Realtime Segmentation of Range Data Using Continuous Nearest Neighbors

Klaas Klasing Dirk Wollherr Martin Buss
Institute of Automatic Control Engineering
Technische Universität München
80290 Munich, Germany
{kk,dw,mb}@tum.de

***Abstract*— In mobile robotics, the segmentation of range data is an important prerequisite to object recognition and environment understanding. This paper presents an algorithm for realtime segmentation of a continuous stream of incoming range data. The method is an extension of the previously developed RBNN algorithm and proceeds in two phases: Firstly, the normal vector of each incoming point is estimated from its neighborhood, which is continuously monitored. Secondly, new points are clustered according to their Euclidean and angular distance to previously clustered points. An outline of the algorithm complexity as well as the parameters that influence the segmentation performance is provided. Three benchmark scenarios in which the algorithm is deployed on a mobile robot with a laser range finder confirm that the method can robustly segment incoming data at high rates.**

I. Introduction

For many research disciplines within the field of robotics a central goal is to reach a level of semantic environment understanding. The recognition and correct interpretation of objects in the environment serves as a prerequisite for many higher-level tasks, such as manipulation and action planning. However, object recognition and understanding is in itself dependent on a proper segmentation of the raw data into meaningful portions. In the context of range data these portions correspond to patches of homogeneous surfaces which make up the perceived objects.

This paper presents an algorithm that segments a stream of incoming range data, for example from a laser range finder, into distinct homogeneous surfaces in a realtime fashion. In contrast to the majority of existing work [1], [2], [3] the range data is not required to be delivered in the form of *range images* but can stem from a continuous stream of measurements of a sweeping 2D range sensor. Range images constitute a central data structure for surface segmentation because they allow for the direct application of many of the powerful segmentation algorithms developed in the vision community over the past decades. For many kinds of sensor, such as stereo cameras or time-of-flight cameras, range images are the ideal data structure, because the data periodically arrives in portions of 2D lattices with corresponding depth values. Many roboticists use actuated 2D laser range finders to produce 3D range data [4]; depending on the scanning setup - for example for a rotating or periodically sweeping scanner - a range image may be the most suitable data structure. However, range images also have a number of known disadvantages, including projection-related problems , limited or distorted depth resolution, limited field of view etc. A central philosophy of range images is that each image is captured from a constant viewpoint, thus, in mobile robotics one is faced with the cumbersome problem of fusing multiple 2.5D range images from multiple viewpoints to build a meaningful environment model. While these problems are not impossible to solve [5] and have been tackled in a 2D context by the vision community for a long time, in this paper it is argued that a growing and sliding *3D point cloud* over range data acquired by a sweeping 2D range sensor offers a more natural and flexible approach for a number of robotics scenarios. As one example, imagine a mobile robot driving around with one or several vertically-mounted fixed laser range finders for the exploration of a building or an outdoor environment. As another example, consider an articulated robot arm with a gripper end effector and a laser scanner mounted underneath. Both scenarios yield an incoming stream of sweeping range data that exhibits no periodicity.

In [6] an algorithm is presented that efficiently segments a given 3D point cloud using a *radially bounded nearest neighbor* (RBNN) clustering strategy. While the required computation time renders the algorithm an interesting candidate for realtime processing, the method cannot directly be used for a continuous stream of data, mainly because nearest neighbor searching is carried out by means of a static kd-tree.

This paper presents a refined version of the RBNN algorithm that continuously monitors nearest neighbors and uses a feature space consisting of both the incoming points and their estimated normal vectors for clustering. Accordingly, the remainder of this paper is organized as follows: Section II describes how Euclidean nearest neighbors can be monitored efficiently for a stream of incoming range data. Section III explains the process of normal vector estimation from the so-established local neighborhood. In Section IV the actual clustering algorithm, which makes use of a Euclidean and an angular distance measure, is described. Experimental results are presented in Section V and a discussion and an outlook are provided in Section VI.

II. Continuous Nearest Neighbors

The efficient retrieval of nearest neighbors of a given query point is a fundamental problem in computational geometry that spans such different domains as database search, ray tracing or path planning. The well-known *kd-tree* [7] is one of the most efficient data structures for exact and approximate nearest neighbor searching in low- and high-dimensional Minkowski metric spaces [8]. Unfortunately, virtually all existing implementations, including the efficient ANN library[1], build a static kd-tree from a fixed set of points. Although the computational geometry literature abounds with elaborate kd-tree extensions, the authors are unaware of efficient kd-tree implementations. The main trouble with making a kd-tree dynamic is that it is not possible to efficiently balance the tree after insertions or deletions of points. Thus, all dynamic extensions rely on using a set of static kd-trees, mostly by means of the so-called *logarithmic method* which makes any static data structure dynamic with logarithmic complexity overhead.

