PROCESSING TERRAIN POINT CLOUD DATA *
RONALD DEVORE†, GUERGANA PETROVA‡, MATTHEW HIELSBERG§, LUKE OWENS¶, BILLY CLACK∥, AND
ALOK SOOD∗∗

Abstract. Terrain point cloud data are typically acquired through some form of LiDAR sensing. They form a rich resource
that is important in a variety of applications including navigation, line of sight, and terrain visualization. Processing terrain
data has not received the attention of other forms of surface reconstruction or of image processing. The goal of terrain data
processing is to convert the point cloud into a succinct representation system that is amenable to the various application
demands. The present paper presents a platform for terrain processing built on the following principles: (i) distortion is
measured in the Hausdorff metric, which we argue is a good match for the application demands, (ii) a multiscale representation
based on tree approximation using local polynomial fitting. The basic elements held in the nodes of the tree can be efficiently
encoded, transmitted, visualized, and utilized for the various target applications. Several challenges emerge because of the
variable resolution of the data, missing data, occlusions, and noise. Techniques to identify and handle these challenges are
developed.

Key words. surface reconstruction, point clouds, Hausdorff metric, compression, adaptive splines

AMS subject classifications. 65D17, 65D18, 41A15

1. Introduction. Terrain point clouds are now quite ubiquitous and used in a variety of applications
including autonomous navigation, change detection, and field of view calculations. The point clouds them-
selves are too cumbersome and large to be used for these purposes. They need to be converted to a simpler
platform that is more efficient and still contains all of the features of the terrain, present in the point cloud,
that are needed for these applications. A naïve approach would be to take local averages of data heights
to obtain pixel intensities (and therefore a pixelized image) and then employ the techniques of image pro-
cessing to make a conversion into a wavelet or other multiscale representation. However, this approach is
not successful for several reasons. Foremost among these is that terrains are not images. They have certain
topology and geometry that must be extracted and maintained for successful applications. A second related
point is that the usual least squares metrics used in image processing do not match the intended applications
for terrain maps. For example, capturing long thin structures such as poles, towers, and wires are essential
for navigation, but are not given priority in PSNR metrics employed for images. Another important point is
that terrain point clouds usually have missing data, occlusions and noise which do not appear in most other
applications.

While surface reconstruction from point clouds is now a dominant theme in computer graphics, very
little of this work addresses terrain data per se. The notable exception is the paper [26], which proposes a
Morse tree structure to represent terrain, but falls short of providing an implemented processing platform.
Of course, one could argue that one can simply apply one of the vast number of surface processing algorithms
in computer graphics. However, these algorithms are typically built for high resolution data which is not the
case for general terrain data, which suffers from occlusions, missing data, noise, and variable resolution.

The purpose of the present paper is to give a terrain point cloud processing algorithm based on the
following basic principles.

• Distortion is measured in the Hausdorff metric, which we argue matches well the intended applica-
tions.
• The decomposition is organized in a multiscale octree giving coarse to fine resolution.

---

* This research was supported by the ARO/DoD Contract W911NF-07-1-0185; the NSF grants DMS-0810869, DMS-0900632;
the Office of Naval Research Contracts ONR-N00014-08-1-1113, ONR-N00014-09-1-0107, and ONR-N00014-11-1-0712; the
AFOSR Contract FA9550-09-1-0500; the DARPA grant HR0011-08-1-0014. This publication is based in part on work supported
by Award No. KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).
†Department of Mathematics, Texas A&M, College Station, TX, 77843 (rdevore@math.tamu.edu)
‡Department of Mathematics, Texas A&M, College Station, TX, 77843 (gpetrova@math.tamu.edu)
§Institute of Scientific Computation, Texas A&M, College Station, TX, 77843 (hielsber@tamu.edu)
¶Automated Trading Desk, Mount Pleasant, SC, 29464 (lowens@atdesk.com)
∥Department of Computer Science and Engineering, Texas A&M, College Station, TX, 77843 (clac81@neo.tamu.edu)
∗∗Department of Computer Science and Engineering, Texas A&M, College Station, TX, 77843 (alok169@tamu.edu)
Each node of the tree corresponds to a dyadic cube and is ornated with a low degree polynomial fit to the point cloud on this cube and other attributes, such as the local Hausdorff error of this fit.

