Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Parking lot exploration strategy

Student Scientific Conference

Authors
Ádám Anna Barbara
Kocsány László

Supervisor
Gincsainé Dr. Szádeczky-Kardoss Emese

October 25, 2019

Contents

Összefoglaló

Napjainkban egyre több jármű közlekedik az utakon, ennek következtében egyre nehezebb megfelelő parkolóhelyet találni a lakóhelyünk környékén, bevásárlóközpontokban és különböző parkolóházakban. A parkolóhely keresése időigényes lehet, de az autonóm járművek elterjedése lehetőséget ad egy autonóm parkolási rendszer megalkotására, ami megtakaríthatja ezt az időt.

Jelen dolgozat egy autonóm parkolási rendszert mutat be, amely képes a parkoló bejárásának megtervezésére, miközben a jármű tetejére erősített LiDAR segítégével észleli a jármű számára megfelelő parkolóhelyeket, illetve ezek után a parkolási manővert is képes megtervezni. A parkolási rendszer kimenete szolgáltatja az alapjelet (referencia pályát, referencia beavatkozó jeleket) egy zártkörű szabályozáshoz.

A fellelhető szakirodalom a hatékony parkolóhely keresésére elsősorban parkolókban elhelyezett szenzorokat alkalmaz, ezen módszerek azonban csak az erre felkészített parkolókban alkalmazhatóak. Jelen dolgozat egy olyan általánosan alkalmazható megoldást mutat be, mely a parkoló bejárásával, a járművön elhelyezett érzékelők segítségével detektálja a szabad parkolóhelyeket. A dolgozat célja több különböző megoldás bemutatása ezen problémára. A parkoló bejárásához minden esetben ismert a bejárandó parkoló felülnézeti térképe. Az első módszer a celladekompozíciót alkalmazva a térképet sokszög alakú cellákra bontja. A cellák szomszédossági gráfjának felépítése után meghatározható a bejárási sorrend, melynek felhasználásával útvonal tervezhető. A második módszer ezzel szemben nem végez dekompozíciót. Az algoritmus a térképen elhelyezkedő utak Voronoi-diagramjából képes egy bejárandó útvonal tervezésére figyelembe véve a lehetséges parkolóhelyeket.

A szabad parkolóhelyek észlelését egy LiDAR szenzor segíti elő, ahol az implementált algoritmus egy fuzzy alapú osztályozást végez a megfelelő parkolóhely kiválasztására. Ezen osztályozás több minőség jellemző figyelembevételére képes, úgy mint a környező járművekhez való igazítás, illetve egy optimális távolság figyelembevétele az úttest közepétől.

A parkolási manővert megtervező alrendszer folytonos görbületű pályatervezést végez, kielégítve az autószerű mobilis robotokra vonatkozó kényszereket.

A kidolgozott algoritmusok Matlabban kerültek implementálásra és szimulációkban történt a tesztelésük.

Abstract

Nowadays, more and more vehicles are running on the roads, that is the reason why it is getting more difficult to find an adequate parking space near our home, in shopping centers and parking lots. Searching for free parking spaces can be time-consuming. As autonomous vehicles are becoming widely used, the opportunity is given to design a system which performs this activity autonomously and spares this time.

This thesis presents an autonomous parking system, which performs the exploration of the parking lot, the parking space detection with a LiDAR attached to the top of the vehicle, and it plans the path of the parking maneuver itself. The output of this system provides the reference signal (reference path, reference control signal) for a closed-loop control.

The literature provides methods of parking space detection, in which multiple sensors are installed in parking lots. These methods can only be used in parking lots which are equipped with these sensors. This thesis presents a generally applicable method, which detects parking spaces with parking lot exploration using sensors attached to the vehicle. The purpose of this thesis is to present multiple solutions for this problem. The bird's-eye view map of the parking lot is known in each case. The first method applies cell decomposition to divide the map into polygonal cells. After the construction of the adjacency graph of the cells, the graph traversal can be defined, from which a path can be planned. The second method does not decompose the map. The exploration path is planned using the Voronoi-diagram of the routes and taking the location of parking spaces into consideration.

The detection of free parking spaces is based on a LiDAR sensor, and the algorithm makes a fuzzy based classification in order to select the adequate parking space. Throughout the classification more quality features can be taken into consideration, like the alignment to the neighboring vehicles and an optimal distance from the center of the road.

The path planning of the parking maneuver is performed with continuous curvature path planning, in order to fulfill the constraints of car-like mobile vehicles.

The presented algorithms are implemented in Matlab and they were tested in simulations.

Chapter 1

Introduction

Since automobiles came into general use, the lack of sufficient parking spaces is forcing drivers to circle around, sitting in the vehicle searching for free parking spaces. As autonomous vehicles are becoming widely used, the opportunity is given to design a system which performs this activity autonomously [1]. The biggest advantage of this system would be that it spares highly valuable time for the driver.

The literature provides methods of parking space detection, in which multiple sensors are installed in parking lots. In [2] a method is presented, which utilizes the installed CCTV (Closed-Circuit Television) system for the detection based on image processing. [1] shows multiple solutions for this problem. It presents expert systems, fuzzy logic based systems, wireless sensor based systems, GPS (Global Positioning System) based systems, vehicular communication systems, vision based systems and other miscellaneous systems. Internet of Things (IoT) and smart city ecosystem based parking systems can also be the appropriate base of the smart parking according to [3]. CirPark is an active solution for efficient parking [4]. It provides Intelligent Parking Guidance System (iPark), Efficient Led Lighting System (LEDPark) and Electric Vehicle Charging System (EVPark). The common part of the mentioned methods is the need for extra infrastructure, devices and sensors in order to detect the vacant parking spaces. This fact is the most serious disadvantage of these methods, because developing systems like them can be very expensive and time-consuming.

Since autonomous vehicles are becoming available for everyone, it is a manifest idea to design a smart parking system in which the vehicle is smart, instead of installing additional sensors in the parking areas. A smart vehicle is equipped with sensors in order to be able to drive autonomously and detect the free parking spaces. The advantage of a system like this is that it can perform the autonomous parking in a traditional parking place, so it is not needed to make the parking places smart. This approach seems to be the more practical way to create autonomous parking systems.

Available self-parking cars can detect the free parking spaces of appropriate size and perform the parking maneuver, but they are not capable of exploring the whole parking lot

autonomously (e.g [5]). Because of this, chauffeurs need to drive around the parking lot searching for parking spaces. The main target of this thesis is to present a parking system, which can perform the whole parking task autonomously.

The main tasks of the smart parking system are the exploration of the parking lot and meanwhile detecting the free parking spaces. In this thesis, methods for parking lot exploration and parking space detection are presented.

The heart of the matter of the parking lot exploration is driving around the whole parking lot until an appropriate parking space is found. This task is similar to the Coverage Path Planning methods [6].

Trapezoidal cell decomposition method is commonly used as the first step of the coverage path planning of a robot in a polygonal environment. Complete Coverage Path Planning (CPP) is used to determine the path of a mobile robot passing through all points of the workspace while avoiding obstacles. The coverage path planning algorithms have wide application in robotics including cleaning, mowing, exploration of areas etc. In that approach trapezoidal cell decomposition can be used to divide the the environment into smaller regions for effective coverage [7].

In the approach presented in this thesis, cell decomposition is used to provide a map that is covered by polygonal cells. From this map an adjacency matrix can be determined which describes the adjacency graph. Knowing the adjacency graph the graph traversal can be specified.

An algorithm was also implemented, that provides a one step method, that uses the Voronoi diagram of the map, regarding the possible parking zones.

A parking space detection method is also presented in this thesis. The detection is performed with a help of a LiDAR, and different quality factors are assigned to the detected parking spaces in order to select the most appropriate one.

The thesis is organized as follows: Chapter 2 describes the whole system to which the parking lot exploration belongs. In Chapter 3, the background and the steps of the trapezoidal cell decomposition are presented. This chapter also discusses some details of the actual implementation. The next steps of the parking lot exploration, which include the creation of the adjacency matrix based on the decomposition, and creating the graph traversal from the adjacency graph are discussed too. Chapter 4 discusses a modified version of the trapezoidal cell decomposition, and methods for creating the adjacency matrix and the traversal. The last step of the actual offline exploration is the feasible path planning from the points of graph traversal, which is presented in Chapter 5.

Another method of parking lot exploration has also been implemented, which is discussed in Chapter 6. In Chapter 7, the second subsystem of the autonomous parking system is discussed, where a fuzzy based parking space detection method is also presented. Finally the conclusions and the future work are presented in Chapter 8.

Chapter 2

Parking system

The parking lot exploration is part of an autonomous parking system, which can perform the whole parking task (see Fig. 2.1). This autonomous system is made up of subsystems. Different subsystems require different types of maps.

The exploration is the first step of the procedure. In the first step a map of the whole parking lot is needed, as the exploration is a global procedure. This map considers all the free and occupied parking spaces as obstacles. When the global planning has finished, the vehicle starts moving along a planned path, exploring the parking lot.

During this, the second subsystem is activated, where a LiDAR attached to the top of the vehicle sweeps the parking lot searching for a free parking space of adequate size. This subsystem performs a local search, for which only the sensor is needed. The LiDAR provides a 3 dimensional image of the surroundings from which a bird's-eye view map can be created (described in Chapter 7), where free parking spaces are considered as free space.

When the LiDAR detects an appropriate parking space it activates the third subsystem where a feasible path to the parking space, and the continuous curvature path of the parking maneuver are planned. The last step is driving along the planned path to the parking space autonomously, and executing the parking maneuver.

As the literature does not provide solution to parking lot exploration, like the method presented in this thesis, Section 2.1 gives the formulation of the parking lot exploration problem.

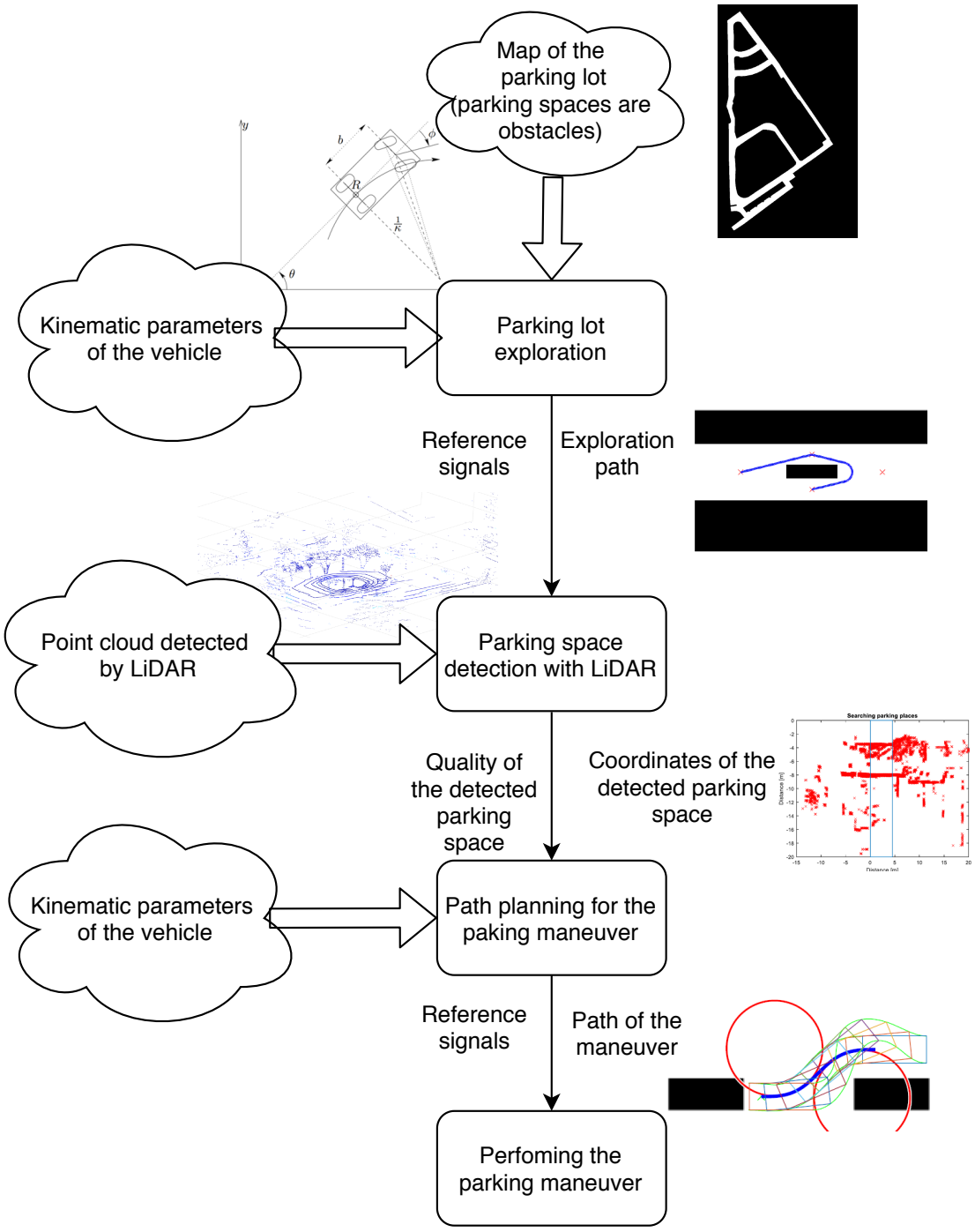


Figure 2.1: *Flowchart of the parking task*

2.1 Formulation of the parking lot exploration problem

Let $C \subset \mathbb{R}^2$ define the workspace of the parking lot exploration and $\mathcal{A}$ denotes the vehicle. The state of the vehicle is $q = \begin{bmatrix} x \\ y \end{bmatrix}$, where $\mathcal{A}(q) \subset C$. $\begin{bmatrix} x \\ y \end{bmatrix}$ denotes the position of the vehicle in a fix frame. (The orientation of the vehicle is not taken into account.) The workspace consists of obstacles ($C_{obs} \subset C$) and free spaces ($C_{free} = C \setminus C_{obs}$), some of which are needed to be visited ($C_{vis} \subseteq C_{free}$). The vehicle can move only in free spaces ($\forall q \in C_{free}$).

The vehicle moves on a collision free path (τ), where $s_i \in \mathbb{R}$ is a scalar path parameter ($s \in [0, T]$, T is the length of the whole path).

$$\tau : s \mapsto q \quad , \; \forall s \in [0, T] : \; \tau(s) \in C_{free} \tag{2.1}$$

The movement of the vehicle can be described with a nonlinear equation:

$$q(t_{k+1}) = f(q(t_k), u_k) \tag{2.2}$$

where u_k is the control signal in the k^{th} sampling time. In this case $\forall k \; \exists \; u_k$, where $\tau(s_{k+1}) = f(\tau(s_k), u_k)$.

$L(q) \subset C$ denotes the points, that can be seen from a given position. Taking into consideration the range of the LiDAR (δ):

$$L(q) = \{z \in C \mid \|q - z\| \leq \delta\}, \tag{2.3}$$

where $\|q - z\|$ is the Euclidean distance between q and z points.

Points seen during traversing the path:

$$L(\tau(t)) = \bigcup_{s \in [0,t]} L(\tau(s)) \tag{2.4}$$

The target is to reach every position, that should be visited during the exploration:

$$C_{vis} \subseteq L(\tau(T)) \tag{2.5}$$

Other constraints, that should be taken into consideration:

- The start position is given $\tau(0) = q_{init}$
- Cost can be assigned to the path: $w_1(\tau)$ (e.g. length of the path)

- Given preferable positions, near which the parking spaces are looked for firstly. $w_2(q)$ cost can be assigned to $q \in C_{free}$ position (e.g. distance from the goal position)
- The traversal can be interrupted, when a condition is met (parking space detected with LiDAR)
- When stopping on the flying the cost of the traversal is $w_1(\tau(s_1)) + w_2(\tau(s_1))$ (cost of the path up to now plus cost of the current position)
- There could be constraints for the order of configurations in τ path (e.g one-way streets)

Chapter 3

Trapezoidal cell decomposition

As the first step of the parking system, the parking lot exploration is needed to be performed. In order to plan a path, which drives along all the possible parking spaces, the map of the parking lot is required to be divided into smaller parts, in which the path planning can be executed easily. It is assumed, that every point of a cell can be seen from any point of the cell.