Fortunately, it turns out that for range sensing applications one can resort to a much simpler data structure because firstly, the space is only three-dimensional and secondly, heuristics of the range sensor can be exploited. Consider a sequence of range scans as depicted in Figure 1. For a 2D range finder that is swept over the environment the following observations apply:

- The points within one scan $\boldsymbol{P}_i = \{\boldsymbol{p}_{i,1}, \dots, \boldsymbol{p}_{i,n_i}\}$ are ordered by increasing index, i.e. neighboring indices correspond to neighboring scanning angles.
- There is a continuity in consecutive scans, i.e. scan $\boldsymbol{P}_{i+1}$ is bound to be closer to $\boldsymbol{P}_i$ than $\boldsymbol{P}_{i+2}$.

This structure can be exploited in the search for nearest neighbors. Assume that one wants to find the k nearest neighbors of each point within a scan $\boldsymbol{P}_i$. The term *k-window* will refer to the window of neighboring indices centered around a given point $\boldsymbol{p}_{i,j}$ in which the k nearest neighbors are searched. Testing different variants and sizes of k-windows, a square k-window of width

$$w_k = 2\max\left(\left\lfloor \frac{k}{4} \right\rfloor, 1\right) + 1 \tag{1}$$

was found to yield a good balance between the number of examined points n_{ex} and the number of true nearest neighbors retrieved. Since the k-window is square, at most $n_{ex} = w_k^2 - 1$ neighboring points are examined; n_{ex} increases in a squared fashion with k. The overhead of examining all points in the square k-window to find only k nearest neighbors may seem costly, however, keep in mind that for points at the end of a scan the k-window contains as little as $w_k \cdot \left\lceil \frac{w_k}{2} \right\rceil - 1$ points. As an example, Figure 1 shows the 14 examined points in the 8-window (in blue) which contain all 8 nearest neighbors of the query point $\boldsymbol{p}_{i,n_i}$ (in green).

For all points in the k-window the Euclidean distance to the query point is computed, the results are sorted in a temporary list by ascending distance and the first k points

[1]http://www.cs.umd.edu/~mount/ANN/

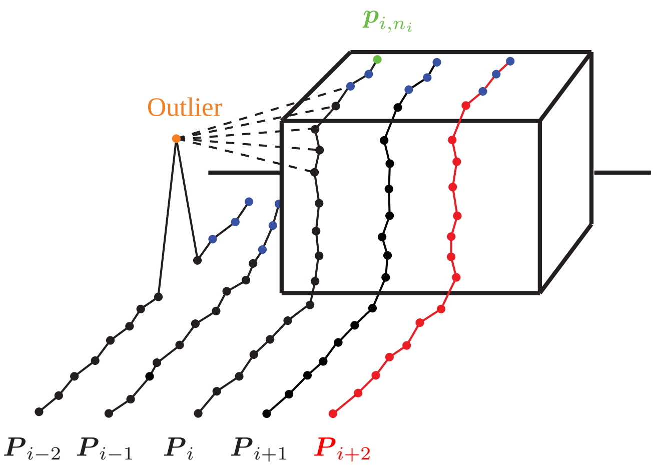

Fig. 1. Consecutive noisy scan profiles from a sweeping 2D range sensor. The most recent scan is depicted in red. For query point $\boldsymbol{p}_{i,n_i}$ (in green) the 8 nearest neighbors are searched and found among the points in the *8-window* with width $w_8 = 5$. For an individual outlier (in orange) the neighboring indices do not correspond to neighboring points. In the depicted case, only 5 true nearest neighbors (indicated by dashed lines) are contained in the *8-window*.

are saved in the neighborhood matrix $\boldsymbol{Q}$. Using the heap-sort algorithm, this incurs a cost of $O(n_{ex} \log n_{ex})$. Since $n_{ex} \propto k^2$, the overall runtime complexity of finding k nearest neighbors of a given point is dominated by $O(k^2 \log k)$ for large k. Obviously, the k-window will not always contain all true nearest neighbors of the query point. For the outlier in Figure 1 (in orange) only 5 out of 8 nearest neighbors are contained in the 8-window. Depending how the local neighborhood $\boldsymbol{Q}$ is further processed, this effect can be neglected. For the computation of normal vectors the neighbors obtained from the described k-window approach yield good results. This is mainly due to the fact that the incoming scans represent a sufficiently dense *surface sampling* of objects in the real world and not a *volumetric sampling*, thus the true nearest neighbors most of the time coincide with the neighboring scan points. A description of the *continuous nearest neighbors* algorithm in pseudo-code is given in Algorithm 1.

