Navigation on an implicit Voronoï diagram: experimental evaluation and implementation issues

Christian Gehring, Yves Pilat, Cédric Pradalier, François Pomerleau and Roland Siegwart

Abstract Obstacle avoidance and safe exploration are very important subjects in autonomous mobile robotics and cornerstones for most high level control algorithms. Victorino et al. introduced in [6] a powerful navigation strategy especially for small robots when low computational effort is an issue. The approach does not need to calculate the Voronoï graph explicitly and does not require any a-priori knowledge of a map. The sensor-based control framework depends only on laser scans and includes obstacle avoidance by its nature of design. In this paper we show an implementation of their work on a real world platform and discuss the problems we encountered. Moreover we show some additional approaches needed to complete the safe navigation task and discuss our results obtained in simulations as well as in real world environments.

1 Introduction and Motivation

In small autonomous mobile robots such as micro-helicopters, computational effort, memory storage, energy consumption and space for sensors are additional challenges of the navigation problem. The energy supply and space limit lead to the logic conclusion that as few sensors as possible should be used for navigation. At a first glance, a camera is more convenient than a laser range finder. This is because it is smaller and consumes less energy. However the pre-processing of the signals of a computer vision system needs a lot more computational effort. Thus, the whole system namely the sensors and the navigation technique must be considered when designing a mobile robot with such limitations. There are mainly two concepts of navigation, both have their own benefits and drawbacks. A map-based (or model-based) approach implies the well-known simultaneous localisation and mapping (SLAM) problem which has been solved using different techniques. Map-based navigation...
approaches have many key advantages. However, they require more memory and computational power. This is due to the heavy computational load involved in map-based path planning. A behaviour-based navigation system interacts directly with its environment and thus avoids explicit path planning. The main disadvantage is that it consists of a set of behaviours that are programmed for a specific environment and goal, and therefore cannot be easily applied to other places and targets.

This paper follows-up on work presented by Victorino et al. [6, 7]. In these contributions, a control strategy was introduced that follows the Voronoi graph without computing it explicitly. This method provides a powerful set of control laws to explore the unknown environment without using any a-priori map. Following the Voronoi diagram is a very safe strategy for navigation since the diagram is always as far away as possible from the two nearest obstacle. Unfortunately the computation of the diagram is very costly and requires an a-priori map. However, if the control system guarantees that the robot motion occurs on the diagram, then navigation becomes a 1D problem, simplifying simultaneously the problems of localisation, exploration and topological mapping.

In this paper we report on the practical challenges raised when implementing the work of Victorino et al. on a robotic platform in an indoor environment. We hope that this article will be a good starting point for the reader intending to implement [6, 7] on a real robot equipped with a laser range finder. However, we focus here only on the navigation part, leaving the topological mapping for a future publication.

We first tested our approach of the navigation algorithm in MATLAB to evaluate if it is applicable to a micro-helicopter. The simulations indicated that some assumptions may not hold in the real world. Therefore, the experiments we present here are based on a wheeled robot.

Sect. 2 describes our implementation of the work of Victorino et al., Sect. 3 presents our experiments and results and Sect. 4 discusses the raised challenges and our conclusions.

2 Technical Approach

The Implicit Voronoi Navigation strategy requires to implement a set of control laws or behaviours that will be merged based on sensory measurements. Hence, we defined a set of operating modes that correspond to the navigation tasks as described in [6, 7] and additional functions that are required for safe autonomous exploration. The navigation algorithm switches between the following operating modes: the robot starts from an arbitrary position and orientation, generally not on a Voronoi graph. The first task is to reach and align its heading direction with the nearest Voronoi edge, by controlling the distance to the two nearest objects towards equality (mode 0 in Fig. 1). As soon as a third object is within some predefined range, the robot changes its mode and stops at the so-called bifurcation point, uniquely defined by the fact that the three closest objects are at equal distances (mode 1 in Fig. 1). At this point, the robot decides which Voronoi branch it should follow subsequently,
Fig. 1 The robot depicted as a circle with indicated heading direction follows the Voronoi branch (green) and switches between the following operating modes: reaching and following the Voronoi edge (mode 0, blue), stopping on a bifurcation point (mode 1, red), leaving a bifurcation point (mode 2, orange) and avoiding a dead end (mode 3, cyan) and rotates on the spot to align itself with that branch (mode 2 in Fig. 1). Finally, the edge following mode is re-activated and the robot continues exploring. Note that moving on the Voronoi graph does not ensure in itself a safe navigation. On one hand, the dimensions of the robot are not considered, which might be a problem in narrow corridors. On the other hand, a Voronoi branch can lead to a dead end. For instance, a L-shaped corridor has a Voronoi branch equivalent to the angle bisector of the corner that leads to a dead end. To prevent the robot from such dangerous situations, a security loop has been implemented that monitors such situations and immediately reacts on them by turning 180° and flying back for example (mode 3 in Fig. 1). This is traditionally solved by performing a Minkowski expansion of the environment.

As described in [6], the control algorithms of the first two operating modes, namely ‘reaching and following a Voronoi branch’ and ‘stopping on a bifurcation point’ are formulated in terms of task functions. If the output of such a task function is zero, then the robot has reached the goal, i.e., it is located on the Voronoi branch or on a bifurcation point, respectively. The robot’s position \( \mathbf{r}(t) \) in the world frame belongs to the Voronoi diagram if the following navigation function is equal to zero (cf. Eq. 10 in [6]):

\[
e_1(\mathbf{r}(t)) = \left( \frac{\delta_1 - \delta_2}{\theta_1 - \theta_2} \right),
\]

(1)

where the two points \((\delta_1, \theta_1)\) and \((\delta_2, \theta_2)\) described in polar coordinates in the robot’s frame correspond to the closest points of the closest objects with respect to the robot as shown in Fig. 1. If the robot is on a Voronoi bifurcation point, the following navigation function is equal to zero (cf. Eq. 21 in [6]):

\[
e_3(\mathbf{r}(t)) = \left( \frac{\delta_1 - \delta_2}{\theta_1 - \theta_2} \right),
\]

(2)

where the point \((\delta_3, \theta_3)\) corresponds to the third closest object. The basic idea of the task functions is to move the robot in such a way that the outputs of those
functions converge to zero. This objective is accomplished by a feedback controller that considers the laser measurement and computes the command velocities $\tau_C = (v_x, v_y, \omega)^T$, which are then achieved by an appropriate cascade controller as will be detailed in the following paragraph.

The design of the control laws is directly derived from the appropriate navigation functions. A mapping of a generic navigation function $e$ to the command velocities $\tau_C$ is deduced by considering the total derivative of the navigation function with respect to time $t$ (cf. Eq. 27 in [6]):

$$\frac{de}{dt} = \frac{\partial e}{\partial r} \frac{dr}{dt} + \frac{\partial e}{\partial t}$$

Due to $\frac{dr}{dt} = \tau_C$ and aiming at an exponential convergence of the task function $\frac{de}{dt} = -\lambda e$ with a tuning gain $\lambda$, the desired command velocities are given by

$$\tau_C = \left( \frac{\partial e}{\partial r} \right)^{-1} (-\lambda e - \frac{\partial e}{\partial t})$$

(cf. Eq. 28 in [6]). Thus, a navigation function $e$ just needs to be differentiated with respect to $r$ and $t$ and plugged in (4) in order to obtain the control law. For the operating mode 0, the navigation function $e_1$ must be extended in such a way that the robot follows the Voronoi edge at constant speed (cf. Eq. 20 in [6]). To calculate the control laws from the navigation tasks some further steps must be considered. The interested reader may consult the original work of Victorino et al. [6, 7] for more details.

In case of a wheeled robot with a non-holonomic constraint that does not allow lateral velocities, a further problem has to be overcome. In [6], they propose a method that extends the robot state with a virtual degree of freedom that represents the mounting of the sensor on a pan with motorized axis. We applied another approach that maps the lateral velocity $v_y$ to the heading velocity $v_x$ and yawing velocity $\omega$ (see [1] for details), but then the navigation function of the first mode as described in Eq. 20 in [6] has to be slightly adapted.
In conclusion, the feedback control laws are just simple matrix calculations involving matrix addition and multiplication and computation of pseudo inverses, which require low computational effort and memory usage.

The control laws take the distances and angles $(\delta_i, \theta_i)$ between the robot and the three closest obstacles into account, which have to be extracted from the environment first of all. This so-called pre-processing step is an essential part of the algorithm, which is hardly described in [6]. The preprocessing is computationally heavy, because in practice no obstacles are given, but laser scan points, i.e. the scan points have to be segmented to identify the objects.

Due to the assumption that most indoor environments consist of rooms and corridors that are described by simple geometrical shapes, such as polygons, the cross section of such an environment can be approximated by a set of segments. Since the cross section is given as a set of points measured by a 2D laser range finder, we use an efficient implementation of the Split-and-Merge algorithm from [4] to extract the segments. In conclusion, we simplified the search of the relevant distances for control by grouping the scan points to segments.

![Simulation in MATLAB. Left: the robot (blue) is following the Voronoi branch (green) in a T-shaped corridor. Right: the scan points (black), the extracted segments and the sectors used to classify the segments are depicted in the robot’s frame. The thin red lines visualize the relevant distances used for control.](image)

A further processing step is needed to choose the appropriate segments to compute the control laws. Just taking the three closest segments is not sufficient. If, for example, the robot starts in a corridor with an initial position that is closer to one wall than to the others and this wall is described by two segments cause of discontinuities, then the two closest segments belong to the same wall. This set-up entails a Voronoï branch leading through the actual wall and the robot might finally collide with the wall. Consequently, a further procedure is required that labels the segments to the objects. The assumption of a corridor-like environment allows to apply a simple approach. First, the closest point of each segment with respect to the
robot’s position is evaluated and its corresponding distance and angle to the robot’s orientation (heading direction) are calculated. Second, the segments are classified according to the angle between the robot’s heading direction and their closest point. Therefore, the field of the robot’s view is divided into three classes, i.e. sectors, as depicted in Fig. 3 on the right. Third, the closest segment in each sector is chosen based on the previously calculated distances to compute the control laws.