The tree is organized into subtrees each of which corresponds to a certain accuracy of resolution in the Hausdorff metric.

The tree and nodal information can be efficiently encoded using predictive encoding.

Upon receiving the tree and nodal information, the user can easily convert this information to a format that matches the intended application.

Primitives such as normals, curvature, and other information are easily extracted from the tree and nodal information.

Multiscale decompositions and local polynomial fits are often used in surface fitting (see [8, 15, 16, 17, 19, 25, 27, 28]). Among the things that separate our work from others is the measure and guarantee of performance in the Hausdorff metric and the fact that our methods can be applied to nonhomogeneous and non dense data.

A terrain point cloud $D$ is a finite set of three dimensional data points $x := (x, y, z)$. Such point clouds are typically obtained from a sensor or from images using Structure From Motion calculations. The terrain point clouds utilized in this paper are all drawn from some form of LiDAR sensing (real data) or a simulation of LiDAR sensing (synthetic data). In contrast to image processing or computer graphics, a canonical collection of point clouds that could be used to develop and test algorithms is not available. We propose such a collection here, which can be used and added to by other researchers. One limiting factor is that several interesting data sets are not available to the general public because of priority restrictions. Let us also note, that our algorithms are developed for data sets which only contain the point cloud. In some settings, one also has available the position and orientation of the sensor, which makes the task of processing the point cloud much simpler.

Some of the point clouds that we use in order to demonstrate the results of our algorithm are given in Figure 1.1. Among these are data sets taken from AFRL/MNG VEAA Data Set #1, which was created by taking a LiDAR scan of some real world topology. We have also developed a synthetic LiDAR sensor which allows us to create data sets with representative features. The data sets in this paper can all be downloaded from the web site http://www.math.tamu.edu/~hielsber/MURI_PointCloud/index.html.

This paper is organized as follows. In the following sections, we shall discuss the various tasks that have to be completed in our terrain processing engine. For each of these tasks, we describe the current algorithm which completes the task. After this, we describe in §7, our terrain processing algorithm. That section is then followed by a discussion how the terrain processing algorithm is used for some of the directed applications.

2. Dyadic Cubes and Octrees. Given a point cloud $D$, we process $D$ by using a dilation (with each coordinate dilated by the same factor) and a shift of the data in $D$ so that the output data $D'$ lies in the unit cube $\Omega = [0, 1]^3$. There are many such transformations from which we choose the following one. After the transformation, the range of the new $x_i$ should be $[1/2 - \delta_1, 1/2 + \delta_1]$ with $0 \leq \delta_1 \leq 1/2$. Similarly, the
range of the $y_i$ should be $[1/2 - \delta_2, 1/2 + \delta_2]$ with $0 \leq \delta_2 \leq 1/2$. For the new $z_i$, we require that the range be of the form $[2^{-\kappa - 1} - \delta_3, 2^{-\kappa - 1} + \delta_3]$ where $2^{-\kappa - 2} \leq \delta_3 \leq 2^{-\kappa - 1}$. Thus the $z$ values will all lie in $[0, 2^{-\kappa}]$. Finally, we require that one of the $\delta_i$, $i = 1, 2, 3$, equals $1/2$. We denote by $\text{INIT}$, the subroutine which takes as input $D$ and has as output $\text{INIT}(D) = (D', \kappa)$.

