Data Structures for Efficient Dynamic Processing in 3-D

Jean-François Lalonde, Nicolas Vandapel*, and Martial Hebert
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, USA
Email: {jlalonde,vandapel,hebert}@ri.cmu.edu

March 15, 2007

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 problems. We propose a new, efficient data structure and corresponding algorithms 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.

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 arriving 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 a 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 [29, 7]. These sensors provide hundreds of thousands of 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.

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. A critical operation is the retrieval of all points which fall within a certain distance of a given point, and we call this operation *range search*. The goal of the approach is to minimize range search computations by re-using previously computed intermediate results. While this dynamic programming approach has already been used in the 2-D image processing case (in stereo, for example), there are three fundamental

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

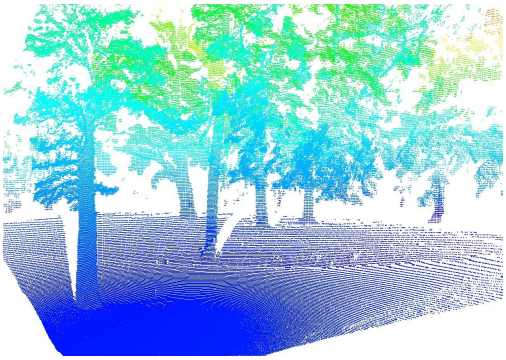


Figure 1: Example of raw 3-D point cloud from a ladar. The points are color-coded by elevation, from blue (low) to red (high), and are enlarged for clarity. This dataset is made of 259,528 points.

differences which prevent a straightforward adaptation to 3-D. First, 3-D data is sparse, which means that previously computed data will not always be available, as opposed to the dense 2-D case. Second, the additional dimension significantly increases the complexity of algorithms. Finally, the 3-D volume occupancy will vary as a function of the observed scene, so different techniques might be suitable for different environments.

The main contribution of this paper is a novel data structure that allows fast processing of sparse, dynamic 3-D data. It takes advantage of the data sparsity to reuse previously computed results only when available. Furthermore, it *automatically adapts* to the environment by analyzing the local distribution around each point. Finally, it is suitable to efficiently process *dynamic* data in an on-line fashion and runs at high speed in three dimensions. Whereas this data structure is based on the original work of [14] that was suitable only for static data, this paper introduces the crucial extension to handle dynamic data, and also presents results obtained by performing live experiments on-board a ground mobile robot, the Demo-III eXperimental Unmanned Vehicle [4], for ladar-based terrain classification [30]

The remainder of the paper is divided as follows. First, Section 2 presents related works, mainly in robotics and efficient data structures. Although we originally devised this new data structure to improve the speed of previous algorithms for terrain classification [15], we show in Section 3 that it can be applied to a generic category of 3-D processing methods. In Section 4 the proposed approach is presented in details. Its implementation is outlined in Section 5. Experimental results on static and dynamic data processed off-board are provided Section 6. Finally, on-board, real-time data processing results are presented in Section 7.

2 Related work

2.1 Data structures

Traditional pre-computed tree-based data structures (Kd-tree, range tree) are efficient for performing range search. Unfortunately, there is a large processing overhead at initialization, and their performance rapidly degrades as additional data is inserted after construction [23]. In [16], 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 [30] but it is performed off-line. A related approach is the bucket-space used in [3] for on-line surface reconstruction from a laser line scanner. The raw data points are stored in the data structure but for the task of terrain classification we collapse the data by computing the sums, sums of squares and sums of cross products of point location, reducing the amount of memory required.

3-D point clouds are a popular method to represent objects with solid surfaces acquired by laser scanners. Typically, such sensors produce a high point density and techniques have been developed to efficiently store, transmit and render their data. Some algorithms are driven by the local geometry [22], some are based on octrees [24] or hierarchical clustering [11]. Such techniques are centered around computer graphics applications and compresses the point cloud. We are more interested in data structures for data processing than visual presentation.

In different applications, efficient approximate range search techniques [2] can be used. But in our case, we choose to

emphasize accuracy over speed in order to better capture the local geometry of the point cloud.

Gao proposes an interesting work on efficient proximity search in 3-D for kinetic data [9]. 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 [17, 10]. However, most of the attention is focused on high dimensional data sets rather than dynamic data sets of low dimensionality.

Three-dimensional medical imaging devices [26] produce data very much different than the one we use. The data resolution is much higher (sub-millimeter voxel size versus 10 cm in our case) as well as the density. The physical sensing process is also very different as we are recovering the location of object surface while in medical imaging each voxel is associated with a physical measurement (diffusion tensor for example). Traditional spatial data structure are well suited for such application.

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 2-D 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 2-D 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 [29]. Such data structures are dense, fixed resolution, scrolling grids. In contrast, Montemerlo [19] proposes a multi-resolution grid to account for the limited sensor resolution. In all cases the processing is not performed in fully 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 [13, 7]. 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 [21]. Operations in 3-D can sometimes be reduced to 2-D operations as shown in [18] 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 class of computation in which we are interested 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 $i \neq k$ and with $\| p_i - p_k \|_\infty = \max(|x_i - x_k|, |y_i - y_k|, |z_i - z_k|)$. This is referenced in the paper as range search;
- $F(N(p_i, s))$: 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, our approach 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 voxel size is chosen such that the desired resolution is achieved and allows to faithfully capture high spatial frequency details in the data. The local support volume is therefore defined in number of voxels around a voxel of interest. The content of each voxel, the voxel and support volume size, and the binning process are task-dependent.

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.

