Data Structure for Efficient Dynamic Processing in 3-D

Jean-François Lalonde

CMU-RI-TR-06-22

May 2006

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

© Carnegie Mellon University

Submitted in partial fulfillment of the requirements for the degree of Master of Science

Abstract

In this paper, we consider the problem of the dynamic processing of large amounts of sparse three-dimensional data. It is assumed that computations are performed in a neighborhood defined around each point in order to retrieve local properties. This general kind of processing can be applied to a wide variety of applications. We propose a new, efficient data structure and corresponding algorithm that significantly improve the speed of the range search operation and that are suitable for on-line operation, where data is accumulated dynamically. The method relies on taking advantage of overlapping neighborhoods and the reuse of previously computed data as the algorithm scans each data point. To demonstrate the dynamic capabilities of the data structure, we use data obtained from a laser radar mounted on a ground mobile robot operating in complex, outdoor environments. We show that this approach considerably improves the speed of an established 3-D perception processing algorithm.

Prepared through collaborative participation in the Robotics Consortium sponsored by the U.S Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-209912.

Thesis supervisor: Prof. Martial Hebert

Contents

1 Introduction **1**

2 Related work **2**
2.1 Data structures . . . 2
2.2 Robotics . . . 2

3 Nature of the computation **3**
3.1 Admissible conditions . . . 3
3.2 Examples . . . 4
3.3 Dynamic aspects . . . 4

4 Approach **5**
4.1 General principle . . . 5
4.2 Direct computation approach . . . 6
4.3 Naive approach . . . 6
4.4 Optimized scan approach . . . 7

5 Implementation **9**
5.1 Algorithm . . . 9
5.2 Data structure for efficient range search . . . 9
5.2.1 Static case . . . 9
5.2.2 Dynamic case . . . 11
5.3 Examples, revisited . . . 11
5.4 Application of interest . . . 12
5.4.1 Voxel-wise classification . . . 12
5.4.2 Voxelization process . . . 12
5.4.3 Previous implementation . . . 13

6 Experiments **14**
6.1 Data collection . . . 14
6.2 Static data experiments . . . 15
6.2.1 Validation of theoretical results . . . 15
6.2.2 Comparison for different terrain types . . . 15
6.2.3 Parameters influence . . . 16
6.3 Dynamic data experiments . . . 19
6.3.1 Live parameters . . . 19
6.3.2 On-line timing . . . 21
6.3.3 Comparison for different terrain types . . . 21
6.3.4 Parameters influence . . . 22

7 Conclusion **23**

1 Introduction

The problem of processing large amounts of dynamic, sparse three-dimensional data is very challenging because computations must keep up with the continuous flow of data coming in at a high rate. Traditional algorithms and data structures designed for batch processing are inadequate in that particular setting because they cannot handle dynamic data efficiently.

An example of such situation arises in the domain of perception for ground mobile robots. Recent advances in sensor design have enabled the use of laser radars (or ladars) to improve the three-dimensional (3-D) perception capabilities of outdoor robots [18, 4]. These sensors provide up to one hundred thousand 3-D points per second, with range resolution on the order of one centimeter. For example, Figure 1 shows a typical 3-D point cloud obtained from a ladar sensor[1]. As the input data rate increases, it is critical to design data structures that can efficiently store large amounts of data and quickly perform basic operations such as insertion, memory access, and range search.

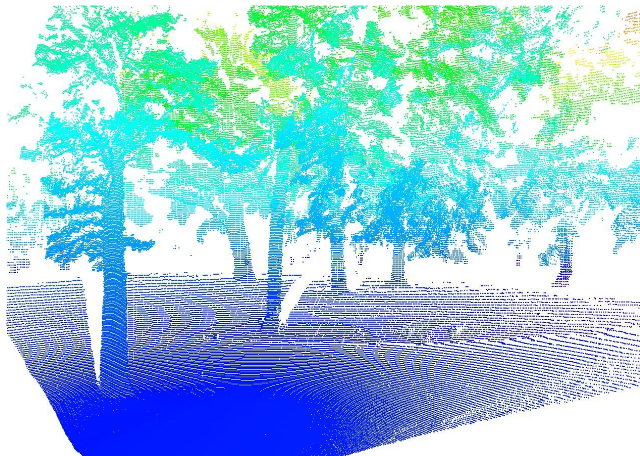


Figure 1: Example of raw 3-D point cloud from a ladar. The points are color-coded by elevation, from blue (low) to red (high), and are enlarged for clarity. This dataset is made of 259,528 points.

In this paper, we present a new approach for handling 3-D data for processing of dynamic data. The data processing we are concerned with are local methods, where for a given point, some operations are performed using a support volume centered around that particular point. The core of the approach is to minimize computation by re-using previously computed intermediate results. The approach is demonstrated with data from a ground mobile robot, the Demo III XUV [2], for ladar-based terrain classification [19]. Instead of using different data structures for different types of terrain, the approach presented in this paper *automatically adapts* to the environment by analyzing the local distribution around each point. We review the method, previously introduced

[1]The figures in this paper are best viewed in color.

in [9], that was suitable only for static data, extend the approach to handle dynamic data, and present results obtained by performing live experiments that simulate the conditions on-board the robot.

The remainder of the paper is divided as follows. First, Section 2 presents related work, mainly in robotics and efficient data structures. Then, Section 3 presents the generic category of 3-D processing methods suitable to our method, which is then presented in details in Section 4. We then present the implementation details in Section 5, followed by experiments on static and dynamic data in Section 6.

2 Related work

In this section, we review related work in efficient data structures as well as in ground mobile robotics.

2.1 Data structures

Traditional pre-computed tree-based data structures (Kd-tree, range tree) are efficient for performing range search. Unfortunately, because there is a large processing overhead due to their initialization, their performance rapidly degrades as additional data is inserted after construction [15]. In [11], Lersch presents a data structure for structural segmentation of 3-D point-cloud data, called the windowed priority queue. The approach focuses on the indexing of the data for fast retrieval. The computation performed is similar to the one used in our work [19] but it is performed off-line. Approximate search is sometimes proposed, but it is not considered here because we aim to obtain exact results.

Gao proposes an interesting work on efficient proximity search in 3-D for kinetic data [6]. The author extends Voronoi diagram and Delaunay triangulation to an environment made of 3-D voxels. A simple example is provided. However, it is unclear how this can be scaled efficiently to handle higher point density.

Machine learning and statistical methods require efficient data structures for nearest neighbor search, range search, regression or kernel operations [12, 7]. However, most of the attention is focused on high dimensional data sets rather than dynamic data sets of low dimensionality.

2.2 Robotics

Three-dimensional data has been extensively used for outdoor robot navigation, using stereo cameras or laser radars. If the obstacles are expected to lie on the ground (in desert or planetary environments, for example), one common approach is to create a 2D grid of the terrain that stores projected classification results in each cell. The processing can take place in the sensor reference frame (range image) and the results are then back-projected in that 2D grid. An alternative is to create an intermediate digital elevation map by projecting the 3-D data into a 2D-1/2 map and then doing some processing, convolving a robot model with the terrain for example [18]. In both cases, processing is not performed in 3-D.

If the terrain contains vegetation such as trees, grass or bushes, the previous approach is not sufficient. A full 3-D representation is necessary to represent the environment and to produce a better and higher level of scene interpretation. One such approach has been demonstrated for vegetation detection and ground surface recovery in [8, 4]. In both cases, a dense 3-D grid representation of the environment is maintained around the robot and scrolled as it moves. A ray tracing algorithm updates each voxel by counting the number of times it has been traversed by or stopped a laser ray. Such statistics are then used to determine if the voxel is likely to be the load bearing surface or vegetation. The data processing requires data insertion and retrieval, but no range search.

Similarly, 3-D occupancy grid approaches create a voxelized 3-D model by performing insertion and access but are not optimized for range search [14]. Operations in 3-D can sometimes be reduced to 2-D operations as shown in [13] for natural environment navigation in specific cases. Unfortunately, in general, we cannot follow such an approach.

3 Nature of the computation

The focus of the analysis is directed toward a generic category of 3-D processing methods that possess similar characteristics. In this section, we first present these characteristics, show examples of techniques that possess such attributes, then discuss important aspects related to dynamic processing.