3.1 Method

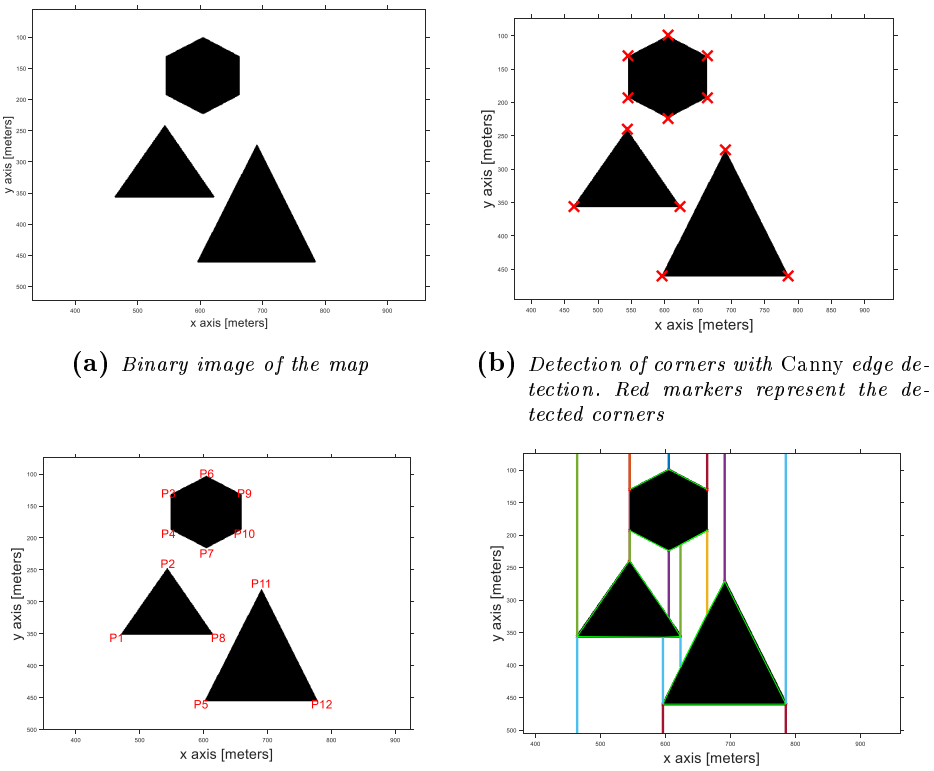
The input of the algorithm is a binary image (Fig. 3.1a), that represents the map of the parking lot from bird's-eye view. The goal is to determine a path through which the vehicle reaches all possible parking zones. The first step of this method is the decomposition of the map.

Trapezoidal cell decomposition [8, 9] is a method used to decompose the free space into convex polygonal[1] cells. Let C_{obs} define the set of points that belongs to the obstacles, and C_{free} denote the set of points that belongs to the non-obstacle points. As a consequence $C_{obs} \cup C_{free} = C$ includes all points of the map. Let P denote the set of vertices used to define C_{obs}. The goal is to extend rays from $\forall p \in P$ upwards and downwards through C_{free} until C_{obs} is hit or the edge of the image is reached.

Steps of cell decomposition:

1. Performing low pass filtering, in order to avoid false corner detection
2. Performing any corner detection (e.g. Harris [10]) (Fig. 3.1b) - determining the set of vertices P
3. Sorting corners (p) in ascending order based on x coordinate (Fig. 3.1c)
4. Extending rays from $\forall p \in P$ (Fig. 3.1d)

[1] trapezoidal or triangular



(a) *Binary image of the map*

(b) *Detection of corners with* Canny *edge detection. Red markers represent the detected corners*

(c) *The obstacles are depicted as black colored objects with the vertices indexed in ascending order from* $P1$ *to* $P12$

(d) *Rays are extended from* $\forall p \in P$

Figure 3.1: *Steps of cell decomposition*

5. Determining the corner points describing each cell:
   - Trapezoidal cells have 4 distinct corner points
   - Triangular cells have only 3 distinct corner points[2]

Step 1 creates a blurred image. As a consequence the transitions from C_{free} to C_{obs} are continuous, which draws the consequence, that only the real corners of the image will be detected by Harris corner detection.

As the map has a finite resolution Steps 2 and 4 might decrease the robustness of the method. In order to make the method robust, double-thresholding is applied (see Fig. 3.2), while exectuting the actual ray extensions (Step 4).

- In order to increase the scale invariance of the decomposition, the use of ε_1 is introduced. This threshold eliminates the influence of the extension of the corner points. Threshold ε_1 should describe the radius of a circle, within which, all $c \in C$ points are considered as part of the corner point $p \in P$.
- ε_2 is required after the use of ε_1. It is used to detect cases when it is not possible to extend ray in a given direction from the corner point. ε_2 should be smaller than the

[2]For the implementation, all cells are stored as trapezoidal cells, but triangular cells have 2 corner points with the same coordinates.

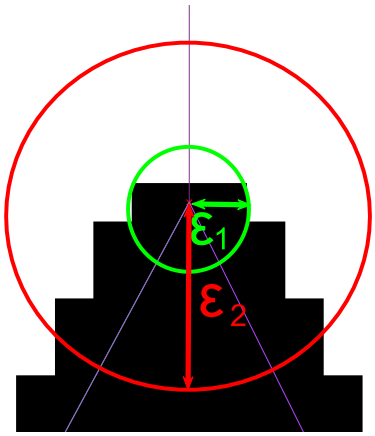


Figure 3.2: *Each point within ε_1 (inside the green circle) belongs to the corner point*

minimal distance that is needed to get to the closest object. The application of this thresholds is introduced, as the binary image has a finite resolution.

The following correlation between ε_1 and ε_2 should be true:

$$\varepsilon_2 > \varepsilon_1 \tag{3.1}$$

As the corners are sorted in ascending order (based on x coordinate) in Step 3 (see Fig. 3.1c), the map is scanned along x axis in Step 4. Algorithm 1 performs both the scanning process and the extension of rays upwards and downwards from the vertices of the obstacles. The upper and the lower points of the rays are stored as cell borders as these rays will become the sides of the cells. The output of Algorithm 1 provides the input of Algorithm 2, where the corresponding cell borders are to be found, which describe a cell. In order to decrease the scale invariance of the decomposition, the use of ε_1 and ε_2 is required while performing the ray extensions (Algorithm 1).

Algorithm 1 Ray extension upwards and downwards

Require: $cornersInAscendingOrder$, C_{obs}

Ensure: $cellBorders$

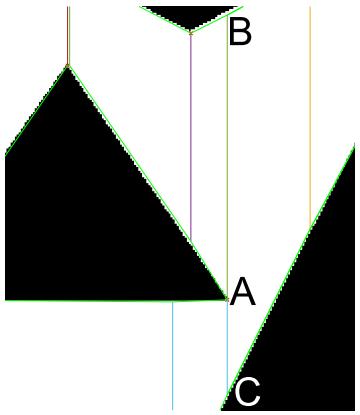
1: $cellBorders = [\,]$;
2: $j = 1$;
3: **for** $i = 1$ **to** $numberOfCorners$ **do**
4: $initialCorner \leftarrow cornersInAscendingOrder[i]$;
5: $upperCellPoint \leftarrow$
findUpperCellPointWithSameXCoordinate ($initialCorner$, C_{obs}, ε_1);
6: **if** dist($initialCorner$,$upperCellPoint$)$> \varepsilon_2$ **then**
7: $cellBorders[j] \leftarrow [upperCellPoint, initialCorner]$;
8: $j \leftarrow j + 1$;
9: **end if**
10: $lowerCellPoint \leftarrow$
findLowerCellPointWithSameXCoordinate ($initialCorner, C_{obs}, \varepsilon_1$);
11: **if** dist($initialCorner$,$lowerCellPoint$)$> \varepsilon_2$ **then**
12: $cellBorders[j] \leftarrow [initialCorner, lowerCellPoint]$;
13: $j \leftarrow j + 1$;
14: **end if**
15: **if** No ray could be extended in any direction **then**
16: $cellBorders[j] \leftarrow [initialCorner, initialCorner]$;
17: $j \leftarrow j + 1$;
18: **end if**
19: **end for**

Fig. 3.3a-3.3b show an example for the application of ε_1, where the corner point A had an extension, but it did not influence the decomposition, as the algorithm found both B and C points, with which two cell borders $B - A$ and $A - C$ are found.

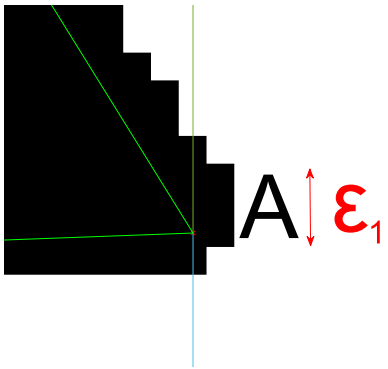
Fig. 3.3c-3.3d show an example for the application of ε_2, where from corner point D no ray could be extended upwards, as the distance of the closest point $c_{closest} \in C_{obs}$ was smaller than ε_2, and $c_{closest}$ was definitely not part of the cornerpoint D, as the distance between D and $c_{closest}$ was higher than ε_1 . The algorithm found point E downwards within C_{obs}, where the distance measured between D and E was higher than ε_2. As a consequence, points D and E will make the border of a cell.

The last step of decomposition is to find the corresponding cell borders, as all cells have both left and right side borders.

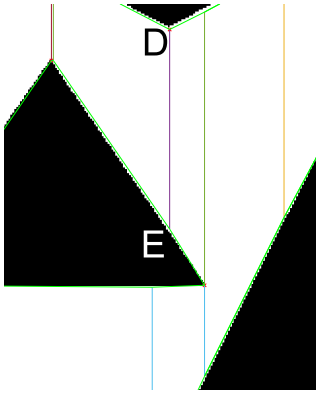
The adjacent cells must have a common border, that is not required to be of the same length. As a consequence, the only requirement for these common borders is the overlapping of the rays. Due to this requirement, in Step 5, it is not only the correspondence of the cell borders that should be determined, but the possibility of the cell border extensions should



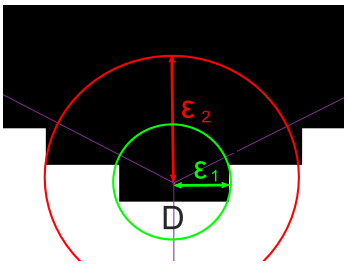
(a) *From corner point A rays could be extended both upwards and downwards*



(b) *Extension of corner A*



(c) *From corner point D ray could be extended only downwards*



(d) *No ray can be extended upwards from corner point D, as obstacle is detected upwards between ε1 and ε2*

Figure 3.3: *Examples for ray extension*

be checked too, while the cell remains convex. (Convex cell should have a nonzero area, and no points of C_{obs} should be inside the cell.) Algorithm 2 performs the search of the corresponding cells.

Algorithm 2 Finding final cells

Require: $cellBorders$
Ensure: $finalCells$

```
finalCells = [ ];
j = 1;
for i = 1 to numberOfCellBorders − 1 do
    leftSideOfCell ← cellBorders[i];
    for k = i + 1 to numberOfCellBorders do
        rightSideOfCell ← cellBorders[k];
        newCell = [leftSideOfCell, rightSideOfCell];
        if newCell is convex then
            break;
        end if
    end for
    if newCell is convex then
        if rightSideOfCell can be extended and newCell remains convex then
            rightSideOfCell ←extendCellBorder(rightSideOfCell, cellBorders);
            newCell = [leftSideOfCell, rightSideOfCell];
        end if
        if leftSideOfCell can be extended and newCell remains convex then
            leftSideOfCell ←extendCellBorder(leftSideOfCell, cellBorders);
            newCell = [leftSideOfCell, rightSideOfCell];
        end if
        finalCells[j] = newCell;
        j ← j + 1;
    end if
end for
```

3.2 Creating the adjacency matrix

As the cells are given with the coordinates of their corners, the coordinates of each side of the cells are known. If the decomposition is executed along the x axis, two cells are adjacent if the left side of one cell has common points with the right side of the other cell. If the decomposition is executed along the y axis, the up and downsides of the cells should be taken into consideration. In order to avoid false detection of adjacent cells a threshold is determined for the minimal number of common points. The width of the vehicle in pixels is a manifest choice for this threshold (see Fig. 3.4).

The creation of the adjacency matrix is presented in Algorithm 3. This algorithm creates a symmetric $n \times n$ matrix, where n is the number of the cells.

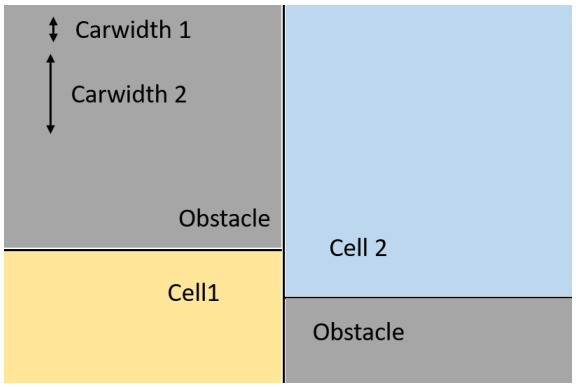


Figure 3.4: *In case of Carwidth 1 Cell 1 and Cell 2 are adjacent, but in case of Carwidth 2 they are not.*

Algorithm 3 Creating the adjacency matrix for trapezoidal cells

Require: $finalCells$

Ensure: $adjacencyMatrix$

1: $adjacencyMatrix \leftarrow$ zeros($numberOfFinalCells$);
2: **for** $i = 1$ **to** $numberOfFinalCells - 1$ **do**
3: **for** $j = i + 1$ **to** $numberOfFinalCells$ **do**
4: **if** rightXCoordinateOf($finalCells[i]$)==leftXCoordinateOf($finalCells[j]$) **then**
5: $commonPoints = [\ ]$;
6: $rightSide \leftarrow$ getRightSide ($finalCells[i]$);
7: $leftSide \leftarrow$ getLeftSide ($finalCells[j]$);
8: $commonPoints \leftarrow$intersect ($rightSide, leftSide$);
9: **if** length($commonPoints$) $>$ $carWidth * 1.5$ **then**
10: $adjacencyMatrix[i, j] = 1$;
11: $adjacencyMatrix[j, i] = 1$;
12: **end if**
13: **end if**
14: **end for**
15: **end for**

If the map has two-way streets only, the adjacency matrix is symmetric. In case of one-way streets the adjacency matrix is a general matrix, as it is permitted to go from one cell to the other but it is prohibited to go in the opposite direction. In this case Algorithm 3 should be slightly modified.

3.3 Determining the graph traversal

Algorithm 4 Creating the graph traversal of the trapezoidal cells

Require: $finalCells, adjacencyMatrix$
Ensure: $cellsInOrder$

markCellAsUnvisited($finalCells$);
$i \leftarrow 1$;
markCellAsVisited($finalCells[i]$);
$cellsInOrder[1] = finalCells[i]$;
$stackList = [\,]$;
while any(isUnvisited($finalCells$)) **do**
  for $j = 1$ **to** $numberOfFinalCells$ **do**
    $newCellFound =$ **false**;
    if $adjacencyMatrix[i, j] == 1$ **and** isUnvisited($finalCells[j]$) ==**true then**
      markCellAsVisited($finalCells[j]$);
      $newCellFound \leftarrow$ **true**;
      $stackList[end + 1] \leftarrow cellsInOrder[end]$;
      $cellsInOrder[end + 1] \leftarrow finalCells[j]$;
    end if
  end for
  if $newCellFound ==$ **false then**
    $cellsInOrder[end + 1] \leftarrow stackList[end]$;
    $stackList[end] \leftarrow [\,]$; {deleting the last element}
    $i \leftarrow$ indexOf($cellsInOrder[end]$);
  end if
end while

The adjacency matrix of the cells describes if two cells are adjacent or not for each pair of cells. If two cells are adjacent the vehicle can pass from one cell to the other [11].

Firstly, every cell should be marked as unvisited. (Initially, it is possible to mark only those cells as unvisited which must be visited (C_{vis}, described in Section 2.1) and the other cells ($C_{free} \setminus C_{vis}$) can be initialized as visited.)

The initial cell is marked as visited and it is the first cell in the order-list. After that, one of the unvisited neighboring cells is chosen as the following cell in the visit order and this cell goes to the end of the order-list. If there is no unvisited adjacent cell, a backtrack is needed to the cell which has an unvisited neighboring cell. This backtrack can be implemented with a stack-list.

Stack-list is used to store the cells in order, and when backtracking is needed, cells are visited again in reverse order from the end of the stack-list. So during backtracking cells are pushed to the end of the order-list in reverse order from the end of the stack-list and they are deleted from the stack-list when they are visited again.