A dyadic subcube $Q$ of $\Omega$ of side length $2^{-i}$ (volume $2^{-3i}$) is of the form $2^{-i}([k_1, k_1 + 1] \times [k_2, k_2 + 1] \times [k_3, k_3 + 1])$ with the integers $k_1, k_2, k_3$ satisfying $0 \leq k_1, k_2, k_3 < 2^i$. We use the standard convention that the above intervals are closed on the right when the right endpoint is one. The children of $\Omega$ are the eight dyadic subcubes with side length $2^{-1}$. Each of these children has itself eight children and so on. Thus, the dyadic cubes organize themselves into an octree with root $\Omega$. We denote the set of all dyadic subcubes of $\Omega$ by $Q$. Some of the cubes in $Q$ will not contain data from $D$. We denote by $Q^*$ the set of all cubes in $Q$ that are occupied and for each cube $Q \in Q^*$, we denote by $D_Q := D' \cap Q$ the set of all data points in $Q$. The cubes $Q \in Q^*$ determine a subtree of the full dyadic octree.

A finite collection $T \subset Q$ is called an octree if whenever a cube is in $T$ then its parent and all of its siblings are in $T$. Our processing algorithm will input the data set $D$ and a tolerance $\eta$ and output a finite octree $T = T_\eta$.

3. Distortion Metrics. The development and assessment of terrain processing algorithms requires a metric which measures the distortion between two surfaces. In the case of image processing, this metric is usually chosen as a least squares metric and is tabulated in the Peak Signal to Noise Ratio (PSNR). We argue in this section that least squares metrics are not appropriate for the intended applications of terrain maps and that a more suitable way to measure distortion is through the Hausdorff metric.

Given two sets $A, B$ in $\mathbb{R}^3$, the one-sided Hausdorff distance from $A$ to $B$ is given by

$$\delta(A, B) := \sup_{a \in A} \inf_{b \in B} |a - b|,$$

where $|\cdot|$ is the standard Euclidean distance in $\mathbb{R}^3$. Obviously, this guarantees that for any $a \in A$, there is $b \in B$ whose Euclidean distance to $a$ does not exceed $\delta(A, B)$. The Hausdorff distance between two sets $A$ and $B$ is then defined as

$$\delta_H(A, B) := \max\{\delta(A, B), \delta(B, A)\}.$$

Notice that $\delta_H$ is a metric on sets, and in particular satisfies the triangle inequality

$$\delta_H(A, C) \leq \delta_H(A, B) + \delta_H(B, C)$$

for any sets $A$, $B$, $C$.

A depiction of the Hausdorff distance between a point cloud $D$ and a curve $C$ in two dimensions ($2D$) is given in Figure 3.1, where $\delta(D, C)$ is the one-sided distance from $D$ to $C$ and $\delta(C, D)$ is the one-sided distance from $C$ to $D$.

Given two finite sets $A, B \in \Omega$, we denote by $\text{DIST}(A, B)$ the subroutine that returns the one-sided Hausdorff distance from $A$ to $B$, and by $\text{HAUS}(A, B)$ the subroutine that computes $\max\{\text{DIST}(A, B), \text{DIST}(B, A)\}$, which is the Hausdorff distance between $A$ and $B$. For our implementation of $\text{HAUS}$, we utilized the spatial searching features in the CGAL library [2] to reduce the computational complexity of this algorithm from quadratic to logarithmic.

We shall frequently need to compute the Hausdorff distance between a finite set $A$ and a continuum surface $S$, such as a plane or quadric surface. There are fast algorithms for computing the Hausdorff distance between two surfaces [3, 6, 14, 29], most of which involve the discretization of the surfaces in question.

Normally, as is traditional in numerical analysis, we do not indicate the fact that a subroutine does not provide exact computation. However, we will make an exception in the case of computing Hausdorff distances between a point set $A$ and a surface $S$, since this subroutine plays a special role in our algorithm. With this in mind, we choose a numerical tolerance $\gamma = 2^{-m}$ with $m$ a positive integer. We consider the tiling $Q_m$ of $\Omega$ into
Fig. 3.1: Hausdorff distance from point cloud to curve and curve to point cloud.