1. Possibly the simplest computation is the recovery of 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 is the retrieval of the neighboring points themselves (sometimes also referred to as *range search*). 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

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

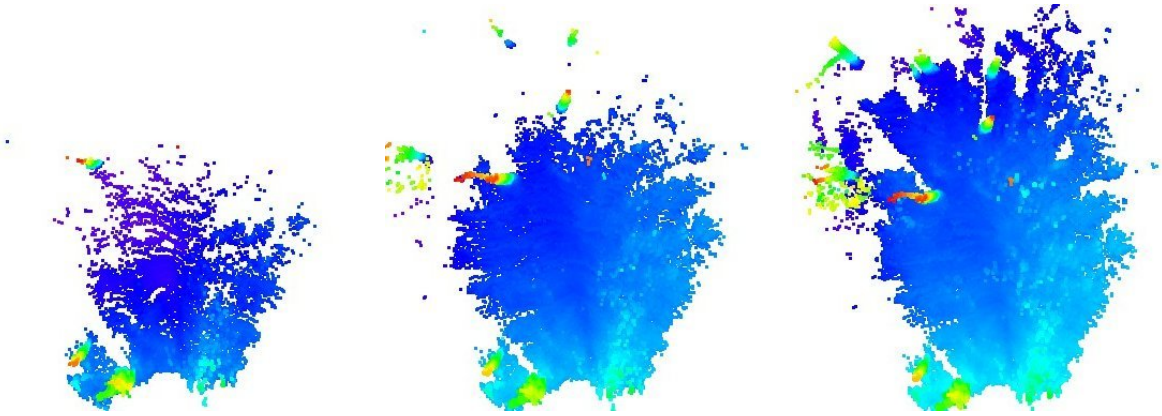
   where $\bar{p} = \frac{1}{|N(p_i, s)|} \sum_{p \in N(p_i, s)} p$.
4. A final example is kernel density estimation [31], which can also be used in 3-D data processing (see [28] for instance). Its local neighborhood function, for d-dimensional data, is given by

$$F_4(N(p_i, s)) = \frac{1}{|N(p_i, s)| s^d} \sum_{p \in N(p_i, s)} \kappa_s \left(\frac{\|p_i - p\|}{s} \right) \tag{2}$$

   where κ is a kernel with finite-support and $\int \kappa(x) dx = 1$ with s serving as the bandwidth parameter.

3.3 Dynamic aspects

The dynamic aspect of the problem justifies the need for the data structure presented in this paper. Since the amount, composition and distribution of the data is unknown a priori, it is impossible to build the efficient, nearest-neighbor oriented data structures that exist in the litterature and were presented in Section 2.1. These data structures are typically built from the entire dataset during a lengthy initialization process. Only once they are initialized can they be used for fast processing, which clearly is not suitable for the case of dynamic data. Figure 2 shows an example of such data, in this case accumulated from a ladar sensor placed on a ground mobile robot. Since the data boundaries might grow arbitrarily, 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. This situation is commonly encountered in the context of mobile robot perception.



(a) (b) (c)

Figure 2: Top-down view of an example of dynamic data, as it is accumulated from a robot driving through the woods. Note the presence of tree trunks. 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 Efficient scrolling data structure

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 [8], 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, for natural environment terrains, varies between 2 % and 12 % for voxel sizes of 10 cm to 1 m of edge length respectively. This contrasts with images in which every pixel contains information. It is important to note that those two differences increase significantly the complexity of the algorithms.

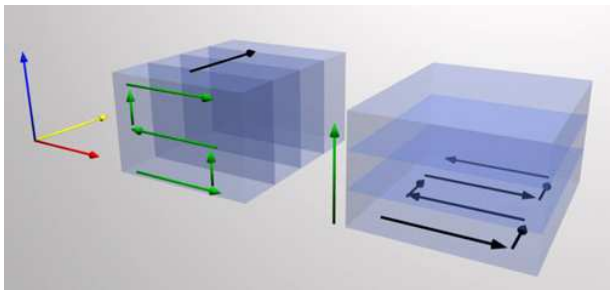


Figure 3: 2 possible scanning strategies in 3-D

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;
- $k = (2r + 1)$ 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 assume that the support region is isotropic, that is, k is equal in each dimension. We also do not take into account the differences that may arise at the boundaries of the volume for two reasons. First, the fraction of voxels that are at a distance of r or less than the boundaries is expected to be small for large volumes. Second, the conditions we encounter in practice are that the point density is higher at the center of the map (centered at the vehicle location) as it is closer to the sensor and the central area is swept more often by the mobility ladar.

4.2 Direct computation approach

Direct implementations usually rely on an exhaustive approach that visit every k^3 voxels in every neighborhood each time (see [30] 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 [30]. However, on the dense regions, much of the computations are repeated many times because of overlapping neighborhoods.

4.3 Naive approach for data reuse

A naive approach would be to act as if the volume were 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.

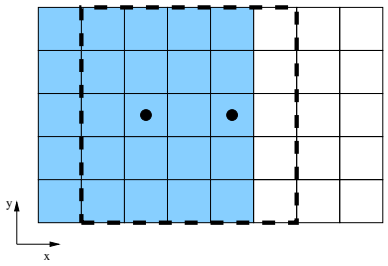


Figure 4: Illustration of naive approach in 2-D. Each square is a voxel, and dots indicate occupied voxels. The shaded area represents the local support region defined for the leftmost occupied voxel (dotted). The dashed outline is the support region of the middle voxel, computed even though the voxel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

An appropriate data structure is the dense voxel representation, in which memory space must be reserved, but not necessarily allocated immediately, for each voxel of the volume at the beginning. The drawback of this 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 (corresponding to the central part of the volume) and recomputes two neighborhood slices (add/remove) 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

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

Note that the initialization cost is not considered here as explained earlier. 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 :

$$2nk^2 < vk^3$$

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

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, Table 1), the v/n ratio tends to be very low, typically under 2%, justifying the need for a better approach.

4.4 Optimized scan approach for data reuse

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, 10 voxels (instead of 25) must be visited.

