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	Point n made up of the coordinate 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

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

We want to classify navigational situations by extracting contextual features.

1.1 Motivation

This work aims to set the foundation to improve the ease of indoor navigation of an electric wheelchair. 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. This list of terms includes *Walls*, *Doors*, *Corners*, and *Corridors*. By having semantic terms be something a computer can understand it is much easier for a human to communicate with the machine. With these defined, future works will be able to take these terms and provide user friendly navigation options, that will make navigating a wheelchair as easy as talking to another person. While the thesis aims to aid electric wheelchair navigation as well, from here on the term 'Robot' will be used, as an electric wheelchair is not actually needed.

1.2 Problem Statement

In a world comprised of laser scan points, the question is: Can semantic features such as Doors, Walls, Corners and Corridors be defined?

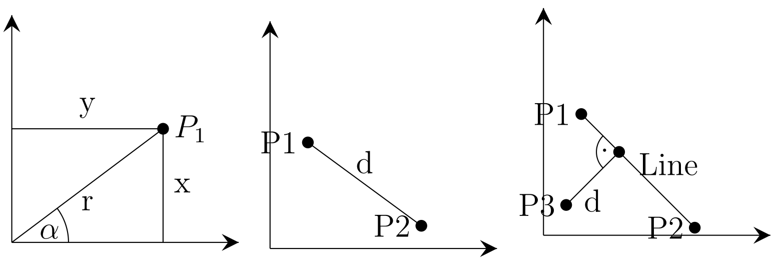
These laser scan points are recorded multiple times a second and are stored in a list. This list is made up of distances. The first entry of the list is defined to have the minimum angle provided by the laser scanner. The angle of the following indices is determined by Formula (1.1) where $\Delta\phi$ is the angle increment.

$$\alpha_{n+1} = \alpha_n + \Delta\phi \tag{1.1}$$

1.3 Basic Formulas

In this chapter the formulas used throughout this thesis are introduced. These formulas should provide the reader with the tools to fully understand this thesis.

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 list of distances. The length



(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

of this list is determined by the angle ranges the Laser scanner is capable of and the angle increment. Where each index of the list relates to the (minimum angle + (angle increment) * index), the value of said index is the distance. This means index i contains the information of α_i and r_i. Figure 1.1a shows an example of how the angle and distance can represent a point in a coordinate system. By using the Formula (1.2) we can extract the x and y coordinates, which can be used easier to apply various algorithms to.

$$\begin{aligned} x &= r \cdot cos(\alpha) \\ y &= r \cdot sin(\alpha) \end{aligned} \tag{1.2}$$

Figure 1.1b shows an example of the Euclidean distance between two points. We determine this distance with the Formula (1.3). An example for where we use this distance function is in the Breakpoint Detection, which is covered in Chapter 2.

$$d(P_1, P_2) = \sqrt{(p_{2x} - p_{1x})^2 + (p_{2y} - p_{1y})^2} \tag{1.3}$$

Figure 1.1c offers a visualization for the Formula (1.4). 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.

$$d(P_1, P_2, P_3) = \frac{|(p_{2y} - p_{1y}) \cdot p_{3x} - (p_{2x} - p_{1x}) \cdot p_{3y} + p_{2x} \cdot p_{1y} - p_{2y} \cdot p_{1x}|}{\sqrt{(p_{2y} - p_{1y})^2 + (p_{2x} - p_{1x})^2}} \tag{1.4}$$

The Formula (1.5) uses the two parameter arctan, to determine the angle between two points. The atan2 function <add reference to math.atan2 from python> determines the angle between 0 and a single point. To compensate for this we translate one point to the origin and the second in relation to that. Figure 1.2 shows this process in action.

$$\alpha(P_1, P_2) = atan2(P_2 - P_1) = atan2(p_{2y} - p_{1y}, p_{2x} - p_{1x}) \tag{1.5}$$

The last formula involves applying Formula (1.5) twice, as shown in Formula (1.6). This is shown in Figure 1.3. Once again the order of points matters.