Algorithm 1 Continuous Nearest Neighbors

```
1:  FindNearestNearestNeighbors($\boldsymbol{p}_{i,j}$, $k$)
2:  $w_k = 2\max\left(\left\lfloor \frac{k}{4} \right\rfloor, 1\right) + 1$
3:  list<distance, point> = $\emptyset$;
4:  for ($r = i - \lfloor \frac{w_k}{2} \rfloor, \dots, i + \lfloor \frac{w_k}{2} \rfloor$) do
5:      left = max ($r - \lfloor \frac{w_k}{2} \rfloor$, 0);
6:      right = min ($r + \lfloor \frac{w_k}{2} \rfloor, n_r, n_i$);
7:      for (s = left , . . . , right) do
8:          dist = $\|\boldsymbol{p}_{i,j} - \boldsymbol{p}_{r,s}\|$;
9:          list.append(dist,$\boldsymbol{p}_{r,s}$);
10:     end for
11: end for
12: list.sortByAscendingDistance();
13: for ($l = 1, \dots, k$) do
14:     $\boldsymbol{Q}_{i,j} = [\boldsymbol{Q}_{i,j};$ list[$l$].point$^T$];
15: end for
16: return $\boldsymbol{Q}_{i,j}$
```

III. Normal Vector Estimation

Given the neighborhood $\boldsymbol{Q}_{i,j}$ of each point $\boldsymbol{p}_{i,j}$, an estimate for the normal vector is computed using the *PlanePCA* method [9], because it was shown to offer the best performance in terms of both quality and computation time [10]. The *PlanePCA* method proceeds as follows: The point $\boldsymbol{p}_{i,j}$ itself is appended to its neighborhood matrix $\boldsymbol{Q}_{i,j}$, which yields the augmented neighborhood matrix $\boldsymbol{Q}^{+}_{i,j}$. The empirical mean $\overline{\boldsymbol{Q}}^{+}_{i,j}$ is calculated and subtracted from $\boldsymbol{Q}^{+}_{i,j}$. Finally, a *singular value decomposition* ($\boldsymbol{U\Sigma V}^{T}$) of $\boldsymbol{Q}^{+}_{i,j}$ is performed and the resulting normal vector $\boldsymbol{n}_{i,j}$ is obtained directly as the singular vector $\boldsymbol{v}_{min} \in \boldsymbol{V}$ that corresponds to the smallest singular value in $\boldsymbol{\Sigma}$. The so-obtained vectors are normal to an optimally fitted local plane; however, they are not oriented consistently, meaning some vectors will point towards the range sensor and others away from it. While consistent orientation represents a tough problem in many surface modelling scenarios in computer graphics, in this case the knowledge of the sampling process can again be exploited. For every incoming scan $\boldsymbol{P}_i$, let the origin of the range sensor be denoted by $\boldsymbol{o}_i = [o_{i,x}\ o_{i,y}\ o_{i,z}]$. Every normal vector $\boldsymbol{n}_{i,j}$ is oriented consistently by assessing the sign of its inner product with the line-of-sight-vector $\boldsymbol{p}_{i,j} - \boldsymbol{o}_i$ and inverting all normal vectors with positive sign:

$$\langle \boldsymbol{n}_{i,j}, \boldsymbol{p}_{i,j} - \boldsymbol{o}_i \rangle > 0 \Rightarrow \boldsymbol{n}_{i,j} = -\boldsymbol{n}_{i,j} \tag{2}$$

This yields normal vectors that consistently point towards the range sensor. Algorithm 2 provides a pseudo-code description of the *normal vector estimation* algorithm.

Algorithm 2 Normal Vector Estimation

```
1: EstimateNormalVector($\boldsymbol{p}_{i,j}$, $\boldsymbol{Q}_{i,j}$, $\boldsymbol{o}_i$)
2: $\boldsymbol{Q}^{+}_{i,j} = [\boldsymbol{Q}_{i,j};\ \boldsymbol{p}^{T}_{i,j}]$;
3: $\boldsymbol{Q}^{*}_{i,j} = \boldsymbol{Q}^{+}_{i,j} - \overline{\boldsymbol{Q}}^{+}_{i,j}$;
4: $[\boldsymbol{U}, \boldsymbol{\Sigma}, \boldsymbol{V}^{T}]$ = performSVD($\boldsymbol{Q}^{*}_{i,j}$);
5: $\boldsymbol{n}_{i,j} = \boldsymbol{v}_{min}$;
6: if ($\langle \boldsymbol{n}_{i,j}, \boldsymbol{p}_{i,j} - \boldsymbol{o}_i \rangle > 0$) then
7:     $\boldsymbol{n}_{i,j} = -\boldsymbol{n}_{i,j}$;
8: end if
9: return $\boldsymbol{n}_{i,j}$
```