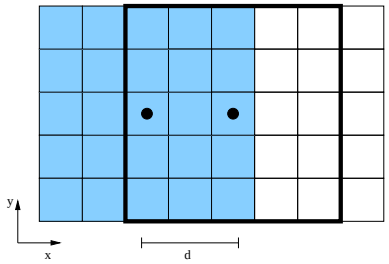


Figure 5: Illustration of data reuse with sparse 2-D data with the neighborhood size $k = 5$. Each square is a voxel, and dots indicate occupied voxels. The shaded area represents the local support area defined for the leftmost occupied voxel. The outlined area is the support area of same dimensions 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 operations 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 $d < \frac{k}{2}$:

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

If $d < \frac{k}{2}$, $2\bar{d}k^2$ voxels need to be visited. If $d \geq \frac{k}{2}$, k^3 need to be visited. The latter case corresponding to no overlap in the neighborhood. $t_{optimized}$ is then computed as the expected number of visited voxels:

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

This equation assumes that: 1) the cost associated to finding whether a voxel is valid or not is constant, this is represented by the last term (n) for the complete volume 2) the cost for finding the nearest neighbor is zero.

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

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

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 the next section will show, 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.

4.5 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 (v/n) 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.

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. It automatically determines in what direction to look for re-usable data.

Algorithm 1: General scanning algorithm

Input: V the voxelized volume and its boundaries

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

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. 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 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.

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.

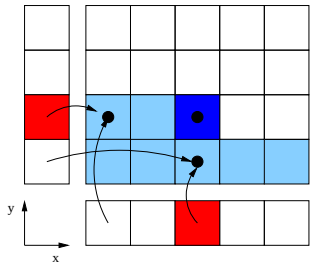


Figure 6: 2-D example of the proposed data structure. Dense voxel representation augmented with additional side vectors that store the location of the previously computed results, in each dimension. The dotted squares represent occupied voxels. The dark blue square represent the current voxel, and the 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). The origin (0,0) in voxel coordinates is the bottom-left voxel.

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 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 directly.

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 squares 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. This approach can be seen as maintaining sufficient statistics as introduced in [20].

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: laser-based terrain classification

An interesting application of 3-D processing based on local computations, is point-wise classification to improve the perception capabilities of ground mobile robots, as shown in [30, 15]. 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 and Section 7.

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 saliency features via PCA. For each voxel, 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 saliency features. Three saliency features are computed, capturing locally the "linear-ness", "surface-ness" and "scatter-ness" of the point cloud. A model of the saliency features distribution is learned off-line, prior to the mission, from labeled data. As the robot traverses the terrain, data is accumulated, saliency 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 [30, 15].

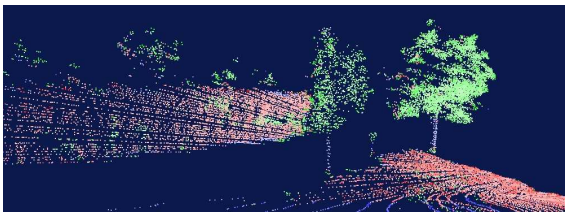


Figure 7: Example of terrain classification. Surfaces, linear structures and scattered points are colored in red, blue and green respectively.

5.4.2 Voxelization process

In order to handle the high data rate coming from the laser sensor (in the order of a hundred-thousand points per second), we previously implemented a compression process that dramatically reduces the amount of data to handle without compromising the saliency features computation accuracy [30]. 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 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 Classifiers

Once the saliency features are computed at a given voxel, a classifier is used to determine the class of that voxel. To do this, two well-known classifiers have been used and will be briefly described here.

Gaussian Mixture Model The first classifier models the saliency features by fitting a Gaussian Mixture Model (GMM) using the Expectation-Maximization (EM) algorithm on a hand-labelled training data set, see [6] for more details. If $M^k = \{(\omega_i^k, \mu_i^k, \Sigma_i^k)\}_{i=1\ldots n_g^k}$ is the mixture model of the k^{th} class and $S = (S_{scatter}, S_{linear}, S_{surface})$ the feature vector of a voxel to classify, then the conditional probability that the voxel belongs to the k^{th} class is given by

$$p(S|M^k) = \sum_{i=1\ldots n_g^k} \frac{\omega_i^k}{(2\pi)^{d/2}|\Sigma_i^k|^{1/2}} e^{-\frac{1}{2}(S-\mu_i^k)^T \Sigma_i^{k\,-1}(S-\mu_i^k)} \tag{8}$$

where d is the dimension of the feature vector S. The class is chosen to satisfy

$$k_{max} = \underset{k}{\text{argmax}}\{p(S|M^k)\} \tag{9}$$

Support-Vector Machines The second classifier uses the multi-class Support Vector Machine (SVM) formulation introduced in [27]. This approach is a generalization of the traditional binary SVM formulation [5] that supports structured outputs, such as multiple classes, labeled trees, or others. The SVM formulation can be summarized by finding the hyperplane that best separates two feature sets by maximizing the *margin*, i.e. the distance from the hyperplane to the closest feature point. This can be represented (see [27] for details) as a minimization problem in Equation 10, where the weights $\mathbf{w}$ are the hyperplane parameters.

$$\min_{\mathbf{w},\xi} \frac{1}{2}\|\mathbf{w}^2\| + \frac{C}{n}\sum_{i=1}^{n} \xi_i, \text{s.t.} \forall i, \xi_i \geq 0 \tag{10}$$

subject to the following constraints:

$$\forall i, \forall \mathbf{y} \in \mathcal{Y} \backslash \mathbf{y}_i : \langle \mathbf{w}, \delta\Psi_i(\mathbf{y}) \rangle \geq 1 - \xi_i \tag{11}$$

where the ξ_i are slack variables to allow for non-linearly separable saliency features and $\mathbf{y} \in \mathcal{Y}$ is the labeling where each y_i is taken from the set of all possible labels $\mathcal{Y}$. $\delta\Psi_i(\mathbf{y})$ is a measure of the difference between two labelings, and must be defined according to the task at hand, multiclass classification in this case. Note that we keep the same notation as in [27] to allow for convenient reference. $C > 0$ is a constant that controls the tradeoff between training error minimization and margin maximization, and must be determined experimentally. The value $C = 1$, obtained by cross-validation, was used in all the experiments presented in this paper.