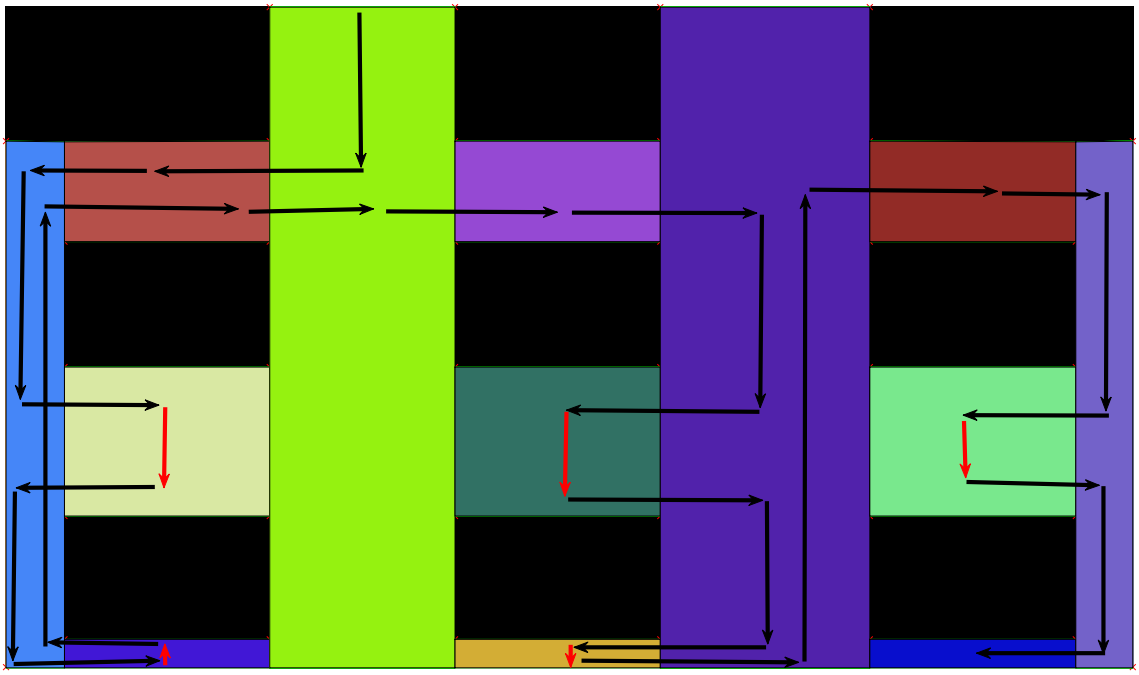


Figure 3.5: *Example of adjacency graph traversal in a map (black areas represent the parking areas and the obstacles, colorful areas represent the road surfaces), red arrows represent the backtracks*

Finally, when no unvisited cells are left, the order-list stores the cells in order, which gives the actual graph traversal (see Fig. 3.5).

Algorithm 4 presents the creation of the traversal.

Chapter 4

Rectangular cell decomposition

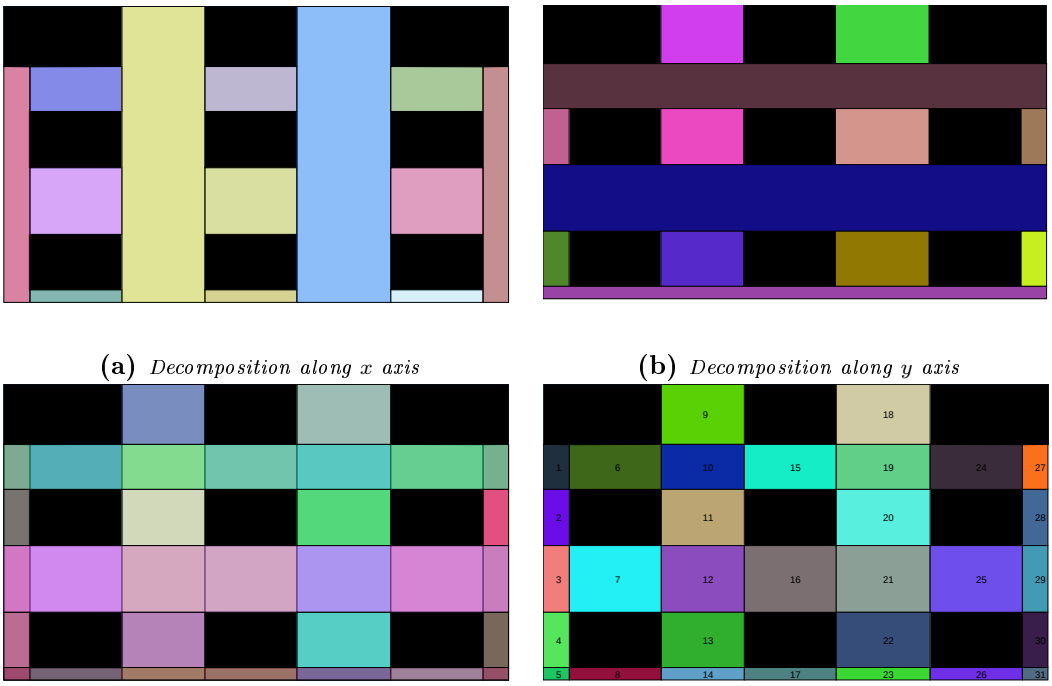
As Fig. 3.5 shows, there are several cells with large areas. It is disadvantageous because the path planning for these cells is not clear. (E.g. if the path goes through the center of the cell, which is a manifest solution, the distance from the parking spaces might be too large to detect them with LiDAR, or sometimes the path does not go through all the points of the cell, which should be visited (C_{vis}, described in Section 2.1).) The presented trapezoidal cell decomposition (see Section 3.1) divides the map into polygonal cells along the x axis. It is a subjective choice along which axis the decomposition is performed. The decomposition can be executed along the y axis, too. This gives the base idea of the modified decomposition: first, decompose the map along the x axis, then along the y axis.

4.1 Steps of the decomposition

This modified decomposition can be executed by the following steps:

1. Performing trapezoidal cell decomposition along x axis (see Fig. 4.1a)
2. Performing trapezoidal cell decomposition along y axis (see Fig. 4.1b)
3. Finding the intersections of the cells decomposed along x and y axis (see Fig. 4.1c)
4. Storing the cells

The method of Step 1 is detailed in Section 3.1. Step 2 is nearly the same as Step 1, the only difference is the direction of the decomposition. It is possible to rotate the image of the map by $\pm 90°$and use the same method as is Step 1. In this case the coordinates of cells are needed to be rotated by $\mp 90°$. In Step 3 the intersections of the cell areas are needed to be determined for each cell-pair. These intersected cells are the final decomposition of the map.



(a) *Decomposition along x axis*

(b) *Decomposition along y axis*

(c) *Intersections of the cells from decompositions along x and y axis*

(d) *Final cells generated by the modified decomposition*

Figure 4.1: *Rectangular cell decomposition*

4.2 Creating the adjacency matrix

By modifying the trapezoidal cell decomposition method, the creation of the adjacency matrix becomes different from the method detailed in Chapter 3.2. The main difference is in the arrangement of the cells. In this case every cell has 4 adjacent cells (upper, lower, left and right neighbor), so it is manifest to store the indices of the neighboring cells for every cell. The adjacency matrix stores the indices of the adjacent cells in the following order: left, right, upper, lower neighbor. Cells neighboring obstacles and cells in the edges of the map do not have 4 neighboring cells, in this case the neighboring obstacles and edges are considered as their neighbors. (An implementational solution can be if these false neighbors are stored with dummy indices e.g -1 or not a number). The adjacency matrix is a $n \times 4$ matrix (instead of the $n \times n$ matrix, presented in Section 3.2), where n is the number of the cells. Two cells are adjacent if the left side of one cell has common points with the right side of the other cell, or the upper side of one cell has common points with the lower side of the other cell. Algorithm 5 presents the creation of the adjacency matrix. The algorithm handles the roads as bidirectional, but with a minor modification it can handle one-way streets, too.

Algorithm 5 Creating the adjacency matrix for rectangular cells

Require: $finalCells$

Ensure: $adjacencyMatrix$

1: $adjacencyMatrix \leftarrow$ zeros($numberOfFinalCells, 4$); {$adjacencyMatrix$ stores the indices of the adjacent cells in the following order: left, right, upper, lower neighbor}
2: **for** $i = 1$ **to** $numberOfFinalCells - 1$ **do**
3: $rightSide1 \leftarrow$ getRightSide($finalCells[i]$);
4: $leftSide1 \leftarrow$ getLeftSide($finalCells[i]$);
5: $upperSide1 \leftarrow$ getUpperSide($finalCells[i]$);
6: $lowerSide1 \leftarrow$ getLowerSide($finalCells[i]$);
7: **for** $j = i + 1$ **to** $numberOfFinalCells$ **do**
8: $rightSide2 \leftarrow$ getRightSide($finalCells[j]$);
9: $leftSide2 \leftarrow$ getLeftSide($finalCells[j]$);
10: $upperSide2 \leftarrow$ getUpperSide($finalCells[j]$);
11: $lowerSide2 \leftarrow$ getLowerSide($finalCells[j]$);
12: $commonPointsLR \leftarrow$ intersect($leftSide1, rightSide2$);
13: $commonPointsRL \leftarrow$ intersect($rightSide1, leftSide2$);
14: $commonPointsUD \leftarrow$ intersect($upperSide1, lowerSide2$);
15: $commonPointsDU \leftarrow$ intersect($lowerSide1, upperSide2$);
16: **if** length($commonPointsLR$) $> carWidth * 1.5$ **then**
17: $adjacencyMatrix[i, 1] = j$;
$adjacencyMatrix[j, 2] = i$;
18: **end if**
19: **if** length($commonPointsRL$) $> carWidth * 1.5$ **then**
20: $adjacencyMatrix[i, 2] = j$;
$adjacencyMatrix[j, 1] = i$;
21: **end if**
22: **if** length($commonPointsUD$) $> carWidth * 1.5$ **then**
23: $adjacencyMatrix[i, 3] = j$;
$adjacencyMatrix[j, 4] = i$;
24: **end if**
25: **if** length($commonPointsDU$) $> carWidth * 1.5$ **then**
26: $adjacencyMatrix[i, 4] = j$;
$adjacencyMatrix[j, 3] = i$;
27: **end if**
28: **end for**
29: **end for**

4.3 Determining the graph traversal

In spite of the fact, that the structure of the adjacency matrix differs from the one detailed in Chapter 3.2, it can be used nearly the same way to determine the graph traversal. The traversal method presented in Chapter 3.3 can be used with a little modification (only the handling of the matrix is needed to be modified). In this case every cell has 4 neighbors, just one in each direction (left, right, up, down).

Taking the advantage of this, a traversal can be planned which avoids reversing when it is possible (in case of dead ends it is impossible to avoid reversing)(see Algorithm 6). Fig. 3.5 shows a traversal in which there are a lot of reversing, and some of them could be eliminated. From the initial cell it is possible to go to any of the unvisited adjacent cells, and the passing direction is stored. After that, those neighboring cells are preferred, which are unvisited and are not in the reverse direction. (E.g. if one of the cells is visited from the left side, from that cell the least preferred direction is to the left. The traversal direction will be left only in case of dead ends.) The neighboring cells are tested in the preferred order whether they are unvisited. If an adjacent cell is unvisited, this cell is chosen to be visited. It is possible, that every neighboring cell is visited, in this case the adjacent cell in the most preferred direction is revisited. In case of dead ends the only adjacent cell is in the reverse direction, so this cell is going to be revisited. Every visited cell is added to the end of the order-list. The algorithm runs till all the cells are visited or the order-list of the cells is longer than a specified value (e.g. 3 times the number of the cells). Finally, the order-list stores the cells in the adequate order.

It is possible to assign preference to the directions. In this case the order of testing the cells - as possible following cells - is based on the preference. Of course random preference is also permitted, then directions are preferred equally.

Compared to the method detailed in Chapter 3.3, the main advantage of Algorithm 6 is avoiding reversing, but the disadvantage is that it visits a cell multiple times and there may remain unvisited cells in the end (see Fig. 4.2).

Algorithm 6 Creating the graph traversal of the rectangular cells

Require: $finalCells, adjacencyMatrix$

Ensure: $cellsInOrder$

$cells \leftarrow finalCells$;
$actualCell \leftarrow cells[1]$;
markCellAsVisited($actualCell$);
$cellsInOrder[1] \leftarrow actualCell$;
$preference \leftarrow [up, down, left, right]$;
while there is any unvisited cell **do**
  $neighbors \leftarrow$ get4Neighbours($actualCell, adjacencyMatrix$);
  $neighborToGoTo \leftarrow$ getBestUnvisitedNeighbor($neighbors, cells, preference$);
  if no $neighborToGoTo$ found **then**
    $neighborToGoTo \leftarrow$ getBestVisitedNeighbor($neighbors, cells, preference$);
  end if
  $preference \leftarrow$disFavorReversal($preference, actualCell, neighborToGoTo$);
  {Reordering $preference$ array, where the reverse direction is the last element}
  $actualCell \leftarrow neighborToGoTo$
  $cellsInOrder[end + 1] \leftarrow actualCell$
  markCellAsVisited($actualCell$)
  if length($cellsInOrder$)>3∗length($cells$) **then**
    break;
  end if
end while

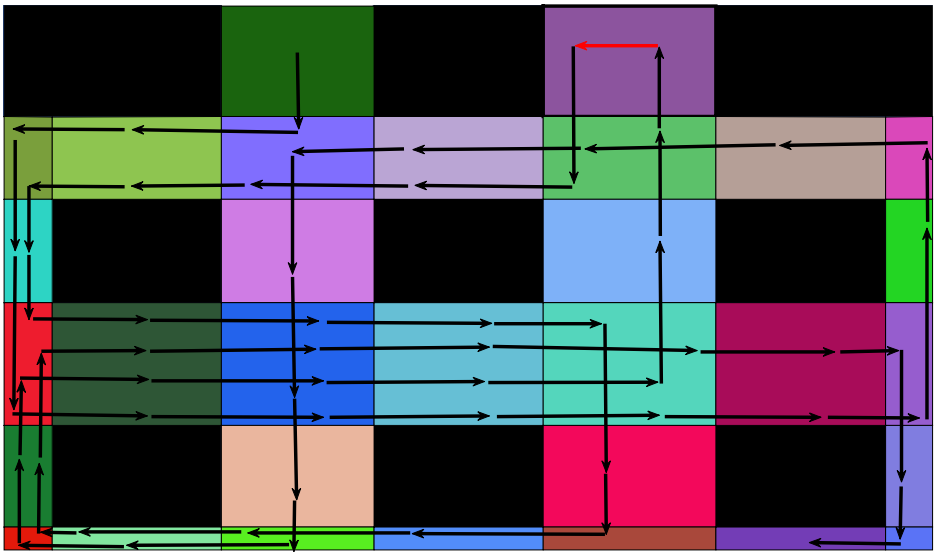


Figure 4.2: *Example of adjacency graph traversal in a map (black areas represent the parking areas and the obstacles, colorful areas represent the road surfaces) red arrows represent the backtracks*

4.4 Wavefront algorithm based traversal

A more effective method of determining the traversal is inspired by the grid-based coverage using the wavefront algorithm, usually used in Coverage Path Planning tasks [6]. This algorithm is applied in grid-based coverage, so in this approach the decomposed map is treated as a grid. As cells are of different size, grid is different from the traditional grid. A start and a goal point are also needed for the algorithm, so it is expedient to choose the initial cell as both the start and goal destination (see Fig. 4.3). In this case the adjacency matrix presented in Section 4.2 is used, but a distance value is assigned to every cell. The distance value is 0 at the goal cell, then every neighboring cell gets one bigger distance value. This step is repeated until there are unmarked cells left.

This algorithm is very similar to Algorithm 6, the main difference between them is in the preference of the directions. The traversal starts from the starting point and the wavefront algorithm firstly visits those unvisited neighboring cells, which have the highest distance value. This algorithm also tries to avoid reversing, so the reverse direction is the least preferred direction. If a cell has only visited adjacent cells, the one with the highest distance value will be the following cell (see Fig. 4.4). If the start and goal cells are the same, the traversal leads to the furthest cells at first.

The presented traversal does not drive back to the initial cell, but every cell is visited in the end. However, in reality it is a frequent occasion when no parking space is found in the first traversal around the parking lot. In this case the best strategy is to repeat this traversal until an adequate parking space is found. By the application of this method the algorithm can be forced to return to the initial cell by remarking the initial cell as unvisited when there are no unvisited cells left. Another solution can be marking the last cell as initial cell and recreating the traversal.