IV. Incremental Segmentation

In order to cluster the incoming data in a realtime fashion, the similarity difference between the newly arrived points and the previously clustered points must be assessed in a suitable feature space. In its original form, the RBNN algorithm [6] is already well suited for this job because of its agglomerative nature. Since the goal is to segment homogeneous surfaces, a natural choice is to use both the point coordinates $[p_x\ p_y\ p_z]$ as well as the estimated normal vectors $[n_x\ n_y\ n_z]$ as features. This choice, however, raises the issue of a suitable (dis)similarity measure. Surprisingly, preliminary studies in which the RBNN algorithm was applied to a $[p_x\ p_y\ p_z\ n_x\ n_y\ n_z]$ feature space, yielded good offline segmentation results already. Figure 2 shows the resulting clusters for a simple data set with two boxes and a cylinder taken from [10]. The result is surprising because the kd-tree was built using a Euclidean distance metric over *all* entries of the feature vector, which does not respect the topology of the angular (dis)similarity of the normal vectors. While it is reassuring to know that even with a one-fits-all Euclidean metric over the feature space proper segmentation can be achieved, the threshold r associated with this metric has very little descriptive meaning. To remedy this, two thresholds on two measures are introduced. As before, a threshold $\rho_{e,max}$ on the Euclidean dissimilarity metric

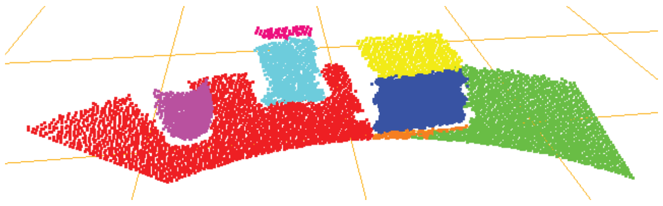

Fig. 2. Segmentation of a simple data set from [10] containing two boxes and a cylinder. The clusters were produced using the offline RBNN algorithm with $r = 0.14$ and $n_{min} = 40$ and a Euclidean distance metric over a $[p_x\ p_y\ p_z\ n_x\ n_y\ n_z]$ feature space. Note that for such a feature space r must be dimensionless and has little interpretable meaning.

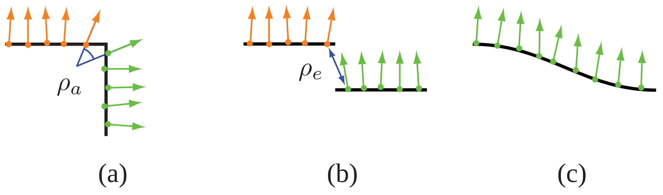

Fig. 3. Effect of the two thresholds on segmentation: (a) Corners are detected with ρ_a. (b) Steps are detected with ρ_e. (c) Smooth surfaces are not segmented.

$$\rho_e(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left\| \boldsymbol{p}_i - \boldsymbol{p}_j \right\| \tag{3}$$

specifies how far the 3D coordinates of two oriented points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ can be away from each other to still belong to the same cluster. The threshold $\rho_{a,max}$ on the angular dissimilarity measure

$$\rho_a(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{1}{2} - \frac{1}{2} \frac{\langle \boldsymbol{n}_i, \boldsymbol{n}_j \rangle}{\|\boldsymbol{n}_i\|\ \|\boldsymbol{n}_j\|} \tag{4}$$

defines the maximum angle that the normal vectors of two oriented points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are allowed to span to still belong to the same cluster. In (4) is normalized ρ_a to the interval $[0, 1]$; alternatively one can specify the angle in degrees. Using these two thresholds enables us to choose meaningful values and to control the segmentation process. Figure 3 illustrates how both thresholds affect the segmentation of different surface discontinuities.

Using these two thresholds, the segmentation process can be summarized as follows:

1) For each new point $\boldsymbol{p}_{i,j}$ with normal vector $\boldsymbol{n}_{i,j}$, go through the list of all neighbors in $\boldsymbol{Q}_{i,j}$ that have already been assigned a normal vector.
2) For each current neighbor compute ρ_e and ρ_a.
3) For all neighbors that fulfill $\rho_e \le \rho_{e,max}$ and $\rho_a \le \rho_{a,max}$, check whether any of the neighbors has been

assigned to a cluster. If so, assign $\boldsymbol{p}_{i,j}$ to the same cluster, then assign all remaining neighbors to the same cluster. Merge clusters if necessary.

4) If $\boldsymbol{p}_{i,j}$ has not been assigned to any cluster, create a new one and assign all previously examined neighbors to it.

To remove noise, the offline RBNN algorithm provides the parameter n_{min}, which specifies the minimum number of data items a cluster should have. The same parameter can be used for the incremental segmentation, however, since an existing cluster might still grow with the arrival of the following scans, a time offset t_k is introduced, which specifies the maximum number of new scans that are processed before deleting a cluster that has not grown and contains less than n_{min} points. Assuming that the same k-window buffer is used for segmentation and normal vector computation, t_k should be chosen in the interval $[1, \lfloor \frac{w_k}{2} \rfloor]$, because only points within this window will connect to the current cluster. Algorithm 3 provides a pseudo-code description of the outlined *incremental segmentation* algorithm. The input arguments $\boldsymbol{N}_i$ and $\boldsymbol{Q}_i^n$ denote the set of caculated normal vectors and the set of neighborhood matrices of oriented points that have already been assigned a normal vector, respectively.