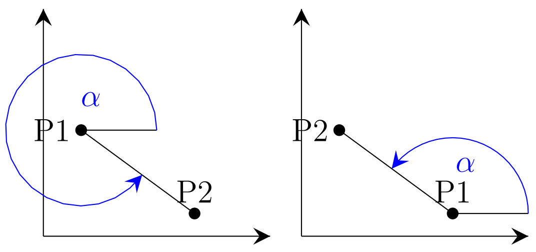


Figure 1.2: Angle between two Points (aka Angle of a line)

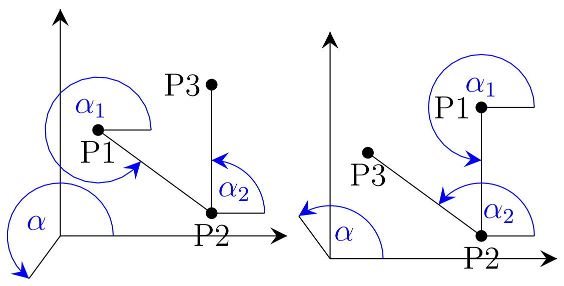


Figure 1.3: Angle α between three Points (aka Angle between two lines)

$$\alpha(P_1, P_2, P_3) = \alpha_1(P_1, P_2) - \alpha_2(P_2, P_3) \tag{1.6}$$

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 used to group segments of connected walls, 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

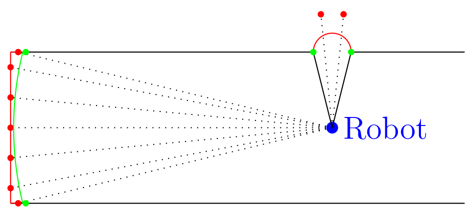


Figure 2.1: Rupture Points shown in red. Detectable points shown in green

A Rupture Point thus is determined if the distance of point n to the laser scanner is greater than $d_{rupture\_max}$ and the flag is assigned to the first available non ruptured points, which in Figure 2.1 are the green points.

$$d_n > d_{rupture\_max} \tag{2.1}$$

2.1.2 Breakpoint Detection

The idea of the Breakpoint Detection closely resembles that of the Rupture Point Detection. Where the Rupture Point detection analyzed the distance to the Laser scanners, the Breakpoint Detection analyzes the distances between two consecutive points P_{n-1} and P_n. A Breakpoint occurs, when the distance between the two points is greater than d_{break_max}.

$$d(P_{n-1}, P_n) > d_{break\_max} \tag{2.2}$$

d_{break_max} is defined in Formula (2.3), where 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.

$$d_{break\_max} = P_{n-1} \cdot \frac{sin(\Delta\phi)}{sin(\lambda - \Delta\phi)} + (3 \cdot \sigma) \tag{2.3}$$

The distance of d_{break_max} is exceeded for example, when we are dealing with two separate 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

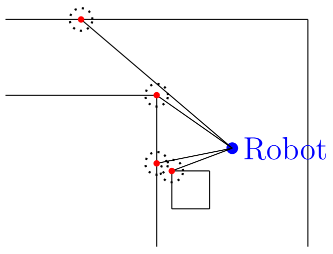


Figure 2.2: Breakpoints are shown in red and d_{break_max} is indicated by the dotted circles around each Point

Should a Breakpoint occur between points P_{n-1} and P_n both need the Breakpoint flag to be set true.

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 Points together until one 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 it 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 every Point between the first and last point is checked for the maximum distance to the 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.

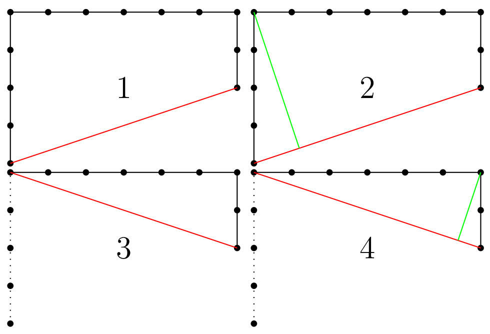


Figure 2.3: Iterative Endpoint Fit process

Figure 2.3 depicts 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.