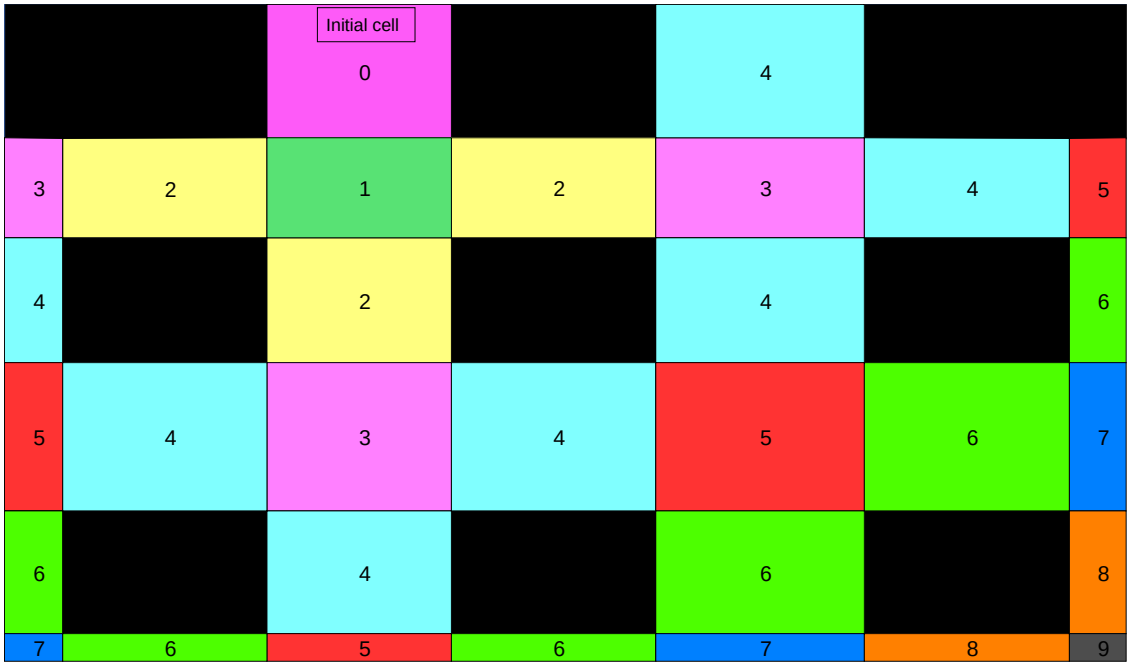


Figure 4.3: *Distance values assigned to the cells*

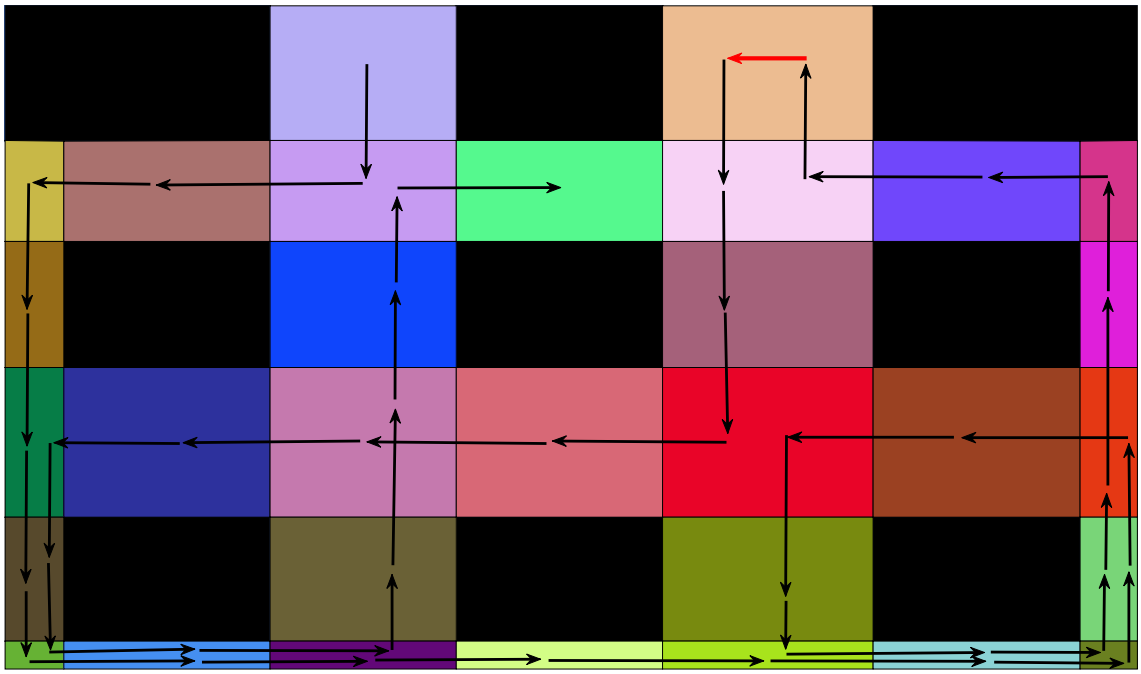


Figure 4.4: *Example of adjacency graph traversal in a map (black areas represent the parking areas and the obstacles, colorful areas represent the road surfaces) red arrows represent the backtarcks*

Chapter 5

Path planning

Knowing the graph traversal it is possible to plan a feasible path for the vehicle [12]. In order to avoid collisions with the obstacles in the map, the obstacles should be inflated by the radius of the circumscribed circle of the vehicle. In this case the vehicle can be considered as a point. It is possible to plan the path of the corners of the vehicle, but it is more compute-intensive. (The vehicle is approximated with circle only for the parking lot traversal, during the parking maneuver planning the paths of the corners of the vehicle are calculated, too.)

From each cell at least one point is needed to be chosen, which is used for the path planning. This point can be the mass center of the cell, one of the corners or other parameter of the cell.

There are several methods to plan feasible paths between the chosen points. For example continuous curvature path planning [13] can be used to design a path that uses the chosen points in adequate order to calculate the path (see Fig. 5.1). In case of a simple map it can be enough to choose only one point from each cell, but in case of a more complex map the going-over of the cell should be planned, too.

Fig. 5.1 shows an example for a continuous curvature path, planned using the mass centers of the cells. In case of backtracks the mass center of the cell should be pushed by the double size of the minimal turning radius of the car to the left and to the right (or up and down depending on the situation of the cells). Because of this, in the cell, where the turnaround is performed, there are two points used for path planning instead of one. This duplication of the point is needed so that there should be enough space between the points for the vehicle, in order to be able to perform the turnaround in one maneuver. It is possible that the cell is too small for the vehicle to turn around, in this case the vehicle can reverse on the same path, it came into the cell. Backtracks can be easily detected from the cell traversing, as the cell before and after the cell, in which the turnaround is needed, are the same.

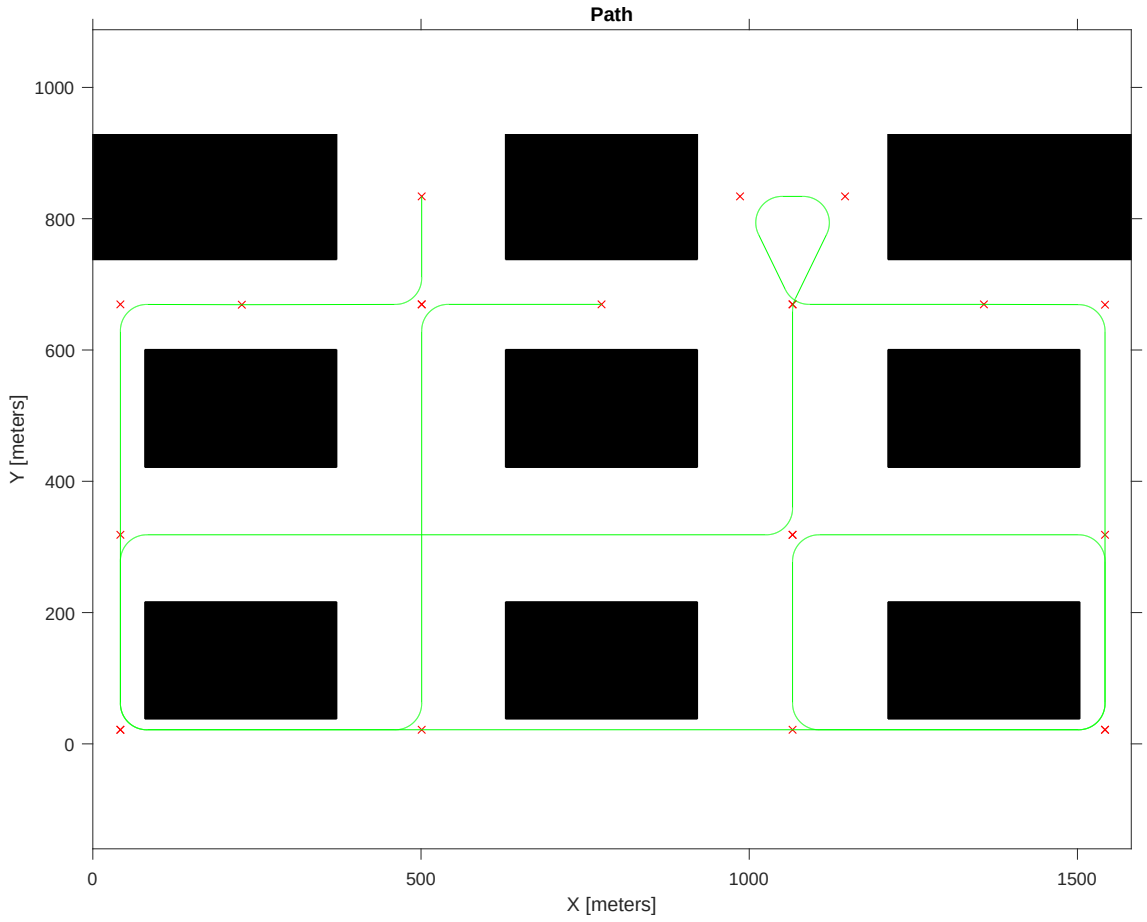
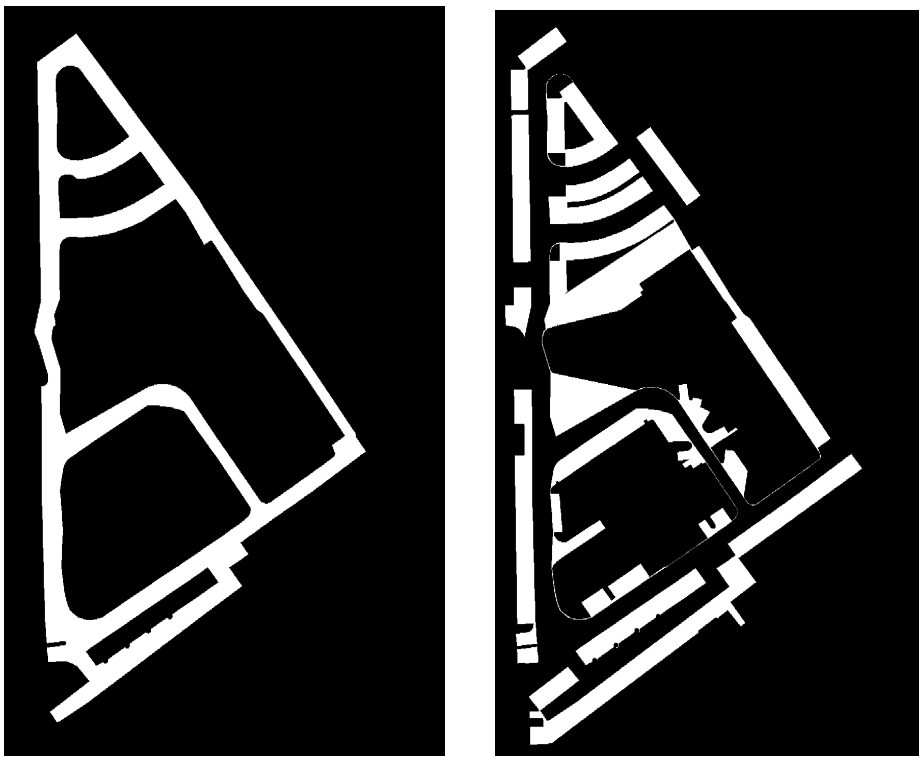


Figure 5.1: *Example for a map exploration in a parking lot, this path belongs to the traversal shown in Fig. 4.4*

Chapter 6

Voronoi diagram based method

Chapter 3-4 described the trapezoidal cell decomposition, which can realize the second subsystem of the autonomous parking system, described in Chapter 2. In that approach the free space had been decomposed into polygonal cells, with which a graph traversal could be planned. In this chapter another possible realization of the subsystem is presented, which creates the Voronoi diagram of the binary image. These methods do not differ in terms of output and the input types, as both require a binary image, and provide a possible exploration path. The only differences are the method, and the exploration path provided by the algorithms. The presented method considers the roads as bidirectional.



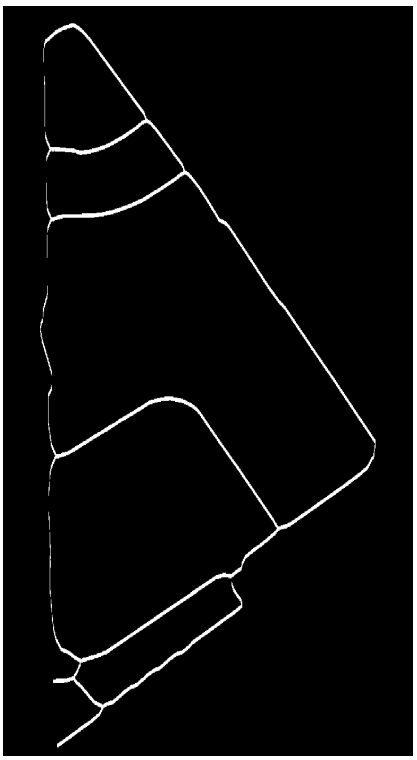
(a) *Binary image of the road surface* **(b)** *Binary image of the parking zones*

Figure 6.1: *Binary images storing road surface and parking zones. Black color represents obstacles*

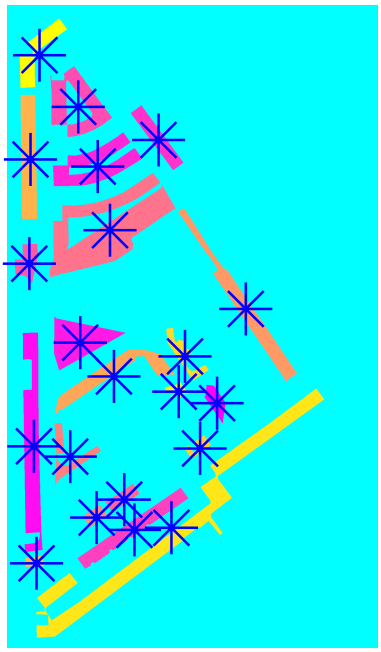
The Voronoi diagram based method requires a binary image. From the binary image, the Voronoi diagram / skeleton can be determined. The Voronoi diagram of the road surface provides a set of points, that lies at the same distance from both sides of the road or from both sides of the lane. As the map of the road surface of the parking lot is known (see Fig. 6.1a), and the places where parking spaces (see Fig. 6.1b) are to be looked for are also known, a route can be planned throughout the parking lot. In order to determine the route, descriptive properties [14] of the different parking zones should be determined, such as the centroids, or the extremum. With the help of these properties those points of the Voronoi diagram can be pointed out that are close to these parking space descriptive points.