Near a bifurcation point the assumption of a corridor-like environment does not hold any more with the result that the closest objects change after every motion step, because the set of minimum distances and corresponding angles belong to another Voronoï branch. Thereby the computed command velocities entail strong oscillations of the robot motion and in the worst case, the robot never stops on the bifurcation point. A possible solution of this problem is to track the segments in operating mode ‘stop on a bifurcation point’, when the robot is arriving on a bifurcation point. When switching from operating mode ‘follow a Voronoï branch’ to operating mode ‘stop on a bifurcation point’, the tracker is initialized with the closest segments in the sectors, and the current pose of the robot obtained by the odometry sensors is memorised. The pose is needed to compute the transformation from the previous robot pose to the new robot pose, which is required to predict the segments described in the previous robot frame in the new robot frame. After having predicted the tracked segments from the previous time step, the error with respect to each segment is computed. The error is the sum of the squared difference between the distance between the robot and the predicted segment and the distance between the robot and the segment from the new laser readings and of their corresponding angles, respectively. Finally, the segment with the lowest error is identified as the new tracked segment. Since the assumption of a corridor-like environment is not fulfilled, the classification using sectors is disregarded in operating mode ‘stop on a bifurcation point’. Note that in case of a helicopter, odometry data is not available. A possible circumvention is the Iterative Closest Point algorithm that estimates the transformation matrix from two consecutive point clouds from the laser readings in an iterative way [5]. Hence, the only required sensor is a laser range finder.

The Voronoï diagram for a 2D environment is well defined and a complete exploration is guaranteed. Due to the fact that the shapes of real obstacles are not just simple geometric shapes, such as triangles or rectangles, there are a lot of bifurcation points in reality. It is not necessary to stop on every bifurcation point, since some Voronoï branches lead to areas that are not interesting to explore according to their size in comparison to the robot’s size. Moreover stopping on a bifurcation point consumes a lot of time. This issue results in an additional design part of the navigation system, namely a criterion to figure out whether the robot is near a bifurcation point and has to switch to mode ‘stop on a bifurcation point’ to reach and stop on the bifurcation point. There is no unique solution to this problem, but an optimal one that depends on the environment. In case of corridors approximated by segments in sectors, a threshold for the distance to the third object in front of the robot is feasible, which depends on the distances to the two other closest objects.
Furthermore, a criterion is also necessary to examine if the robot has stopped on a bifurcation point and the controller can switch to mode ‘find an exit branch’ (see [1] for details).

Once stopped on a bifurcation point, a new branch leaving this point must be found and the problem of being re-attracted to the same bifurcation point has to be negotiated. Our approach to find an exit is to virtually move the robot on a circle with a radius equal to the robot’s radius around the bifurcation point and to calculate the command velocities at 72 uniformly distributed points on that circle. If a point is close to a Voronoi branch, the magnitude of the control commands will be lower than at a point far away from any branch. Therefore, searching for the minimal control magnitude on this circle will lead to a new direction for the robot to leave the bifurcation point. Consequently, every Voronoi edge results in a local minimum of this mapping function. An exit branch can be chosen either based on a randomly pick or on an available map. As soon as a Voronoi edge has been chosen, the robot performs a yawing manoeuvre, i.e. it turns on the spot and orients itself according to the computed angle and switches back to mode ‘follow a Voronoi branch’.

The problem of re-attraction by the bifurcation point that the robot wants to leave is already solved by the classification of the segments using sectors since the segments lying in the back of the robot are not considered to identify a bifurcation point.

Most of the insights into the navigation methodology described in this section were gained from the simulations and experiments.

3 Experiments

The control framework was initially implemented in MATLAB in conjunction with a simple kinematic model of the robot and its sensor. At first, the robot was modelled as a point mass with an inertia to consider a helicopter, and later a non-holonomic constraint was added to consider the wheeled robot. The environment was described as an image to allow investigating the effect of the resolution of the laser range finder as well as to have a quick method to simulate different kind of maps. At the stage of the development of the navigation framework, neither computational effort nor memory usage nor time played a role. The focus was on the verification of the method and on detecting the weakness and the crucial part of the algorithm.

As soon as simulations showed positive results, the implementation was ported to a real robot, using C++ and the open-source Robot Operating System (ROS) [2]. The first goal of the experiments was to tune the parameters of the navigation controller to a certain environment, e.g. adapt the conditions to change between the different operating modes. Once acceptable results were achieved, the robot was tested in a more complex corridor to verify its behaviour.

In the subsequent sections we present the results of the experiments made with the real robot.
3.1 Experimental Setup

The robot depicted in Fig. 4 which was used for the experiments, is equipped with four wheels, two driven and two caster wheels. Two laser range finders provide distance measurements at 10Hz from 720 points on a horizontal plane, with 360° field of view, seven meter range and half a meter above the ground. In addition, odometry data is available from the wheel encoders. The desired command velocities computed by the navigation controller are converted to the desired rotational speed of the wheels by a cascade controller.

First experiments were made in an artificial environment with straight walls consisting of plasterboards or wooden boards to reduce the perturbations to a minimum. Furthermore only simple surroundings were tested, such as corridors and T- or L-shaped junctions. Such an artifical environment is depicted on the right hand side in Fig. 4. This kind of setup allowed us to test each operation mode individually and debug our code on the one hand, and to implement and test the different approaches to solve the problems mentioned in Sect. 2 on the other hand.

The second part of the experiments was to verify the behaviour obtained in the artificial environment in a real corridor with perturbations such as people walking by, short passages of walls not visible to the laser range finder or other perturbations to the laser range finder.

3.2 Results

In this section an overview of the problems and the results of the experiments is given. The detailed discussion and the explanation of the problems will be subject to Sect. 4.

The experiments showed a very good behaviour if the robot follows a branch in a simple corridor. The robot is able to regulate the error between the Voronoï
branch and its current position to zero. Thereby, the convergence rate can be tuned with a parameter. However, there is a case when the navigation method may fail. If the starting position is chosen so that the angle between the heading direction of the robot and the Voronoi branch is greater than 75°, the classification of the segment might fail and therefore the robot will not be able to align itself with the local branch. Nevertheless, if one stick to this limitation of the starting position a very good behaviour in straight corridors is ensured.

In a L-shaped corridor the detection of a bifurcation point as well as the method to find a new Voronoi branch to leave the bifurcation point can be tested. The robot follows the branch and stops on the bifurcation point as expected. As depicted on the right hand side in Fig. 5 the robots trajectory (blue) is not always perfectly aligned to the Voronoi branch (green). Furthermore if the robot has stopped on a bifurcation point, the robot might take the branch leading towards the corner as a new trajectory to follow. However this problem is not serious, because the safety function, as mentioned in Sect. 2 will eventually detect that this branch leads to a dead end. The function will interact, turn the robot 180° and let it then follow the same branch into the opposite direction until the robot reaches the last bifurcation point again.

In Fig. 5 different colours are used which are clarified in Tab. 1.

Table 1 Colours used in Fig. 5

<table>
<thead>
<tr>
<th>colour</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>orange octagon</td>
<td>robot in mode 1</td>
</tr>
<tr>
<td>red octagon</td>
<td>robot in mode 0</td>
</tr>
<tr>
<td>blue</td>
<td>trajectory of the robot</td>
</tr>
<tr>
<td>light green</td>
<td>Voronoi diagram</td>
</tr>
<tr>
<td>light green lines</td>
<td>extracted segments of sector 1</td>
</tr>
<tr>
<td>red lines</td>
<td>extracted segments of sector 2</td>
</tr>
<tr>
<td>blue lines</td>
<td>extracted segments of sector 3</td>
</tr>
<tr>
<td>thick black lines</td>
<td>segments taken to calculate the control input</td>
</tr>
</tbody>
</table>

Fig. 5 Visualisation of the experiment in a L-shaped corridor; left: the robot stops on the bifurcation point, right: the robot left the bifurcation area and follows the new branch

The classification of the segments into three different sectors has been tested using a T-shaped junction. The problem when a branch leads to a dead end does
not occur with this type of junction. Most of the T-shaped junctions with different widths of the single corridors were passed flawlessly by the robot. However, if the robot drives from the left to the right side (cf. right hand picture in Fig. 5) and the corridor perpendicular to the current one is much smaller in width, the algorithm sometimes fails to detect the bifurcation point and therefore the junction.

Experiments in real corridors showed passable results. On the one hand, the robot runs smoothly in corridors and is even capable in coping with people passing by, however on the other hand, the algorithm sometimes fails to stop on a bifurcation point. In Fig. 5 a test run in a real corridor is depicted. Fig. 6 clearly shows that the robot detected the second corridor correctly and changed its operating mode. The trajectory of the robot in blue corresponds to the operation mode 'follow the branch', whereas red stands for the operation mode 'stop on a bifurcation point'. The robot stopped on the bifurcation point as shown in Fig. 5 and will decide next, which branch it should follow. A few problems are visible in this figure: There are some points near the robot which are not represented by a segment. In this particular case those points belong to the frame of a glass door. If the robot follows the Voronoï edge leading to the small corridor, it might collide with the frame of the glass door. Additionally, the robot might not fit through the gap at the end of this corridor. However, this information is neither considered by the segmentation nor by the calculation of the Voronoï branch.