3 Feature Extraction

As described in the last chapter 2.1 [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 the 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. This side is always to the left of the wall.

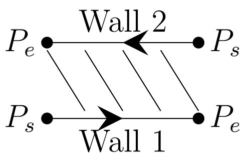


Figure 3.1: Two walls with their direction indicated by an arrow. The shaded area is navigable by the scanner

- 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.

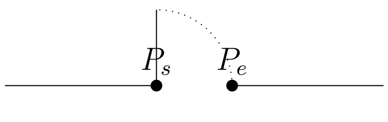


Figure 3.2: Here we have 2 Walls separated by an open Door

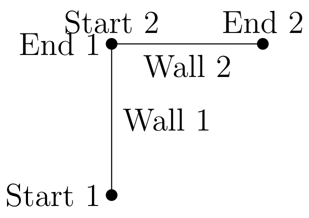


Figure 3.3: A display of what a Corner looks like with its Walls

- 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.

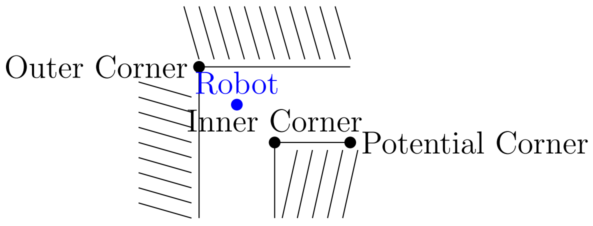


Figure 3.4: Corner Type Examples

- 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. Lines 2, 3 and 6 do not intersect with a wall. The wall in Line 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.

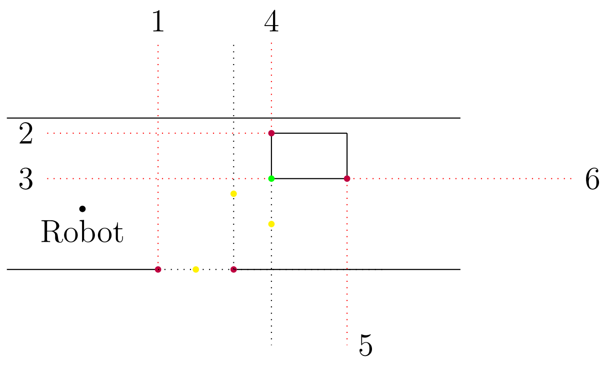


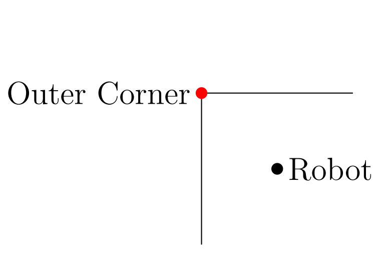
Figure 3.5: The red dotted lines indicate that the search for another wall yielded for one reason or another no match

3.2 Corner Detection

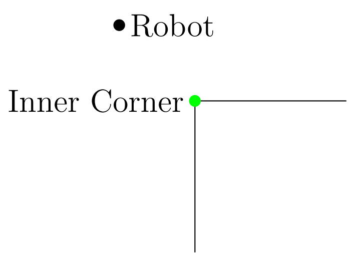
Now that a corner has been defined, the next step is detecting them. Currently this is done by comparing the two Walls. Should there be a pair $P_{nstart} == P_{mend}$ the Walls m and n are combined and the corner point is set as P_{mend} An improvement on this system could involve analyzing the distances 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 Types

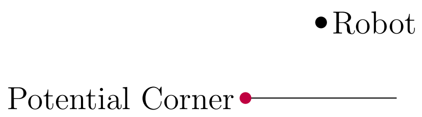
- Outer Corner is assigned 0. This Corner type points away from the laser scanner as shown in Figure 3.6a.
- Inner Corner is assigned 1. This corner type points to the laser scanner as shown in Figure 3.6b
- Potential Corner is assigned 2. This corner type is defined as the start or end of a wall that 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 or rupture leads to a chance of there being a corner, a potential corner.