In this work, the linear SVM formulation is used primarily for its speed, since it only requires dot products and can be implemented very efficiently. In addition, we do not observe any significant increase in classification accuracy as higher-order kernels are used because of significant overlap in the saliency features distribution. We use the readily available C++ implementation dubbed SVM^{struct}, an extension of the SVM^{light} package [12]. The implementation also requires the definition of a termination parameter ϵ. In all our experiments, we set $\epsilon = 0.001$. SVM^{struct} is used both for off-line training and on-line classification.

5.4.4 Previous implementation

An earlier data structure using the voxelization scheme presented in the previous section has been implemented and its capabilities demonstrated in [30]. 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, off-board the vehicle. Results performed in real-time on-board the vehicle will be presented in Section 7. This section will be used to show the influence of the different parameters and allow the analysis of the results in controlled, repeatable experiments.

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 DEMO-III eXperimental Unmanned Vehicle (XUV, [13]). 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 a hundred thousand 3-D points per second with centimeter range resolution and a maximum range of a hundred meters. The laser is mounted on a pan and tilt scanning turret. Additional information on this specific version of the laser used can be found in [25].

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.

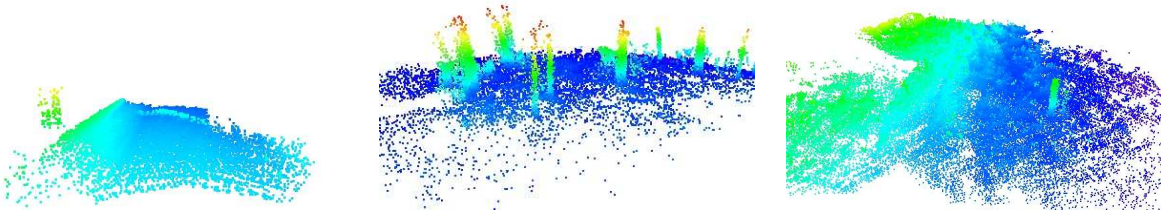
Terrain	Size n (cells)	Raw data	v	$\frac{v}{n}$	$\bar{d}$	$P[d < \frac{k}{2}]$
Flat ground	200x200x30	2.0×10^6	59,275	0.049	1.0263	0.9917
Forest	160x250x40	1.7×10^6	112,001	0.070	1.0519	0.9923
Tall grass	200x300x30	1.2×10^6	117,756	0.065	1.0678	0.9856

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 [14].

6.2.1 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. Note that the GMM classifier was used to produce those results. The initial results seem to show that the type of terrain does not influence $\bar{d}$ and $P[d < \frac{k}{2}]$. These values are computed for the best direction (as determined by Algorithm 1) at each voxel. Therefore, given a roughly constant point density across different scenes, the scene geometry should not greatly affect these values. Since the point density is generally very high, both $\bar{d}$ and $P[d < \frac{k}{2}]$ are expected to be close to 1.



(a) (b) (c)

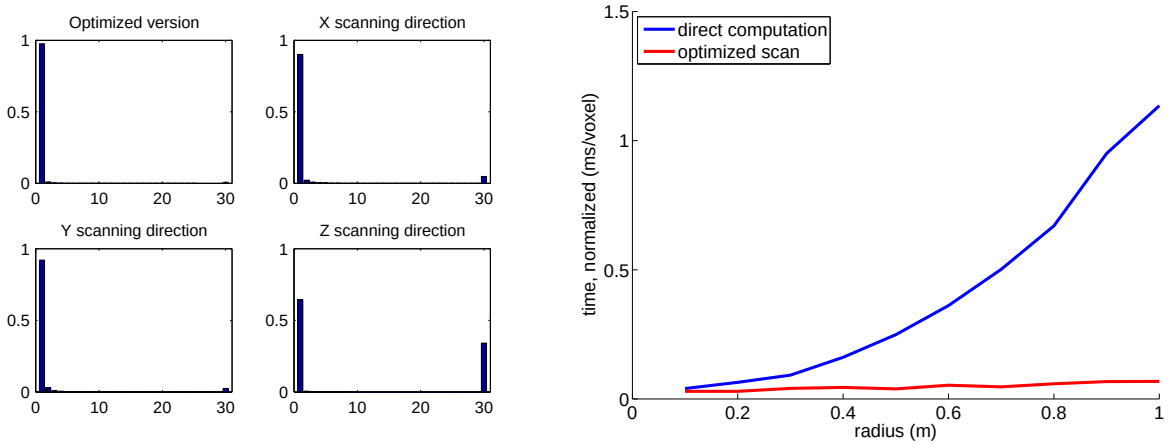
Figure 9: Illustration of the three data sets taken over three different terrains to evaluate their influence on the algorithm. (a) Flat ground, (b) Forest and (c) Tall grass

The left column of Figure 10 illustrates the histograms of the distribution of distances between current voxel and previous occupied voxel 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 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 5.4.4 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.

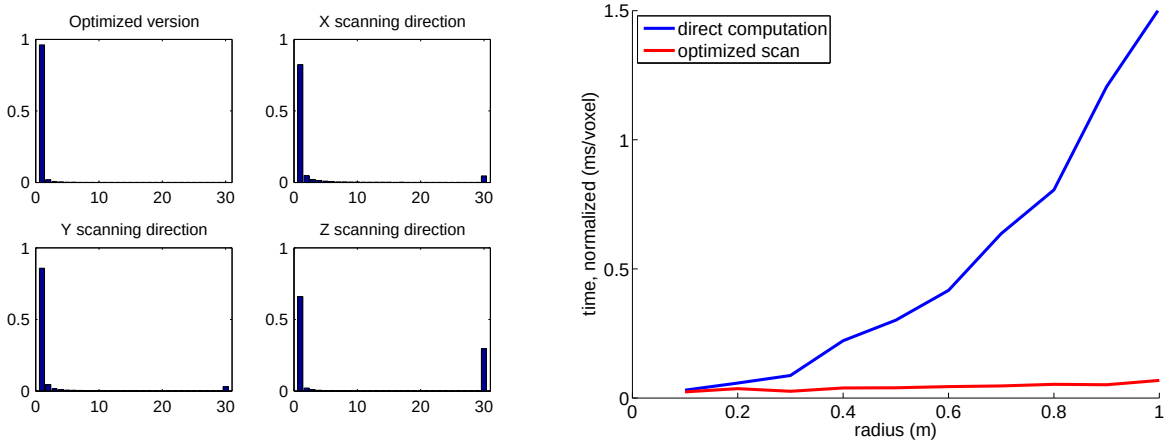
6.2.2 Parameters influence

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.



