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

"Doors and Corners, this is where they get you." -Josephus Miller (The Expanse)

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 n-th Point made up of the tuple (p_{nx}, p_{ny}). The reference frame has the scanner at (0,0). Along positive x 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.

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 user friendly navigation options. To include the option of telling 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
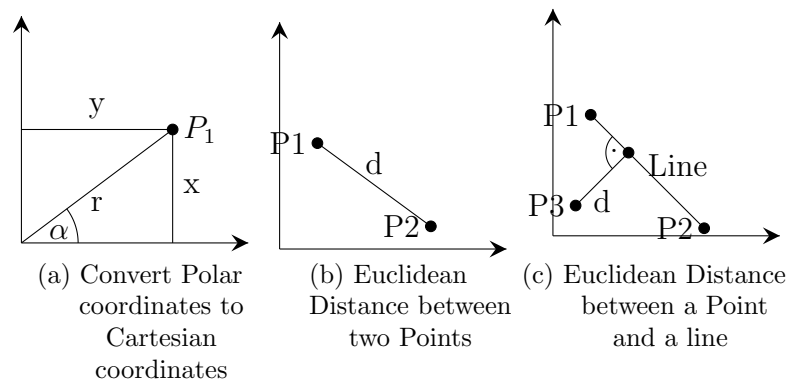
(c) Euclidean Distance between a Point and a line

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.

$$x = r \cdot cos(\alpha)$$
$$y = r \cdot sin(\alpha)$$

(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.

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

(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.

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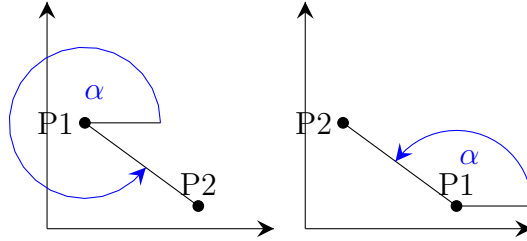
(1.3)

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.

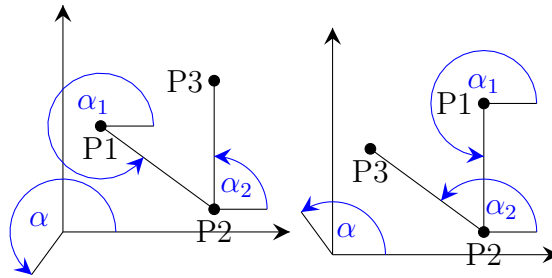$$\alpha = atan2(p_{2y} - p_{1y}, p_{2x} - p_{1x})$$

(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

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 Sections 2.1.1 and 2.1.2. These are then used 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 exceed the laser scan maximum or a given threshold d_{max} we know there to be 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
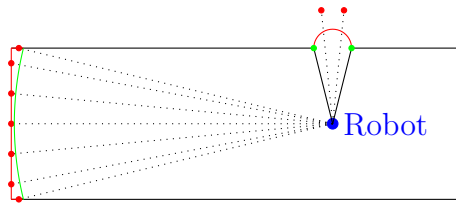

Figure 2.1: Rupture Points shown in red. Detectable points shown in green

A Rupture Point thus is determined if the distance of point n to the laser scanner is less than d_{max}

$$d_n > d_{max} \tag{2.1}$$

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 (2.2). Here P_{n-1} is the previously detected point, $\Delta\phi$ is the angle between two laser beams, 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 tolerance. 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

$$D\_max = P_{n-1} \cdot \frac{sin(\Delta\phi)}{sin(\lambda - \Delta\phi)} + (3 \cdot \sigma) \tag{2.2}$$
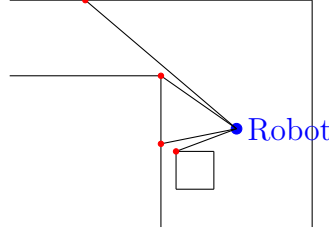

Figure 2.2: Breakpoints shown in red

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. The process starts with one continuous wall segment, indicated in black. Step 1 then creates the line between the first and last points indicated by the line in red. Step 2 shows the max distance with the green line. The dotted wall in Step 3 was split from the rest as this wall does not contain any more corners. Step 3 also creates a line between the new start and the end, again indicated by the line in red. Step 4 shows the max distance with the line in green. After step 4 the process sees there are no more corners and returns all three walls.
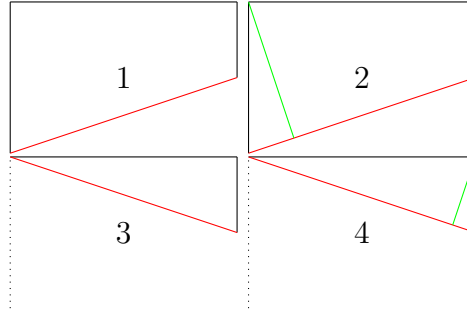
Figure 2.3: Iterative Endpoint Fit process

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

As described in the last chapter [insert paper here] provides us with a method for extracting walls. From here we will use the walls to extract the doors, corners and corridors by comparing walls against each other. The more useful features we can extract, the easier the navigation process will be in the future.