Algorithm 3 Incremental Segmentation

```
1:  SegmentNewScan($\boldsymbol{P}_i, \boldsymbol{N}_i, \boldsymbol{Q}_i^n, \rho_{e,max}, \rho_{a,max}, r_{min}, t_k$)
2:  for ($j = 1, \ldots, n_i$) do
3:    if (hasCluster($\boldsymbol{p}_{i,j}$)) then
4:      continue;
5:    end if
6:    for all ($[\boldsymbol{p}_l \ \boldsymbol{n}_l] \in \boldsymbol{Q}_{i,j}^n$) do
7:      $\rho_e = \|\boldsymbol{p}_l - \boldsymbol{p}_{i,j}\|$;
8:      $\rho_a = \frac{1}{2} - \frac{1}{2}\frac{\langle \boldsymbol{n}_{i,j}, \boldsymbol{n}_l \rangle}{\|\boldsymbol{n}_{i,j}\|\|\boldsymbol{n}_l\|}$;
9:      if ($\rho_e > \rho_{e,max} \vee \rho_a > \rho_{a,max}$) then
10:       continue;
11:     end if
12:     if (hasCluster($\boldsymbol{p}_{i,j}$)) then
13:       if (hasCluster($\boldsymbol{p}_l$)) then
14:         if (clusterOf($\boldsymbol{p}_{i,j}$) $\neq$ clusterOf($\boldsymbol{p}_l$)) then
15:           mergeClusters(clusterOf($\boldsymbol{p}_{i,j}$),clusterOf($\boldsymbol{p}_l$));
16:         end if
17:       else
18:         clusterOf($\boldsymbol{p}_l$) = clusterOf($\boldsymbol{p}_{i,j}$);
19:       end if
20:     else
21:       clusterOf($\boldsymbol{p}_{i,j}$) = clusterOf($\boldsymbol{p}_l$);
22:     end if
23:   end for
24:   if ($\neg$hasCluster($\boldsymbol{p}_{i,j}$)) then
25:     clusterOf($\boldsymbol{p}_{i,j}$) = createNewCluster();
26:     for all ($\boldsymbol{p}_l$ previously examined) do
27:       clusterOf($\boldsymbol{p}_l$) = clusterOf($\boldsymbol{p}_{i,j}$);
28:     end for
29:   end if
30: end for
31: for all ($C_i \in$ Clusters) do
32:   if ($\|C_i\| <$ nMin $\wedge$ $(i - C_i.t_{lastModified}) > t_k$) then
33:     delete $C_i$;
34:   end if
35: end for
36: return Clusters;
```

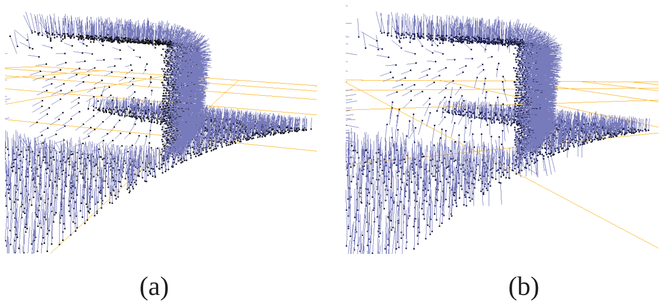

Fig. 4. Estimated normal vectors for a simple data set and k=20: (a) Using nearest neighbors obtained from a static kd-tree (b) Using the proposed *continuous nearest neighbor* approach

V. Results

To validate the efficiency of the proposed segmentation strategy, the algorithm was benchmarked on the three data sets shown in Figure 5. The data was recorded with a SICK LMS400 laser range finder mounted vertically on the mobile robot *ACE* [6]. The robot was commanded to slowly rotate on the spot; for the *Primitives* data set it turned with 0.1 rad/sec, whereas for the *Living Room* and *Human* data sets its speed was 0.2 rad/sec. The LMS400 was queried at a rate of 25 Hz and the laser data was fused with the robot odometry data, which arrived at 16.6 Hz. The algorithms were implemented in C++ and run on a Linux desktop machine with a 3.0 GHz AMD Athlon XP processor and 2GB RAM. Sensor polling and segmentation were run in separate threads that communicated via shared memory. The reader is advised that when implementing Algorithm 3, special care must given to the design of the cluster data structure. For dense data sets, such as the *Primitives* data set, the merge operation in Line 15 can quickly become the bottleneck of the algorithm. Also, the speed-up in Line 4 explained in [6] is essential for the algorithm to operate efficiently.