![Fig. 6 robot on the bifurcation point in a real corridor](image)

### 4 Conclusion

Implementing the algorithm of [6] highlighted the large difference that may occur between reality and theory. Even with a working simulation model in MATLAB the step from the simulation to a real world hardware and environment caused some problems. In this conclusion, we want to discuss the advantages and disadvantages of our approaches as well as the results of the experiments.
One source of the errors during the experiments has been the classification of the segments. The control laws strongly depend on the order of the three closest segments: just taking the three minimal distances is not sufficient, however. It is crucial to assign the segments to certain objects and take then the minimal distances to those objects. Our approach, as mentioned in Sect. 2, to classify the segments according to their relative angle with respect to the robot, is an approach that works, but has one big disadvantage: As the results in Sect. 3.2 showed, the control algorithm sometimes fails to detect all bifurcation points. Due to the strict range of each sector some segments could be assigned to a wrong object and therefore a corridor perpendicular to the current Voronoi branch might be ignored. Another problem occurs if the velocity demand in y-direction and the angular velocity cancel each other out. As mentioned in Sect. 2, the non-holonomic constraint has been solved by converting the velocity in y-direction to an angular velocity. With this approach it is possible that the mapping of the y-velocity leads to a negative yaw velocity, whereas the control laws demand exactly the same velocity but in the opposite direction. However this problem is solved by a dynamic weighting between those two demands depending on the operation mode. In the worst case a slightly biased movement will be noticeable for a short distance (cf. Sect. 3.2 and Fig. 5).

The main advantage of the navigation algorithm is that it is very simple and reliable. The calculation of the control laws depend only on the extracted minimal distances and their corresponding angles, no further information is needed. Once the parameters and the behaviours are adapted to the surroundings, a safe navigation in this kind of environments is ensured.

The control design is reactive, hence obstacle avoidance lies in the nature of this design. No additional low level controller is needed for this task. In addition the control algorithm can run with low computational effort: we managed to run the whole navigation algorithm stably at 10 Hz. In our opinion, the navigation method presented by Victorino et al. has a high potential for small autonomous robots. On that account the algorithm must be tuned in terms of computation time and the additional functions must be reduced to an absolute minimum. One approach to achieve this goal is to reduce the number of scan points, since the time of pre-processing depends strongly on the number of points. As showed in our experiments, we were able to obtain good results with only 720 points, in contrast to Victorino et al. who used 2000 scan points.

References


6 Appendix

In Sect. 6.1 the state of the work is outlined, whereas the code written in Matlab is described in Sect. 6.2 and the code written in C++ in Sect. 6.3, respectively. In Sect. 6.4 some recommendations for a future work or a possible improvement are proposed.

6.1 State of the Work

Section 6.1.1 shows how we implemented the control algorithm presented by Victorino et al. and how we solved the problems that occurred. In Sect. 6.1.2 and in Sect. 6.1.3, respectively, we discuss some additional improvements of the algorithm in terms of a more reliable design and a reduction of computational effort.

6.1.1 Issues Regarding the Work of Victorino et al. and our Solutions

This section discusses some issues regarding the work in [2] and our approaches to solve them. The navigation framework presented by Victorino et al. does not meet all requirements to safely navigate and autonomously explore the unknown environment. The subsequent paragraphs will go into the details, while Table 2 gives an overview of our implemented task functions.

<table>
<thead>
<tr>
<th>Operating mode</th>
<th>Task</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reach the nearest Voronoi edge from any point of the free space, corresponds to the task function $e_{\text{branch}}$ in [2], which is align the robot’s heading direction with it, and move ahead along modified in our implementation the Voronoi branch</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>reach and stop the robot on a bifurcation point</td>
<td>corresponds to the task function $e_{\text{BP}}$ in [2], which is modified in our implementation</td>
</tr>
<tr>
<td>2</td>
<td>find all Voronoi edges that leave the bifurcation point, decide which one to follow next, and turn the robot on spot to align its heading direction with the corresponding edge</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>turn the robot 180° on spot to avoid a dead end</td>
<td>not mentioned in [2]</td>
</tr>
</tbody>
</table>

Extracting the Set of Minimal Distances and Corresponding Angles for Control

In our point of view, the procedure of extracting the set of minimal distances and corresponding angles between the robot and the obstacles for control from the laser scans is the most important and difficult part of the algorithm. Victorino et al. modelled the obstacles as a set of polygons and proposed to perform a Hough transform or to use a polygonal approximation algorithm in order to obtain those polygons. We decided to apply the Split-And-Merge-Plus (SMP) algorithm that is described in [1] in details, because it is the most efficient way to extract segments from a laser scan according to [1].

How to extract the three minimum distances between the robot and the obstacles from this set of polygons is not mentioned in [2]. Just taking the minimum distances referring to the three closest segments is not working as first simulations showed immediately. If for example the robot starts in a corridor with an initial position that is closer to one wall than to the others and this wall is described by two segments cause of discontinuities, then the two minimum distances belong to the same wall. This setup entails a Voronoi branch leading through the actual wall and the robot might finally collide with that wall. As a result of this, a further procedure is required that labels the segments to the objects.
To keep the algorithm simple, we divided the polar range of the robot’s view into three sectors as depicted in Fig. 2 and choose from each sector the closest segment, except when the robot is arriving on a bifurcation point as described soon. Note that the angles defining the sectors are further tuning parameters of the navigation algorithm. Section 6.1.2 follows up other approaches to deal with the pre-processing of the point cloud.

The distance between the robot and a segment is either the distance from the robot’s position to one of its end points or the normal distance between the robot’s position and the segment line. The minimal distance of those three does not always define the Voronoï graph as a L-shaped corridor in Fig. 7 shows. When the robot crossed the inner corner of the corridor, the Voronoï diagram is defined by the inner corner point, but the minimum of the three distances of the inner segment is the distance to the line model of the segment. Hence, we defined a criterion to choose the minimum distance to the segment as follows: First, the coordinates \((xT,yT)\) of the tangent point, which lies on the intersection of segment and a circle with radius \(p_L\) and centre on the origin are calculated as follows:

\[
xT = p_L \cos \alpha_L, \tag{5}
\]

\[
yT = p_L \sin \alpha_L, \tag{6}
\]

where \(p_L\) is the distance and \(\alpha_L\) the angle of the line model describing the segment. Second, the two distances \(l_{TP1}\) and \(l_{TP2}\) between the tangent point and the end points of the segment and the distance \(l_S\) between the two end points are computed:

\[
l_{TP1} = \sqrt{(xT - x_{P1})^2 + (yT - y_{P1})^2} \tag{7}
\]

\[
l_{TP2} = \sqrt{(xT - x_{P2})^2 + (yT - y_{P2})^2} \tag{8}
\]

\[
l_S = \sqrt{(x_{P1} - x_{P2})^2 + (y_{P1} - y_{P2})^2} \tag{9}
\]

Third, the relevant distance for control \(p_S\) is chosen according to the following conditions:

\[
p_S = \begin{cases} 
p_{P1} & \text{if max}\{l_{TP1}, l_{TP2}\} > l_S \text{ and } p_{P1} < p_{P2}, \\
p_{P2} & \text{if max}\{l_{TP1}, l_{TP2}\} > l_S \text{ and } p_{P1} \geq p_{P2}, \\
p_L & \text{otherwise.} \end{cases} \tag{10}
\]
Tracking of Objects

The simulations in MATLAB showed that it is essential to track the segments when the robot is arriving on a bifurcation point in operating mode 1. Victorino \textit{et al.} only mention the tracking of the objects in section “VI. Experimental Results”, but do not explain why it is needed and how they realized it. The reason is that near a bifurcation point the corridor-like environment assumption does not hold any more and the closest objects change after every motion step, because every time the set of minimum distances and corresponding angles belong to another Voronoï branch. This results in strong oscillations of the robot motion and in the worst case the robot never stops on the bifurcation point.

We only track the segments in operating mode 1 when the robot is arriving on a bifurcation point, otherwise the closest segments in the sectors are chosen as input of the control laws. When switching from operating mode 0 to 1, the tracker is initialized with the closest segments in the sectors and the current pose of the robot obtained by the odometry sensors is memorised. The pose is needed to compute the transform matrix that describes the transformation from the previous robot pose to the new robot pose, which is needed to predict the segments described in the previous robot frame in the new robot frame. After having predicted the tracked segments from the previous time step, the sum of the errors with respect to each segment, which are the squared difference between the distance to the predicted segment and the current segment, and their angles, respectively, are computed. Finally, the segment with the lowest error is identified as the new tracked segment.

In MATLAB, the transform matrix is directly determined by the Iterative Closest Point (ICP) algorithm from François Pomerleau, which is available on \url{https://aslforge.ethz.ch/projects/icp/}. It estimates the transformation matrix from two consecutive point clouds from the laser readings in an iterative way. This approach allows to omit the odometry, which is not available on a helicopter. The ICP is tunable in terms of calculation speed, but which is inversely related to the accuracy of the transformation matrix. Both properties are an issue. The former is obvious and the reason for the latter is discussed in Sect. 6.1.3. We did not test the ICP algorithm on ROS, because the C++ code of the ICP algorithm was not available at that time and programming it by us would have been beyond the scope of this work.