dyadic cubes of sidelength $\gamma$ and create the point set $S_\gamma$ which consists of all points $x$ such that $x$ is the center of a cube $Q \in Q_n$ which contains points of the surface $S$, see Figure 3.2. Then $\text{DIST}_\gamma(A, S) := \text{DIST}(A, S_\gamma)$ takes as input the finite set $A$, the surface $S$ and the numerical tolerance $\gamma$ and returns the distance between $A$ and $S_\gamma$. Since $\delta_H(S, S_\gamma) \leq \sqrt{3} \gamma$, this computation is an approximation to $\delta(A, S)$ which is accurate to tolerance $\sqrt{3} \gamma$. We can similarly compute $\text{DIST}_\gamma(S, A)$. The subroutine $\text{HAUS}_\gamma$ takes as input any pair of $A, S$ and the tolerance $\gamma$ and returns $\text{HAUS}_\gamma(A, S) := \max\{\text{DIST}_\gamma(A, S), \text{DIST}_\gamma(S, A)\}$.

Fig. 3.2: Discretization of a 2D curve for Hausdorff computation.

To understand why the Hausdorff metric is appropriate for terrain applications, let us consider two such applications. First consider the problem of navigating an unmanned vehicle, for example a Micro Air Vehicle. Sensors extract a point cloud, which describes surfaces that the vehicle must avoid (no fly zones). Suppose that we know that the true surface $S$ has a Hausdorff distance $\epsilon$ from the point cloud; this is an assumption on the quality of the data which is necessary to proceed with any certainty. Suppose we use the point cloud to find a surface $\hat{S}$ which is within $\eta$ of the point cloud in the Hausdorff metric. Then, whenever the vehicle remains a distance greater than $\eta + \epsilon$ from the approximate surface, it is guaranteed to avoid the true surface. Moreover, no weaker metric (such as least squares) can guarantee such a performance.

As a second example, consider observing a terrain surface $S$ from an observation point $x$. The field of view describes what the observer can see from this vantage point taking occlusions into consideration. If we construct an approximation $\hat{S}$ to $S$ from given point cloud data, then the field of view will not be accurate but will have false positives and false negatives. The quality of the approximate field of view will depend on the metric used to compute the approximation $\hat{S}$. In Figure 3.3, we give a comparison of computing the field of view using three metrics with a comparable error. The left image uses the least squares metric, the center image uses a maximum $z$-value deviation, and the right uses the Hausdorff metric. The points colored yellow are false positives, and those colored red are false negatives. The grey colored points are correctly classified. One sees that when distortion is measured in the Hausdorff metric we obtain the largest agreement with the true field of view.
4. Multiscale Decompositions. Our processing algorithm is based on a multiscale decomposition of the surface using low degree algebraic surfaces to locally approximate the given point cloud $D$. In the algorithms implemented in this paper, the algebraic surfaces are either planes (corresponding to linear polynomials) or certain quadric surfaces (corresponding to special choices of quadratic polynomials described below). In this section, we explain the multiscale structure which is based on dyadic cubes and also how we extract the polynomial fits.

We wish to associate a local polynomial fit to the point cloud on the cube $Q$. To do this, we need to assume that $Q$ contains sufficiently many points from the point cloud. In our algorithms, we assign a tolerance $K$ and require $Q$ to have at least $K$ points from $D$ whenever we ask our algorithm to assign a polynomial fit. The value of $K$ can be set by the user. Of course, it should be larger than the number of degrees of freedom in the local polynomial fits. But it also should be large enough to avoid fitting noise in the data. In our implementation, we set $K = 10$ for the examples in this paper. This value of $K$ was motivated by considerations from learning theory, see [9, 10].

4.1. Planar Fits to the Data. Suppose that $D_0$ is a subset of $D$. We look to fit the data $D_0$ by a plane. Any plane can be described as the zero set of a linear function $L(x) = n \cdot x + c$ on $\mathbb{R}^3$, where $n$ is a unit normal to the plane and $c$ is a suitable constant. Also $|L(x)|$ is the distance of any given point $x$ to this plane.

While we have emphasized that we measure the distortion of our data fitting in the Hausdorff metric, it turns out that finding the best planar fit to the data $D_0$ in the Hausdorff metric is too computationally intensive. So we will take another approach to finding a linear fit by using Principal Component Analysis (PCA). We emphasize, however, that we continue to evaluate the performance of this fit in the Hausdorff metric.