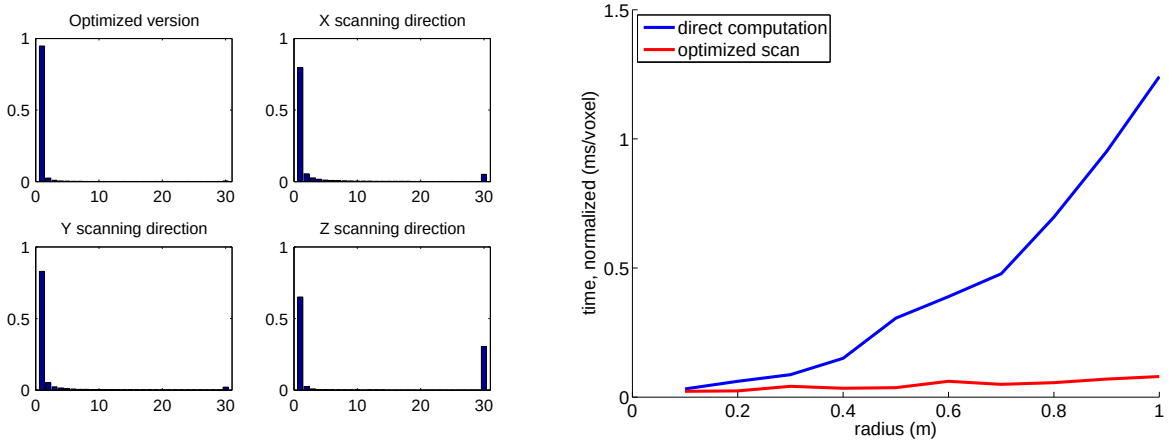
Flat ground statistics: $\frac{v}{n} = 0.049$; $\bar{d} = 1.0263$; $P[d < \frac{k}{2}] = 0.9917$

(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 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. Results were produced using the GMM classifier.

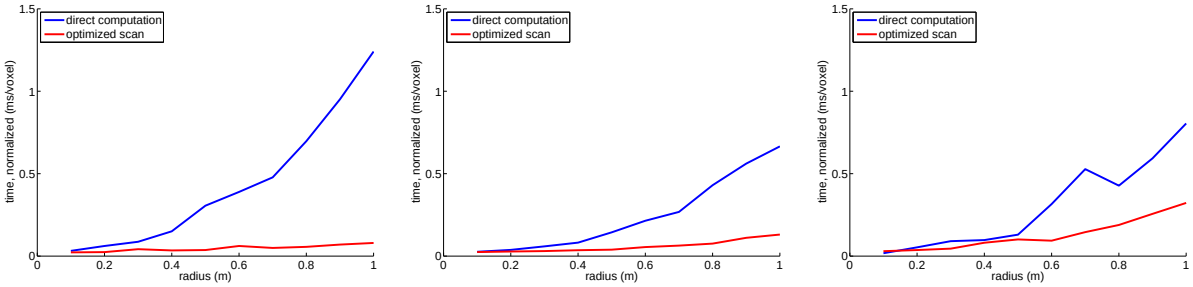


Figure 11: Point density influence for the tall grass data set. (a) With raw data (117,000 occupied voxels). (b) With data sub-sampled 10 times (55,000 occupied voxels). (c) With data sub-sampled 100 times (9500 valid voxels).

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 sub-sampling 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 [30]) 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.

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

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 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.

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

These results show the important compromise relative to the voxel size. 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 the running time will be much slower. The best parameter (10 cm in our case) is found by running benchmark tests on typical examples.

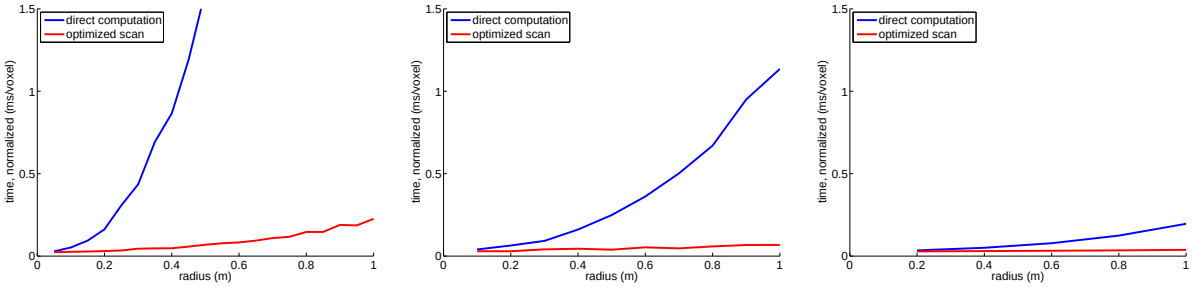


Figure 12: Voxel size influence for the flat terrain data set at full density. (a) 5 cm voxel size. (b) 10 cm voxel size. (c) 20 cm voxel size.