3.1 Admissible conditions

The computations performed belong to a general class of processing methods that require the retrieval and the use of the data within a support volume around a point of interest. To help formalize the problem, a specific notation is introduced:

- $PCD = \{p_1, p_2, ..., p_m\}$: point cloud data, a set of m unorganized 3-D points;
- $p_k = (x_k, y_k, z_k)$: a 3-D point in PCD;
- $N(p_i, s)$: the set of points around a given point p_i, within a volume of radius (also referred to as *scale*) s, such that $\| p_i - p_k \|_\infty \leq s$ with $i \neq k$;
- $F(N(p_i, s))$: a function over the neighborhood points.

An admissible 3-D processing technique must define the function $F(N(p_i, s))$, with constant s.

In addition, the approach proposed in Section 4 is suitable for voxelized data only, that is 3-D data that has been binned into discrete containers (the 3-D analogy to pixels). Voxels are used to reduce the amount of memory needed to store the data and to simplify computations. The local support volume is therefore defined in number of voxels around a voxel of interest.

3.2 Examples

Many well-known 3-D processing techniques obey the conditions presented in the previous section. We now present four of them, along with their corresponding F, and show in Section 5.3 how they can each be implemented within our framework.

1. Possibly the simplest computation would be to simply recover the number of neighbors within a support region. Using the notation previously introduced, this is represented by $F_1(N(p_i, s)) = |N(p_i, s)|$.

2. A slightly more complex procedure would be the retrieval of the neighboring points themselves. This requires the knowledge of each of the p_k individually. F_2 is represented by the set of points $\{p_k\}$ such that $p_k \in N(p_i, s)$.

3. One popular processing technique is the extraction of the principal components of a region by Principal Components Analysis (PCA). This requires the computation of the local covariance matrix and the extraction of its eigenvectors and corresponding eigenvalues. We can represent this by

$$F_3(N(p_i, s)) = \sum_{p \in N(p_i,s)} (p - \bar{p})^T (p - \bar{p}) \quad (1)$$

   where $\bar{p} = \frac{1}{|N(p_i,s)|} \sum_{p \in N(p_i,s)} p$.

4. A final example is kernel density estimation [3], which can also be used in 3-D data processing (see [17] for instance). Its local neighborhood function is given by

$$F_4(N(p_i, s)) = \frac{1}{|N(p_i, s)|h_\theta} \sum_{p \in N(p_i,s)} \kappa \left(\frac{p_i^T \theta - p}{h_\theta} \right) \quad (2)$$

   with κ a kernel function scaled to the bandwidth h_θ, and θ is a direction in $\mathcal{R}^3$. The optimization then consists in finding the direction θ that maximizes Equation 2.

3.3 Dynamic aspects

We also restrict our analysis to dynamic data that is produced sequentially by a single source (a mobile sensor, for example). As we mentioned in Section 2.1, traditional efficient data structures are ill-suited for this kind of data, because the bounds are unknown a priori, and might grow as more data is inserted in the data structure. Figure 2 shows an example of data accumulated from a ladar sensor placed on a ground mobile robot. Since the data boundaries might grow arbitrarily large, it will be assumed that the dynamic data comes from a single source that is moving in an environment, and that we are interested in performing local point-wise computations on a restricted volume that is following the sensor as it moves. As the sensor moves, the data is assumed to be co-registered in a common reference frame.

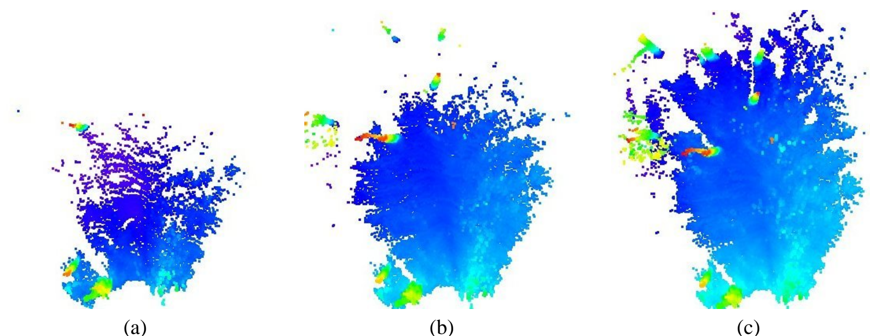


Figure 2: Top-down view of an example of dynamic data, as it is accumulated from a robot driving through the woods. Snapshots of the data are shown at (a) $t = 2.5$ s, (b) $t = 12.5$ s and (c) $t = 25$ s. The raw data is color-coded by elevation, from blue (low) to red (high).

4 Approach

In this section, we first present the general idea used to increase the processing speed of the general class of techniques presented in the previous section. We then compare various approaches that exploit this idea.

4.1 General principle

The approach draws its inspiration from the 2-D image processing domain. Real-time area-based correlation stereo algorithms take advantage of the overlap in computations and reuse previously-computed data to achieve greater execution speed. For example, in [5], Faugeras et al. decompose the zero mean normalized correlation score into partial sums, and add/remove only columns contribution as the epipolar line is scanned. A similar approach is used to handle change of line by removing/adding line contribution at the image borders. We apply the same principle to a voxel representation in 3-D.

However, there are two fundamental differences between the 2-D and 3-D cases that justify the need for a novel approach. First, in stereo, correlation is performed along the epipolar line, resulting in a unidirectional scanning. However, many different strategies exist to scan the 3-D space (obtained by permuting the order of axes). Figure 3 illustrates two examples. Second and most importantly, 3-D data is usually very sparse, that is, a large number of voxels are empty. In fact, it is estimated that the percentage of occupied voxels varies between 2 % and 12 % for voxel sizes of 10 cm to 1 m of edge length respectively. This contrasts with images in which each pixel contain information that can easily be retrieved in a subsequent step.

First, we introduce some notation. Given a volume V subdivided into voxels, let:

- $n = n_x \times n_y \times n_z$ the total number of voxels in the volume;
- v be the number of occupied (non-empty) voxels in the volume;

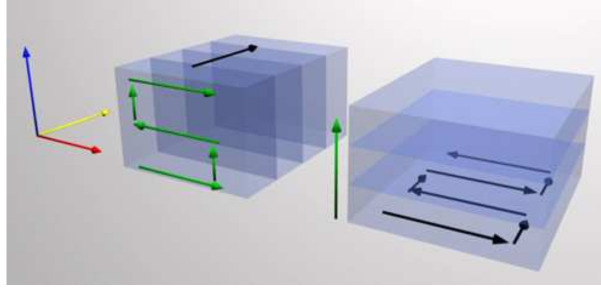


Figure 3: 2 possible scanning strategies in 3-D

- $k = (2r + 1)$ the neighborhood size (for range search), where r is the radius, in number of voxels. The neighborhood (also referred to as *support region*) is rectangular as opposed to spherical.

We derive the following equations to compute the number of voxels that need to be visited for range search computation. To simplify the expressions, we do not take into account the differences that may arise at the boundaries of the volume: for a large volume, they are assumed to be negligible. We also assume that the support region is isotropic, that is, k is equal in each dimension.

4.2 Direct computation approach

Typical implementations usually rely on an exhaustive approach that visit every k^3 voxels in every neighborhood each time (see [19] for example), without data reuse. By assuming that the occupied voxels can be retrieved directly (in a lookup table, for example), the total number of visited voxels is simply

$$t_{direct} = vk^3 \tag{3}$$

This method takes advantage of the sparseness of the data: only occupied voxels are visited. A suitable data structure for this approach is a sparse voxel representation, where only the occupied voxels are stored, and accessed via a hash table [19]. However, on the dense regions, much of the computations are repeated many times because of overlapping neighborhoods.

4.3 Naive approach

A naive approach would be to act as if the volume was densely populated, that is, to scan the whole volume in an ordered way while executing neighborhood computation for every voxel, even the empty ones. This is the direct translation of the 2-D approach for dense correlation using 3-D data, see Figure 4 for a graphical example.

An appropriate data structure is the dense voxel representation, in which memory space is reserved for each voxel of the volume at the beginning. The drawback of this

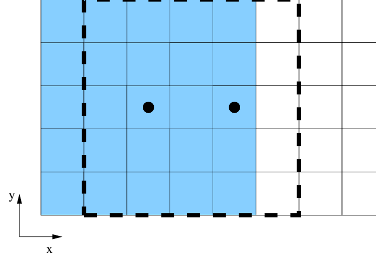


Figure 4: Illustration of naive approach in 2-D. Each square is a voxel, and dots indicate occupied voxels. The shaded area represents the local support region defined for the leftmost occupied voxel (dotted). The dashed outline is the support region of the middle voxel, computed even though the voxel contains no data.

choice is that memory usage is mostly inefficient, thus limiting the volume of interest. However, it allows the efficient and ordered traversal of the volume.