Arriving on a Bifurcation Point

The Voronoï diagram for a 2D environment is well defined and a complete exploration is guaranteed. However, it sometimes makes no sense to stop on every bifurcation point. Figure 8 shows a typical example for such a case. There is a L-shaped corridor the robot should explore. If the robot follows the Voronoï branch by operating in mode 0 all the time, then the robot will follow the Voronoï edge as desired. The two induced bifurcation points (red) of the cupboard at the side of the right wall are ignored owing to the classification of those segments, i.e. all segments on the right side belong to the same sector. Furthermore, the bifurcation point at the corner of the corridor will also be ignored. The robot will turn inwards and as soon as the outer segment of the vertical corridor is classified as a segment of the side sector, it will be considered as a candidate for the new closest distance. In the worst case, i.e. the robot gets too near to the outer segment, the safety function that monitors the closest distances will interact and prevent the robot from crashing. There are cases where it is essential to follow edges leading to dead ends to completely explore the environment, because a further corridor
might not be seen from the middle of the corridor. Due to the fact that a real environment does not consist of nice and simple geometric objects, there are usually a lot of bifurcation points that are not interesting to stop on. Furthermore, the manoeuvre of stopping on all of them takes a lot of time. This issue results in an additional design part of the navigation system, namely the criterion when to stop on a bifurcation point.

In [2] a threshold for the distance to a third object is used to identify a bifurcation point, which is not further described. Our solution is to identify the third closest object using the sector approach as aforementioned and to use a dynamic threshold for the distance between the robot and the third object to decide when a bifurcation point occurs. If the distance $\delta_{O3}$ that belongs to the third object is smaller than the threshold $\delta_{\text{thresh}}$, hasBP = $\alpha_{\text{BP}}(\delta_{O1} + \delta_{O2})$, (11)

where $\delta_{O1}$ and $\delta_{O2}$ are the distances belonging to the first and second object, respectively, and $\alpha_{\text{BP}}$ is a tuning parameter, the robot switches to operating mode 1 and stops on the bifurcation point. In case of the controller version using the SMP, the distances refer to the distance between the robot and the segments.

When the robot is operating in mode 1, the robot stops automatically on the bifurcation point cause of the design of the control law. Hence, a criterion is needed to figure out whether it reached the bifurcation point in order to switch to the next operating mode. If the following inequality is fulfilled:

$$\max\{|\delta_{O1} - \delta_{O2}|, |\delta_{O1} - \delta_{O3}|, |\delta_{O2} - \delta_{O3}|\} \leq \delta_{\text{thresh}, \text{isBP}},$$

where $\delta_{\text{thresh}, \text{isBP}}$ is a tolerance of 0.05 m, the robot has arrived on the bifurcation point.

Leaving a Bifurcation Point

Once stopped on a bifurcation point, a new branch leaving this point must be found and the problem of being attracted to the same bifurcation point again must be negotiated. Victorino et al. do not mention how to proceed from a bifurcation point and how to overcome the problem of being re-attracted.

Our approach to find an exit is to virtually move the robot on a circle with a radius of 0.3 m around the bifurcation point and to calculate the command velocities at 72 uniformly distributed points on that circle. If a point near to a branch is taken to calculate the virtual control inputs, it will result in a lower magnitude of control command than at a point far away from any branch. Therefore searching for the minimal control magnitude on this circle will lead to a new direction and orientation for the robot to proceed. If a new orientation has been found, the robot performs a yawing manoeuvre, i.e. it turns on the spot and aligns itself with the chosen branch. The yawing manoeuvre is implemented as a PI-Feedback-Controller using odometry data. So far, no mapping is implemented. Only the global minimum is considered and no information about the directions of the branches is stored. When exploration is considered in conjunction with mapping, the algorithm can slightly be adapted to be able to save all information about branches leaving this bifurcation point.

The second problem was to not being re-attracted to the same bifurcation point. The only difference between arriving on a bifurcation point and leaving one is that in the first case the third segment lies in the front of the robot, whereas in the second case it is in the back. The problem of finding a third segment has been solved by the classification of the segments using sectors as aforementioned. The segments are assigned to one of three sectors, two orientated at each side and one in the front. This leads to the conclusion that leaving a bifurcation point without being attracted again is already solved with this classification. As soon as the robot is orientated in direction of the new branch, the segments on the right hand side and left hand side are considered to calculate the control laws and the third segment in the back of the robot does not have an effect.

Avoiding Dead Ends and Safety Issues

The navigation methodology assumes that the robot is a point with a heading direction, but a real robot has always three dimensions in space. Hence, the robot is not able to follow every Voronoï path, i.e. when a corridor is narrower than the robot, it must detect it and avoid it. In addition, some Voronoï branches lead to a dead end, e.g. in a corner. As a consequence, a safety system is needed that detects those critical environments and avoids them. The SMP version of the
navigation controller has a function that monitors the first two segments for control. If the line models of those segments intersect in front of the robot and the distances to the segments fall below a certain threshold, the operating mode is switched to mode 3 in order that the robot stops immediately, turns 180°, switches back to operating mode 0 and finally follows the Voronoï edge which it came from. Due to lack of time this safety mode has not been tested very well.

Computing the Control Laws

In [2], the following motion constraint is used to calculate the navigation function $e_2$ (cf. Eq. (16) in [2]):

$$F = X(t) - X_0 - V_d t = 0,$$

(13)

where $X(t)$ is the absolute position vector of the robot, $X_0$ the initial position, $V_d$ a constant desired velocity vector and $t$ the time. The desired velocity $V_d$ allows the robot to follow the Voronoï branch at constant speed. The problem in our case is that the absolute position of the robot $X(t)$ is not available due to the lack of accurate position measurement.

The gradient $g^T_s$ of the motion constraint $F$ w.r.t. the position $X(t)$ is calculated and projected into the controlled frame and is part of the control law. The goal of the calculation is to get an expression for the projected desired velocity w.r.t. the current direction of movement. The final equation for the navigation task $e_{\text{branch}}$ is then (cf. Eq. (20) in [2]):

$$e_{\text{branch}} = W^+ C e_1 + \alpha t (I_3 - W^+ W) g^T_s,$$

where the last part corresponds to the projected constant velocity as aforementioned. However, this approach is only applicable if the non-holonomic constraint problem is solved as proposed in [2].

We implemented another way to deal directly with this non-holonomic constraint. Furthermore we weakened the motion constraint and allowed a smaller velocity than the desired one if the robot is either not on the Voronoï branch or near a bifurcation point. This leads to the following new navigation task $e_{\text{branch}}$:

$$e_{\text{branch}} = W^+ C e_1 + \alpha g_s_{\text{own}},$$

(15)

where $g_s_{\text{own}} = (v_{\text{des}}, 0, 0)^T$.

The control law, i.e. the command velocities $\tau_C = (v_x, v_y, \omega)^T$, is then calculated as follows:

$$\tau_C = -\lambda e_{\text{branch}} e_{\text{branch}},$$

(16)

where $\lambda_{\text{branch}}$ is a tuning parameter. The non-holonomic constraint is considered as described in the next paragraph.

In addition, some of the equations in [2] needed to calculate the navigation task $e_{\text{branch}}$ are not consistent in terms of matrix dimensions. For instance, the motion constraint $F$ is a two dimensional constraint, which contains no information about the angular speed $\omega$. However, its derivative with respect to the position (cf. $g^T_s$ in [2]) is part of the calculation of $e_{\text{branch}}$, which needs a three dimensional vector as input.

Non-holonomic Constraint

In order to circumvent the non-holonomic constraint implied by the wheeled robot that does not allow lateral velocities, Victorino et al. virtually mounted the laser range finder on a pan motorized axis [2]. They extended the robot state by an angle $\beta$ to obtain a virtual degree of freedom. We accomplished another approach instead. The control laws output the command velocity vector $\tau_C = (v_x, v_y, \omega)^T$. Then the lateral velocity $v_y$ is just added to the yawing velocity $\omega$ using the functions below, so that the final command velocity vector is $\tau'_C = (v'_x, v'_y, \omega')^T$, and the calculations are

- in operating mode 0:
\[ v_x' = v_x \quad (17) \]
\[ v_y' = 0 \quad (18) \]
\[ \omega' = \omega + \alpha_{t0} \arctan \left( \frac{v_y}{v_x} \right), \quad (19) \]

- and in operating mode 1:

\[ v_x' = v_x \quad (20) \]
\[ v_y' = 0 \quad (21) \]
\[ \omega' = (1 - \alpha_{t1}) \omega + \alpha_{t1} \arctan \left( \frac{v_y}{v_x} \right), \quad (22) \]

where \( \alpha_{t0} \) is a constant parameter with value equal to 0.75 and \( \alpha_{t1} \) is a function of the velocities \( v_x \) and \( v_y \) as shown in Fig. 9.

Fig. 9 The function of the tuning parameter \( \alpha_{t0} \) depending on the heading and lateral velocity \( v_x \) and \( v_y \), respectively, which is used to circumvent the non-holonomic constraint

The non-holonomic constraint results in a problem when the robot tries to reach the Voronoï path from a starting position that is not on the Voronoï graph. Due to the fact that the navigation function tries to reach it on the shortest path without considering the non-holonomic constraint, the aforementioned hack implicates an overshooting behaviour. This problem does not exist for a helicopter that is able to move instantaneously laterally.

6.1.2 Other Approaches of the Pre-Processing of the Laser Readings

Victorino et al. explain neither how they extract the set of minimal distances between the sensor and the obstacles in detail, nor how they identify the relevant distances for control. There are several possible approaches as listed in Table 3.