PCA finds a linear function $L(x) = L_{D_0}(x) = n_{D_0} \cdot x + c_{D_0}$, from the set $\mathcal{L}$ of all linear functions such that

$$L_{D_0} := \text{argmin}_{L \in \mathcal{L}} \sum_{x \in D_0} |L(x)|^2.$$ \hspace{1cm} (4.1)

Using Lagrange Multipliers, one can solve this minimization problem by first finding the eigenvalues and eigenvectors of the covariance matrix for the components $x, y, z$ of the data $D_0$:

$$\Sigma_{D_0} := \begin{pmatrix} \text{cov}(x, x) & \text{cov}(x, y) & \text{cov}(x, z) \\ \text{cov}(x, y) & \text{cov}(y, y) & \text{cov}(y, z) \\ \text{cov}(x, z) & \text{cov}(y, z) & \text{cov}(z, z) \end{pmatrix},$$ \hspace{1cm} (4.2)

where $\text{cov}(x, y) := \sum_{(x,y,z) \in D_0} (x - \bar{x}_{D_0}, y - \bar{y}_{D_0})$, with $\bar{x}_{D_0}$ the mean of the $x$ components of the data $D_0$, and the sum is taken over all the data points in $D_0$. Then $n_{D_0}$ is a multiple of the eigenvector corresponding to the smallest eigenvalue of $\Sigma_{D_0}$ and $c_{D_0}$ is chosen so that $\sum_{x \in D_0} L_{D_0}(x) = 0$. Notice that there is not necessarily a unique solution to (4.1) when the two smallest eigenvalues are equal. Since the matrix $\Sigma_{D_0}$ is $3 \times 3$ and positive semi-definite, this eigenvalue problem can be solved efficiently using any standard solver.
such as LAPACK, CGAL or Geometric Tools [1, 2, 4]. The plane associated to \(L_{D_0}\) is our default planar fit to the data \(D_0\), and in most instances it performs satisfactorily.

In summary, we define a subroutine \(\text{PCA}\) that takes as input the data set \(D_0\), with the property \(\#(D_0) \geq K\), and returns \(\text{PCA}(D_0) = (\lambda_1, \lambda_2, \lambda_3; v_1, v_2, v_3)\) which are three eigenvalues (written in decreasing order) and corresponding eigenvectors of the matrix \(\Sigma_{D_0}\).

From the algorithm \(\text{PCA}\), we define a new algorithm \(\text{PLANE}\), which takes as input any axis-oriented three-dimensional rectangular box \(B\) (parallelepiped). It outputs the set \(\hat{S}_B := \text{PLANE}(B)\), which is the restriction to \(B\) of the plane obtained from \(\text{PCA}(D_B)\), where \(D_B := D \cap B\).

### 4.2. Quadratic Fits to the Data.

The quality of approximation to the data on a box \(B\) can generally be improved by replacing planes by algebraic surfaces that are zero sets of higher degree polynomials. We use quadratic polynomials in our processing algorithm described below. If one utilizes general quadratic functions in three variables, then the zero sets are quadric surfaces that may have branches. To avoid this, we limit the types of quadratic polynomials, and hence the quadric surfaces that can be used in our algorithms. To describe this limitation, we return to the coordinate system given by \(\text{PCA}(D_B)\).

Given a box \(B\), we use a change of coordinates that replaces the canonical \(x, y, z\) coordinates by the coordinates given by the basis of the three eigenvectors found by PCA written in order of increasing size of the eigenvalues (with ties handled arbitrarily). This maps the coordinates \(x = (x, y, z)\) to new coordinates \(u = (u, v, w)\). We denote by \(\bar{D}_B\) the transformed data points. We shall use quadratics \(P(u,v) \in \mathcal{P}_2\), where \(\mathcal{P}_2\) the set of all quadratics in \(u,v\). We define

\[
P_B := \arg\min_{P \in \mathcal{P}_2} \sum_{(u,v,w) \in \bar{D}_B} |P(u,v) - w|^2, \tag{4.3}
\]

which is a least squares fit to the data on \(B\). We denote by \(\hat{S}_B\) the quadric surface, described as the set of all \((u,v,w) \in B\), such that \(w = P_B(u,v)\). The solution to the least squares problem (4.3) is easy to find from the Moore-Penrose formula for least squares problems.