6.1 Steps of the method

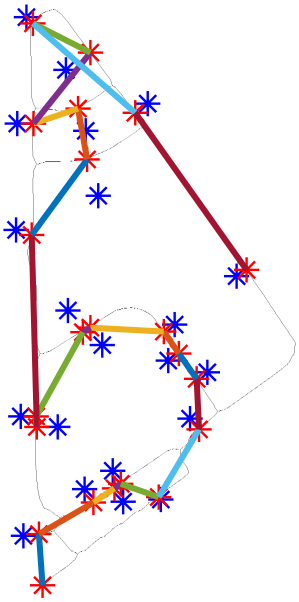
1. Creating the Voronoi diagram (Fig. 6.2a) from the binary image (Fig. 6.1a) of the road surface
2. Defining the connected components (see colorful objects in Fig. 6.2b) from the binary image (Fig. 6.1b) of the parking zones - algorithm explained in Section 6.2
3. Determining the centroids (or other descriptive points) of the parking zones (see blue markers in Fig. 6.2b)
4. Determining the closest points of the Voronoi diagram to the centroids (see red markers in Fig. 6.2c)
5. Determine the sequence of points to be reached (see colorful lines in Fig. 6.2c) - algorithm explained in Section 6.3
6. Finding the shortest route between the set of points (Fig. 6.2d) [15]



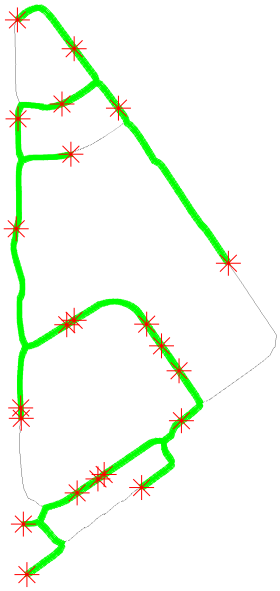
(a) *Voronoi diagram of the road surface*



(b) *Colorful objects are the possible parking zones. Blue markers represent the centroids of the parking zones*



(c) *Sequence of route points. Blue markers are the centroids of the parking zones while the colorful lines are the route sections between closest points (red markers) of the Voronoi diagram*



(d) *Final Route. Green line represents the final route, while red markers are the closest points of the Voronoi diagram to the centroids*

Figure 6.2: *Main steps of Voronoi diagram based method*

6.2 Defining the connected components

A parking lot is made up of parking zones, which have several parking spaces. A set of parking spaces makes a parking zone. These zones are separate from each other, but the road surface makes these zones accesable from other parking zones. The Voronoi diagram based method takes the location of the parking zones into consideration. In order to be able to determine the descriptive properties of the parking zones (such as the centroids or extremum), the pixels of the parking zones should be determined, which describe the parking zone as an object. Algorithm 7 performs the search of the pixels that belong to the same object, by labeling the pixels. [16]

Algorithm 7 Defining the connected components

Require: $binaryOfParkingZones$

Ensure: $connectedComponents$

1: $connectedComponents \leftarrow [\quad]$;
2: $L \leftarrow 1$; {L is the label of the object}
3: $p \leftarrow$ getFirstUnlabeledObjectPixel($binaryOfParkingZones$);
4: **if** No p is found **then**
5: **return** $connectedComponents$
6: **end if**
7: $connectedComponents \leftarrow$ assignLabelToPixel(L,p);
8: $p \leftarrow$getNeighbourObjectPixelOfPixel(p,$binaryOfParkingZones$);
9: **if** No p is found **then**
10: $L \leftarrow L + 1$;
11: GOTO 3
12: **else**
13: GOTO 7
14: **end if**

Algorithm 7 provides all the object pixels, which are the pixels of the parking zones, labeled for each separate zone. As a conseqence all descriptive points of the parking zones can be determined. For example the calculation of the centroid is provided by (6.1). [16]

$$M_{00} = \sum_{x=0}^{W}\sum_{y=0}^{H} I(x,y), \qquad M_{10} = \sum_{x=0}^{W}\sum_{y=0}^{H} x * I(x,y), \qquad M_{01} = \sum_{x=0}^{W}\sum_{y=0}^{H} y * I(x,y)$$
$$C = \left(\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}}\right) \tag{6.1}$$

The input of the calculation (6.1) is the binary image of a given parking zone $I(x,y)$, where W is the width, and H is the height of the image. For the calculation of the centroid (C), the calculation of the moments of zero (M_{00}) and first (M_{10}, M_{01}) orders are required.

6.3 Determine the sequence of points to be reached

The calculation of the sequence of points to be reached is a crucial part of the Voronoi diagram based method. It is not enough to find the points to be reached, but a traversal close to the optimal solution should be found too.

6.3.1 Airline based distance traversal

It seems to be a manifest choice to visit parking zones first, that are close to each other. This would mean that the algorithm would first visit places where the dispersion of the parking zones is dense. The sequence of points provided by this algorithm can be seen in Fig. 6.2c. In that case the airline distance was measured between these parking zones. As Fig. 6.2d shows, all centroids (marked with red markers) were reached.

The huge disadvantage of this method is that no cost optimization is taken into account. Which means that the vehicle can reverse each time it has reached a parking zone, in order to travel to the next one. Another disadvantage of this method can be seen in Fig. 6.2d, where the vehicle only travels till the centroid is reached. This could cause that only half of the parking zone is explored.

6.3.2 Graph based traversal

To find a close optimal solution for the traversal, a graph should be created from the Voronoi diagram, where all centroids would be nodes. The edges represent the road surface between the parking zones. If the Voronoi diagram does not contain any road crossings a parking zone might have only 2 neighbors. In that case the traversal is trivial until all nodes are reachable.

When crossroads are found in the Voronoi diagram, which is the most likely case in reality, multiple traversal options can be found with different costs. To represent the crossroads in the graph, all crossing should be added to the graph as nodes. After that the nodes of the centroids might have only 2 neighbors, but the crossing nodes might have any number of neighbors. The graph created from the Voronoi diagram can be seen in Fig. 6.3.

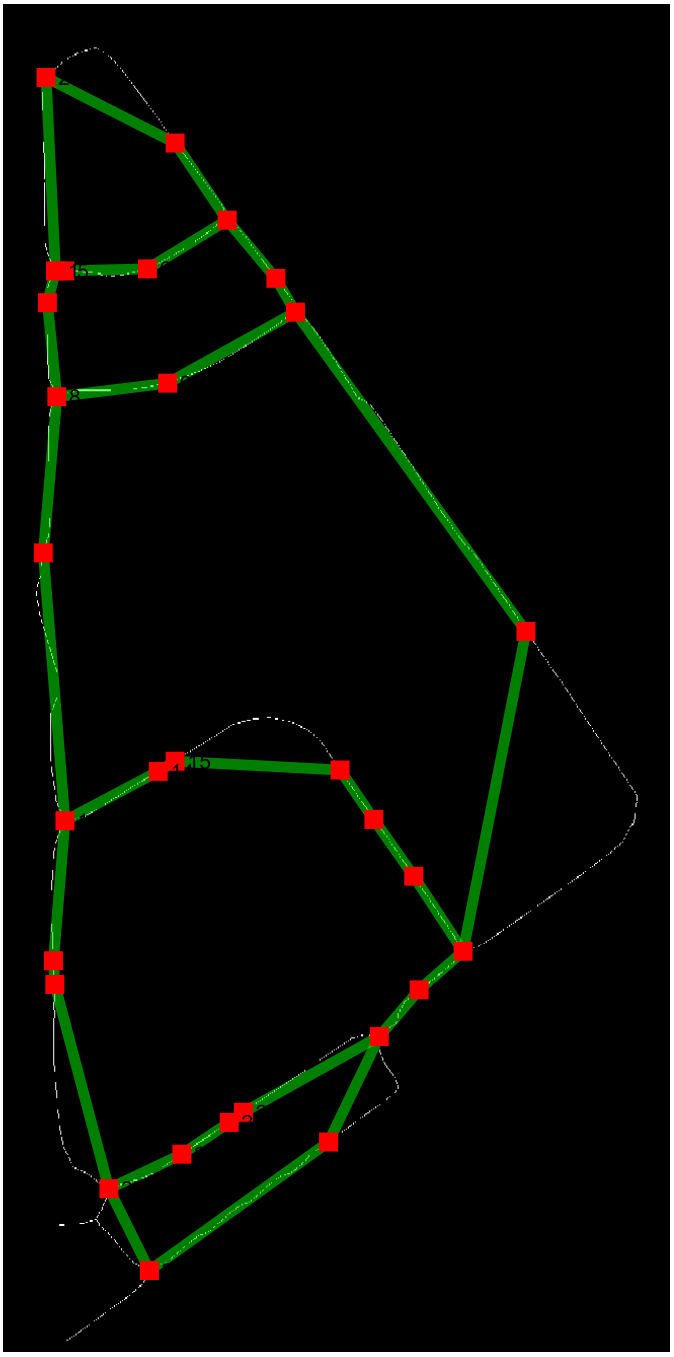


Figure 6.3: *Graph created from the Voronoi diagram*
Note: The edges between the nodes represent the road with the distance between the vertices

Traversal of the graph

Let $G = (V, E)$ denote the undirected graph, where V is the set of vertices with the number of nodes $N < \infty$, and E denotes the set of edges between the vertices. Let $v_c \in V_c$ denote the vertices, that represent the closest points of the Voronoi-diagram to the parking zone centroids and $V_c \subseteq V$. As a consequence $V_{rc} = V \setminus V_c$, where V_{rc} is the set of vertices representing the road crossings. The weight of the edge $\forall e \in E$ is denoted by $w(e)$, where $w(e) > 0$. Between two vertices ($v_i, v_j \in V$), where $i \neq j$, the edges ($e_p \in E, \quad p = 1...n$), appointing the shortest path between v_i and v_j determine the cost function $c(v_i, v_j) = \sum_{\forall e_p \in E} w(e_p)$, where $c(v_i, v_j) > 0$ if $i \neq j$. The goal of the traversal is to find the permutation of vertices $S_N(v_c)$ for which the total cost (6.2) is minimal.

$$C = \sum_{i=1}^{N-1} c(s_i, s_{i+1}) \tag{6.2}$$

Let $G_c = (V_c, E_c)$ denote the graph, where V_c is the set of vertices that represent the closest points of the Voronoi-diagram to the parking zone centroids, and E_c is the set of edges between the vertices. The edge $\forall e_{c,k} \in E_c$ is created so that: $e_{c,k} = e_{c_{i,j}}$, where $e_{c_{i,j}} \equiv \{v_{c,i}, v_{c,j}\} \in E$ and $v_{c,i}, v_{c,j} \in V_c \subseteq V$, or when $\nexists\ e_{c_{i,j}}$ than $e_{c,k} = e_l \cup e_m = \{v_{c,i}, v_{rc,a}\} \cup \{v_{rc,a}, v_{c,j}\}$, where $e_l, e_m \in E$ and $v_{rc,a} \in V_{rc}$.

If the graph $G_c = (V_c, E_c)$ contains a *Hamiltonian* cycle, the traversal should be equal to the *Hamiltonian* path $\mathcal{H}_N$. The *Hamiltonian* path would minimize C, however it is known that it is an *NP*-hard problem to determine whether $G_c = (V_c, E_c)$ has a *Hamiltonian* cycle. What is more, in real parking lots, there are usually more than two dead ends, which means that no *Hamiltonian* cycle can be found.

When speaking about vertex exploration, the manifest choices seem to be the Breadth-first search (*BFS*) and Depth-first search (*DFS*) algorithms [17]. These algorithms use opposite strategies of graph exploration. *BFS* provides a sequence of nodes $S_N(v_c)$, where first the neighboring vertices of the initial vertex, than the neighbors of these neighboring vertices would be explored recursively. This would result in a path where the number of the vehicle reversal is extremely high, which is rather undesirable. In contrast to *BFS*, *DFS* algorithm provides a path where the number of the vehicle reversal is minimal, however the total cost C is not taken into consideration at all.

The lack of optimal algorithmic solution leads to the use of artificial intelligence (AI). The function of the total cost C is provided by (6.2), and the goal is to minimize this function. For this purpose a genetic algorithm has been implemented, which tries to minimize the cost of the graph traversal, which also results in the minimization of the vehicle reversals.

Genetic algorithms [18] are inspired by the process of natural selection. An initial population is provided, from which the fittest individuals are selected for reproduction. The

recombination of the selected individuals provide a new population of offsprings. Similarly to biological effects, an offspring might contain mutated genes, which make the population more diverse. The offsprings are reinserted into the original population, with a specific rate of reinsertion, that is affected by the environmental factors, the fitness of the new offsprings, and other conditions. These steps are repeated in each generation until forever, or until the optimal solution is reached.

In the approach of graph traversal optimality, the initial population is made up of individuals which have the genes of $S_N(v_c)$ in random permutations. The fitness of the individuals is determined by the order of the individual in terms of cost effectiveness. The objective function that should be minimized is the total cost function described by (6.2).

As a consequence the individual with the lowest cost is the fittest, and the one with the highest cost is the least fit. After a certain number of generations, by selecting the the fittest individual of the last generation, the cost function is close enough to the optimal solution.

The genetic algorithm was tested with the following parameters:

- Number of elements in V_c : 20
- Number of individuals in each generation: 50
- Maximal number of generations: 200
- Generation gap: 80%
- Probability of mutation: 1%
- Reinsertion rate of offsprings: 40%

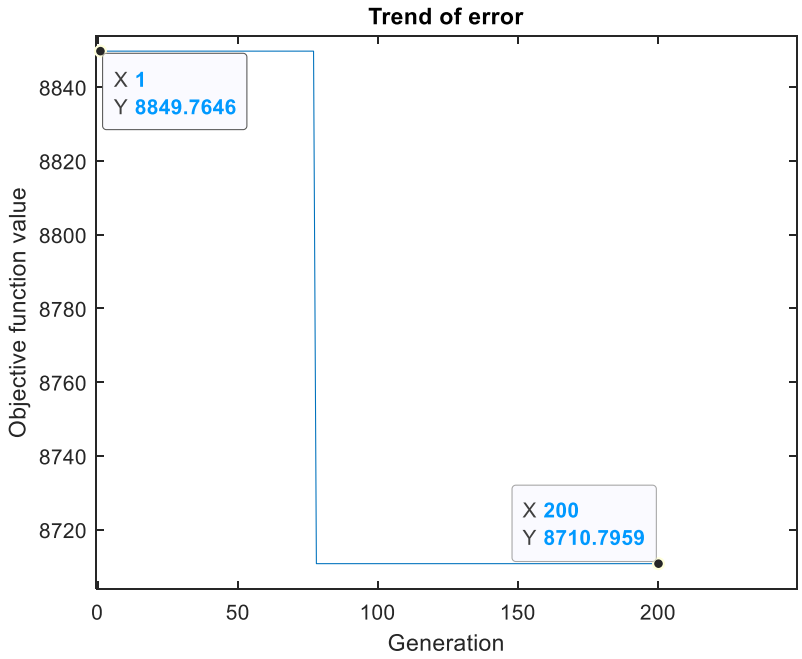


Figure 6.4: *The values of the cost function (6.2) for the best individual in each generation*

Fig. 6.4 shows that the randomly initialized first population had a best individual with total cost of 8849. The next 60 generations could not improve the cost of the traversal. The last generation had a lower cost of traversal with the total cost of 8710.

The final path shown in Fig. 6.5, starts from the vertex marked with green numbers of visit order , and travels along all vertices of the graph G_c, that represent the closest points of the centroids to the Voronoi diagram. Although the traversal contains some reversal of the vehicle, the cost function has decreased in comparison to the random traversal.

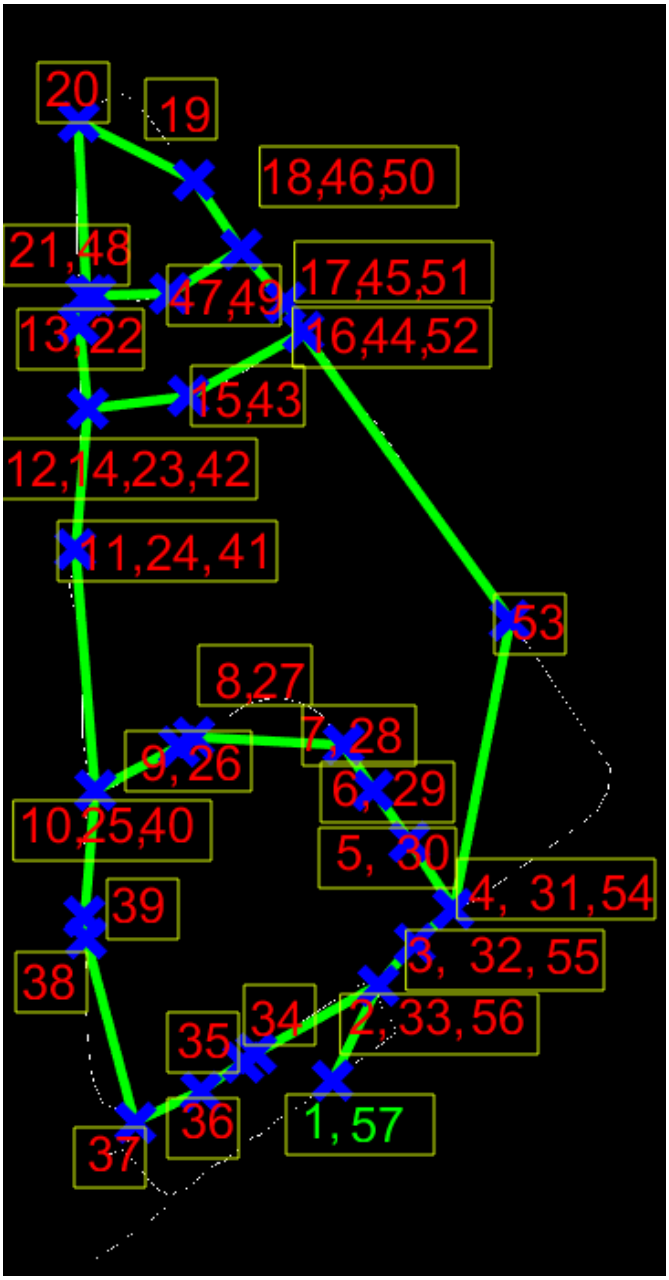


Figure 6.5: *The traversal after the last generation, where the visit orders are assigned to each vertex. The vertex marked with green numbers is the initial and final point of the traversal*

Chapter 7

Parking space detection

Chapters 3-6 described possible realizations of the first subsystem of the whole autonomous parking system, presented in Chapter 2. This chapter gives the basics of the second subsystem, which is the parking space detection. The sensor that has been used is a VLP-16 LiDAR [19].