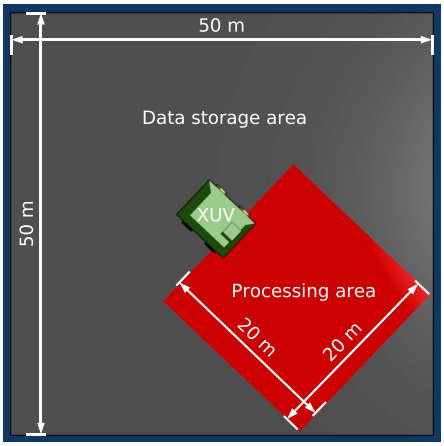


Figure 13: Top-down view of the parameter choice for live processing. The robot (XUV) is located at the center of the data structure that covers an area of 50×50 m around the robot (illustrated in gray). The interest region, where the processing is performed, is a 20×20 m area in front of the robot, 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.

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

Since the number of data points can grow to an arbitrarily large number, several additions are necessary to allow real-time processing. First, the number of voxels to be processed is greatly reduced by limiting 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. Only the former update is implemented. 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 confidence measure. We define the conditional probability of a feature S to pertain to the k^{th} 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:

$$r = \frac{P(S|M^{k_1})}{P(S|M^{k_2})} \tag{12}$$

If $r \geq r_{max}$, the current classification is considered accurate and the voxel with feature S is tagged 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.

The GMM classifier outputs these conditional probabilities, so they can be used directly to compute r. However, the SVM classifier outputs an unbounded signed distance $d(S, H^k)$ from feature S to the decision hyperplane H^k, and therefore Equation 12 cannot be used. Instead we use the ratio of confidence measure defined by

$$r = \frac{f(d(S, H^{k_1}))}{f(d(S, H^{k_2}))} \tag{13}$$

With $f(d) = \frac{2}{1+e^{-d}} - 1$, k_1 is the most likely class with $d(S, H^{k_1}) > 0$. If $d(S, H^{k_2}) < 0$ or $r > r_{max}$, the current classification is considered accurate and the voxel with feature S is tagged to prevent further processing as for the GMM classifier.

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. Thus we *do not* use this traditional metric.

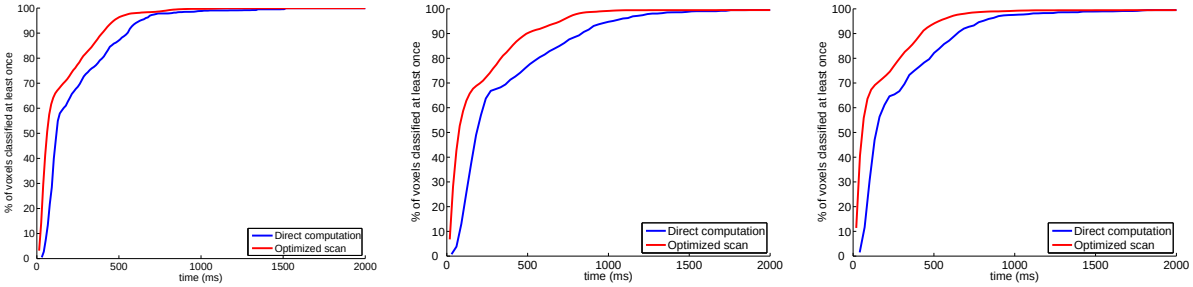
Instead, we use a different way of reporting timing performance, more appropriate for voxel-based perception approaches and introduced in [15]. The idea is to compute, for each voxel, the delay between the time of its creation (first time a point is inserted in that voxel), and the time at which its 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 6.2.1. 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.

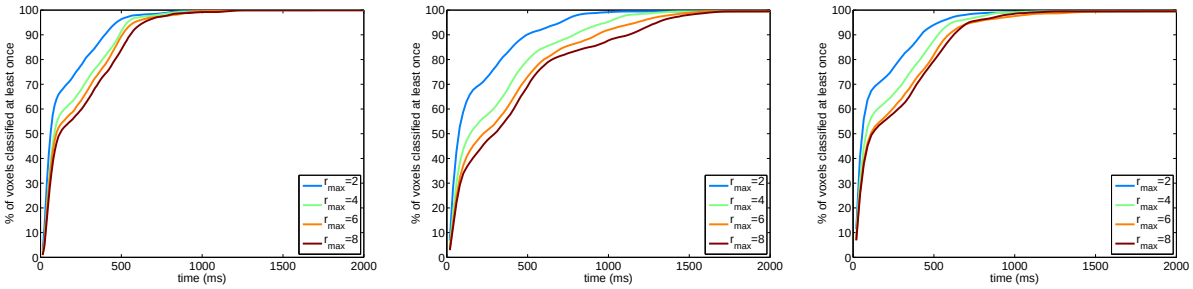
Using the previous approach, 90% of the voxels are classified within at most 550 ms, 825 ms and 625 ms for the flat, forest and tall grass data sets respectively. Even though the new data structure improves only the range search part of the algorithm, it allows a decrease of 27%, 41.8% and 36% respectively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

In these live timing results, the overhead due to voxel creation, data insertion and classification that contribute to the decrease in speedup observed in the static case, where none of these was 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.



(a) (b) (c)

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 5.4.4. The red curve is obtained with the new data structure introduced in this paper. The parameters used are $r_{max} = 2$ and $m_{max} = 5$.



(a) (b) (c)

Figure 15: Effect of the confidence measure 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.

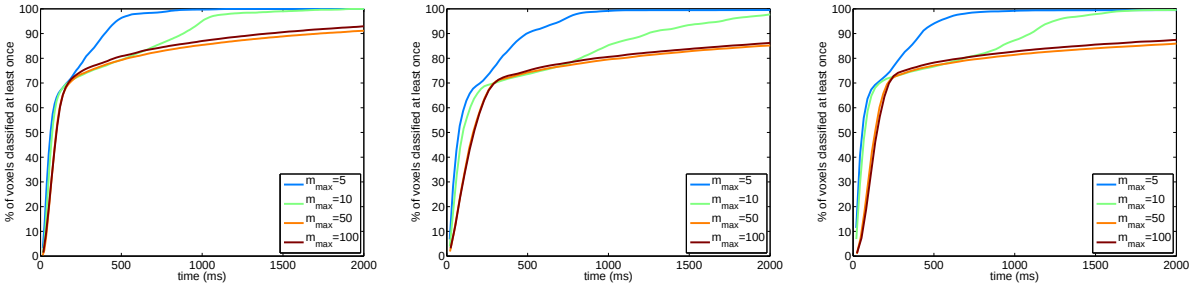
6.3.4 Parameters influence

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.