We define a subroutine \(\text{QUAD}\) that takes as input a box \(B\) with \(\#(D_B) \geq K\) and returns the quadric surface \(\hat{S}_B\). We want to emphasize once again that \(\hat{S}_B\) is not the best Hausdorff fit to the data from quadric surfaces of the above type. This would be too expensive to compute. However, we shall still measure the quality of fit of \(\hat{S}_B\) to our point cloud by using the Hausdorff metric.

[Fig. 4.1: Comparison of planar and quadratic fits.]

Figure 4.1 shows the planar and quadratic fits to a portion of test data from Figure 1.1, representing part of the building’s wall and nearby ground. Table 4.1 gives the one-sided Hausdorff distances between the point cloud and the fits. Clearly, for these data points a quadratic fit substantially outperforms the planar fit.

### 4.3. General Fitting.

It is clear that \(\text{QUAD}\) will always give at least as good of a distortion as \(\text{PLANE}\). However, we found that in applications such as compression, the additional overhead required to encode the output of \(\text{QUAD}\) sometimes outweighed any benefit. It should be noted that while we found
Table 4.1: Comparison of one-sided Hausdorff distances for Figure 4.1 (values are given with respect to the original data units).

<table>
<thead>
<tr>
<th></th>
<th>PLANE</th>
<th>QUAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta(D_Q, S_Q)$</td>
<td>0.9799</td>
<td>0.2722</td>
</tr>
<tr>
<td>$\delta(S_Q, D_Q)$</td>
<td>0.4192</td>
<td>0.3164</td>
</tr>
</tbody>
</table>

5. Improving Hausdorff Fits. The planar and quadric surfaces $\hat{S}_Q$ on a cube $Q \in Q^*$, described in the previous sections, are generated from least squares methods and may not be accurate in the Hausdorff metric. One way to remedy this situation is to subdivide $Q$ into its children and repeat the fitting on each child. However, often the reason for the poor fit is simply because the planar/quadratic fit spreads over the entire cube, whereas the data is very localized on the cube. This manifests itself in having $\delta(D_Q, \hat{S}_Q)$ small, but $\delta(\hat{S}_Q, D_Q)$ large, see Figure 5.1. There are ways to remedy this situation by localizing the surface. In this section, we describe some methods for implementing this, as well as testing and applying such methods when $\delta(D_Q, \hat{S}_Q)$ is smaller than our preassigned Hausdorff error threshold but $\delta(\hat{S}_Q, D_Q)$ is not.

5.1. Bounding Boxes. Our first technique is to find one or two clusters of the data in $Q$, described by bounding boxes, in order to more accurately represent the underlying geometry. The first step of the bounding box method is to find the smallest axis-aligned box that contains $D_Q$. The octree data structure keeps track of the minimum and maximum $x$, $y$ and $z$ coordinates for $D_Q$ in each cube $Q$ dynamically as the tree is being built. These minimum and maximum coordinates provide three intervals, $I_x, I_y, I_z$ representing the smallest possible fit of an axis-aligned bounding box around the data. Thus, the single bounding box on $Q$ is simply $B_Q := I_x \times I_y \times I_z$. We now compute $\delta(\hat{S}_{B_Q}, D_Q)$ and $\delta(D_Q, \hat{S}_{B_Q})$, where $\hat{S}_{B_Q} := \text{FIT}(B_Q)$. If both quantities are smaller than our preassigned Hausdorff error tolerance, we accept $\hat{S}_{B_Q}$ and do not further refine $Q$. Figure 5.1 shows a synthetic example (in two dimensions), where the algorithm terminates with an accurate Hausdorff fit using the surface $\hat{S}_{B_Q}$ in $B_Q$; therefore in this case, the use of bounding boxes eliminates the necessity to continue subdividing $Q$.