Since it re-uses previously-computed data and recomputes neighborhood slices each time, the number of visited voxels at each step is $2k^2$. Since it is done over the whole volume, $n = v$, the total number of visited voxel is

$$t_{naive} = 2nk^2$$

This method does not take advantage of the sparseness of the data, and must scan the whole volume each time. The condition for t_{naive} to be less than t_{direct} is :

$$t_{naive} < t_{direct} : 2nk^2 < vk^3$$

$$\frac{v}{n} > \frac{2}{k} \tag{4}$$

If $k = 9$, then $v/n > 0.23$. At least 23% of the voxels in the volume must be valid for the naive method to be faster than the exhaustive one. As our experimental results show (Section 6), the v/n ratio tends to be very low, typically under 2%, justifying the need for a better approach.

4.4 Optimized scan approach

This method takes advantage of the dense regions in the volume, but also avoids unnecessary loops over large portions of empty space. The principle is the same as in Section 4.3, but the computations are done only on the occupied voxels. Therefore, this algorithm will need to find the *previous occupied voxel* and determine if it is close enough. This concept is related to the volume traversal order because it directly depends on the scanning direction.

Since this approach requires the voxels to be scanned in predetermined order, let d be the distance (in number of voxels) between the current voxel and the previous

occupied voxel in the volume, along the scanning direction. d is a random variable, with an unknown distribution that depends only on the data.

We note that reusing data implies that $2dk^2$ voxels must be visited (instead of k^3 otherwise). Therefore,

$$2dk^2 < k^3 \Longrightarrow d < \frac{k}{2} \tag{5}$$

which is the criterion that indicates if previous data should be reused. Figure 5 illustrates the situation in 2-D. In this example, $d = 2$ so the two rightmost columns are added to the neighborhood of the voxel at $(2, 2)$, and the two leftmost columns are subtracted. In total, 20 voxels (instead of 25) must be visited.

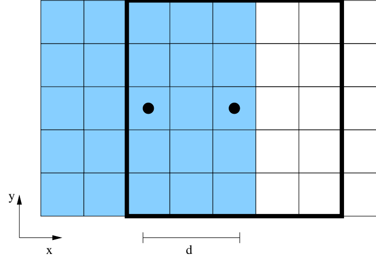


Figure 5: Illustration of data reuse with sparse 2-D data with $k = 5$. Each square is a voxel, and dots indicate occupied voxels. The shaded area represents the local support area defined for the leftmost occupied voxel (dotted). The outlined area is the support area of same dimensions for the other occupied voxel. This approach takes advantage of the overlap (3 central columns) in the support areas. As opposed to the naive approach (c.f. Section 4.3), this approach performs range search only at voxels that contain data.

Since the number of visited voxels depends on d, its expected value is used in the analysis to derive an expected bound. We define $\bar{d}$ as the expected value of d over the voxels for which $d < \frac{k}{2}$:

$$\bar{d} = \sum_{i=1}^{\lfloor \frac{k}{2} \rfloor} i \frac{P(d = i)}{P(d < \frac{k}{2})}$$

$t_{optimized}$ is then computed as the expected number of visited voxels:

$$\begin{aligned} t_{optimized} &= v\left(2\bar{d}k^2 P[d < \frac{k}{2}] + k^3 P[d \geq \frac{k}{2}]\right) + n \\ &= v\left(2\bar{d}k^2 P[d < \frac{k}{2}] + k^3 (1 - P[d < \frac{k}{2}])\right) + n \\ &= v\left((2\bar{d}k^2 - k^3) P[d < \frac{k}{2}] + k^3\right) + n \end{aligned} \tag{6}$$

The condition for $t_{optimized}$ to require less operations than t_{direct} becomes:

$$vk^3 > v\left((2\bar{d}k^2 - k^3)P[d < \frac{k}{2}] + k^3\right) + n$$

$$\frac{v}{n} > \frac{1}{\left(k^3 - 2\bar{d}k^2\right)P[d < \frac{k}{2}]} \tag{7}$$

In the worst case, $P[d < \frac{k}{2}] = 0$, Equation 6 becomes $t_{optimized} = vk^3 + n$, which is equivalent to $t_{optimized} = t_{direct} + n$. The only difference with the exhaustive method is the need to visit each voxel in the volume.

As shown in Section 6.2.1, Equation 7 is a lower limit on which we can guarantee a lower number of visited voxels, which in turns results in a decrease in computation time.

5 Implementation

The approach described in the previous section derives bounds on the expected number of visited voxels for different range search techniques. We now describe the algorithm itself, and the corresponding data structure implemented to test it. We also show implementation ideas for the examples presented in Section 3.2, and finally introduce the application used to evaluate the algorithm on real data.

5.1 Algorithm

The proposed algorithm implementing the approach described in Section 4.4 is illustrated in pseudo-code by Algorithm 1. The algorithm automatically determines in what direction it should look for re-usable data.

Depending on the type of processing used in a particular application, the n_p, s_p, s_c and n_c from Algorithm 1 are defined differently (see Section 5.3 for the implementation of the examples introduced in Section 3.2). The rest of the algorithm is exactly the same, which makes it suitable for a large variety of 3-D processing algorithms.

5.2 Data structure for efficient range search

We now present a data structure that supports the range search technique described by Algorithm 1. We first describe its static implementation, then proceed to the dynamic version, suitable for on-line applications.

5.2.1 Static case

A static dense voxel grid representation, as used in Section 4.3 is insufficient for the needs of Algorithm 1. We therefore propose a variant of the dense voxel representation that maintains an array of pointers to previously visited occupied voxels in memory. Figure 6 illustrates the principle in 2-D, but it is easily generalizable to three dimensions. The 5×4 grid is the original dense voxel representation, and the two additional

Algorithm 1: General scanning algorithm

Input: V the voxelized volume and its boundaries

1 **for** *every occupied voxel in* V **do**
2 $v_c \leftarrow V(x, y, z)$, the current occupied voxel
3 Retrieve d, the distance to the closest occupied voxel that has already been visited, and dir, the direction associated with it
4 **if** $d \geq \frac{k}{2}$ **then**
5 $n_c \leftarrow$ the whole k^3 neighborhood {There's no sufficiently close occupied voxel}
6 **else**
7 Retrieve v_p, the previous occupied voxel located at distance d in direction dir of v_c
8 $n_p \leftarrow$ neighborhood computation of v_p
9 $s_p \leftarrow$ the d rightmost (along dir) slices of v_p
10 $s_c \leftarrow$ the d leftmost (along dir) slices of v_c
11 $n_c \leftarrow n_p - s_p + s_c$
12 **end**
13 **end**

arrays are pointers to previously-visited, occupied voxels in memory. In this example, the scanning order is x then y, and the current voxel v_c is filled in blue, whereas the previously visited voxels are drawn in a lighter shade of blue. The occupied voxels are dotted. The algorithm has access to the nearest previously visited occupied voxel just by looking at the cells in red, which correspond to the (x, y) position of v_c. v_p (position $(2, 0)$ in this example), can then easily be retrieved.

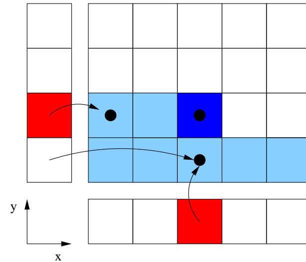


Figure 6: 2-D example of the proposed data structure. Dense voxel representation augmented with additional side vectors that store the location of the previously computed results, in each dimension. The dotted squares represent occupied voxels. The dark blue square represent the current voxel, and the previously scanned voxels are shown in light blue. The algorithm examines the cells in red, which point (illustrated by the arrows) to previously computed range search results. It then chooses the closest results for v_p (c.f. line 7 in Algorithm 1.

A 3-D version of the data structure illustrated in Figure 6 is implemented in C++ for Linux OS. It is templatized and can be used for a wide variety of applications without having to change its core implementation. Basic optimization rules are followed for efficiency. For example, run-time operations such as polymorphism are avoided and data is accessed using references only.

Since it is expected that the majority of the voxels in the data structure will be empty, each voxel is initialized to a NULL pointer, and memory is allocated each time a voxel is created. In this static version of the data structure, memory is freed only when the program terminates.