3.1 Definitions

Before the extraction process is explained, some terms need to be defined first.

- 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 direction of the wall indicates the side of the wall the laser scanner is facing.
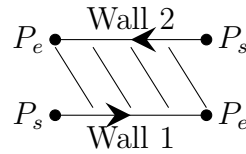

Figure 3.1: Two walls with their direction indicated by an arrow. The shaded area is navigable by the scanner

- While under the hood a Door may share the structure of a Wall, that being $D = (P_s, P_e)$, the way it is detected is by finding a gap between two walls. This gap will have at least one rupture point associated with it. Here the assumption is made, that a doorway exists only when the door is open. Because of this assumption and the check for a gap we only need one rupture point, as the door might still be attached to one side. The gap between the two points is approximately 1 meter.
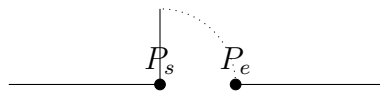

Figure 3.2: Here we have 2 Walls separated by an open Door

- 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,
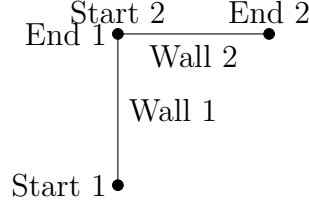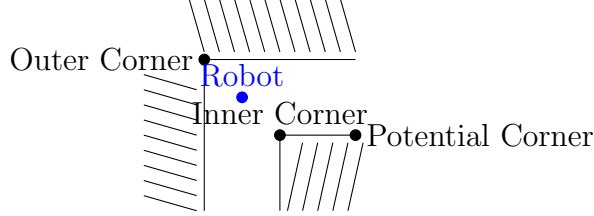
Figure 3.3: Caption

Figure 3.4: Corner Type Examples

outer Corner and potential Corner. The corner type impacts how the Robot can approach the Corner.

- Finally a *Corridor* is a single Point P_c. This point is located between a corner and a wall, which indicates the entrance or exit into a given corridor.

Due to how the midpoints are detected, the same object humans would describe as a corridor will contain multiple different corridor entrances. This is due to the fact that a corridor can have open doors or objects in its way that create more corners. However by having more of these points in the same corridor, a path finding system can have an easier time navigating past objects, such as pillars. Figure 3.5 shows in what instances the probing walls get used and when they yield a corridor midpoint. The 6 red lines indicate probing walls that do not yield a corridor midpoint. The line marked with 1 does not provide a corridor midpoint as it is too close to the laser scanner. 2, 3 and 6 do not intersect with a wall. The wall in 4 is too close to the corner from where the probing wall originates. 5 is too far from the laser scanner and 6 both does not intersect with a wall and is too far from the laser scanner. All other probing walls yield a midpoint, even the door.
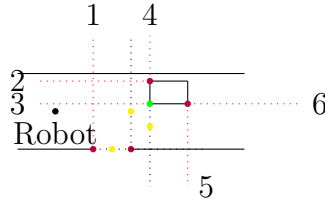
Figure 3.5: The red dotted lines indicate that the search for another wall yielded for one reason or another no match

3.2 Corner Detection

Now that a corner has been defined, the next step is detecting them. The approach used in the code is to simply analyze each pair of walls and check if a start point overlaps an end point. If such a point exists we know this point is a corner. An improvement on this system could involve analyzing the distances between start and endpoints. If the two are within a certain distance the corner point can be set in an area that makes sense between the two walls. This could be useful in the event that the laser scanner did not perfectly capture the corner and the two resulting points are too far from each other to be grouped by the program. Now that the corner has been identified we need to determine on which side of the corner the laser scanner can traverse. This information is not directly obvious just based on having two walls, as the corner in question could either be facing to the laser scanner and be an inner corner or be facing away from the laser scanner and be an outer corner. To distinguish between the two the Corner Type Detection has been implemented.

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 and an integer, representing the corner type. When looking at a Corner one can make the observation, that the construct is simply a triangle with the hypotenuse 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

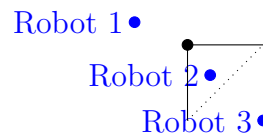3. We have an outer corner where the robot is outside of the triangle

Figure 3.6: Possible Robot Relative Locations

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 (r-c)

- the shortest distance from the robot to the missing hypotenuse of the triangle (r-h)

- the shortest distance from the corner to the missing hypotenuse of the triangle (c-h)
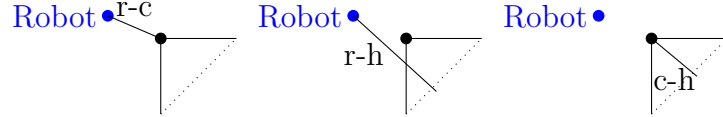

Figure 3.7: All relevant distances

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:

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

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

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

By going through each possible combination and creating a visualization one is left with the results of Figure 3.8.

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-h < c-h > r-c or r-c > r-h > c-h < 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 is shown in Table 3.1 where a refers to $(r - c < r - h)$, b refers to $(r - c < c - h)$ and finally c refers to $(r - h < c - h)$.