7.1 LiDAR description

The VLP-16 sensor uses an array of 16 infra-red (IR) lasers paired with IR detectors to measure distances to objects. The device is mounted securely within a compact, weather-resistant housing. The array of laser/detector pairs spins rapidly within its fixed housing to scan the surrounding environment, firing each laser approximately 18,000 times per second, providing, in real-time, a rich set of 3D point data.

Advanced digital signal processing and waveform analysis provide highly accurate long-range sensing, as well as calibrated reflectivity data, enabling easy detection of retro-reflectors like street-signs, license plates, and lane markings. Combining 16 laser/detector pairs into one VLP-16 sensor and pulsing each at 18.08 kHz enables measurements of up to 300,000 data points per second – or double that in dual return mode. [19]

7.2 Processing LiDAR data in MATLAB

Data processing is performed in MATLAB, where a preprocession of raw data is performed in order to increase the efficiency of the calculations.

7.2.1 Raw data from LiDAR

Data from LiDAR is stored in pointCloud object, which is a 3D-object storing $[x, y, z]$ coordinates of obstacles points. As a consequence the pointCloud determines the surroundings

of the LiDAR in Cartesian coordinate system, where the LiDAR is located at the origin (0,0,0).

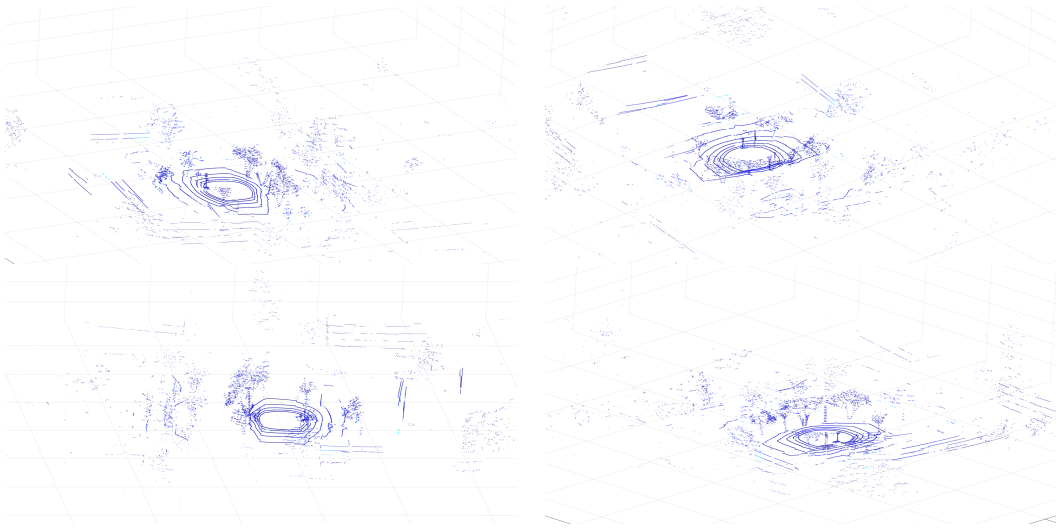


Figure 7.1: *Example pointCloud object read from LiDAR. Figure depicts the same measurement from different viewpoints*

Fig. 7.1 shows an example of the data read from LiDAR. As it can be seen it is a 3 dimensional representation of the surroundings, where the LiDAR is located at the origin.

7.2.2 Effect of the orientation of the LiDAR sensor

The orientation of the LiDAR differs from the reference coordinate system. The center of the vehicle represents the origin of the reference coordinate system. In order to simplify further calculations let the orientation matrix, denoted by $\mathbf{A_o}$, be the identity matrix, and let the position ($\mathbf{p_o}$) of the vehicle be at the origin. Let $\mathbf{K_o}$ denote the reference coordinate system.

$$\mathbf{K_o} = \begin{bmatrix} \mathbf{A_o} & | & \mathbf{p_o} \\ - & | & - \\ \mathbf{0^T} & | & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & | & 0 \\ 0 & 1 & 0 & | & 0 \\ 0 & 0 & 1 & | & 0 \\ - & - & - & | & - \\ 0 & 0 & 0 & | & 1 \end{bmatrix} \tag{7.1}$$

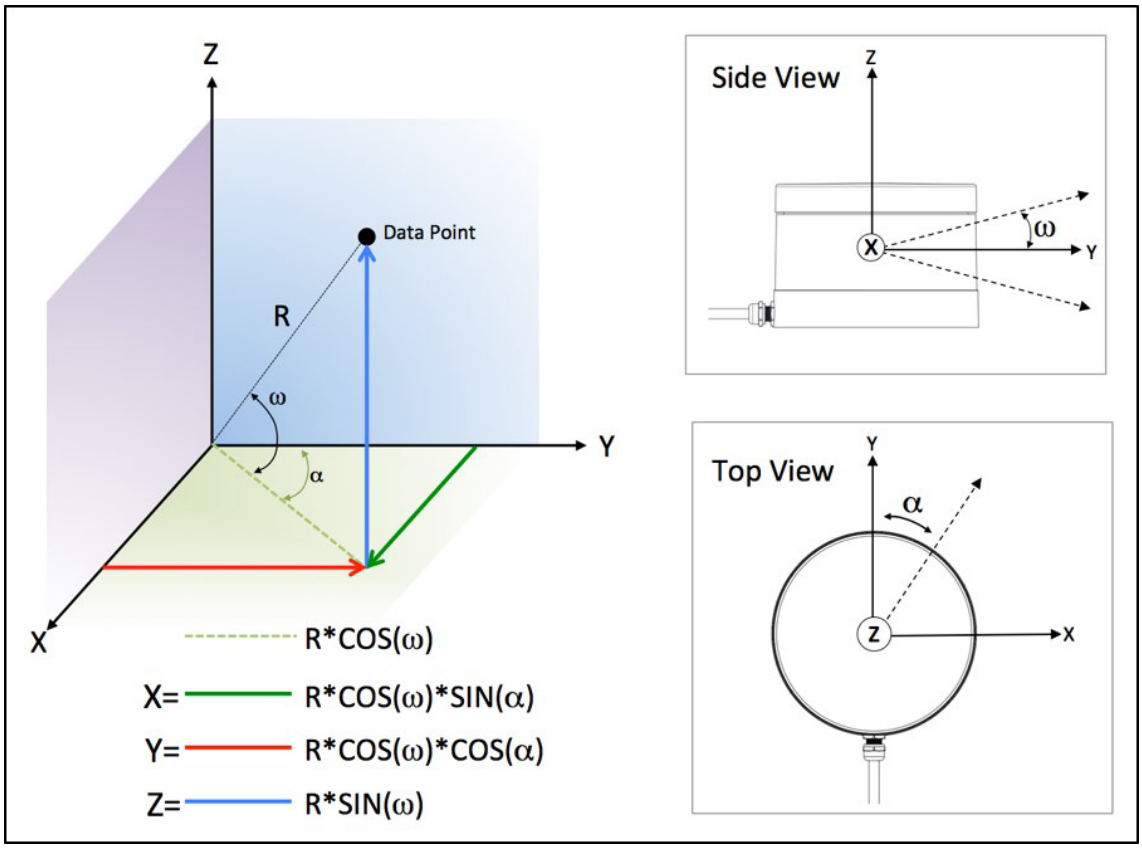


Figure 7.2: *Reference system of the LiDAR [19]*

The vehicle is moving along x axis (see Fig. 7.2.). Let K_l denote the frame fixed to the middle point of the LiDAR[1]. As in most cases the K_l coordinate system is not identical with K_o, a transformation is required in order to determine distances relative to the frame fixed in the middle point of the vehicle.

The transformation can be expressed with a transformation matrix called *homogenous transformation matrix* ($T_{o,l}$).

$$\mathbf{K_l} = \mathbf{T_{o,l}}\mathbf{K_o} \Longrightarrow \mathbf{T_{o,l}} = \mathbf{K_l}\mathbf{K_o^{-1}} \tag{7.2}$$

As $\mathbf{K_o}$ is an identity matrix in (7.1) there is no need for inverse calculation in (7.2). Which means that for the further calculations the *homogenous transformation matrix* is the frame fixed to the center of the sensor.

$$\mathbf{T_{o,l}} = \mathbf{K_l} = \begin{bmatrix} \mathbf{A_{o,l}} & | & \mathbf{p_{0,l}} \\ - & | & - \\ \mathbf{0^T} & | & 1 \end{bmatrix} \tag{7.3}$$

[1]Or any other sensor that is used for detection

7.2.3 Ground segmentation

The LiDAR is mounted on top of the vehicle at the height of h_0 (see Fig. 7.3).

Figure 7.3: *LiDAR location on top of the vehicle [20]*

By specifying both h_0 and the maximal height of interest h_{max} a simple ground and high object segmentation can be performed. The eliminations of the obstacle points that do not influence the occupancy of a given parking place, is performed by (7.4).

$$f(x_i, y_i, z_i) = \begin{cases} [\quad], & \text{if} \quad z_i < -h_0 \quad \vee \quad z_i > h_{max} \\ [x_i, y_i, z_i], & \text{otherwise} \end{cases} \quad ,\forall i \tag{7.4}$$

This method of ground segmentation was introduced, to make faster execution possible. Another possibility for ground segmentation is based on the surface normal, which in case of a 3 dimensional space would be $n_3 = [0, 0, 1]$ vector. In the future phase of the development, the segmentation based on the surface normal will be used.

As the planning of the parking maneuver is a two-dimensional problem, after ground segmentation by projecting all the 16 layers of data to the $x-y$ plane, the map is created in bird's-eye view (See Fig. 7.4).

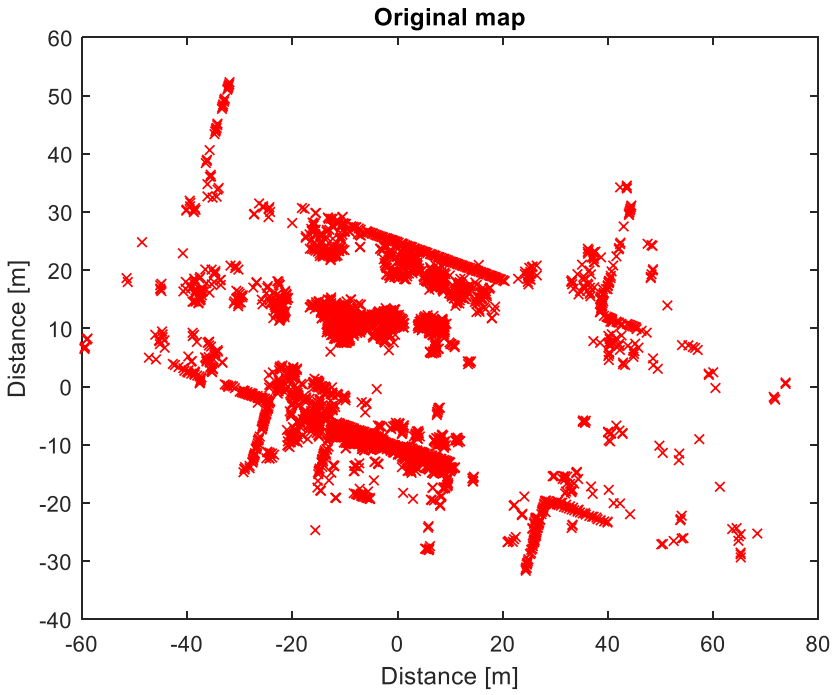


Figure 7.4: *Bird's-eye view map, after projection to the $x-y$ plane*

7.2.4 Alignment of coordinate system

The parking places are to be searched parallel with, or perpendicular to the vehicle. As the vehicle might not be perfectly aligned, there might be a need for alignment, so that the parking places are to be searched parallel with, or perpendicular to the orientation of the vehicle.

As Fig. 7.4 depicts, the roadside is not parallel with the x axis, which is the orientation of the vehicle. In order to solve the misalignment, an angle of rotation α should be determined.

The steps of alignment are the following:

1. Creating a binary image form the 2 dimensional matrix storing all obstacle coordinates
2. Performing Canny edge detection on the image [16]
3. Using Hough transformation [16] to determine the most significant lines between the angles of -45^o and 45^o.
4. The angle of the most significant line determines the original rotation.

Creating a binary image

Figure 7.5: *Binary image of the bird's-eye view map*

The binary image (Fig. 7.5) is created in order to be able to calculate the derivative with the help of *Canny* edge detection.

Performing Canny edge detection on the image

As *Hough* transformation provides a more robust *Hough-line* detection in derivative images, *Canny* edge detection is performed before executing the transformation to *Hough-space*.

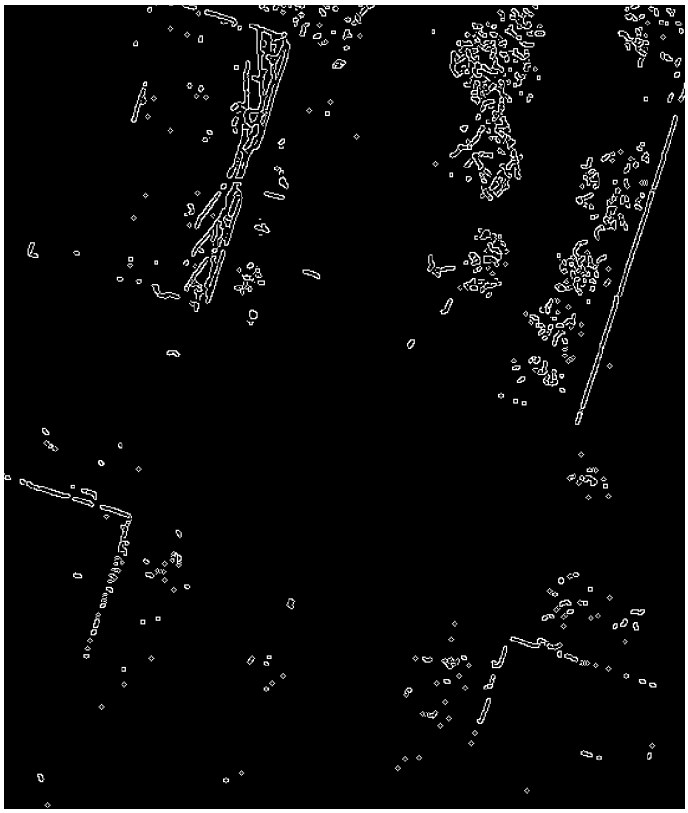
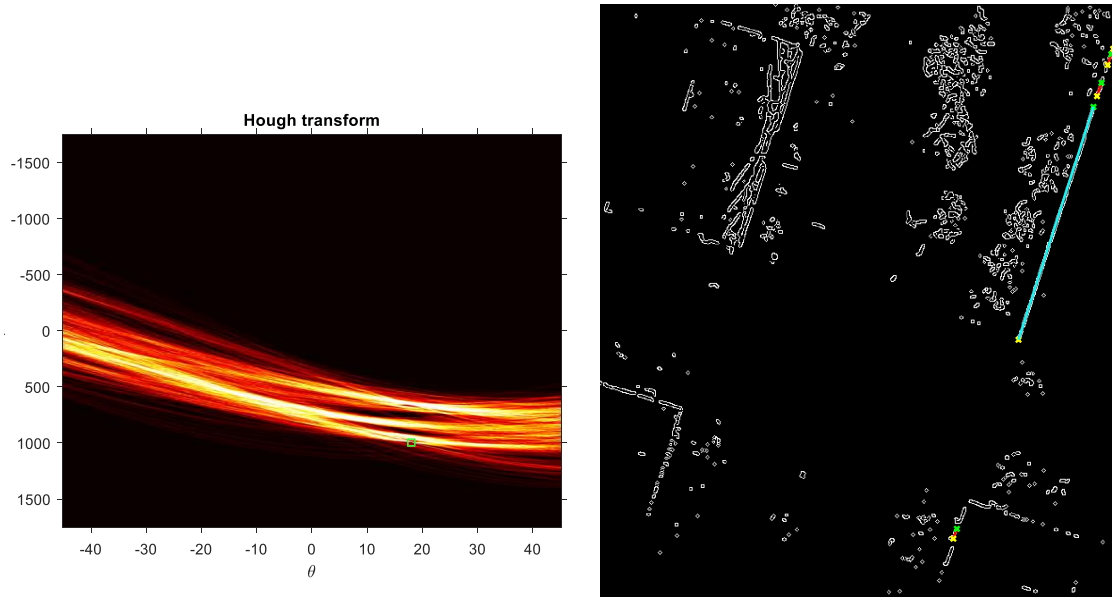


Figure 7.6: *Performing Canny edge detection on binary image*

Hough transformation

In order to get the parameters of the significant edges, the image needs to be transformed to (θ, ρ) parameter space, where ρ is the length of the normal of the line, and θ is the angle between x-axis and the normal of the line. In this case the equation of the line is: $\rho = x * \cos(\theta) + y * \sin(\theta)$. As a consequence in this parameter space, lines are points, and points are sine curves.