<table>
<thead>
<tr>
<th>No.</th>
<th>Approach</th>
<th>Progress / Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identify objects by segments using the Split-And-Merge algorithm and identify the closest ones by means of sectors.</td>
<td>Works in MATLAB and C++ well in corridor-like environments as aforementioned.</td>
</tr>
<tr>
<td>2</td>
<td>The first approach, but instead of using only the newest scan points, the scan points from the last few scans are taken, i.e. from a transformation of the robot poses are exactly given, otherwise points belonging to a line describe a cloud and the SMP fails to identify the segment.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Closed polygons are laid around the scan points including half of the robot’s shape to make the walls smoother.</td>
<td>It is implemented in C++ and the polygons are extracted properly, but it does not work because of the selection of the closest points.</td>
</tr>
</tbody>
</table>
The points from the laser scan will be grouped with the condition that the change of the gradient between a point and the subsequent one in both direction must not be larger than a certain, maybe dynamic, value. Then the minimal distance to this group of points will be taken as candidate of the minimal distances.

The approach of modelling the objects as polygons or segments and using sectors to classify the objects in the environment in order to choose the relevant distances is working well in narrow corridor-like environments (cf. first approach in Table 3). However, as soon as the environment gets more complex, such as a hall or room as shown in Fig. 10, which cannot be described by connected straight lines, segments may not be the best choice. Since the relevant segments change after every small motion of the robot in such an environment, the computed velocities and heading direction ‘jump’ after every calculation step, which results in an oscillating robot motion. The robot may even be trapped in a region in the worst case if the criterion of a bifurcation point is never fulfilled.

Jumps in the robot motion also occur due to missing segments as a consequence of the limited range of the laser scanner, especially enhanced by undetectable glass, or limited field of view. The field of view might be limited either because the sensor does not have a field of view of 360°, or cause of occultation. Better results are achieved if the scan points of the last few scans are saved in a local map, and the SMP is applied to all of those points, instead to only the scan points of the last laser scan (cf. second approach in Table 3). This approach has been implemented in MATLAB and on ROS.

So far, the objects were modelled as segments, but there are other approaches that avoid the aforementioned problems of the segments. The main advantage of using segments is that the computational effort is acceptable. More complicated algorithms may give better results, but may be impracticable due to long computation time.

Another method is to lay closed polygons around the scan points, which are expanded by half of the robot’s shape. This Generic Polygon Clipper (GPC) algorithm, an implementation of the Polygon Expansion, allows to smooth the walls and to avoid small gaps between adjacent segments. The approach refers to the original idea of using anti-snakes. In computer vision, closed polygons, so-called snakes, are adapted in an iterative way, so that they resemble the shape of an object. The GPC algorithm was implemented on ROS and tested (cf. third approach in Table 3 and Sect. 6.3.5). The polygons were extracted properly, but the implemented choice of the closest points method did not lead to a working navigation controller. Due to lack of time, this problem was not solved.

Owing to the fact that each piecewise continuous curve represents an obstacle, the gradient of those curves is a useful criterion to label the scan points to objects. For example, if the gradient between to subsequent points in the polar coordinate system changes more than a threshold, which might be adaptable, there must be the end of the object. The minimal distance to an object is then defined as the minimum of the curve belonging to that object (cf. fourth approach in Table 3).
6.1.3 Reduction of the Number of Scan Points and Field of View

As mentioned in Sect. 6.1.2, a reduction of the number of scan points can lead to a smaller calculation time of the algorithm. However, it might be assumed that due to less points, less information will be available and thus the extracted segments might be less accurate than the ones calculated with more points. Furthermore, some short walls might be completely omitted since less points mean less resolution. The simulations in MATLAB have shown that no noticeable reduction of accuracy occurred, even with only one point per five degrees, i.e. 72 points in total. Nevertheless with such a low resolution the problem of omit short segments became serious. Due to lack of time, the reduction of the number of scan points were not investigated in the experiments in the real world.

A helicopter with a Hokuyo laser range finder was primarily considered as a real world platform to implement the algorithm on. This sensor provides data from a 270 degrees field of view with one point per degree. Hence in MATLAB the field of view is limited to this 270 degrees, i.e. the robot has a blind spot of 90 degrees orientated backwards. As a consequence of this, the robot may fail to stop on a bifurcation point. When the segments from the corridor the robot comes from are lost due to this blind spot, the aforementioned tracking can not work properly, so that a successful stop on that bifurcation point is not guaranteed. The simulation has shown some severe crashes and missed bifurcation points.

A possible approach to deal with this problem is to consider the points from the last few laser scans in terms of a local map (cf. second approach in Table 3). Therefore more scan points of the surroundings are available. Thus, the transformation matrices between the robot poses are needed to build the local map. If those matrices are not exact, the scan points of a line, for example, will be mapped to a cloud, which is a serious problem for the Split-And-Merge algorithm.

Another approach might be to let the robot spin around its z-axis with a constant velocity to avoid having a blind spot, however this leads to problems regarding the control and the stability of the helicopter.

In a nutshell, our simulations in MATLAB showed that a reduced number of scan points has no noticeable impact on the navigation, whereas the reduction of the field of view caused some serious problems. If the computational effort must be lowered the reduction of scan points might be a good point to start with. In case of a reduced field of view, a further procedure has to be applied to enable a complete field of view.
6.2 MATLAB Code

The first goal was to implement the algorithm in MATLAB to get familiar with it on the one hand, and to reveal the weakness of it on the other hand. Furthermore, suitable solutions to the problems that are either only briefly discussed or not mentioned in [1] have been tested and implemented. The MATLAB code as well as the C++ code are available on https://aslforge.ethz.ch/projects/voronavigation/.

In Sect. 6.2.1 the simulation and parameter files are described as well as an introduction to start a simulation is given. Section 6.2.2 shows an overview of all files regarding the preprocessing. In Sect. 6.2.3 and 6.2.3, respectively, the model of the robot and the maps are described.

6.2.1 Simulation and Parameter Files

To start a simulation in MATLAB the following steps should be executed:

1. add all subfolders of the directory in “File - Set Path”
2. choose an existing map or make a new one (store the map in the folder ‘Maps’)
3. set the parameter "Params(1).map" in "Sim_Init.m" to the chosen map (see Sect. 6.2.4 for more details)
4. run Sim_Init.m
5. choose a starting position for the robot (run the file: Tool_GetRobotInitPose.m)
6. set the parameter "CoaxModel.x_0" and "CoaxModel.y_0" in the file “Params_CoaX_PointMass.m” to the starting values found at the step before
7. adjust the parameters in Params_VNC.m as desired
8. run "Sim_Master" to start the simulation

A short overview of the parameter and simulation files is given in Table 4. In addition, there are some scripts, so-called tools, to help the user.

Table 4: Simulation and parameter files

<table>
<thead>
<tr>
<th>File Name</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sim_Init</td>
<td>Initializes the simulation. The file loads all parameters and the map. If the map was already loaded once, a .mat file was created and the map will not be converted again. When the map is used for the first time, a occupancy map will be created. The walls will be black (255), free spaces white (0) and the Voronoï branches will be green. After the map was successfully loaded, the local map will be initialized and the plots, according to the parameters defined in Params_VNC, will open in a separate window.</td>
</tr>
<tr>
<td>Sim_Master</td>
<td>The master file is the only file one has to run to start the simulation. First it performs a ‘tabula rasa’, loads then the Sim_Init file, defines some global variables and runs then the simulation.</td>
</tr>
<tr>
<td>Params_CoaX_PointMass</td>
<td>defines the parameter for the CoaX point mass model used in the simulation, such as mass, maximal allowed velocities, inertias and some parameters for the internal engine controller.</td>
</tr>
<tr>
<td>Params_LRF</td>
<td>defines the parameters for the laser range finder, such as field of view, resolution, error in distance and angle, minimal and maximal range.</td>
</tr>
<tr>
<td>Params_VNC</td>
<td>defines all parameters needed for the Voronoï navigation. It defines the initial starting position of the robot, the tuning parameters for the control algorithm, the desired plot parameters and all parameters for the split-and-merge plus algorithm.</td>
</tr>
<tr>
<td>Tool_CreateImageFromLocalMap</td>
<td>can be run to export the local map as a picture. The picture will be stored as .Cache/LocalMapImg.png</td>
</tr>
<tr>
<td>Tool_EstPoseFromRealLRFData</td>
<td>a test function to estimate the pose transformation from real LRF data. This function was needed to learn dealing with real data</td>
</tr>
<tr>
<td>Tool_GetRobotInitPose</td>
<td>function that returns the x and y position of the current cursor’s position on the map. Can be run to get an initial position for the simulation</td>
</tr>
<tr>
<td>Tool_ProprocessRawLRFData</td>
<td>If real laser scans from a text file are used, this function converts the raw data to a data struct that is required by other functions.</td>
</tr>
</tbody>
</table>
### 6.2.2 Pre-Processing Files