We define the subroutine $\text{BOX}_\gamma$ that takes as input a dyadic cube $Q$ and the numerical tolerance $\gamma$. It finds the bounding box $B_Q$, the fit $\hat{S}_{B_Q}$, and the Hausdorff distance $\hat{\eta}_Q := \text{HAUS}_\gamma(\hat{S}_{B_Q}, D_Q)$. Thus $\text{BOX}_\gamma(Q) = (B_Q, \hat{S}_{B_Q}, \hat{\eta}_Q)$.

If $\hat{\eta}_Q$ is bigger than the prescribed Hausdorff tolerance then we attempt to cluster the data into two groups. There are several existing algorithms in the literature for clustering data sets that could be used for this purpose. We have implemented two methods: the sliding-partition method and the $k$-means clustering algorithm.

The sliding-partition method described here does not operate on $Q$, but instead on the single bounding box $B_Q$ found in the previous step. The reason for doing this is that $B_Q$ is already the tightest axis-aligned fit
also compute the best linear (or quadratic) fit to the data on each of these bounding boxes using the sliding-partition method. A typical iterative algorithm begins by choosing a point in the points in \( D_Q \) onto the \( x \)-axis written in increasing order \( x_i \leq x_{i+1}, \ i = 1, \ldots, M - 1 \). Let \( P_d := [x_1, x_1 + d \cdot L] \) and \( P'_d := (x_1 + d \cdot L, x_M], \ d = 1, \ldots, N \), be the interval choices along the axis, where \( L := (x_M - x_1)/(N + 1) \). If we fix a value of \( d \), then \( D_Q \) can be partitioned into two sets \( D_1(d) \) and \( D_2(d) \), where \( D_1(d) \) contains all the data points \( x \in D_Q \) whose projection is in \( P_d \), and \( D_2(d) \) contains all the data points whose projections are in \( P'_d \). We find the two bounding boxes \( B_1(d), B_2(d) \) for \( D_1(d), D_2(d) \), respectively, each by the single bounding box method \( \text{BOX}_i \). We also compute the best linear (or quadratic) fit to the data on each of these bounding boxes using \( \text{FIT} \). The Hausdorff errors on each box are also computed and the maximum of these two errors is stored as the error on each of these boxes.

To evaluate a given cluster assignment, the sum of the squares of distances from the points to their corresponding cluster mean is computed. One then determines the \( \eta^{(1)} \) error on each box are also computed and the maximum of these two errors is stored as the error for this partition. We then choose the minimum of these errors over all partitions and denote this value by \( \eta_Q^{(1)} \). We do a similar calculation starting with the \( y \) (respectively \( z \)) values of the data and thereby obtain error \( \eta_Q^{(2)} \) (respectively \( \eta_Q^{(3)} \)). We now choose the bounding boxes corresponding to the smallest of the three errors \( \eta_Q^{(1)}, \eta_Q^{(2)}, \eta_Q^{(3)} \), which we denote by \( \hat{\eta}_Q \).

We denote by \( \text{BOX} 2_\gamma \) the above algorithm, which takes as input a dyadic cube \( Q \) and the numerical tolerance \( \gamma \), and outputs two boxes \( B'^{(1)}_Q, B'^{(2)}_Q \), their surface fits \( \hat{S}_{B'^{(1)}_Q} = \text{FIT}(B'^{(1)}_Q), \hat{S}_{B'^{(2)}_Q} = \text{FIT}(B'^{(2)}_Q) \) and the Hausdorff error \( \hat{\eta}_Q \), which is the maximum of the \( \text{HAUS}_\gamma \) error on each of these boxes. Thus, we have \( \text{BOX} 2_\gamma (Q) = (B'^{(1)}_Q, B'^{(2)}_Q, \hat{S}_{B'^{(1)}_Q}, \hat{S}_{B'^{(2)}_Q}, \hat{\eta}_Q) \). An example of the sliding-partition method for a synthetic two dimensional data set is given in Figure 5.2, where the result of \( \text{BOX}_\gamma \) is shown on the left, an intermediate partition from the sliding-partition method is shown in the middle, and the result of \( \text{BOX} 2_\gamma \) is shown on the right.