5.2.2 Dynamic case

When dynamic data is used, the data structure presented in the previous section is inadequate because data might appear outside the volume defined by the dense voxel representation. However, the volume cannot be made arbitrarily large because of prohibitive memory requirements. Therefore, a limited volume is used, and it is scrolled as the sensor moves in the environment.

To scroll the volume, the modulo operator (% in C++) is necessary but can be efficiently implemented using a logical AND operator (& in C++), as long as the data structure's dimensions are a power of two. For example, $5\%2^2 = 5\&(2^2 - 1) = 1$. No integer division is thus needed, making memory access a very fast operation. In addition, when the volume is scrolled in one dimension, a 2-D slice of voxels must be invalidated. To avoid filling memory over time as the volume is scrolled multiple times, invalidated memory is freed and the pointers reset to NULL.

5.3 Examples, revisited

We now show the implementation of the various examples introduced in Section 3.2. For each of them, we need to define what is stored in the n_c, s_p and s_c variables introduced in lines 5, 9 and 10 respectively in Algorithm 1. We also need to define how to add and subtract the partial results, as required in line 11.

1. To recover the number of neighbors, recall that $F_1(N(p_i, s)) = |N(p_i, s)|$. Therefore, we simply need to count the number of occupied voxels in each slice and store that number as an integer. It can then be added and subtracted straightforwardly.

2. The retrieval of the points within the neighborhood ($F_2(N(p_i, s)) = N(p_i, s)$) requires a more complex implementation, since it requires spatial information about the points, which was not the case in the previous example. Although it has not yet been implemented, n_c can be a binary search tree (BST) storing the memory addresses of the points. The addition of the partial results is simply the insertion of new points in the BST. The subtraction operator needs to find each of the points ($O(\log n)$ operation in a BST) and remove them. Since it requires additional operations, the lower bound derived in Equation 7 is not applicable. This is the object of our current work.

3. The covariance matrix can be computed from Equation 1. However, it can be shown that the matrix can be decomposed in terms of sums, sums of squared and sums of pairwise cross products of the 3-D point coordinates. This information is stored in a vector of nine floating point elements v_{sums}, which define n_c. The addition and subtraction are performed element-wise on that vector. Therefore, the local covariance matrix computation does not require spatial information about each of the individual points and the partial sums vector can be computed directly. The extraction of the eigenvectors and corresponding eigenvalues can be performed once the complete neighborhood has been scanned.

4. Equation 2 shows that the kernel function depends on the distance between the current point p_i and the mean of the points in the neighborhood $\bar{p}$, therefore it also requires spatial information. It is thus impossible to re-use the previous results directly, because the kernel functions do not overlap between two neighboring points. In that situation, the best option is to use the implementation presented in example 2 and retrieve the neighbors efficiently, and to apply the kernel function afterwards. It cannot be computed directly, as in example 3.

5.4 Application of interest

An interesting application of 3-D processing based on local computations it point-wise classification to improve the perception capabilities of ground mobile robots, as shown in [19, 10]. We now proceed to describe this application in greater details, as it will be used for evaluating the performances of the data structure and algorithm in Section 6.

5.4.1 Voxel-wise classification

Using 3-D ladar data as input, we perform voxel-wise classification to detect vegetation, thin structures and solid surfaces. The method relies on the use of the covariance matrix to extract features via PCA. For each voxels, the approach computes the matrix within a support volume and then extracts its principal components (eigenvalues). A linear combination of the components, and the associated principal directions define the features. A model of the features distribution is learned off-line, prior to the mission, from labeled data. As the robot traverses a terrain, data is accumulated, features computed and maximum likelihood classification performed on-line. Figure 7 presents an example of such terrain classification. For additional information about the classification process details please see [19, 10].

5.4.2 Voxelization process

In order to handle the high data rate coming from the laser sensor (roughly one hundred-thousand points per second), we previously implemented a compression process that dramatically reduces the amount of data to handle without compromising the features computation accuracy [19]. The compression scheme relies on a voxelization of the data at a specific scale (typically, 10 cm side voxels are used). As explained in example 3 in Section 5.3, the covariance matrix can be recovered from a partial sums vector

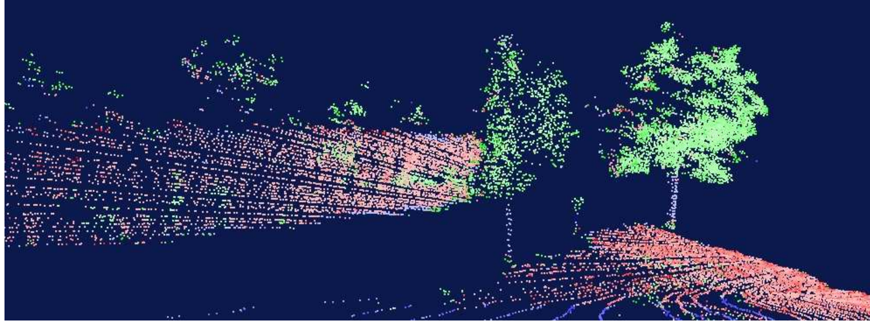


Figure 7: Example of terrain classification. Surfaces, linear structures and scattered points are colored in red, blue and green respectively.

v_{sums}. Each voxel stores such a vector, that represents the partial sums of all the raw points that fall into it. Therefore, storing the location of each 3-D point is not needed, resulting in significant data compression while keeping all the information necessary to recover the true covariance matrix. To compute the saliency features for a given voxel, range search is performed and the neighboring voxels' v_{sums} are added together to recover the scatter matrix.

5.4.3 Previous implementation

An earlier data structure using the voxelization scheme presented in the previous section has been implemented and its capabilities demonstrated in [19]. This previous data structure is a sparse voxel representation: it stores only the occupied voxels in a continuous vector in memory. The memory index can then be reconstructed via a hash-map, the key being constructed from the voxel's 3-D coordinates. The hash key is 64 bits long and made of the concatenated index value of the z, y and x coordinates. The length of the key ensures that the global map is large enough and does not need to be scrolled. Because it is not suitable to implement Algorithm 1, it uses the *direct computation* range search technique presented in Section 4.2.

This approach has been extensively tested on-board a ground mobile robot. The processing time on current hardware (Pentium IV at 3GHz, with 3 GB of RAM) allows operation at slow speed (1-2 m/s) with a hundred-thousand input points per second, depending on the complexity of the terrain. The motivation behind the new data structure introduced in this paper is to increase the processing speed to handle higher terrain representation resolution and enable faster robot navigation speed. The next section shows comparative experiments of the two data structures that both implement the voxelization process described earlier, using static and dynamic data collected from a ground mobile robot.

6 Experiments

This section presents experiments performed on static and dynamic data. First, we present the data collection procedure, then show results that demonstrate the superiority of the new data structure (which will be identified as *optimized scan*) over the previous implementation (*direct computation*) using static data. Then, we show that the approach is suitable for live processing by presenting results obtained with playback data, simulating the conditions on-board the robot.

6.1 Data collection

The data used in the following experiments were collected using the GDRS eXperimental Unmanned Vehicle (XUV, [8]). A similar platform was used in the Demo-III program [1]. This car-sized autonomous vehicle, shown in Figure 8, is equipped with a high-speed rugged range sensor that produces more than 100,000 3-D points per second with cm range resolution and a maximum range up to 80 m. The laser is mounted on a turret scanning the ground surface. Additional information on this specific version of the laser used can be found in [16].

Figure 8: The GDRS eXperimental Unmanned Vehicle (XUV) used to acquire data for the experiments presented in this section.

Field tests were conducted in Central Pennsylvania, USA. The terrain used is several square kilometers in size and includes various natural features such as open space meadows, wooded areas, rough bare ground and ponds, and is traversed by a network of trails. The terrain elevation varies significantly between different areas.

For the experiments on static data, the 3-D points are first accumulated then sequentially stored in a file in random order for batch processing, without any time information.

The algorithm then reads all the points in the data set, inserts them in the data structure, and the experiment is performed. For the dynamic data case, the laser range and robot navigation data are stored on disk in the laser native file format, which can be used for playback and re-create the conditions on-board the robot. The computations are performed using an off-the-shelf computer (Intel Xeon, 2.8 GHz, 1.5 GB RAM).

Table 1: Statistics for the different terrains with 10 cm voxels.