For all three scenarios, the quality of the normal vectors computed online from the proposed *continuous nearest neighbors* was compared to those computed from a static kd-tree built offline over the entire data set. The results are virtually indistinguishable. A detailed presentation is omitted due to space constraints. Figure 4 provides a representative visual comparison of the normal vector quality for the simple data set from Figure 2. The computation of nearest neighbors and normal vectors required ca. 1 msec per scan for k=20 and for all data sets and neighborhood sizes was found to be insignificant compared to the time required for segmentation.

As for the segmentation each data set required different parameters for a good partitioning, as can be seen in Figures 6, 7 and 8. This is not surprising as the data sets exhibited different point densitites and different levels of surface homogeneity. Figure 6 illustrates how the algorithm agglomeratively grows the clusters on the incoming data of the *Living Room* data set at 25 Hz.

The effect of different thresholds $\rho_{e,max}$ and $\rho_{a,max}$ on the segmentation granularity is illustrated in Figure 7. For the densely sampled *Primitives* data set an extremely small

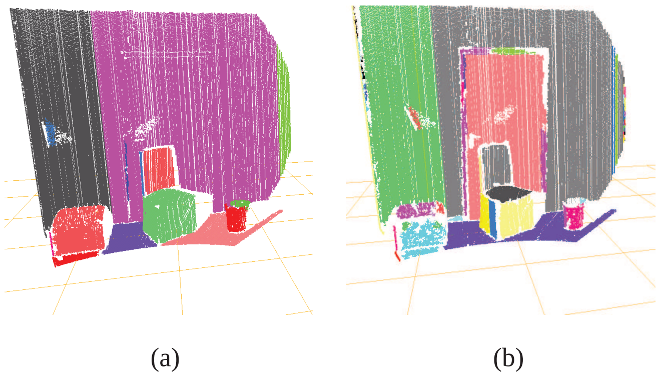
Fig. 7. Effect of different thresholds on segmentation granularity in the *Primitives* data set: (a) k = 20, $\rho_{e,max}$ = 0.04 m, $\rho_{a,max}$ = 0.011 (1.0°), n_{min} = 60, t_k = 1: For medium k and small $\rho_{a,max}$ only the main geometric primitives are separated from each other. (b) k = 15, $\rho_{e,max}$ = 0.04 m, $\rho_{a,max}$ = 0.0055 (0.5°), n_{min} = 60, t_k = 1: The small k and the extremely small $\rho_{a,max}$ uncover fine geometric structures such as the door and the doorframe in the wall.

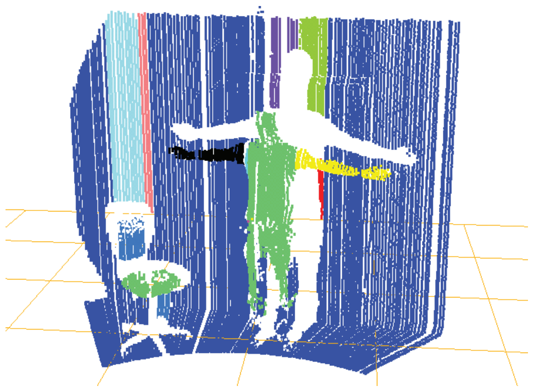
Fig. 8. Realtime segmentation of the *Human* data set at 25 Hz: k = 20, $\rho_{e,max}$ = 0.08 m, $\rho_{a,max}$ = 0.44 (4.0°), n_{min} = 60, t_k = 1

$\rho_{a,max}$ (case (b)) can be used to uncover fine geometric structures (such as the door and the doorframe in the wall). For this case it is also important to use a smaller neighborhood size k, because large neighborhoods tend to 'smooth out' the orientation of normal vectors. In contrast, a medium-sized neighborhood and small $\rho_{a,max}$ (case (a)) yield a good abstraction of the data and less geometric detail.

For all three data sets, the noise removal worked well for the simple choice of t_k=1. Figure 10 shows the number of clusters over time for each data set, using the parameters from Figure 7(b) for the *Primitives*, Figure 8 for the *Human* and Figure 6 for the *Living Room* data set. The very granular *Primitives* segmentation yields many clusters, however, the algorithm does a good job at pruning the overall number of clusters. Figure 9 shows that the time required to segment each incoming scan stays well below 10 msecs for all data sets. This means that - when run on a dedicated processor in its current implementation - the segmentation can be performed at up to 100 Hz.

