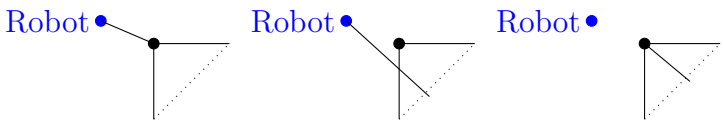
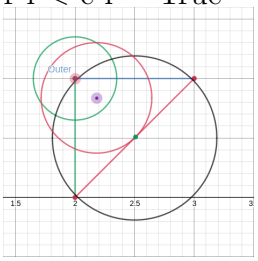


Figure 3.5: All relevant distances

- is the distance from robot to corner less than the distance from robot to the missing leg ($r - c < r - l$)
- is the distance from robot to corner less than the distance from corner to the missing leg ($r - c < c - l$)
- is the distance from the robot to the missing leg less than the distance from the corner to the missing leg ($r - l < c - l$)

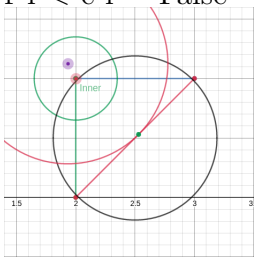
If we go through each possible combination and manually check if we are dealing with an inner or an outer corner we find the following:

r-c < r-l = True
r-c < c-l = True
r-l < c-l = True



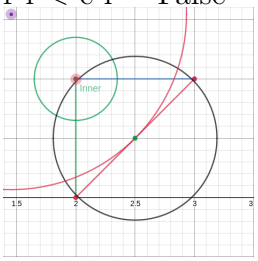
(a) Outer Corner

r-c < r-l = True
r-c < c-l = True
r-l < c-l = False



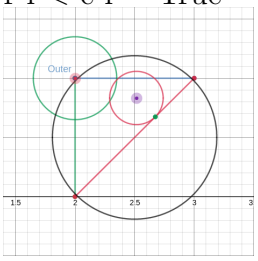
(b) Inner Corner

r-c < r-l = True
r-c < c-l = False
r-l < c-l = False



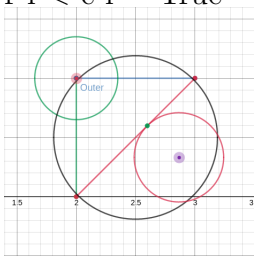
(c) Inner Corner

r-c < r-l = False
r-c < c-l = True
r-l < c-l = True



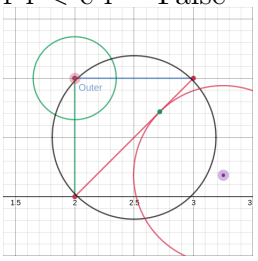
(d) Outer Corner

r-c < r-l = False
r-c < c-l = False
r-l < c-l = True



(e) Outer Corner

r-c < r-l = False
r-c < c-l = False
r-l < c-l = False



(f) Outer Corner

Figure 3.6: Corner Labels

As you may have noticed, the cases True, False, True and False, True, False did not appear. This is because we are dealing with a 2D plane. If we were on a cylinder or other non flat plane there might be three Points where r-c < r-l < c-l > r-c or r-c > r-l > c-l < r-c, however the space we live in does not have Points which could satisfy these conditions.

Having determined under which conditions we have an inner corner and when not we can translate this into a Karnaugh map. The result of which would be:

	b		$\overline{b}$	
	$\overline{c}$	c		$\overline{c}$
a	1	0	0	1
$\overline{a}$	0	0	0	0

Table 3.1: Karnaugh map

3.3 Corridor Detection

3.4 Implementation Overview

3.4.1 Pseudo Code

Data: scan = laserscan data
rupture_points = Detect_Rupture_Points(scan);
break_points = Detect_Break_Points(rupture_points);
walls = Extract_Lines(break_points);
corners = Detect_Corners(walls);
corridors = Detect_Corridor(corners)

Algorithm 1: Order of Functions

4 Experimental Results

4.1 Measurements

4.2 Office Hallway

TODO describe the office hallway in which the scans were taken. Maybe add some pictures

5 Conclusion

5.1 Outlook

5.2 Future Work

5.2.1 Procrustes

The Procrustes algorithm could be a start for looking into recognizing corners. This could be useful to determine relative position at each time step. This in theory can be more precise than just using the odometry information, as we base the position on features in the environment.

References

Appendix