Terrain	Size n (cells)	Raw data	v	$\frac{v}{n}$	$\bar{d}$	$P[d < \frac{k}{2}]$
Flat ground	200x200x30	2.0×10^6	59,275	0.049	1.0263	0.9917
Forest	160x250x40	1.7×10^6	112,001	0.070	1.0519	0.9923
Tall grass	200x300x30	1.2×10^6	117,756	0.065	1.0678	0.9856

6.2 Static data experiments

The data structure version used in this section is the one introduced in Section 5.2.1, and is suitable only for static data. This data structure was also used in [9].

6.2.1 Validation of theoretical results

To validate Equation 7, a set of synthetic random data is generated over a volume of interest with various point density. The values of $\bar{d}$ and $P[d < \frac{k}{2}]$ are then computed on the voxelized data.

First, with $k = 9$ and a volume occupancy of 18%, we obtain $\bar{d} = 1.81$ and $P[d < \frac{k}{2}] = 0.8973$. Because Equation 7 predicts a minimum occupancy of 0.2%, the inequality is satisfied, meaning that we should observe some improvement in execution speed. Indeed, we experimentally observe a speedup of 4.92 over the previous method.

With a volume occupancy of 1%, we obtain $\bar{d} = 2.48$ and $P[d < \frac{k}{2}] = 0.11$. This leads to a minimum occupancy of 2.6%, which doesn't satisfy the inequality of Equation 7. However, the new method is still faster than the first by a factor of 1.34.

These results emphasize the fact that if Equation 7 is satisfied, the new method is guaranteed to be faster than the first. Under this limit, there is no guarantee because the analysis is only considering the average over the whole volume and not taking into account the local clustering of the data.

6.2.2 Comparison for different terrain types

In this section, we compare and analyze the performance of our approach for different types of terrain: bare ground (Figure 9-(a)), highly cluttered forest (Figure 9-(b)) and an open space with vegetation cover (Figure 9-(c)). The bare ground scene includes a gravel trail bordered by a jersey barrier. A concertina wire is laid across the trail. The forest scene is made of large tree trunks scattered over a rough terrain covered with short grass and debris. The last terrain is a side slope covered by dense, dry, waist-high grass, with some large poles. The statistics relative to each data set are presented in Table 1, and results are presented in Figure 10. The initial results seem to show that the type of terrain does not influence $\bar{d}$ and $P[d < \frac{k}{2}]$. All these data set have very high point density, which probably leveled the results.

The center column of Figure 10 illustrates the histograms of the distribution of X for different strategies. The strategies illustrated represent the direction in which the previous occupied voxel is searched (top-right: *x*, bottom-left: *y*, bottom-right: *z*, top-left: best of the three, from the *optimized scan* method). As expected, the optimal strategy always shows the highest peak at 1, whereas the z strategy always gives the

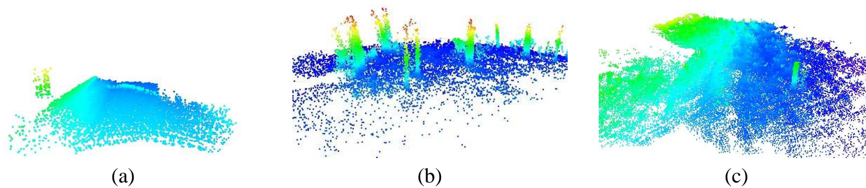

(a) (b) (c)

Figure 9: Illustration of the three data sets taken over three different terrains to evaluate their influence on the algorithm. (a) Flat ground, (b) Forest and (c) Tall grass

lowest. This is because the ground plane is roughly aligned with the xy-plane, so adjacent occupied voxel are more likely to be on that plane than along the z direction.

The speed improvement over the previous implementation presented in Section 5.4.3 is illustrated by the third column of Figure 10. The blue curve indicates the performance (in ms per occupied voxel) of the previous method and the red curve shows the performance of the new method. The speedup is substantial, and increases with the radius r used for range search. For example, at the current voxel size used on-board of the robot and $r = 0.4$ m, the new method is 4.6 times faster on the flat ground example, 5.5 times on the forest example, and 3.6 times on the tall grass, which results in an average speedup of approximately 4.5 over the three examples.

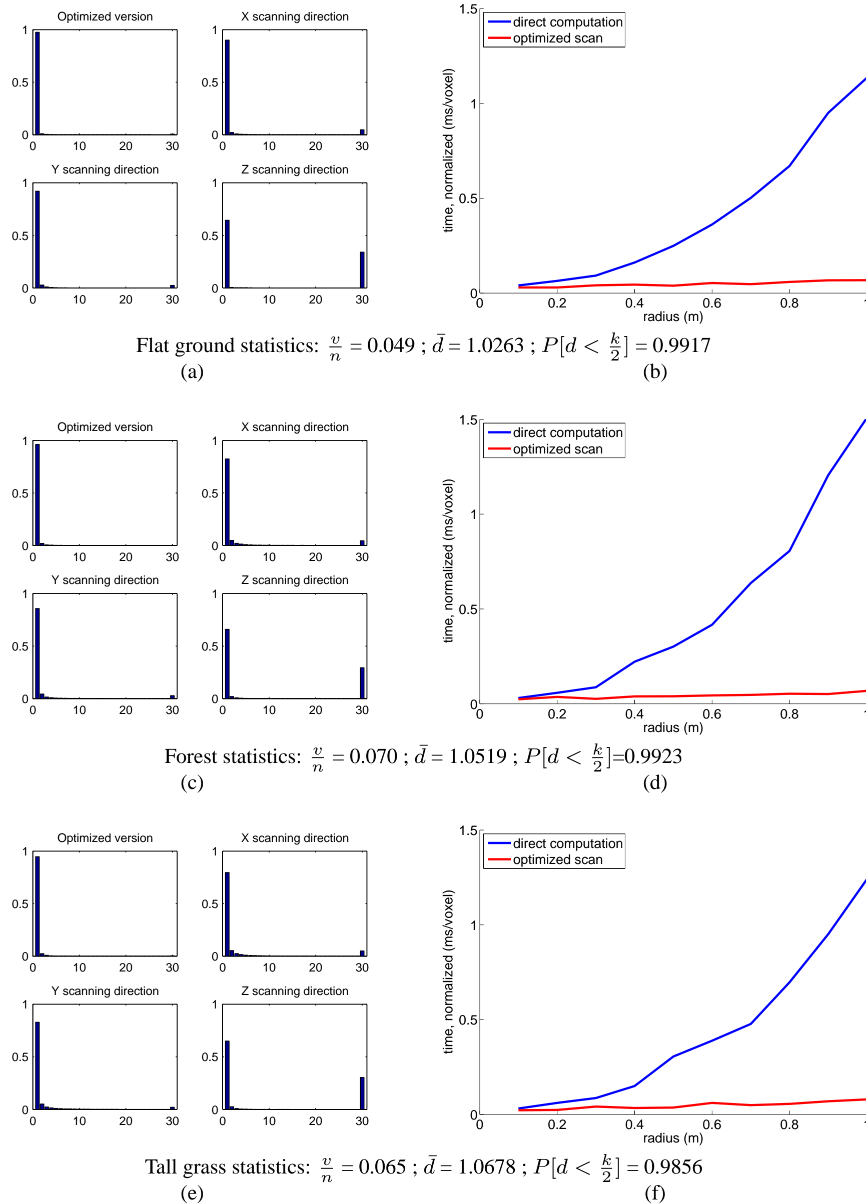
6.2.3 Parameters influence

In this section, we analyze the influence of two important parameters. The first is the point density, which depends on a variety of factors, such as the sensor's characteristics, vehicle speed, turret motion and the environment's geometry. The second is the voxel size, which is determined manually.

Point density Intuitively, denser data means a larger number of occupied voxels, which in turn implies a higher probability of overlapping neighborhoods. This is confirmed by experimental results obtained by artificially varying point density by subsampling the original data set 10 and 100 times. Timing results for the tall grass example are shown in Figure 11. We observe that the new method performs faster with denser data. In addition, Table 2 shows relevant statistics for those three examples. We note that $P[d < \frac{k}{2}]$ increases and $\bar{d}$ decrease with higher point density, which confirms the intuition.

On the other hand, it is interesting to note that the previous method (from [19]) runs *slower* with denser data. This is explained by the fact that, for each voxel, the neighborhood is likely to contain more points than with sparser data. Therefore, the number of visited voxel per occupied voxel is higher, hence the increase in computation time.