(a) Outer Corner



(b) Inner Corner



(c) Potential Corner

Figure 3.6: The Corner Types

3.2.2 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

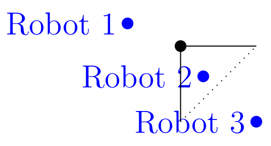


Figure 3.7: Possible Robot Relative Locations

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 (rc)
- the shortest distance from the robot to the missing hypotenuse of the triangle (rh)

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

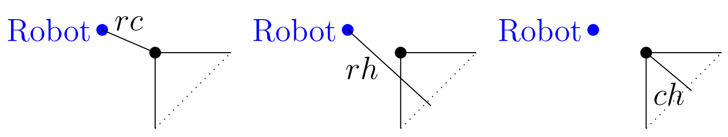


Figure 3.8: All relevant distances

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:

- the distance from robot to corner is less than the distance from robot to the missing hypotenuse ($rc < rh$)
- the distance from robot to corner is less than the distance from corner to the missing hypotenuse ($rc < ch$)
- the distance from the robot to the missing hypotenuse is less than the distance from the corner to the missing hypotenuse ($rh < ch$)

By going through each possible combination and creating a visualization one is left with the results of Figure 3.9.

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 $rc < rh < ch > rc$ or $rc > rh > ch < rc$, 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 or not we can translate this into a Karnaugh map. The result of which is shown in Table 3.1 where a refers to ($rc < rh$), b refers to ($rc < ch$) and finally c refers to ($rh < ch$).

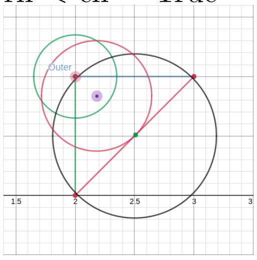
	b		$\overline{b}$	
	$\overline{c}$	c		$\overline{c}$
a	1	0	0	1
$\overline{a}$	0	0	0	0

Table 3.1: Karnaugh map

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. The result of Formula (3.1) is a Boolean. In 3.2.1 an Outer Corner was said to hold the integer value of 0 and an rc < rh = True

rc < ch = True

rh < ch = True

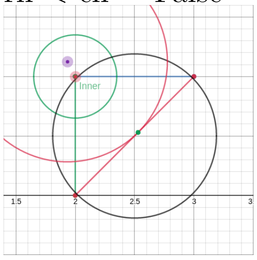


(a) Outer Corner

rc < rh = True

rc < ch = True

rh < ch = False

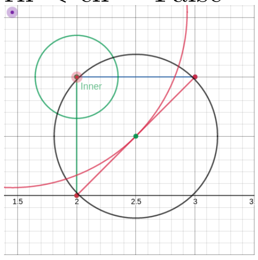


(b) Inner Corner

rc < rh = True

rc < ch = False

rh < ch = False

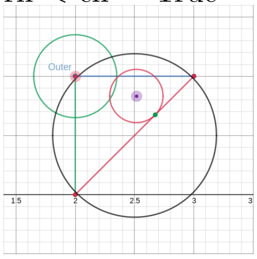


(c) Inner Corner

rc < rh = False

rc < ch = True

rh < ch = True

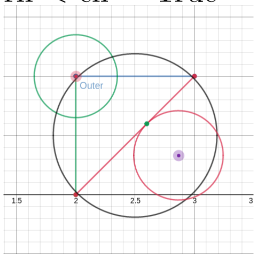


(d) Outer Corner

rc < rh = False

rc < ch = False

rh < ch = True

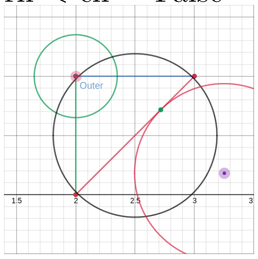


(e) Outer Corner

rc < rh = False

rc < ch = False

rh < ch = False



(f) Outer Corner

Figure 3.9: Corner Labels

Inner Corner to hold the value of 1. So the Boolean is directly transferable to the corner type.

$$corner\_type = ((rc < rh)\ \ \&\ !(rh < ch)) \tag{3.1}$$

3.3 Corridor Detection

To detect the corridor midpoint the corner types will be made use of. Starting 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 2.5m 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.