(a) *Performing Canny edge detection on binary image*

(b) *The most significant line in the binary image belonging to θ parameter*

Figure 7.7: *Hough space detection of the most significant line*

Fig. 7.7a shows the edge detected binary image in Hough space. Where most sine curves intersect, there is the most significant line (marked with green square). Fig. 7.7b shows the line to which the θ parameter belongs.

Inverse rotation of the image

As it has been already mentioned, it is only the angle of rotation that is required for the alignment of the coordinate system. The output of *Hough* transform is the θ parameter. As θ is the angle between the normal of the line, and the x-axis, the original angle of rotation is $\alpha = -\theta$, but for the inverse transformation $-\alpha = \theta$ is needed. This result in the following inverse transformation matrix:

$$\mathbf{R_o^{-1}} = \begin{bmatrix} \cos(-\alpha) & -\sin(-\alpha) \\ \sin(-\alpha) & \cos(-\alpha) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} = \mathbf{R_o^T} \tag{7.5}$$

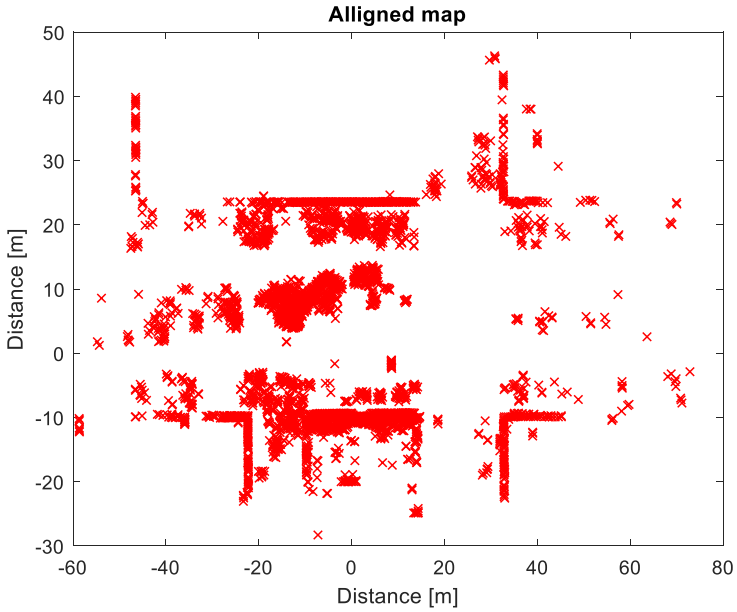


Figure 7.8: *Alignmnet of bird's-eye view map Fig. 7.4*

Fig. 7.8. depicts the aligned map, so in the further steps the parking places are to be detected parallel with, or perpendicular to the x axis.

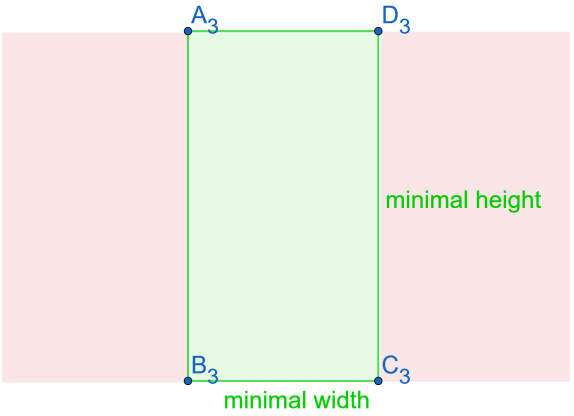
7.3 Recognition of adequate parking spaces

Section 7.2 descriped the preprocessing of the raw data arriving from the LiDAR sensor, including coordinate transforms for orientational corrections, ground segmentation, reduction of the 3 dimensional problem to a 2 dimensional one and the final alignments before parking space recognition. This section presents a method of the parking space detection, based on only LiDAR measurements. All further image processing methods are performed on the aligned bird's-eye view map that can be seen in Fig. 7.8. For the further methods, only the positive x coordinates are taken into consideration, as the vehicle is moving along the x axis, and circling around the possible parking zones. This makes faster execution possible, and decreases the redundancy of the detection.

7.3.1 Searching for adequate sized parking spaces

One of the main purposes of the 2nd subsystem is to find an adequate sized parking space for the vehicle. The most common types of the parking spaces are *parallel* and *perpendicular* parking. For both types, the algorithm is the same, but the parameters are different. The parameters of the algorithm are the dimensions of the vehicle including additional safety distances.

Fig. 7.9 depicts the parameters of the vehicle that are required for the parking space detection. It is important to note, that for both parallel and perpendicular parking, the



(a) *Perpendicular parking parameters*

(b) *Parallel parking parameters*

Figure 7.9: *Minimal sizes of different parking types*

width and *height* parameters might not be the same, and in reality these are usually of different values.

Scanning the map

The manifest method of searching an adequate sized parking space is by scanning the map along the x axis. As the required *width* and *height* parameters of the parking types are known, it is sufficient to create scanning-boxes that span the cordinate system along y axis and have the same *width* as needed for the parking.

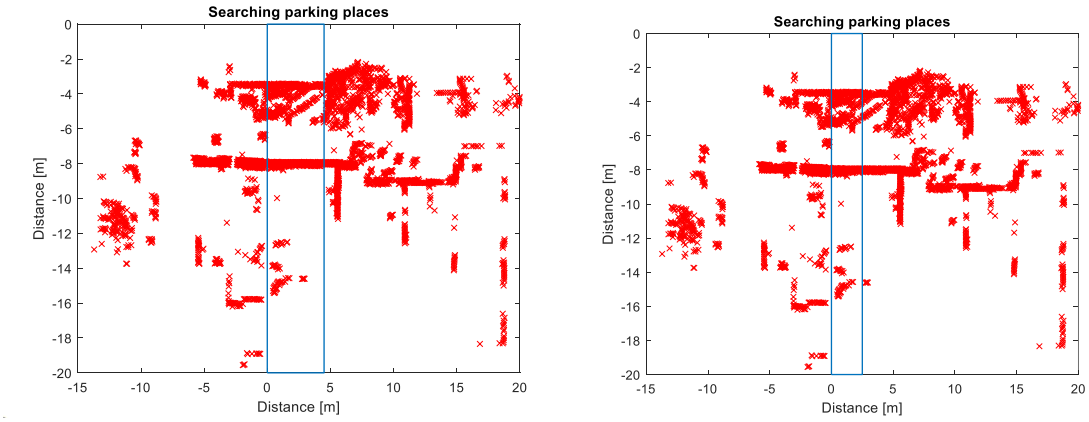


Figure 7.10: *Left - parallel parking space scanning,*
Right - perpendicular parking space scanning

After scanning the map along the x axis (see Fig. 7.10), all the possible bounding boxes can be determined that represent possible parking spaces. These bounding boxes have the same *width* and *height* as the corresponding parking type.

An adequate sized parking space must meet only two requirements:

- No obstacle should be found within the bounding box of the parking space
- The parking space should be reachable along y axis

The aftermath of these requirements is that two sets of bounding boxes are created for both parallel and perpendicular parking.

7.3.2 Grading of parking spaces inspired by fuzzy logic

Section 7.3.1 described the method of finding adequate sized parking spaces. The algorithm only took the car parameters and the safety distances into consideration. It is foreseeable that with this method, the autonomous vehicles would park in the middle of the road, or would overhang the parking spaces. When finding a parking space it is not enough to find a parking space of adequate size, but several aspects, such as the alignment with the nearby cars or the distance measured from the actual location etc, should be taken into consideration.

To satisfy all these aspects we present a method of grading the parking spaces inspired by fuzzy logic. The presented method gives a *quality factor* of the parking spaces in the interval of $[0, 1]$.

The method presented can handle multiple aspects, that influence the quality of a parking space. In this thesis two quality influencing factors are defined. One for the alignment with the nearby vehicles, and one for the distance needed to get to the parking place from the actual location of the vehicle. In order to obtain these quality factors, grading functions are introduced.

Grading the alignment

The alignment of the vehicle to the nearby vehicles is an important quality factor of a parking place. Sometimes it is not possible to park properly, without an overhang, to a parking space, as obstacles might be in the way or for other reasons. In order to detect these cases, it should be checked, whether the vehicle can be aligned with the nearby vehicles. This means, that for parallel parking the vehicle should be aligned with the vehicle in front and the vehicle at the back. In case of a perpendicular parking the vehicle should be aligned with the left and right side vehicles. Let's call this case the *perfect alignment* (see green bounding box in Fig. 7.11). By determining the exact coordinates of the perfect alignment,

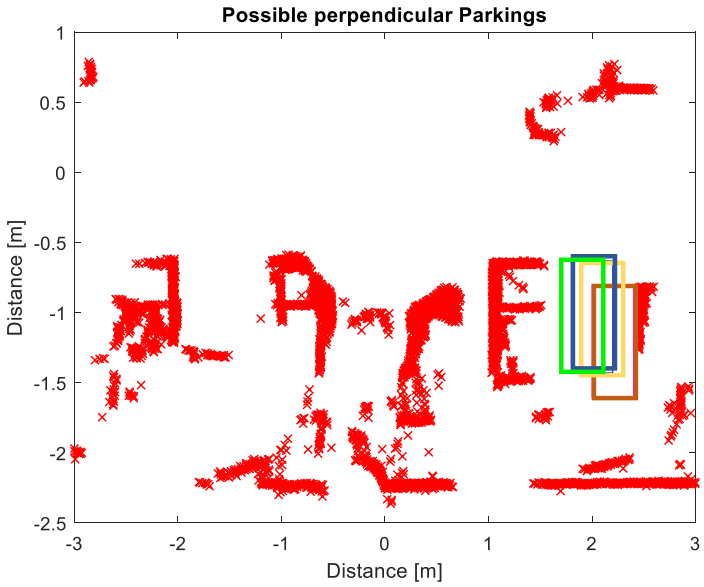


Figure 7.11: *The perfect alignment is marked with a green bounding box, all other depicted bounding boxes get a lower alignment point*

a distance can be measured between this perfect alignment and the best possibly reachable alignment, that is influenced by the obstacles occurring in the parking place.

If due to any reasons the vehicle can not be aligned with the nearby vehicles, a distance can be measured along the y-axis from the perfect alignment (see green bounding box in Fig. 7.11). By assigning a function (in this case a triangular function) to this measurement (see Fig. 7.12), the *quality of the alignment* can be determined.

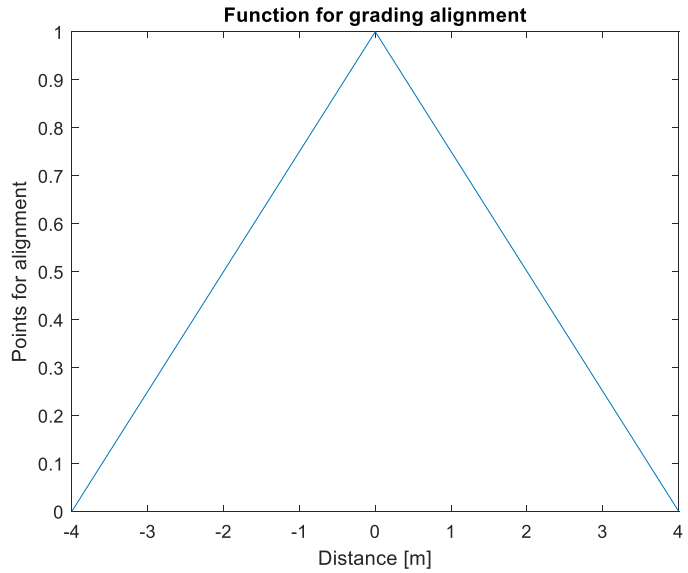


Figure 7.12: *Quality of alignment is provided by a triangular function*

The grading function shown in Fig. 7.12 has the following properties:

- The center of the function is always 0, as the input is the distance measured from the perfect alignment
- The distances from which the quality of the parking is 0, meaning the total lack of alignment, can be chosen arbitrarily

Grading the distance between the actual location and the parking place

The goal of the whole autonomous parking system is to find a parking space as fast as possible. However parking places that are detected too close to the vehicle have a higher chance of being middle road parking places, which is undesirable, although it does not mean that too close parking places should be excluded.

The grading function, that is introduced for grading the parking place based on the distance measured from the actual location, is a Gaussian function. Where the center of the function if a hypothetical optimum. This optimum should be the average of the distances measured between the center of the road and the center of the parking places.

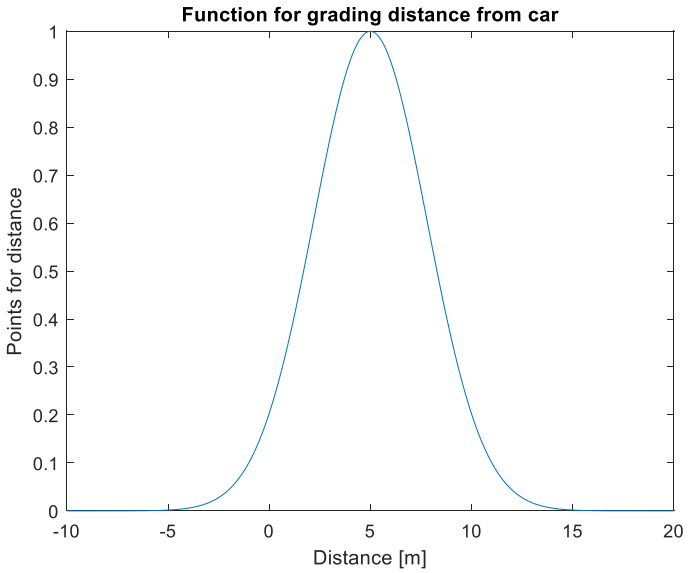


Figure 7.13: *Quality of distance measured from the vehicle is provided by a Gaussian function*

The function shown in Fig. 7.13 has the following properties:

- The center of the function (μ) should be the average of the distances measured between the center of the road and the parking places. This is the hypothetical optimum
- The σ parameter can be chosen arbitrarily, but the range of the LiDAR or any sensor that has been used should be taken into consideration

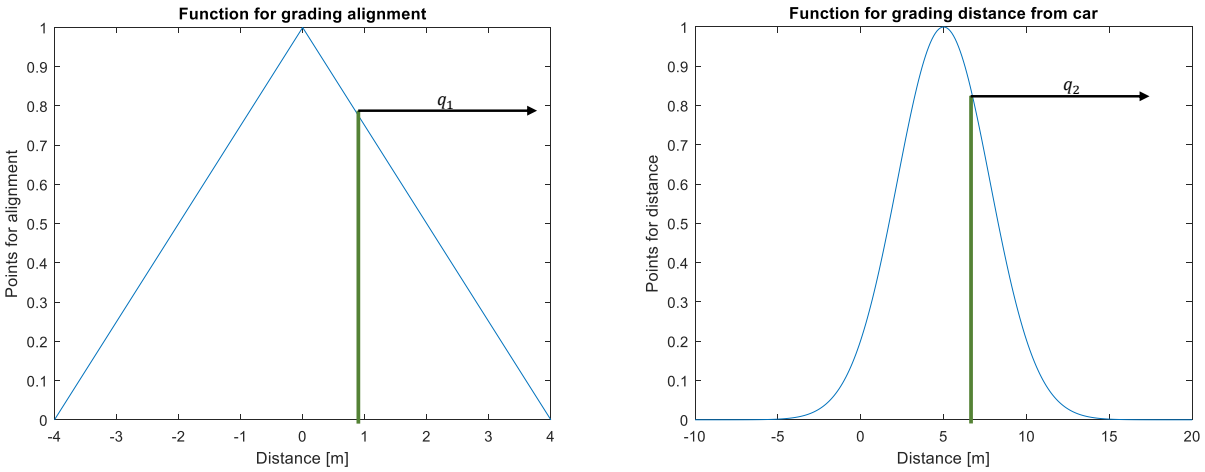


Figure 7.14: *The measured inputs of the funcitons are the green lines (crisp input), and the outputs of the functions are* q_1 , q_2

Getting the final grade

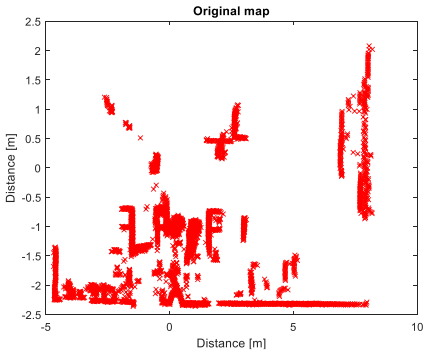
In this thesis 2 quality factors have been introduced. One for grading the alignment to the nearby vehicles, and one for grading the distance measured from the actual location. It is possible to create many grading functions of different quality altering aspects, such as the cost w_2 presented in Section 2.1.