Voxel size We observe that, for a smaller voxel size, the number of voxels must be greater to keep the same range search radius r. For example, if $r = 4$ with voxel size of 10 cm, then $r = 8$ with voxel size of 5 cm, so 8 times more voxels must be visited than



Flat ground statistics: $\frac{v}{n} = 0.049$; $\bar{d} = 1.0263$; $P[d < \frac{k}{2}] = 0.9917$
(a) (b)

Forest statistics: $\frac{v}{n} = 0.070$; $\bar{d} = 1.0519$; $P[d < \frac{k}{2}]$=0.9923
(c) (d)

Tall grass statistics: $\frac{v}{n} = 0.065$; $\bar{d} = 1.0678$; $P[d < \frac{k}{2}] = 0.9856$
(e) (f)

Figure 10: Terrain influence for the static case. Each line corresponds to a terrain or environment type: flat, forest and tall grass (see Figure 9 for illustration of corresponding data sets). The first column contains a snapshot of a 3-D model of the scene, the elevation is color coded. The second column contains histograms of distribution of distances between current voxel and previous occupied voxel, for different scanning directions. The x (y) axis is the distance in number of voxel (the number of voxels). The rightmost peak represents infinite distance, that is, there is no previous occupied voxel in that direction. It is positioned at an arbitrary distance in the graph. The last column shows a comparison of speed of execution of the original versus the new method, with voxel size of 0.1 m.

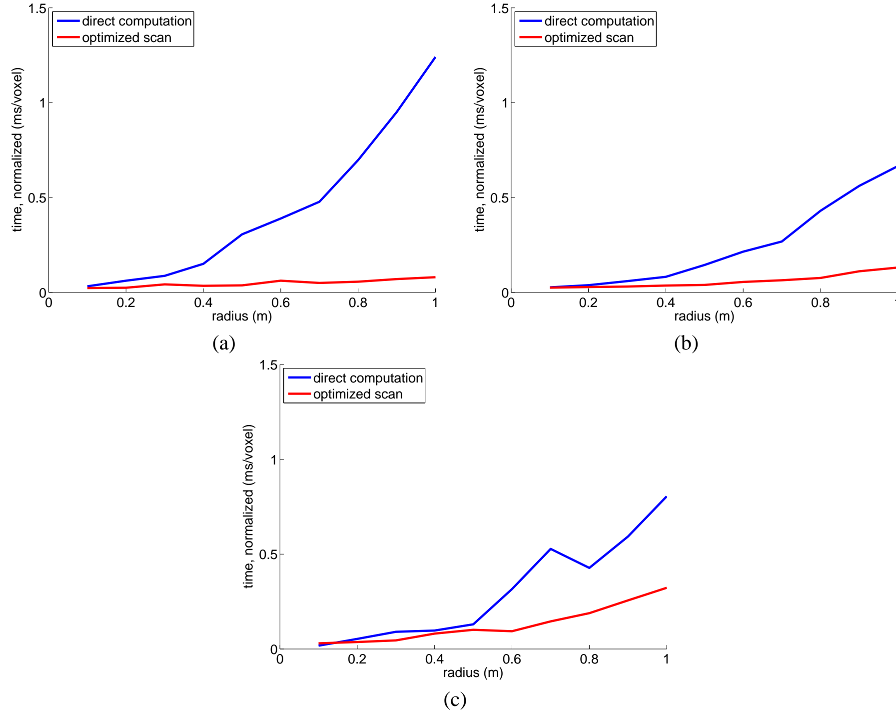


Figure 11: Point density influence for the tall grass data set. (a) With raw data (117,000 occupied voxels). (b) With data sub-sampled 10 times (55,000 occupied voxels). (c) With data sub-sampled 100 times (9500 valid voxels).

Table 2: Results statistics for the point density influence.

Sub-sampling	Raw points	Occupied voxels v	$\bar{d}$	$P[d < \frac{k}{2}]$
0 (raw data)	1,251,402	117,756	1.06	0.9856
10	114,161	54,808	1.2	0.9617
100	10,390	9,469	1.97	0.6926

before. More generally, if v_{size} is the voxel size and n_{neigh} the number of neighbors of a voxel, then with v_{size}/k we get $k^3 n_{neigh}$ neighbors. Moreover, smaller voxel size increases the number of holes in the data, which in turn increases $\bar{d}$ and decreases $P[d < \frac{k}{2}]$, as shown in Table 3. Figure 12 shows timing results obtained by running the previous and new method on the same full resolution data set and varying only the voxel size. The running time is indeed much slower with a voxel size of 5 cm versus 10 cm. Interestingly, the difference is not as obvious when comparing 10 and 20 cm.

These results show the important compromise relative to this parameter. An increasingly large voxel size will result in faster performance, but also in a loss of precision in scene details. Indeed, much of the high frequency content of the scene will be lost. On the other hand, if the voxel size is too low, the details will be preserved, but

Table 3: Statistics for the voxel size influence. 2,046,123 raw data points

Voxel size	Size n (in cells)	Occupied voxels v	$\bar{d}$	$P[d < \frac{k}{2}]$
5 cm	400x400x60	359,327	1.1063	0.993
10 cm	200x200x30	59,275	1.0263	0.991
20 cm	100x100x15	14,485	1.00	0.986

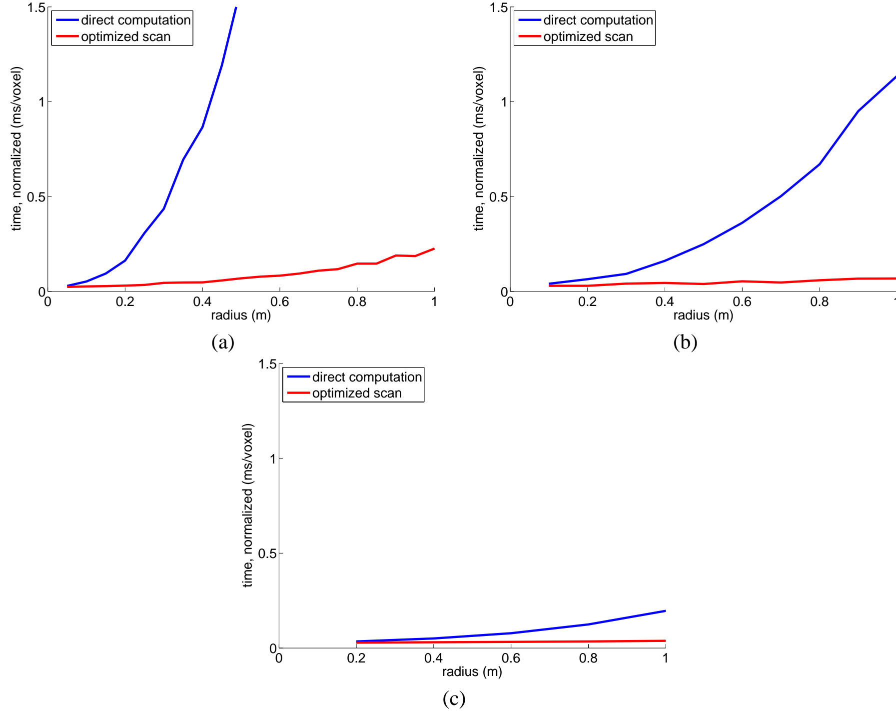


Figure 12: Voxel size influence for the flat terrain data set at full density. (a) 5 cm voxel size. (b) 10 cm voxel size. (c) 20 cm voxel size.

the running time will be much slower. The best parameter (10 cm in our case) is found by running benchmark tests on typical examples.

6.3 Dynamic data experiments

We now present experiments performed with the dynamic version of the data structure presented in Section 5.2.2.