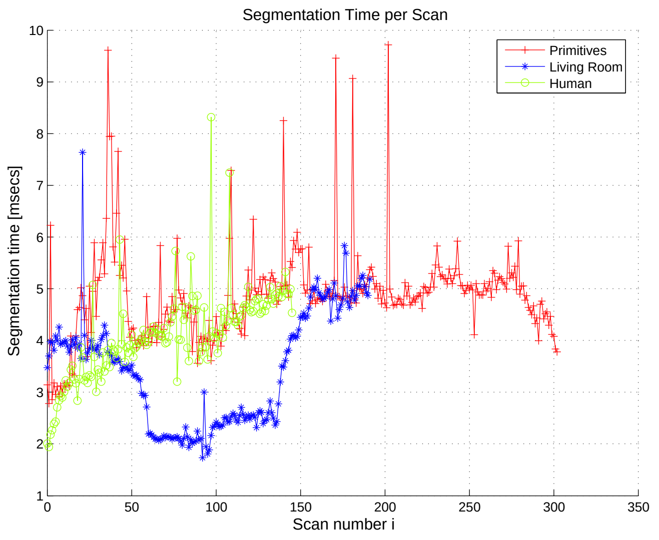

Fig. 9. For all three data sets the segmentation time stays well under 10 msecs, meaning the algorithm can be run at ca. 100 Hz on a dedicated processor.

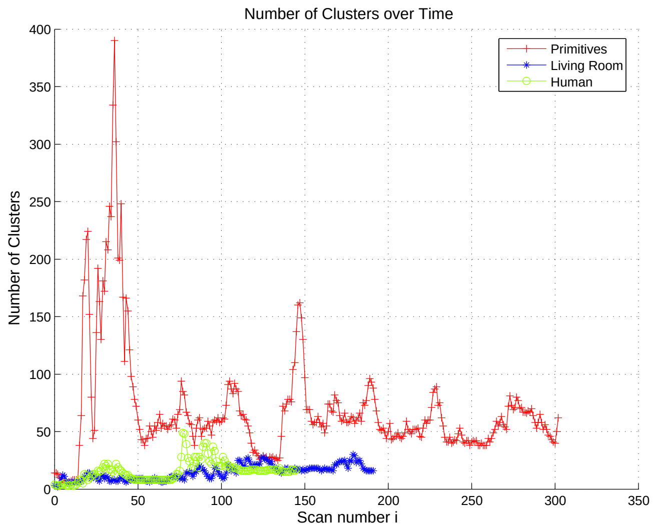

Fig. 10. By merging clusters and filtering noise, the algorithm keeps the number of clusters low for all three data sets. In this diagram the *Primitives* data set segmentation dominates the number of clusters because the parameters from Figure 7(b) were used for segmentation.

VI. Discussion

In this paper an algorithm for segmenting a continuous stream of 3D range data in realtime was presented. The algorithm computes normal vectors of incoming points from their local neighborhood and clusters the new points by assessing their Euclidean and angular distance to previously clustered points. On a set of three benchmark scenarios the algorithm was shown to robustly carry out segmentation in less than 10 msecs per scan and to yield good segmentation results once the appropriate parameters for the given setting have been established. The presented algorithm is a valuable tool for robotics scenarios in which a sweeping range sensor is used for environment perception. The segmented surfaces