In Table 5 the tasks of all functions responsible for the preprocessing of the data are described. Moreover the function of the two additional safety algorithms as well as the method to proceed from a bifurcation point are briefly summarized.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>VNC_PPS_CalcMinDistToSegs</td>
<td>calculates the minimal distances between the robot and the segment</td>
</tr>
<tr>
<td>VNC_PPS_EstPose</td>
<td>pose estimation either based on the control laws, on the ICP or on both, adjusted with a tuning parameter</td>
</tr>
<tr>
<td>VNC_PPS_EstPoseFromControlLaws</td>
<td>computes the transformation matrices based on the control inputs (very low accuracy)</td>
</tr>
<tr>
<td>VNC_PPS_EstPoseFromICP</td>
<td>computes the transformation matrices based on the ICP algorithm</td>
</tr>
<tr>
<td>VNC_PPS_ExClosestSegs</td>
<td>extracts the closest segments from all segments coming from the split-and-merge plus algorithm. In this function the classification in sectors is implemented, it only extracts the closest segments in each sector.</td>
</tr>
<tr>
<td>VNC_PPS_FindObjects</td>
<td>searches for a third segment during following a branch</td>
</tr>
<tr>
<td>VNC_PPS_MonitorMinDistToSeg</td>
<td>safety function: it monitors the minimal distances to all segments. If one segment is getting ‘too close’ to the robot, it is taken immediately as a new relevant segment for control and the robot will react on it. This function prevents the robot to collide with a wall.</td>
</tr>
<tr>
<td>VNC_PPS_SetSegToObjTrack</td>
<td>initializes the tracker with the segments</td>
</tr>
<tr>
<td>VNC_PPS_SortMinDist</td>
<td>sorts the minimal distances to get it in a ascending order</td>
</tr>
<tr>
<td>VNC_PPS_TrackObjects</td>
<td>tracks the objects set in the function VNC_PPS_SetSegToObjTrack</td>
</tr>
<tr>
<td>VNC_PPS_UpdateLocalMap</td>
<td>does an update on the local map</td>
</tr>
<tr>
<td>VNC_PPlot_RobotOnMap</td>
<td>function to plot the robot’s current position on the map</td>
</tr>
<tr>
<td>VNC_PPlot_RobotPoseError</td>
<td>function to plot the distance between the robot’s position and nearest Voronoi branch</td>
</tr>
<tr>
<td>VNC_NCF_AvoidDeadEnd</td>
<td>safety function: prevents the robot from a dead end. The function lets the robot turn 180 degrees, so that the the robot can follow the branch back it came from</td>
</tr>
<tr>
<td>VNC_NCF_FindExitOfBP</td>
<td>searches a new branch at a bifurcation point. In the MATLAB version the method with the minimal magnitude of control inputs is not yet implemented, the algorithm searches somehow parallel walls and computes from this corridors the corresponding angle of the branch. This works quite good for simulation, but does not have good performance in a real environment however. Due to this reason an implementation of another idea was realised in C++.</td>
</tr>
<tr>
<td>VNC_NCF_SetSectorThree</td>
<td>a function that allows to set the angular definition of the third sector (in front) to a new value</td>
</tr>
<tr>
<td>VNC_NCF_Yawing</td>
<td>simple yaw function, the robot turns until it has reached the desired orientation</td>
</tr>
<tr>
<td>VNC_SMP_DelShortSegments</td>
<td>deletes segments that are considered as ‘too short’</td>
</tr>
<tr>
<td>VNC_SMP_DistPointToPoint</td>
<td>calculates the distance between two points that are given in Cartesian coordinates</td>
</tr>
<tr>
<td>VNC_SMP_ExtClusters2D</td>
<td>main function that extracts the clusters from the point cloud</td>
</tr>
<tr>
<td>VNC_SMP_ExtSeg2D</td>
<td>main function of the Split-And-Merge-Plus (SMP) algorithm, note that the cyclic version of the SPACER library is not yet implemented</td>
</tr>
<tr>
<td>VNC_SMP_ExtSeg</td>
<td>main function that applies the Split-And-Merge algorithm</td>
</tr>
<tr>
<td>VNC_SMP_LineFitLQ</td>
<td>fits a line using Least Squares. Full version: all covariance terms are computed.</td>
</tr>
<tr>
<td>VNC_SMP_LineFitLQQuick</td>
<td>fits a line using Least Squares. Quick version: no covariance terms are computed.</td>
</tr>
</tbody>
</table>
VNC_SMP_MegMergeSeqSegs performs the ‘agglomerative hierarchical clustering’ (AHC) algorithm, where distances between the segments are computed and the best candidates are selected first for merging. And continue. This function makes the assumption that the input segments are in sequence, meaning only consecutive segments are checked for merging. Cyclic of segments are not yet taken into account.

VNC_SMP_Mod2pi sets an angle to the range $-\pi < \text{angle} \leq +\pi$

VNC_SMP_PlotSegments plots the extracted segments from the Split-And-Merge-Plus algorithm

VNC_SMP_ProcessScanData plots the extracted segments from the Split-And-Merge-Plus algorithm

VNC_SMP_ProtPointQuick computes the coordinates of the projection of a point on a line. Quick version: No covariance terms are computed.

### 6.2.3 Model of the Robot

The model of the robot consists of a simple point mass model with an inertia in combination with a PI controller that regulates the velocity of the robot according to the control inputs. This controller is much faster than the sampling time of the navigation control algorithm and allows the robot to almost instantaneously change its velocity, i.e. no significant time delay between the demanded and the actual velocity occurs. This is truly a very strong idealisation of the velocity controller, but as the scope of this simulation is to get familiar with the control algorithm and reveal its weaknesses, it meets the requirements. Moreover the model does only consider the very basic Newton’s law of motion but no friction nor any other kind of nonlinearities. Later a non-holonomic constraint was added to meet the limitation of the wheeled robot.

The laser range finder model consists of an algorithm counting the pixels from the current position to the next “wall pixel”. It is obvious that the resolution of the sensor is directly linked to the resolution of the map. No additional noise was considered in the measurement process.

Besides the navigation controller, the model of the robot and the model of the sensor, a fourth block is needed to be able to draw the robot on the map. This transformation from the body fixed control frame to the world frame is done by the position estimation (cf. green block in Fig. 12). However, this part was only used to locate the robot on the map. As long as only the navigation task is considered no such transformation is needed in a real world application.

### 6.2.4 Maps and Computing their Voronoï Diagrams

As mentioned in Sect. 6.2.1 the maps are loaded as pictures in MATLAB. All images have to be stored in the folder ‘Maps’, otherwise the map won’t be found. To change the map, set the value of the parameter ‘Params(1).map’ in the file ‘Sim_Init.m’ to the name of the image file. If the image is loaded the first time, the simplified Voronoï diagram will be computed using the ‘watershed’ command of MATLAB and be added to the matrix ‘Map’ with an entry not equal to 0. This matrix will be saved together with the matrix ‘DMap’ in the .mat file with the same name of the image in the folder ‘Maps’. The distance matrix ‘DMap’ contains the minimum distances to the Voronoï graph at every pixel of the map. This map is needed to calculate the error of the robot’s position with respect to the Voronoï branch. The laser range sensor will only identify a pixel as an obstacle if the pixel is coloured black, i.e. it has a value equal to 0.

In case the image is modified and has to be reloaded, the old corresponding .mat file must be deleted, otherwise the map won’t change. This procedure is implemented to save time, because the calculation of the Voronoï diagram needs some time.

If the initial position of the robot coincides with an obstacle, the ‘Sim_Master’ will show an error. In that case, the initial position has to be changed in the parameter file of the robot. There is the script ‘Tool_GetRobotInitPose’ that helps one to find the coordinates of an initial position. Run the script after having run ‘Sim_Master’, click somewhere on the figure using the mouse and read the coordinates in the MATLAB command window. This procedure has not been automated in order that the simulations can be easily repeated under the same conditions.
Fig. 12 Simulink model of the robot used for the simulation. Cascade and navigation controller are coloured in blue, sensor and robot model in white and the transformation block in green

6.3 ROS Code

The code written in C++ is available on the project’s website https://aslforge.ethz.ch/projects/voronoi_navigation/. The navigation controller is split in different packages as shown in the diagram depicted in Fig. 13. The packages from the Voronoi Navigation project are described in the subsequent sections. The code is documented in doxygen style, so that rosdoc is applicable to generate the package API. Section 6.3.1 explains how to start the navigation controller on the robot robox.

6.3.1 The Launch Files and How to Start Robox

Due to the fact that the navigation algorithm was tested on the robot robox, the launch files of the navigation controller are stored in the repository of the ASL project Robots@home integration on https://aslforge.ethz.ch/projects/robotsathomeros/. The drivers and additional required packages are also in that repository.

The launch and configuration files are shortly explained in Table 6 and the following procedure explains how to start the navigation controller on robox:

1. Download the files of the projects Robots@home integration and Voronoi Navigation to your computer and store them in the ROS directory.
2. Compile the bblos driver (`rosmake –rosdep-install bblos_driver`), the joystick driver (`rosmake –rosdep-install teleoperation`) and all packages of the Voronoi navigation project.
3. Start the robot (two switches) and wait until it started up (stops blinking and beeping).
4. Connect your computer with the robox through the network. Use a static address for your computer (IP: 172.30.50.100, netmask: 255.255.255.0). The robox’ IP address is 172.30.50.243.
5. Plug in the joystick.
Fig. 13 ROS diagram showing how the packages are connected
6. Start roscore on your computer.
7. Manoeuvre the robox using the joystick to its initial position. On that account, launch the file onBoot.launch in the folder launchers/robox first.
8. Launch the navigation controller. There are two ways:
   a. The easiest way is to load everything at once: Launch the file robox_vn_start.launch (see Table 6 where it is stored). Make sure you stopped the nodes you loaded by the onBoot.launch file. This launch file will start the navigation controller with the SMP version.
   b. The first way is impractical to work with, because rviz crashes sometimes and must be reloaded. Thus, load the aforementioned onBoot.launch file first. Kill the joy_to_vel node as soon as you don’t need the joystick any more (“rosnode kill /joy_to_vel”). Launch the navigation controller, e.g. launch smp/robox_vn_paper.launch, and finally, launch rviz using rviz/robox_rviz.launch.
9. Make sure the emergency button is unlocked.
10. Start the navigation system by pressing the second button of the joystick.
11. Press the first button of the joystick to stop the robot immediately. The navigation system won’t shut down, but the published command velocities will be set to zero.
12. Press the second button to set the operating mode to 0 (following a Voronoï branch).
13. Press the third button to reset the published trajectory of the robot. You also might reset the view in rviz to clean it properly.
14. Press the fourth button to set the operating mode to 1 (stopping on a bifurcation point).