Meta Wall

An early version of the corridor detection attempted to create a so called *Meta Wall* as shown in Figure 3.10. 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.

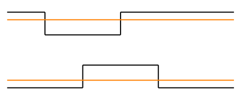


Figure 3.10: Meta Wall

3.4 Implementation Overview

Having explained the theory behind this Thesis, the interesting question becomes how to implement it in a practical manner.

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

Rupture Point Detection Implementation

The way the Rupture Point Detection was implemented was that it started by taking the polar coordinates returned from the laser scan measurements and checking, each individual point if it has a valid length. If this is the case it converts this to Cartesian coordinates using Formula (1.2) and adds a Flag to indicate that this point was not a rupture point. After having converted all points to Cartesian each point is analyzed for a positive rupture flag. Any point that has one of these does not get added for further analysis and the two points before and after it receive the final rupture flag. The final list is then made up of the Point in the Cartesian system, the point in the polar system and the Rupture Flag, which indicates if next to it a rupture occurred.

Breakpoint Detection Implementation

The Breakpoint Detection receives the List of Points and Rupture Flags created in the Rupture Detection. As described in the breakpoint detection the distances between consecutive points is measured and checked against the d_{break_max}. If two Points exceed this distance the breakpoint flag for both is set to true. After having compared all distances this new List is passed on.

Line Extraction Implementation

The Line Extraction now takes the List provided by the Breakpoint Detection

Iterative Endpoint Fit Implementation

The Iterative Endpoint Fit receives a List ”list_of_walls” wherein the separated Walls will be placed, a List of all the Points that were returned in Breakpoint Detection ”breakpoints”, a start index ”start”, and an end index ”end”. From here the distance to the line that is spanned by breakpoints[start], breakpoints[end] to each point between start and end is measured with 1.4. Should the maximum distance of all of these points exceed 6cm this point is likely a corner. This means that the Iterative Endpoint Fit gets called two more times, once where the start remains the same and the end is the corner point and once where the end remains the same and the start is the corner point. Should there be no such point a Wall is created with breakpoints[start] and breakpoints[end] and placed within list_of_walls.

Door Detection Implementation

Corner Detection Implementation

[Explain detect corners implementation]

Corridor Detection Implementation

[Explain detect corridor implementation]

4 Experimental Results

4.1 Measurements

Now that the processes have been described they need to be analyzed.