Extracting the information provided by the Karnaugh map the Formula (3.1) is left. We can now use this Formula to determine the corner type.

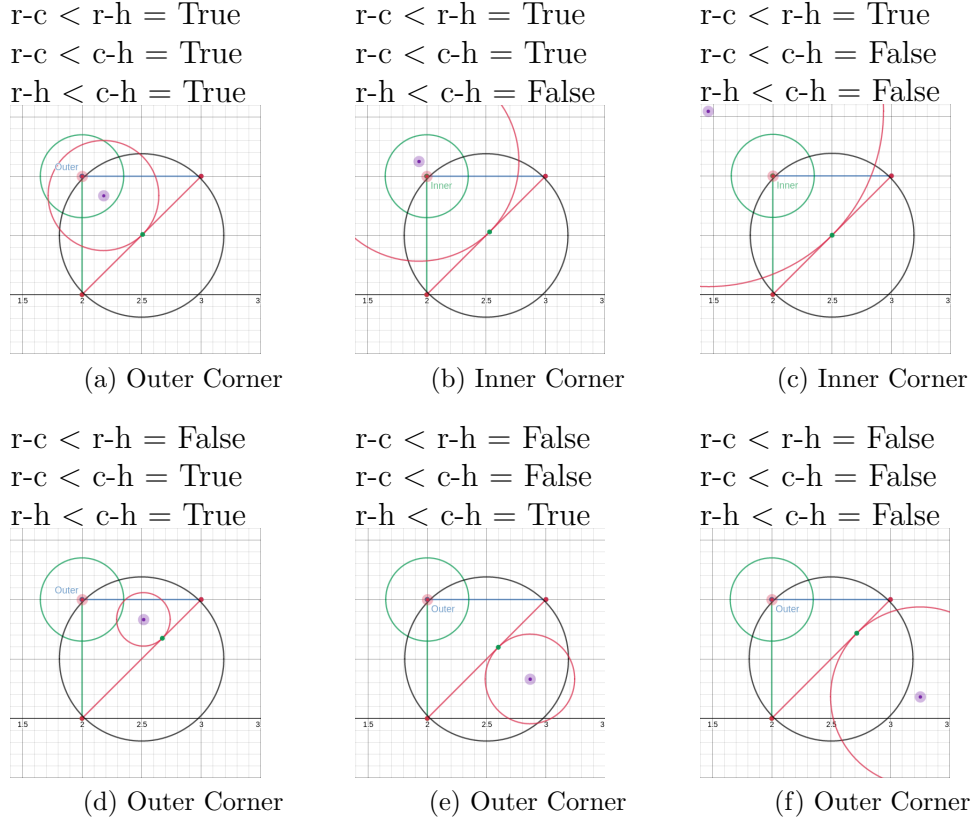$$corner\_type = ((r - c < r - h) \ \& \ !(r - h < c - h)) \tag{3.1}$$

r-c < r-h = True
r-c < c-h = True
r-h < c-h = True

r-c < r-h = True
r-c < c-h = True
r-h < c-h = False

r-c < r-h = True
r-c < c-h = False
r-h < c-h = False

(a) Outer Corner

(b) Inner Corner

(c) Inner Corner

r-c < r-h = False
r-c < c-h = True
r-h < c-h = True

r-c < r-h = False
r-c < c-h = False
r-h < c-h = True

r-c < r-h = False
r-c < c-h = False
r-h < c-h = False

(d) Outer Corner

(e) Outer Corner

(f) Outer Corner

Figure 3.8: Corner Labels

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

Table 3.1: Karnaugh map

3.2.2 Potential Corner

A Potential Corner is defined as the start or end of a wall that contains either a break-point or a rupture point. The reason potential corners are relevant is that there are many situations where a laser scanner will not be able to accurately determine if a wall continues or if the hallway turns off. Thus the assumption is made, that any break and rupture leads to a chance of there being a corner, a potential corner.

3.3 Corridor Detection

To detect the corridor midpoint the corner types will be made use of. Staring from every inner and potential corner two probing walls are created at a 90 and 180 degree offset to the corners walls. These probing walls have a length of 2.5 Meters and search for any intersection with another wall. Should such an intersection exist we can assume

that the midpoint of the corner and intersection point is a location the laser scanner can traverse to.

Looking again at Figure 3.5 the inner and potential corners have probing walls exiting at 90 and 180 degree angles to the walls searching for a wall.

An early version of the corridor detection attempted to create a so called *Meta Wall* as shown in Figure 3.9. One reason the definition ultimately moved away from this sort of a Meta Wall idea, was that comparing each Wall with each other proved more complex than initially assumed.
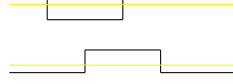

Figure 3.9: Meta Wall

3.4 Implementation Overview

3.4.1 Pseudo Code

forall *scan from laserscan measurements* **do**
    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)
end

Algorithm 1: Overview

[Explain rupture points implementation] [Explain break points implementation] [Explain extract lines implementation] [Explain detect corners implementation] [Explain detect corridor implementation] [these in a similar way as the overview]

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