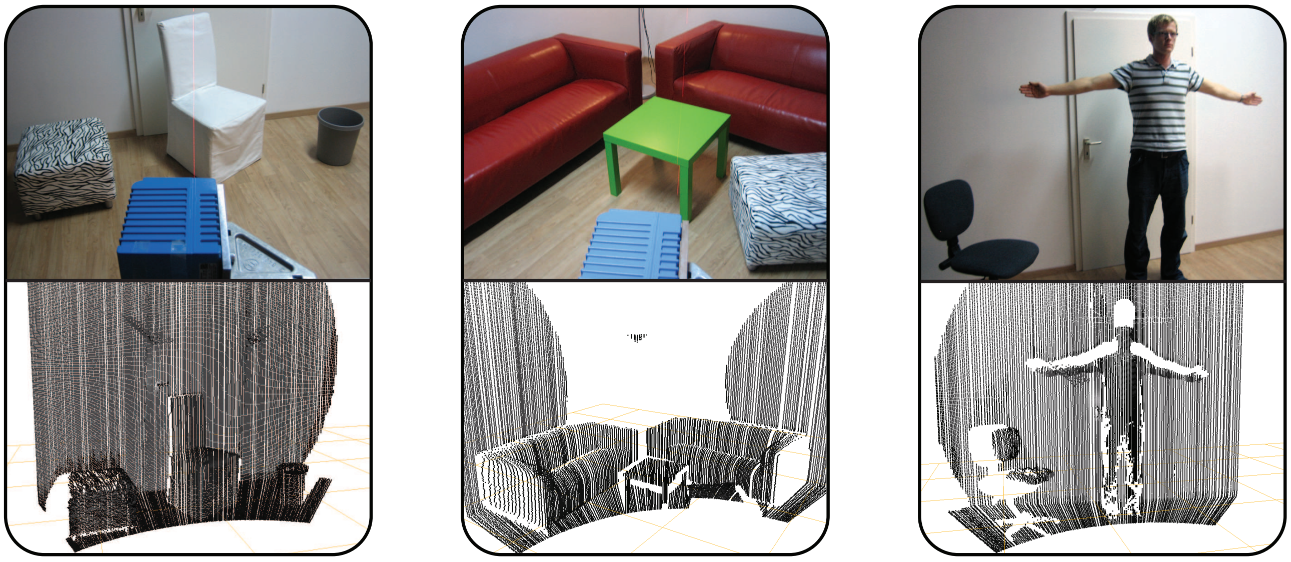

Fig. 5. Three benchmark scenarios. *Top row*: Photos of the scenes; *Bottom row*: Point clouds with remission values; (a) *Primitives*, 310 scans, 72,212 points, 0.1 rad/sec (b) *Living Room*, 206 scans, 34,501 points, 0.2 rad/sec (c) *Human*, 160 scans, 35,465 points, 0.2 rad/sec

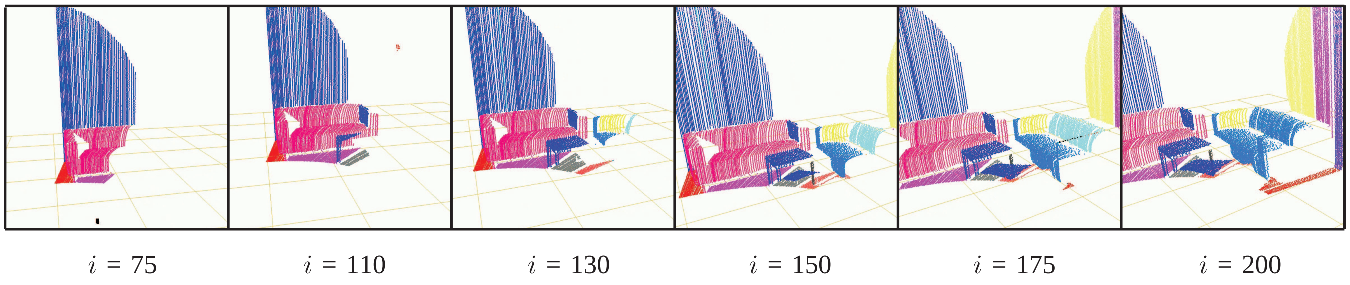

Fig. 6. Realtime segmentation of the *Living Room* data set at 25 Hz: k = 25, $\rho_{e,max}$ = 0.05 m, $\rho_{a,max}$ = 0.22 (2.0°), n_{min} = 50, t_k = 1. The last two frames show how two distinct clusters (light blue and dark blue) on the right sofa are merged into one (dark blue) as the connecting armrest is scanned.

can be used as features in the context of object recognition and understanding, for example as an input for a learning architecture. A central issue of future research will be the handling of occlusions which can be seen to cause discontinuities in some of the clustering results. Also, the automation of some of the required parameter tuning would be a valuable addition.

ACKNOWLEDGMENTS

This work is supported in part within the DFG excellence initiative research cluster *Cognition for Technical Systems – CoTeSys*, see also `www.cotesys.org`.

REFERENCES

[1] P. J. Besl and R. C. Jain, “Segmentation through variable-order surface fitting,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 10, no. 2, pp. 167–192, 1988.

[2] X. Jiang, “Recent advances in range image segmentation,” in *Selected Papers from the International Workshop on Sensor Based Intelligent Robots*, (London, UK), pp. 272–286, Springer-Verlag, 1999.

[3] S. Gächter, V. Nguyen, and R. Siegwart, “Results on range image segmentation for service robots,” in *ICVS ’06: Proceedings of the Fourth IEEE International Conference on Computer Vision Systems*, (Washington, DC, USA), p. 53, IEEE Computer Society, 2006.

[4] O. Wulf and B. Wagner, “Fast 3d-scanning methods for laser measurement systems,” in *Proceedings of the 14th International Conference on Control Systems and Computer Science (CSCS14)*, (Bucharest, Romania), July 2003.

[5] D. L. Elsner, R. T. Whitaker, and M. A. Abidi, “Volumetric reconstruction of objects and scenes using range images,” *Digital Signal Processing*, vol. 9, pp. 120–135, April 1999.

[6] K. Klasing, D. Wollherr, and M. Buss, “A clustering method for efficient segmentation of 3d laser data,” in *Proceedings IEEE ICRA*, 2008.

[7] J. H. Friedman, J. L. Bentley, and R. A. Finkel, “An algorithm for finding best matches in logarithmic expected time,” *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.

[8] S. Arya and D. M. Mount, “Approximate nearest neighbor queries in fixed dimensions,” in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1993.

[9] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, “Surface reconstruction from unorganized points,” *Computer Graphics*, vol. 26, no. 2, pp. 71–78, 1992.

[10] K. Klasing, D. Althoff, D. Wollherr, and M. Buss, “Comparison of surface normal estimation methods for range sensing applications,” in *Proceedings IEEE ICRA*, 2009.