6.3.1 Live parameters

Several additions are necessary to improve classification speed and to allow real-time processing. First, the number of voxels to be processed is greatly reduced by limiting

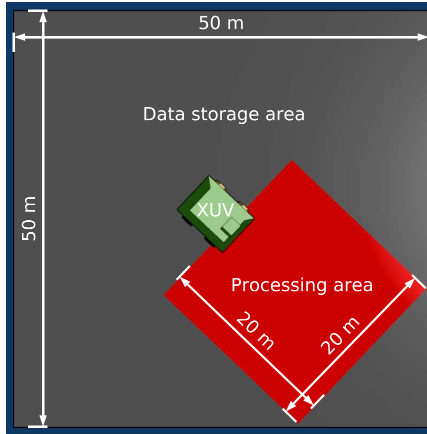


Figure 13: Top-down view of the parameter choice for live processing. The robot (XUV) is located at the center of the data structure that covers an area of 50×50 m around the robot (illustrated in gray). The interest region, where the processing is performed, is a 20×20 m area in front of the robot, and is illustrated in red. Note that the interest region rotates with the robot, whereas the data storage region does not and stays aligned with the global reference frame.

classification to an interest area around the robot. Since it is typically more important to process the data lying at close range, in front of the robot, an area of 20×20 m, as shown in Figure 13 is used in the following experiments. The size of the data structure is 50×50 m, centered around the robot. See Figure 13 for an illustration of these parameters.

In addition, a partial update strategy is implemented to avoid re-classifying the same voxel multiple times. Ideally, one would want to (re)compute the saliency features each time a new voxel is either created or updated, or when one of its neighbors is created or updated. However, because of the small voxel size, this step can be skipped with an acceptable loss of classification performances compared to the gain in processing time. In the worst case scenario, the classification error rate increases by 15% but the processing time is reduced by a factor of ten.

We also use two other parameters that indicate whether a voxel should be classified again or not. The following tests are performed after classification for each voxel:

- m: The number of times each voxel has been classified. If $m > m_{max}$, the current voxel is never classified again. This prevents performing unnecessary classification if the robot is standing still.

- r: The likelihood ratio. We define the conditional probability of a voxel to pertain

to the kth class given a model M by $P(S|M^k)$. If k_1 is the most likely class, and k_2 the second most likely, then:

$$r = \frac{P(S|M^{k_1})}{P(S|M^{k_2})}$$

If $r > r_{max}$, the current classification is considered accurate and the voxel is marked to prevent further processing, although it is still updated by new points. Intuitively, this corresponds to the moment when the classifier is sufficiently confident about the current classification result.

6.3.2 On-line timing

It is generally hard to quantify timing performances of 3-D processing techniques for perception, because it depends on a variety of parameters, such as the sensor's characteristics, the area of interest processed, the speed of the robot, data density and distribution, and others, such as those presented in Section 6.3.1. The traditional way of reporting the number of processed voxels per second is insufficient, as this does not take the dynamic nature of the process into account and is independent from the task at hand.

We introduce a different way of reporting timing performance, appropriate for voxel-based perception approaches. The idea is to compute, for each voxel, the delay between the time of its creation (first time a point is inserted in that voxel), and the time at which its class is known with sufficient confidence. By analogy, we are computing the time it takes for the robot to "see" an object, and "know" what type of object it is.

6.3.3 Comparison for different terrain types

We now compare and analyze the performance of the dynamic version of our approach for the same types of terrain listed in Section 6.2.2. The cumulative histograms obtained by computing the time between voxel creation and classification for different environments are shown in Figure 14. The faster the curve reaches 100 (100% of the voxels classified) the better. These data sets were obtained by manually driving the robot over 10 m in the different environments, at walking speed. As in the static case, there is no apparent difference between the different environments, the general behavior seems to be similar.

Using the previous approach, 90% of the voxels are classified within at most 550 ms, 825 ms and 625 ms for the flat, forest and tall grass data sets respectively. Even though the new data structure improves only the range search part of the algorithm, it allows a decrease of 27%, 41.8% and 36% respectively for each data set following the same order of presentation.

In these live timing results, overhead sources due to voxel creation, data insertion and classification that contribute to the decrease in speedup observed in the static case, where none of these were included. However, these timings reflect the true decrease in processing time due to the new data structure, and cannot be compared directly with those reported in Section 6.2 for batch processing.

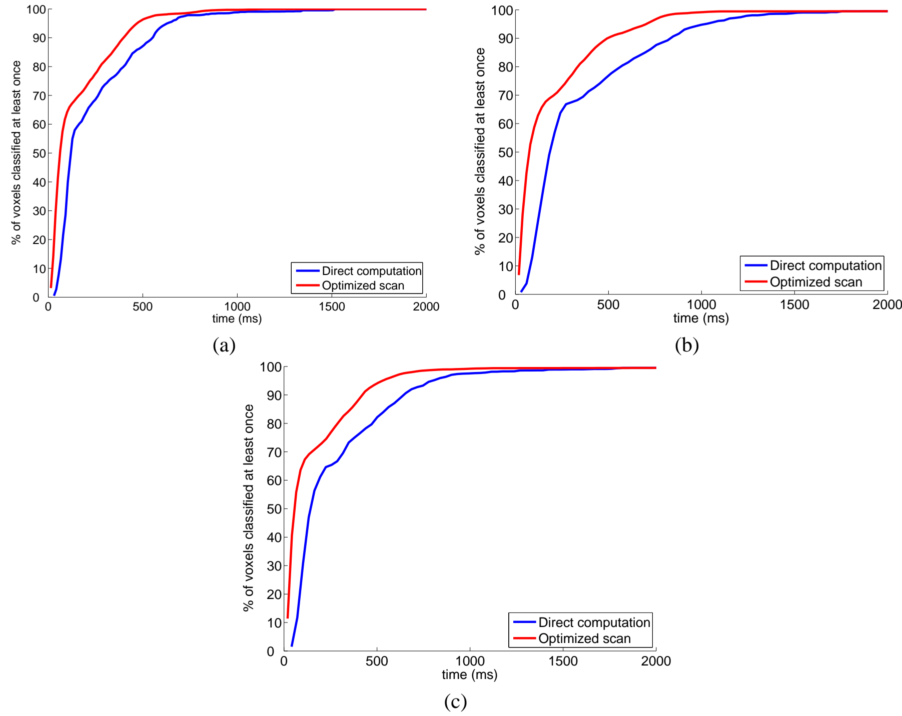


Figure 14: Terrain influence for the dynamic case. (a) is obtained with the flat terrain, (b) with the forest and (c) with the tall grass examples (see Figure 9 for illustrations of corresponding data sets). This shows the cumulative histogram of the time between a voxel is inserted and is classified with sufficient confidence. The blue curve represents the previous implementation, described in Section 5.4.3. The red curve is obtained with the new data structure introduced in this paper. The parameters used are $r_{max} = 2$ and $m_{max} = 5$.

6.3.4 Parameters influence

We now evaluate the influence of the two most important parameters related to live processing: r_{max} and m_{max} (see Section 6.3.1). The results shown in this section are obtained with the *optimized scan* algorithm on the dynamic version of our proposed data structure.

Maximum likelihood ratio As is shown in Figure 15, the r_{max} parameter affects the classification timing by shifting the curve to the right. The effect is shown for different values of r_{max}, ranging from 2 to 8. The results are shown for the new data structure only, on the three terrains previously described.

Again, the three different data sets seem to display a similar behavior. The increase in r_{max} slightly increase the time between voxel creation and classification, which shows that a majority of voxels are classified with good accuracy as soon as there are

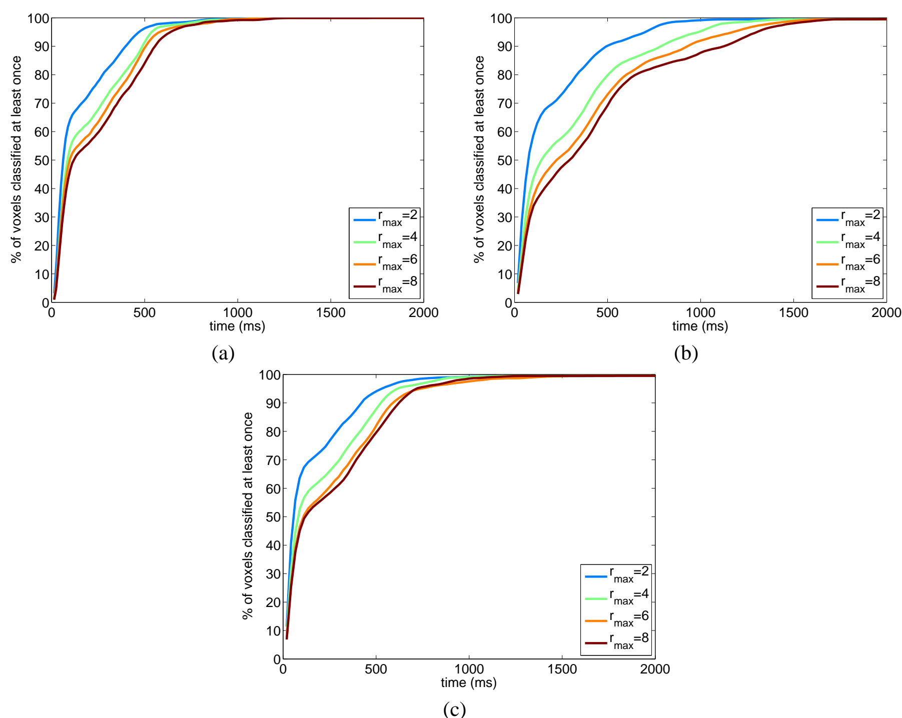


Figure 15: Effect of the maximum likelihood ratio parameter r_{max} on the time between voxel creation and classification with sufficient confidence for the (a) flat ground, (b) forest and (c) tall grass data sets (see Figure 9 for illustrations of corresponding data sets). A value of $m_{max} = 5$ is used.

enough points in their neighborhood to allow classification. Only the voxels that have low confidence contribute to the increase in classification time.