All of these grades are scaled between $[0, 1]$, and if any of these scores are 0, the final score should be 0. As a consequence a simple arithmetical product of these quality factors can give the final score.

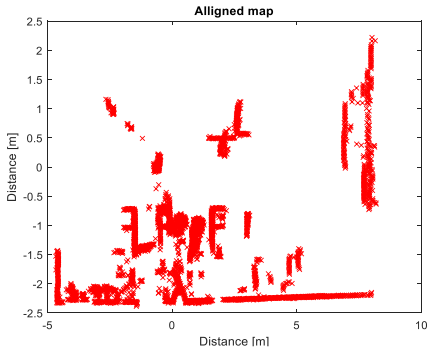
Let's suppose that we have n quality factors (now $n = 2$). Let q_i denote the output of the i-th grading function (see Fig. 7.14), where $i = 1...n$. The final grade q can be determined with the following formula:

$$q = \prod_{i=1}^{n} q_i \tag{7.6}$$

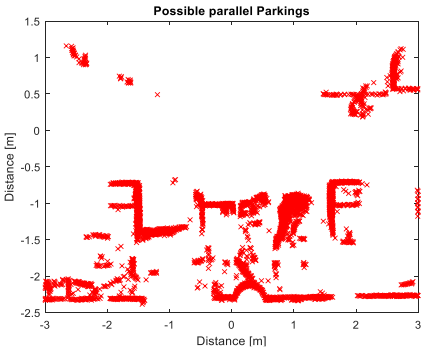
7.4 Simulation results



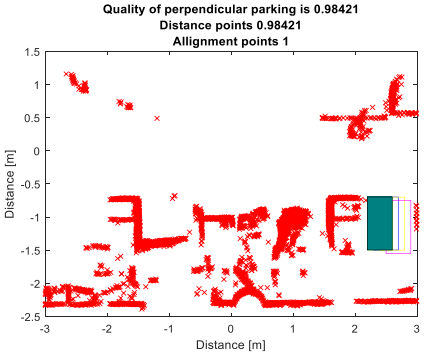
(a) *Bird's-eye view map created from LiDAR data*



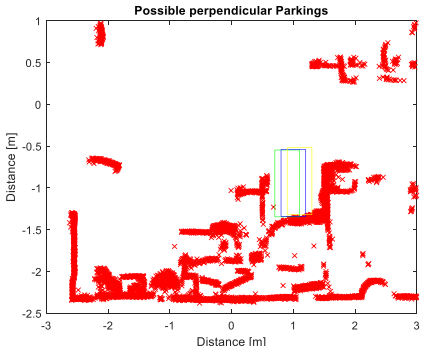
(b) *Aligned map of the simulated parking zone*



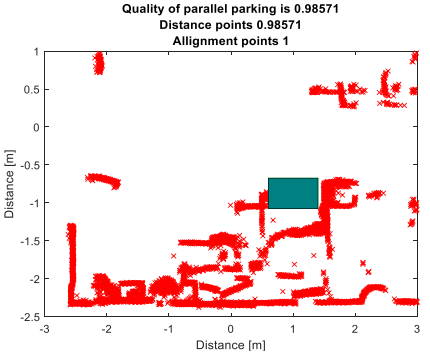
(c) *No parallel parking zones were found during this measurement*



(d) *Possible perpendicular spaces are marked with colorful bounding boxes. The one with the filled green color is the best one*



(e) *Possible perpendicular parkings found during the second measurement*



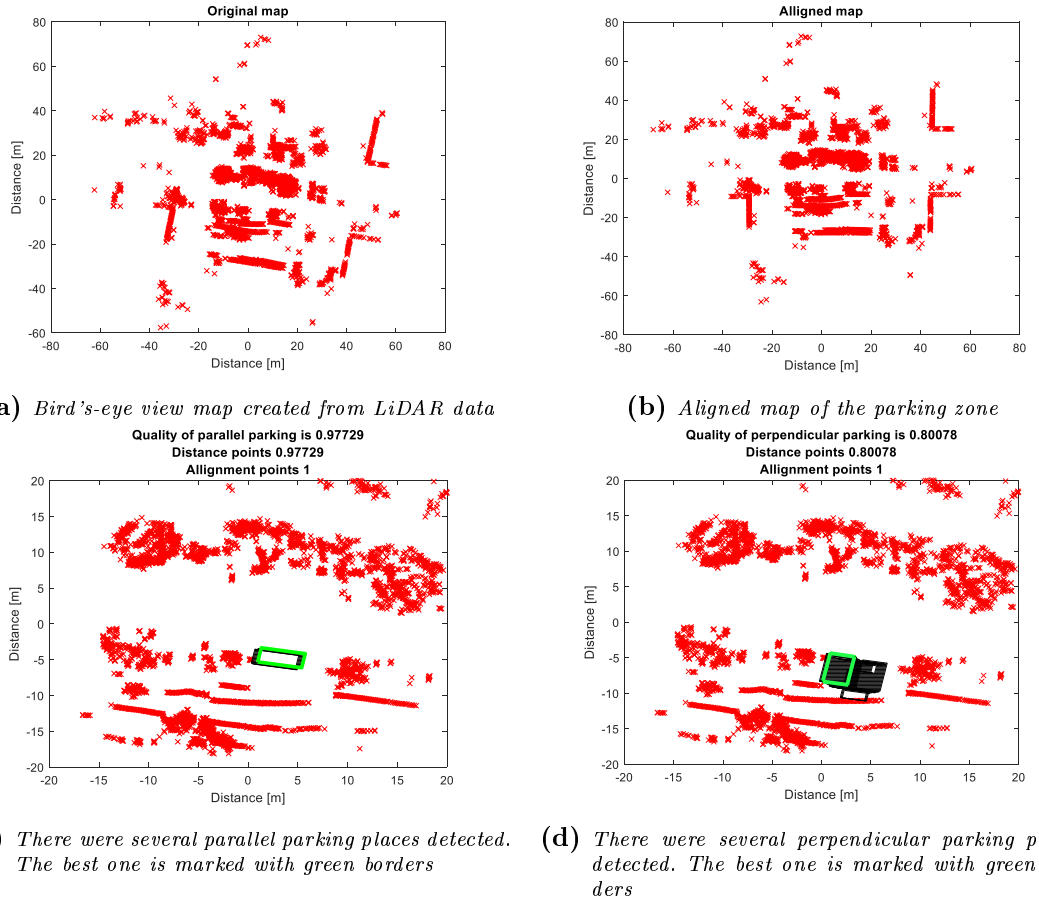
(f) *Possible parallel parkings found during the second measurement*

Figure 7.15: *Measured data in a simulated parking zone*

Fig. 7.15a-7.15f show an example of a set up environment, where a room was simulating the parking zone. Chairs and desks were arranged in specific patterns, so that the LiDAR could detect free spaces as possible parking places. The simulation results show, that due to the physical parameters of the vehicle, and the obstacles found in the map, it was not possible to find an adequate parallel parking space (see Fig. 7.15c). The algorithm found four possible perpendicular parkings (see Fig. 7.15d), from which the box filled with green color was the best one. The best parking place that was found during this scenery received ~ 0.98 points for the distance, and 1 for the alignment.

Fig. 7.15e-7.15f show another measurement, where the best parking place that was found, was a parallel parking place with ~ 0.985 points. In this measurement perpendicular parking places were found too, but the alignment was not as successful as in case of the parallel parking.

After the success of the simulations in a room, the algorithm was tested in a set of data provided by *Velodyne Inc.* The measurement provided by the company was taken place in a busy street, with real traffic and pedestrians. The results of the testing can be seen in Fig. 7.16.



(a) *Bird's-eye view map created from LiDAR data*

(b) *Aligned map of the parking zone*

(c) *There were several parallel parking places detected. The best one is marked with green borders*

(d) *There were several perpendicular parking places detected. The best one is marked with green borders*

Figure 7.16: *Testing the algorithm on a set of data provided by* Velodyne Inc

Fig. 7.16 shows the testing of the algorithm on a real set of data. The results show, that the algorithm found the parallel parking to be the best one, as the perpendicular parking was too far from the middle of the road, which could have led to a possible sidewalk parking.

Further testings of the algorithm on a set of data, provided by *Velodyne Inc* can be seen in a video linked in the bibliography. [21]

Chapter 8

Conclusion and future work

In this thesis methods were presented for parking lot exploration and parking space detection. The literature provides methods of parking space detection, in which multiple sensors are installed in parking lots. This fact is the most serious disadvantage of these methods, because developing systems like them can be very expensive and time-consuming. Our approach in comparison, has provided a method in which a vehicle is equipped with sensors in order to be able to drive autonomously and detect the free parking spaces. Thus, this approach seems to be the more practical way to create autonomous parking systems.

The first presented method for parking lot exploration is the trapezoidal cell decomposition. This method decomposes the map to polygonal cells and then, a traversal can be created knowing the adjacency matrix of the cells. The decomposed map is made up of several cells of big areas. When visiting these cells, it is possible that the planned path does not go through the whole cell.

The solution for this problem can be the modification of the cell decomposition: decompose the map along both x and y axes. In this case the intersections of the decomposed cells are the final cells of the method. These cells have smaller areas and each cell has only 4 neighboring cells, so handling the adjacency matrix becomes easier. This advantage leads to another method for creating the traversal of the cells. The undesirable reversing can be eliminated by forbidding reversal when it is possible. In case of dead ends reversing cannot be eliminated, as the vehicle must go back to the cell, from which it came. This cell traversing method gives the opportunity to assign preferences to the passing directions.

A more effective way to assign preferences to the directions is based on wavefront algorithm. This time an initial and goal cell is needed to be chosen. It is a manifest idea to choose the initial cell as the goal cell, too. Then preference values are assigned to the cells, based on the distance from the initial/goal cell by 4-neighborhood. This method provides a traversal, in which cells are revisited less often. It is possible to avoid reversing the same way, as presented before.

The biggest disadvantage of the presented cell decomposition based exploration methods

is that a polygonal map is required for them. Voronoi diagram based exploration provides a solution for this problem, as it works in case of a general map.

Voronoi diagram based method, in comparison to the trapezoidal decomposition, does not require the environment to be polygonal. This means that the presented method gives a solution for the exploration, in general environments. The biggest disadvantage of the method is the fact, that a graph created from the Voronoi diagram might not contain a *Hamiltonian* cycle, and in addition to this, the decision whether it has one, is an *NP*-hard problem. This thesis discusses several methods of the graph traversal including a simple airline-distance based method, Breadth-first search and Depth-first search algorithms. As these methods do not take the cost function into consideration, we have implemented a genetic algorithm, that is trying to minimize the objective function expressed with the total cost of the path. Voronoi diagram based method gives the exploration path of the parking lot. When the traversal of the path begins, a LiDAR sensor attached to the vehicle starts detecting the adequate parking places.

The second subsystem of the autonomous parking system is the parking space detection. This subsystem uses a LiDAR, attached to the vehicle, to scan the environment. The subsystem performs the necessary alignments that are required, due to the fact, that the vehicle is moving along the planned path, during this process. Deriving from the vehicle parameters, the safe-parking principles and several other requirements for the adequate parking places, a grading of the parking places inspired by fuzzy logic is also presented in this thesis. The presented method gives the possibility to take as many constraints, requirements, and preferences into consideration as the developer or the driver wants.

This thesis also includes simulation of the parking zones, where the second subsystem was tested. Furthermore, after the success of simulations, the subsystem was tested on a set of data provided by *Velodyne Inc.*

In the actual state of development, most of the algorithms were tested in simulations. The future work will include the tests in real environment, involving an autonomous vehicle, equipped with an *AutoBox* and a LiDAR sensor. Future work also includes improving the algorithms to be able to plan the exploration of multi-storey car parks.

Acknowledgement

We are really thankful to our supervisor, Gincsainé Dr. Szádeczky-Kardoss Emese for her coordination, kind guidance and encouragement.

Bibliography

[1] Faheem, S.A. Mahmud, G.M. Khan, M. Rahman, and H. Zafar. A survey of intelligent car parking system. *Journal of Applied Research and Technology*, 11(5):714 – 726, 2013.

[2] Mikael Fernstrom Muftah Fraifer. Designing a smart car parking system (poc) prototype utilizing CCTV nodes: A vision of an IoT parking system via ucd process. *Advances in Science, Technology and Engineering Systems Journal*, 2(3):755–764, 2017.

[3] Fadi Al-Turjman and Arman Malekloo. Smart parking in IoT-enabled cities: A survey. *Sustainable Cities and Society*, 49:101608, 2019.

[4] Circontrol, mobility and emobility solutions. URL: https://circontrol.com.

[5] Ford Active Park Assist. URL: https://www.ford.com.au/technology/active-park-assist/.

[6] E. Galceran and M. Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61(12):1258–1276, 2013.

[7] I. Noreen A. Khan and Z. Habib. On complete coverage path planning algorithms for non-holonomic mobile robots: Survey and challenges. *Journal of Information Science and Engineering*, 33:101–121, 2017.

[8] S. LaValle. *Planning Algorithms*. Cambridge: Cambridge University Press, 2006.

[9] M. A. Akkus. Trapezoidal cell decomposition and coverage. Middle East Technical University, Department of Computer Engineering.

[10] Ch. Harris and M. Stephens. A combined corner and edge detector. In *Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[11] S. Mukhopadhvay V. J. Lumelsky and K. Sun. Dynamic path planning in sensor-based terrain acquisition. *IEEE Trans. on Robotics and Automation*, vol. 6(no. 4):462–472, 1990.

[12] Anna Barbara Ádám. Nonholonomic motion planning for an increment-dimensional path planning method, Thesis. Budapest University of Technology and Economics, 2018.

[13] Th. Fraichard and A. Scheuer. From Reeds and Shepp's to continuous-curvature paths. *IEEE Transactions on Robotics*, vol. 20(no. 6):1025–1035, 2004.

[14] Mathworks. *Image Processing Toolbox: User's Guide (R2006a)*, 2016.

[15] S. Eddins. Exploring shortest paths. MATLAB Central Blogs. Published with MATLAB® 7.13. Posted by Steve Eddins, November 2011.

[16] Márton Szemenyei. Számítógépes látórendszerek jegyzet. Budapest University of Technology and Economics, 2019.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[18] Béla Lantos. Fuzzy systems and genetic algorithms. 10 2019.

[19] Velodyne LiDAR Inc. *VLP-16 User Manual*, 02 2019.

[20] Elephant Insurance. Self-driving cars are closer than we may think, 2017.

[21] Tests of the parking space detection on sample data. URL: https://www.youtube.com/watch?v=Z%5FV0HgxogvM.