Confidence measure 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 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.



(a) (b) (c)

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.

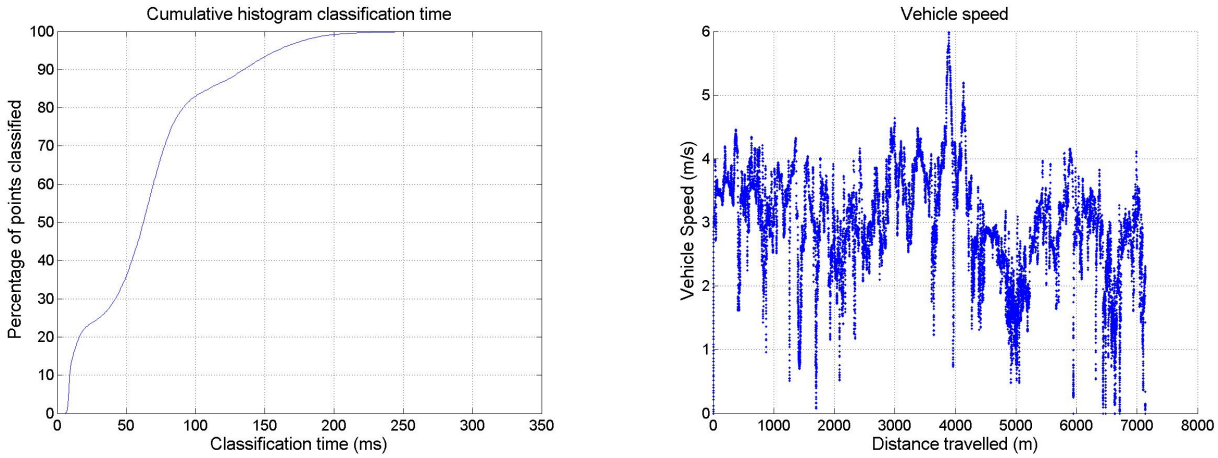
7 Field Tests

In this section, we present terrain classification results obtained from real-time data processing on-board a DEMO-III XUV. The code is first stress-tested over a very long distance, then classification performance is evaluated in various terrain types. Note that the system allows for two additional benefits. First, the low-level classification results are shared with mobility analysis modules via the Neutral Message Language (NML). In addition, classification results can be used in a high-level scene interpretation. Both these aspects are outside the scope of this paper, but the interested reader will find additional details in [15].

7.1 Long range traverses

Our system which uses the data structure presented in this paper was run over a cumulative distance of several dozens of kilometers on various terrains including trails, wooded areas with various tree densities and understorey canopy clutter, and meadows. For the tests reported here, the vehicle was tele-operated from a chase vehicle, reaching top speeds of 6 m/s. Figure 17 shows the classification time and vehicle speed for one of the runs, over a 7 km distance.

During those tests, the algorithm showed stable performance when run over long period of time on a vehicle at high-speed, in contrast to the previous data structure used in [15]. The classification process uses a pipe-line architecture with sequential data acquisition, data accumulation, saliency features extraction and classification. The use of this new data structure, in conjunction with the SVM classifier, produces an increase of up to 50% in point density.



(a) (b)

Figure 17: Endurance test of the code. (a) Cumulative timing duration between data insertion and final classification. (b) Vehicle speed.

7.2 Performance evaluation for different environments

Three environments with different configurations, thus resulting in different point cloud distributions, were selected. *Open Space*, is composed of short grass and rough terrain, and *Forest* is made of trees of different sizes and sometimes cluttered with understorey vegetation. Finally, *Dense Vegetation* is composed of one to two-meters tall dense bushes and tall grass. Note that the tests presented in this section and in Section 6.2.1 were performed at different locations. In each environment, four consecutive tests were performed with different combinations of data structures (optimized scan for dense voxel versus direct computation for sparse voxel) and classifiers (GMM versus SVM). They are respectively noted dveff, sv, gmm and svm in the timing cumulative histograms presented in Figure 18. The timing is reported as the time interval between data insertion and last voxel classification. Note that in Section 6.3.3 we presented results for only the data structure while here we include the classification time in addition. As expected the combination dveff-svm outperformed all others algorithm combinations tested. Each test was run over a distance of several hundred meters.

The GMM and SVM classifier performances were compared against hand-labeled data from several representative scenes. On average, no significant difference between the classifiers could be noted.

8 Conclusion

This paper deals with the challenging problem of processing large amounts of dynamic, sparse three-dimensional data. We present a data structure and an algorithm that improve the speed of range search for 3-D point-cloud processing. The data structure can then be used in a wide class of computations based on the extraction of saliency 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 [30, 15]. For the dynamic case, this data structure is able to reach improvements of up to 40% in speed. Live processing results obtained on an outdoor ground mobile robot also show significant improvements over previous methods.

Acknowledgements

Prepared through collaborative participation in the Robotics Consortium sponsored by the U.S Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0012.

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] S. Arya and D. Mount. Approximate range searching. *Computational Geometry*, 17(3), December 2000.

[3] T. Bodenmueller and G. Hirzinger. Online surface reconstruction from unorganized 3d points for the dlr hand-guided scanner system. In *International Symposium on 3D Data Processing, Visualization, and Transmission*, 2004.