Maximum number of times to reclassify Similarly to the effect of r_{max}, an increase in the parameter m_{max} results in an increase in processing time, as shown in Figure 16. However, its effect is more important than that of r_{max} because its increase results in a higher number of voxels to reclassify at each iteration, independent of the classification results. As the curves for $m_{max} = 50$ and $m_{max} = 100$ show, the processing is not able to keep up with the large number of voxels to re-classify, and the performance degrades rapidly.

7 Conclusion

This paper deals with the challenging problem of processing large amounts of dynamic, sparse three-dimensional data. We present a data structure and algorithm that improve the speed of range search for 3-D point-cloud processing. The data structure can then

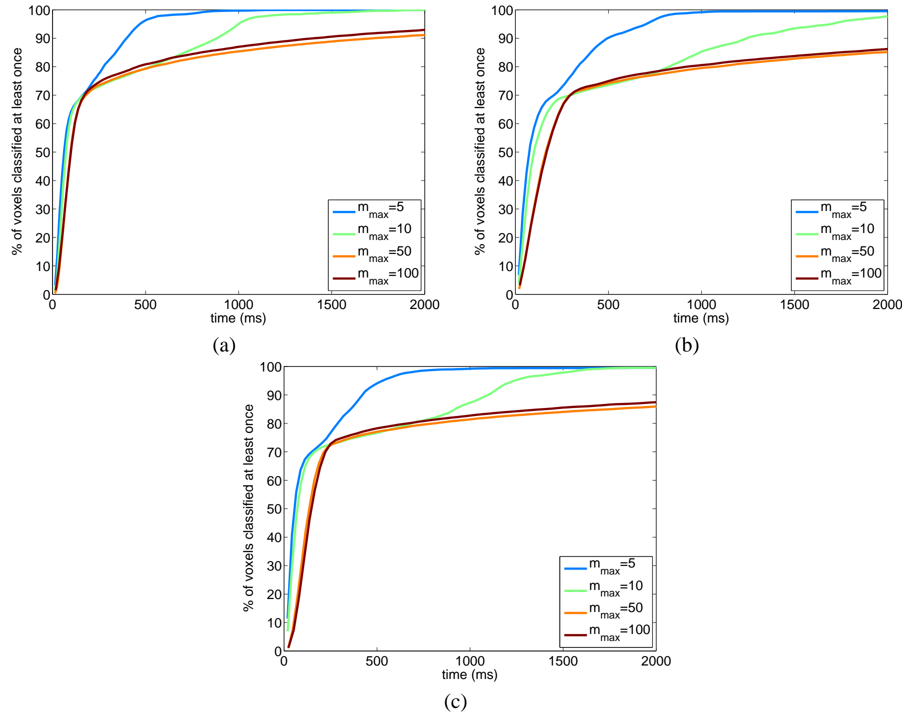


Figure 16: Effect of the maximum number of times to reclassify m_{max} parameter on the time between voxel creation and classification with sufficient confidence for the (a) flat ground, (b) forest and (c) tall grass data sets (see Figure 9 for illustrations of corresponding data sets). A value of $r_{max} = 2$ is used.

be used in a wide class of computations based on the extraction of features defined over a local support volume around each point. The approach takes advantage of the overlap in computation by reusing previously computed results. We show significant speed-up on an application of this technique for classification in the context of perception for ground mobile robots. The approach is validated on ladar data obtained in various environments using the Demo III XUV, both in the static and dynamic data cases.

For the three typical datasets analyzed, we observe considerable improvement in execution speed without noticeable differences between the various terrain types studied. On the static data, we achieve on average a 4.5 fold speedup with a voxel size of 0.1 m and a range search radius of 0.4 m. Those parameters are shown to be suitable to improve ground robot mobility [19, 10]. For the dynamic case, this data structure is able to reach improvements of up to 40% in speed, even though other time-consuming operations such as classification are not affected and are included in the timing reports.

We are currently working on the testing of methods such as Kernel Density Estimation (KDE), that require the recovery of the location of every point in the neighborhood. Although we presented an idea for the implementation, there is a need for a more extensive evaluation to understand the implication of such additional computations.

References

[1] J. Albus, K. Murphy, A. Lacaze, S. Legowik, S. Balakirsky, T. Hong, M. Shneier, and A. Messina. 4D/RCS sensory processing and world modeling on the demo III experimental unmanned ground vehicles. In *International Symposium on Intelligent Control*, 2002.

[2] J. Bornstein and C. Shoemaker. Army ground robotics research program. In *SPIE conference on Unmanned Ground Vehicle Technology V*, 2003.

[3] H. Chen and P. Meer. Robust computer vision through kernel density estimation. In *European Conference on Computer Vision*, 2002.

[4] A. Kelly et al. Toward reliable off-road autonomous vehicle operating in challenging environments. In *International Symposium on Experimental Robotics*, 2004.

[5] O. Faugeras et al. Real-time correlation-based stereo : algorithm, implementations and applications. Technical Report RR-2013, INRIA, 1993.

[6] J. Gao and R. Gupta. Efficient proximity search for 3-D cuboids. In *Computational Science and Its Applications*, volume 2669 of *Lecture Notes in Computer Science*, 2003.

[7] A. Gray and A. Moore. Data structures for fast statistics. Tutorial presented at the International Conference on Machine Learning, 2004.

[8] A. Lacaze, K. Murphy, and M. DelGiorno. Autonomous mobility for the demo III experimental unmanned vehicles. In *Proceedings of the AUVSI Conference*, 2002.

[9] J.-F. Lalonde, N. Vandapel, and M. Hebert. Data structure for efficient processing in 3-D. In *Proceedings of Robotics: Science and Systems I conference*, 2005.

[10] J.-F. Lalonde, N. Vandapel, D. Huber, and M. Hebert. Natural terrain classification using three-dimensional ladar data for ground robot mobility. *To appear in the International Journal of Field Robotics*, 2006.

[11] J. Lersch, B. Webb, and K. West. Structural-surface extraction from 3-d laser-radar point clouds. In *Laser Radar Technology and Applications IX*, volume 5412. SPIE, 2004.

[12] T. Liu, A. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Proceedings of Neural Information Processing Systems Conference*, 2004.

[13] R. Manduchi, A. Castano, A. Talukder, and L. Matthies. Obstacle detection and terrain classification for autonomous off-road navigation. *Autonomous Robot*, 18:81–102, 2005.

[14] H. Moravec. Robot spatial perception by stereoscopic vision and 3d evidence grids. Technical Report CMU-RI-TR-96-34, Carnegie Mellon Univeristy, 1996.

[15] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[16] N. Schneier, T. Chang, T. Hong, G. Cheok, H. Scott, S. Legowik, and A. Lytle. A repository of sensor data for autonomous driving research. In *SPIE Unmanned Ground Vehicle Technology V*, 2003.

[17] R. Unnikrishnan and M. Hebert. Robust extraction of multiple structures from non-uniformly sampled data. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003.

[18] N. Vandapel, R. Donamukkala, and M. Hebert. Unmanned ground vehicle navigation using aerial ladar data. *International Journal of Robotics Research*, 25(1), 2006.

[19] N. Vandapel, D. Huber, A. Kapuria, and M. Hebert. Natural terrain classification using 3-D ladar data. In *IEEE International Conference on Robotics and Automation*, April 2004.