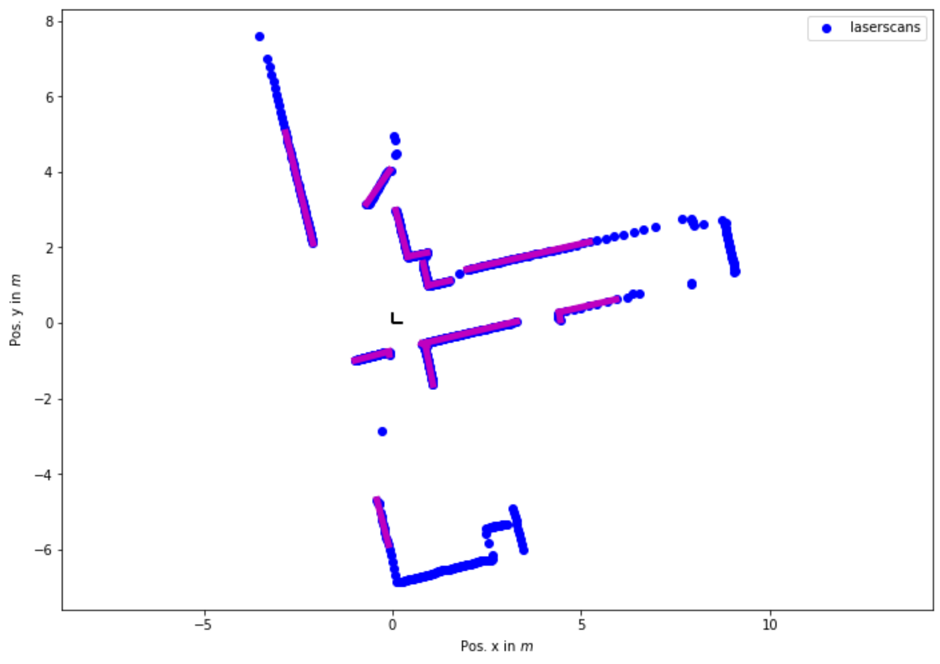


Figure 4.1: Wall Detection 1

Figure 4.1 shows the Wall detection process in action as presented by ¡Paper¿ in Chapter 2.1. The Walls that are within range are detected with little to no problem.

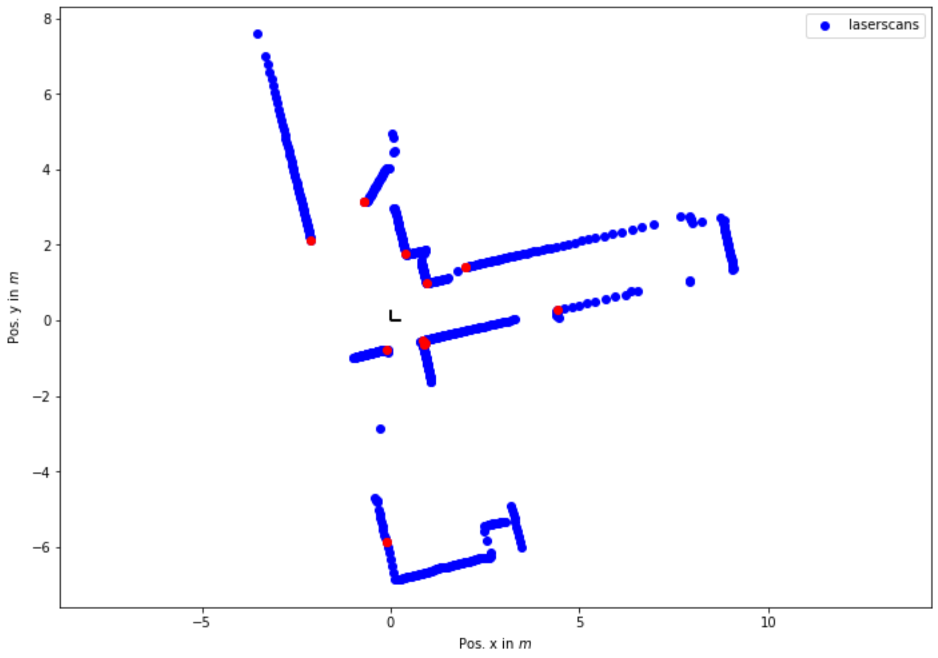


Figure 4.2: Corner Detection 1

Figure 4.2 on the other hand demonstrates some minor inconsistencies. Around (2,-0.5) the detection process has found several corners, despite there only being one. This is likely due to the fact, that the here open door contains gaps around the corner area and thus confuses the detection. Another Corner that has been falsely detected is at (0,-6). Here the wall is a single straight line.