Table 6: Description of the launch files

<table>
<thead>
<tr>
<th>robotsathomeros/launchers/robox/voronoï_navigation/</th>
<th>instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>readme.txt</td>
<td>example launch file to start the voronoï navigation controller on robox with the SMP version</td>
</tr>
<tr>
<td>robox_vn_start.launch</td>
<td>launch files of navigation controller with the Generic Polygon Clipper algorithm</td>
</tr>
<tr>
<td>gpc/</td>
<td>launch files of navigation controller with the Generic Polygon Clipper algorithm</td>
</tr>
<tr>
<td>pcfilt.gpl</td>
<td>gnuplot configuration file, command to use: ‘gnuplot -persist pcfilt.gpl’</td>
</tr>
<tr>
<td>pointcloud</td>
<td>example output of vn_rungpc to plot the point cloud with gnuplot</td>
</tr>
<tr>
<td>polygon</td>
<td>example output of vn_rungpc to plot the polygon with gnuplot</td>
</tr>
<tr>
<td>robox_vn_gpc.launch</td>
<td>launch file to run the navigation controller with the GPC version</td>
</tr>
<tr>
<td>localmap/</td>
<td>launch files of the navigation controller with the local map version</td>
</tr>
<tr>
<td>robox_localmap_params.yaml</td>
<td>costmap_2d configuration file</td>
</tr>
<tr>
<td>robox_SMP_params.yaml</td>
<td>configuration file to overwrite the standard parameter values of the SMP algorithm</td>
</tr>
<tr>
<td>robox_vn_localmap.launch</td>
<td>launch file to run the navigation controller with the local map version</td>
</tr>
<tr>
<td>rviz/</td>
<td>launch files of rviz</td>
</tr>
<tr>
<td>robox_rviz_config_localmap.vcg</td>
<td>configuration file of rviz used with the local map version</td>
</tr>
<tr>
<td>robox_rviz.launch</td>
<td>launch file to execute rviz separately</td>
</tr>
<tr>
<td>robox_vn_paper_rviz_config.vcg</td>
<td>standard configuration file of rviz used by robox_rviz.launch</td>
</tr>
<tr>
<td>robox_vn_paper_rviz_sector_config.vcg</td>
<td>configuration file of rviz to show only the segments in the sectors</td>
</tr>
<tr>
<td>smp/</td>
<td>launch files of the navigation controller with the Split-And-Merge-Plus-Algorithm</td>
</tr>
<tr>
<td>robox_vn_paper_costmap_params.yaml</td>
<td>configuration file of the costmap_2d</td>
</tr>
<tr>
<td>robox_vn_paper.launch</td>
<td>launch file to run the navigation controller with the SMP version</td>
</tr>
</tbody>
</table>
6.3.2 Package vn_initparams

This package initializes the parameter values on the ROS Parameter Server. Actually, the parameter values can also be loaded from a file using the command ‘rosparam load’, however it is more convenient to have the possibility to calculate some parameter values from others. This package only provides parameter values required by the navigation algorithm. Parameters needed to link the single packages, such as names of topics, are specified in the launch files. Table 7 shows more details about the executable.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>This executable initializes the parameter values on the ROS Parameter Server and shuts down afterwards.</td>
</tr>
<tr>
<td>name of node</td>
<td>vn_initparams</td>
</tr>
<tr>
<td>msg callback input</td>
<td>none</td>
</tr>
<tr>
<td>published output</td>
<td>none</td>
</tr>
<tr>
<td>parameters</td>
<td>none</td>
</tr>
</tbody>
</table>

6.3.3 Package vn_msgs

The package vn_msgs contains all messages that are required by the navigation controller. In addition, small common tools are also stored in this package. The message types are described in Table 8, whereas the executables are described in Table 9.

<table>
<thead>
<tr>
<th>segment</th>
<th>describes the properties of a single segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>isSet true if segment is set</td>
</tr>
<tr>
<td>float64</td>
<td>rho minimum distance to the segment (either perpendicular to the line or to the end points)</td>
</tr>
<tr>
<td>float64</td>
<td>alpha corresponding angle to the segment (either perpendicular to the line or to the end points)</td>
</tr>
<tr>
<td>uint8</td>
<td>hook describes to which point the distance is measured (0: perpendicular to the line, 1: to the start point of the line, 2: to the end point of the line)</td>
</tr>
<tr>
<td>uint8</td>
<td>sector the sector the segment belongs to</td>
</tr>
<tr>
<td>float64</td>
<td>LineRho distance of line model</td>
</tr>
<tr>
<td>float64</td>
<td>LineAlpha angle of line model</td>
</tr>
<tr>
<td>geometry_msgs/Point[2]</td>
<td>points end points</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>segments</th>
<th>describes a cloud of segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>header header of the array</td>
</tr>
<tr>
<td>uint8</td>
<td>NSegments number of segments (length of array Segment)</td>
</tr>
<tr>
<td>vn_msgs/segment[]</td>
<td>Segment array of segments</td>
</tr>
<tr>
<td>bool[3]</td>
<td>hasSegsInSector true if there is at least one segment in the sector (index of array corresponds to index of sector)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>controlinput</th>
<th>describes the three very important distances required by the controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>header header</td>
</tr>
<tr>
<td>float64[3]</td>
<td>Delta minimum distance to the segment</td>
</tr>
<tr>
<td>float64[3]</td>
<td>Theta corresponding angle of the segment</td>
</tr>
</tbody>
</table>

| SegmentToMarker  | Converts a vn_msgs/segments message into a visualization_msgs/Marker message in order to visualize the segments in rviz. If the value of the parameter mode is 'standard', all segments are plotted in the specified colour (see parameters). If the value of the parameter mode is 'sector', the segments are plotted in the sector’s colour. Three sectors are defined. |
6.3.4 Package vn_runsmp

This package is part of the pre-processing of the point cloud from the laser range finder. It provides the Split-And-Merge-Plus algorithm (SMP) to extract segment lines from the point cloud. The executable vn_runsmp is linked to the slightly modified spacer library of Viet Nguyen. The library was just adapted to be compatible to the ROS. The SMP algorithm has the following subroutines:

1. Simple clustering algorithm
2. Split-And-Merge algorithm
3. Merging algorithm based on Chi-Square test.
4. Routine to remove short segments

Interested readers may consult [1] to learn more about the principle of the algorithm.

In addition, the executable ‘testpointcloud’ allows to examine the algorithm using laser scan points that are stored in a text file. The example file ‘ASLRawScans-Selected-100.txt’ is stored in the the folder ‘data’.

Table 10 describes the executables in more details.

<table>
<thead>
<tr>
<th>vn_runsmp</th>
<th>description</th>
<th>name of node</th>
<th>msg callback input</th>
<th>published output</th>
<th>private parameters</th>
</tr>
</thead>
</table>
|           | It runs the Split-And-Merge-Plus algorithm as a message callback function and computes all properties of the vnmsgs/segments message. | RunSMP | sensormsgs/PointCloud message | vnmsgs/segments message (see Table 8) | string PointCloudTopicName topic name of the source point cloud  
string SegmentTopicName topic name of the segments  
bool doSMP true if SMP is applied |
|           |             |              |                   |                 | double Sectors/1/angle1 range of sector 1 [angle1 angle2] in rad  
double Sectors/1/angle2 range of sector 1 [angle1 angle2] in rad  
double Sectors/2/angle1 range of sector 2 [angle1 angle2] in rad  
double Sectors/2/angle2 range of sector 2 [angle1 angle2] in rad  
double Sectors/3/angle1 range of sector 3 [angle1 angle2] in rad  
double Sectors/3/angle2 range of sector 3 [angle1 angle2] in rad  
double Offset/phi transform scan points (rotate about z-direction [rad])  
double Offset/x transform scan points (move in x-direction [m])  
double Offset/y transform scan points (move in y-direction [m]) |
|           |             |              |                   |                 | double SickMaxRange maximum range of laser range finder [m]  
double SickMinRange minimum range of laser range finder [m]  
double SickPhiInc angular increment between scan points [deg]  
double SickNumPoints number of points in a raw scan  
int SickCyclic =1 if cyclic scans (2 sick), otherwise =0  
double SickSr radial uncertainty term (covariance entry)  
double SickSp radial/polar uncertainty term (covariance entry)  
double SickSpp polar uncertainty term (covariance entry) |
**Global Parameters**

### VN/VNC/SMP/ExClusters

- **RHOPEAKLEVEL**
  - A threshold of normalized ‘jump’ of absolute rho (distance) differences between consecutive scan points, used for splitting. Default value: 0.12 (you can use: 0.08 .. 0.2, not tested however) Note: You should only adapt MINCLUSTERPOINTS to your specific data. Use small value of RHOPEAKLEVEL for good dataset, large value for noisy one.
  - [m]

- **PHIPEAKLEVEL**
  - A threshold of normalized ‘jump’ of absolute phi (angle) differences between consecutive scan points, used for splitting. [rad]

- **MINCLUSTERPOINTS**
  - Minimum number of scan points for a cluster. Clusters that have less number of points will be discarded. Default value: 7

- **MINCLUSTERLENGTH**
  - Minimum of actual length of a cluster, used to further discard small clusters.