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

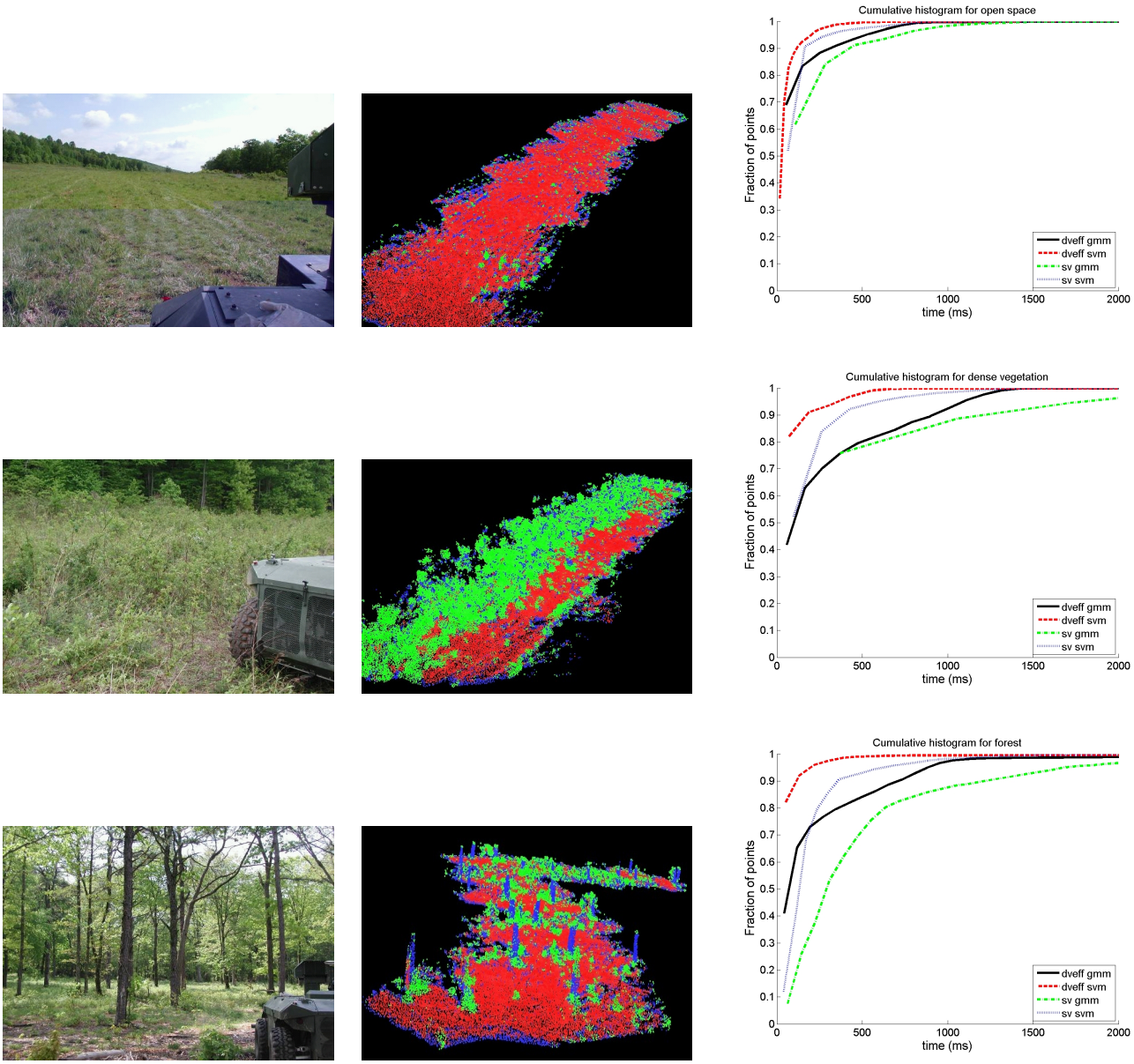


Figure 18: On-board performance evaluation for different environments. The top, center and bottom row contain respectively results from the *Open Space*, *Dense Vegetation* and *Forest* environment. The left/center/right columns respectively represent the scene, the classification results and the cumulative histogram of classification time as defined earlier. In the center column, reb/blue/green represent points classified as surface, linear and scatter as in [30].

[5] C. Burges. A tutorial on support vector machines for pattern recognition. In *Data Mining and Knowledge Discovery*, volume 2, 1998.

[6] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, second edition, 2000.

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

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

[9] 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.

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

[11] M. Hopf and T. Ertl. Hierarchical splatting of scattered data. In *IEEE Visualization Conference*, 2003.

[12] T. Joachims. *Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning*, chapter 11. MIT-Press, 1999.

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

[14] 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.

[15] J-F Lalonde, N. Vandapel, M. Huber, and M. Hebert. Natural terrain classification using three-dimensional ladar data for ground robot mobility. *Journal of Field Robotics*, 23(10), 2006.

[16] 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.

[17] 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.

[18] 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.

[19] M. Montemerlo and S. Thrun. A multi-resolution pyramid for outdoor robot terrain perception. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, San Jose, CA, 2004. AAAI.

[20] Andrew Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67– 91, March 1998.

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

[22] M. Pauly, L. Kobbelt, and M. Gross. Point-based multi-scale surface representation. *ACM Transactions on Graphics*, 25(2), 2006.

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

[24] R. Schnabel and R. Klein. Octree-based point-cloud compression. In M. Botsch and B. Chen, editors, *Symposium on Point-Based Graphics 2006*. Eurographics, July 2006.

[25] 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.

[26] M. Stytz, G. Frieder, and O. Frieder. Three-dimensional medical imaging: algorithms and computer systems. *ACM Computing Surveys (CSUR)*, 23(4):421–499, 1991.

[27] I. Tsochantaridis, T. Hofmann, T. Joachims, and Y. Altun. Support vector learning for interdependent and structured output spaces. In *International Conference on Machine Learning*, 2004.

[28] 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.

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

[30] 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.

[31] M. Wand and M. Jones. *Kernel Smoothing*. Chapman & Hall, 1995.