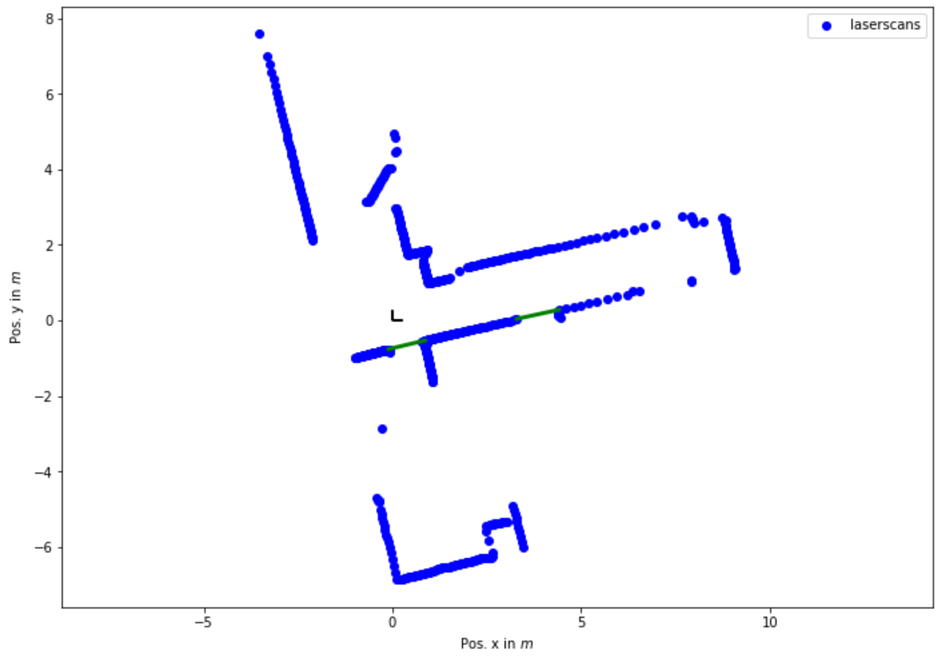


Figure 4.3: Door Detection 1

Figure 4.3 correctly identifies two of the three presently open doors. The door that is not detected is at (1,3). The reason for this door not being detected in this frame is, that the wall behind the open door is obscured, thus the system can not find a wall with a similar angle as the wall on the other side of the door.

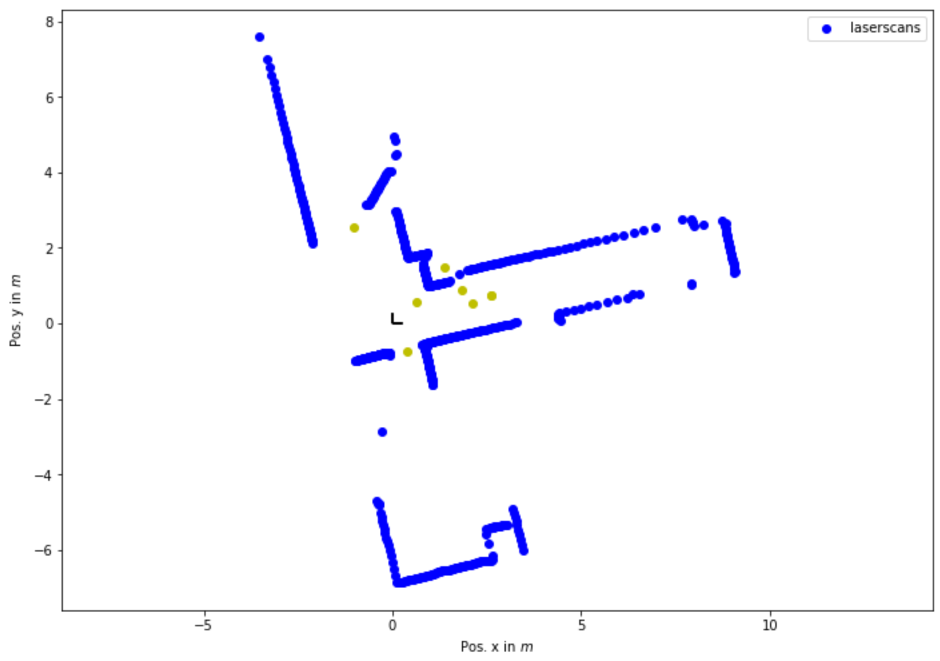


Figure 4.4: Corridor Detection 1

Figure 4.4 shows that the corridor detection can both provide useful points, such as the door and impossible to reach points such as the point behind the pillar.

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