### VN/VNC/SMP/ExSegs

- **INLIERTHRESHOLD**
  - Threshold for a point to be considered inlier to a line. Maximum distance from a scan point to a line that the point is considered inlier to the line. [m]

- **ADJACENTDIST**
  - If the distance between the end points of two adjacent segments is larger than ADJACENTDIST, those segments will not be merged. [m]

- **VALIGATE**
  - The threshold used in the merging routine (≈2.77 which corresponds to the 75% confidence interval, other values: 1.02, 9.21).

- **SEGMINNUMPOINTS**
  - Minimum number of scan points for a line segment.

---

**Testpointcloud**

- **Description**
  - This executable reads point clouds from a single file and publishes them in a sensor_msgs/PointCloud message. The input file must be in this format: The file is in text format. Each scan (a step) is in a separate line. Each line contains ‘Timestamp XR YR ThetaR NumPoints X1 Y1 X2 Y2 …’. The function that reads the file is taken from the spacer library. There are more functions available, which read different format. Have a look at /spacer/robotapi/sick.h to get more information.

- **Name of Node**
  - testpointcloud

- **Msg Callback Input**
  - testpointcloud_publisher

- **Published Output**
  - sensor_msgs/PointCloud

- **Private Parameters**
  - **string** topic_name
    - name of the topic of the PointCloud message
  - **string** DataFileNm
    - name of file containing the point cloud
  - **int** pub_freq
    - the publisher’s frequency
  - **string** ChannelName
    - name of the channel of the PointCloud message
  - **float** ChannelValue
    - value of the channel of the PointCloud message (The value is the same for all scan points and is only needed to avoid an error in rviz.)
  - **string** frame_id
    - frame id of the PointCloud message
  - **bool** loop
    - if true, the publishing of the scans is repeated

---

### Package vn_rungpc

This package is part of the preprocessing of the point cloud from the laser range finder. The executable vn_rungpc contains the Generic Polygon Clipper algorithm to identify the closest points required to compute the control laws, which is an alternative to the SMP algorithm. The executable is linked to the GPC library from Cédric Pradalier and Alan Murta.

The package was tested together with the package vn_calccontrollaws, but did not result in a useful navigation controller. The main reason is that the idea of selecting the three closest points does not work in practice. However, the GPC is working properly.

The executable is described in Table 11.
Table 11 Description of the executable of the package vn_rungpc

<table>
<thead>
<tr>
<th>vn_rungpc</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>description</strong></td>
<td>This executable runs the Generic Polygon Clipper algorithm as a message callback function and selects the closest three points from the extracted polygon. In contrast to the vn_runsmp executable, the output is directly the control input required to compute the control laws. There is the possibility to visualize the polygons in GNU plot.</td>
</tr>
<tr>
<td><strong>name of node</strong></td>
<td>RunGPC</td>
</tr>
<tr>
<td><strong>msg callback input</strong></td>
<td>sensor_msgs/PointCloud message</td>
</tr>
<tr>
<td><strong>published output</strong></td>
<td>vn_msgs/controlinput message</td>
</tr>
<tr>
<td><strong>private parameters</strong></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>PointCloudTopicName</td>
</tr>
<tr>
<td>string</td>
<td>ControlInputTopicName</td>
</tr>
<tr>
<td>bool</td>
<td>doGPC</td>
</tr>
<tr>
<td>bool</td>
<td>doGnuplot</td>
</tr>
</tbody>
</table>

6.3.6 Package vn_calccontrollaws

The package vn_calccontrollaws is the core of the navigation algorithm. It gets as an input the segments from runSMP or the control inputs from runGPC and computes as an output the corresponding velocities for the robot. The vn_calccontrollaws has the following main subroutines:

1. an algorithm to check in which mode the system should run
2. a visualization function to publish the robot’s contour and trajectory for rviz
3. a routine to compute the control laws according to the operating mode
4. a tracking function to track the segments when stopping on a bifurcation point
5. a safety function to monitor the closest distances to the segments
6. a safety function to detect dead ends
7. an algorithm to find a new Voronoi branch to leave a bifurcation point
8. callback functions that react either on messages from the runsmmp or the rungpc package

The executable and its parameters are described in Table 12.

Table 12: Description of the executable of the package vn_calccontrollaws

<table>
<thead>
<tr>
<th>vn_calccontrollaws</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>description</strong></td>
<td>It runs the calculation of the control laws according to the inputs. Furthermore, the executable has some subroutines controlling safety issues like avoiding dead ends and monitoring minimum distances. The node listens on the SMP topic as well as on the GPC topic and is able to deal with both methods. However, the GPC method is not yet working properly and some operating modes are not implemented for it.</td>
</tr>
<tr>
<td><strong>name of node</strong></td>
<td>navigation_controller</td>
</tr>
<tr>
<td><strong>msg callback input</strong></td>
<td>vn_msgs/segments message or vn_msgs/controlinput message (see Table 8)</td>
</tr>
<tr>
<td><strong>published output</strong></td>
<td>geometry_msgs/Twist message</td>
</tr>
</tbody>
</table>
private parameters
bool isPublishingTrajectory true if trajectory shall be published
double RobotContour width thickness of the robots contour in rviz
double Trajectory width thickness of the trajectory in rviz
double /Trajectory/color/a color of trajectory (satisfaction [0...1])
double /Trajectory/color/t color of trajectory (red [0...1])
double /Trajectory/color/g color of trajectory (green [0...1])
double /Trajectory/color/b color of trajectory (blue [0...1])
string ClosestSegs/TopicName topic name for the closest segments
string CmdVelTopicName name of the channel for the velocity output
string ControlInputTopicName name of the channel for the control input
string gpcTopicName name of the channel for the control input from GPC
string JoystickTopicName name of the channel for the joystick input
string MinDistTopicName name of the channel for the minimal distances visualization
string OdometryTopicName name of the channel for the odometry callback
string RobotContourTopicName name of the channel for the robot contour visualization
string SegmentTopicName name of the channel for the segment
string SegmentsTopicName name of the channel for the segments
string TrackedSegsTopicName name of the channel for the visualization of the tracked segments
string TrajectoryTopicName name of the channel for the visualization of the trajectory
string TransformedTrackedSegsTopicName name of the channel for the visualization of the transformed tracked segments
string VISegsTopicName name of the channel for the visualization of the very important segments

global parameters
double ClosestSegTresh condition for a segment to be considered as 'too close'
double AvoidDeadEnd/AlphaThresh angular threshold for a dead end detection
double AvoidDeadEnd/RhoThresh distance threshold for a dead end detection

global parameters
double e_bp_thesh threshold for the condition to detect a bifurcation point (difference between the two distances and the third one in [%])
double AvoidDeadEnd/AlphaThresh condition for a segment to be considered as 'too close'
double AvoidDeadEnd/RhoThresh distance threshold for a dead end detection

global parameters
double eNav_thesh threshold for the condition to have reached a bifurcation point (allowed error in [m])
double v_des desired velocity to follow a Voronoi branch [m/s]
double lambda_ebranch parameter to adjust the convergence during following the branch [0...1]
double lambda_ebp parameter to adjust the convergence during stopping on a bifurcation point [0...1]
double alpha_t weighting parameter between follow a branch and align with a branch [0...1]

global parameters
double xdot_max maximum forward speed [m/s]
double psidot_max maximum rotation speed [m/s]

6.3.7 Package vn_getlocalmap

In case the scan points used to compute the control laws don’t come from the laser range finder directly, but from a local map provided by the package costmap_2d, the point cloud required by the package vn_runsmmp still needs to be ordered by the angles of the points. Therefore, the aim of this package is to extract a useful point cloud from a local map. Table 13 explains the respective executable.

6.4 Future Work

The experiments showed that safe autonomous exploration is not yet guaranteed, i.e. the robot does not stop on each bifurcation point and the size of the robot is not yet taken into account. To overcome this problem and to be able to run our implementation on a helicopter the following further steps should be considered: First, the pre-processing: Although Victorino et al. proposed to use segments to calculate the control inputs, we showed in our experiments that this
representation of the environment does not or only insufficiently allows to safely autonomously explore the unknown environment. We suggest that other approaches as presented in Table 3 yield better results and, even more important, lead to a smoother robot motion cause of their choice of the closest distances. However, if the interested reader intend to use another approach, the functions in our implementation must be adapted since they depend all on segments.

Second, the dependency between the parameters and the performance of the robot was not yet considered and might be an interesting part of some future experiments. Thus, a proper performance measurement should be implemented to get quantitative results from the experiments. This allows to optimize the algorithm by tuning the algorithm’s parameters to a special environment. Once the robot is trained for a certain kind of environment, e.g. the ETH CLA corridor, it should be tested in other surroundings to verify its behaviour.

Third, the computational effort must be considered and lowered if the algorithm should run on-board or on a helicopter. As mentioned in Sect. 6.1 a reduction of points leads directly to a lower computational effort.

Last but not least, there are two particular parts in our implementation which are not compatible with a helicopter. On the one hand, the the transformation matrix needed for the tracking can not be calculated using odometry, but the Iterative Closest Point algorithm (ICP) can be used instead as aforementioned, or the tracking will be completely omitted when another approach is applied. On the other hand, the the mapping of the lateral velocity to the other components of the velocity vector can be omitted owing to the missing non-holonomic constraint.

Acknowledgement

We would like to thank François Pomerleau and Quentin Boussonard for their support, expertise, and help. Special thanks go to Cédric Pradalier for his ideas to improve the algorithm and his helpful advice regarding this paper.

References