![Fig. 5.2: An example of the sliding-partition method.](image)

An alternative to the sliding-partition method is the \( k \)-means algorithm, see [22]. This is the most widely used unsupervised clustering algorithm, and it shows good results in our experiments with terrain data. However, it is more computationally intensive than the sliding-partition method described above.

The ideal \( k \)-means clustering algorithm, when applied to three dimensional point cloud data, groups points into \( k \) distinct clusters based on their Euclidian distances. Given a set of data points \( x := (x, y, z) \) and a cluster count \( k \) (we choose \( k = 2 \) in our case to split the points into two groups), the algorithm outputs a cluster assignment where each input point is assigned to one of the \( k \) clusters. The mean of each cluster is computed. To evaluate a given cluster assignment, the sum of the squares of distances from the points to their corresponding cluster mean is computed. One then determines the \( k \) clusters that minimize this sum over all possible assignments.

The ideal \( k \)-means algorithm is too computationally intensive, so it is typically replaced by an iterative algorithm. A typical iterative algorithm begins by choosing \( k \) points \( P_1, \ldots, P_k \) at random. The clusters \( C^{(0)}_1, \ldots, C^{(0)}_k \) are determined by assigning a point from the data set to cluster \( C^{(0)}_j \) if its distance to \( P_j \) is the smallest. Now the means \( M^{(1)}_j \) of all clusters \( C^{(0)}_j \) are computed, \( j = 1, \ldots, k \). A new cluster assignment \( C^{(1)}_1, \ldots, C^{(1)}_k \) is determined as above where a point from the data set is assigned to cluster \( C^{(1)}_j \) when its distance to \( M^{(1)}_j \) is smallest. This cluster assignment and mean computation is iterated until a user-defined
number of iterations is reached.

Our implementation uses the \textit{k-means++} algorithm [5], where rather than simply picking the points \( P_1, \ldots, P_k \) at random, one uses a greedy method that guarantees their maximum separation. Our implement-
ation of the above ideas is the subroutine \textsc{2MEANS}, (i.e. \( k = 2 \)). This algorithm takes as input a dyadic cube \( Q \) and the numerical tolerance \( \gamma \) and applies the \textit{2-means++} algorithm to find two clusters, \( C_1 \) and \( C_2 \). We then define the smallest axis-aligned boxes \( B_Q^{(1)}, B_Q^{(2)} \), their surface fits \( \hat{S}_{B_Q^{(1)}}, \hat{S}_{B_Q^{(2)}} \) and the Hausdorff error \( \hat{\eta}_Q \), which is the maximum of the result of \textsc{Haus}, on each of these boxes. Thus, \( \textsc{2MEANS}_{2}(Q) = (B_Q^{(1)}, B_Q^{(2)}, \hat{S}_{B_Q^{(1)}}, \hat{S}_{B_Q^{(2)}}, \hat{\eta}_Q) \). We also define the subroutine \textsc{Cluster}, as the generic delegate for the \textsc{Box2}, and \textsc{2MEANS}, subroutines.

6. Preprocessing to Remove Noise and Outliers. The LiDAR acquired terrain point cloud data are typically noisy. The type and amount of noise varies depending on the sensor and data collected. In this section, we discuss a few preprocessing algorithms we have implemented to identify and remove some of the present noise and outliers.

Locally Optimal Projection: We have implemented the locally optimal projection method from [21]. We refer the reader to [21] for its description and motivation. We have applied this method locally on \( D_Q \). This method performs rather well on point cloud coming from a \( C^2 \) surface, but when this is not the case, e.g. for terrain surfaces, it leads to unwanted artifacts and missing detail.

Outlier Measure: We have also implemented the outlier measure method from [32], where a measure for determining whether or not a data point is noise is presented. This method assigns to each point \( P \) a score \( \chi(P) \). A simple thresholding of these scores produces the denoised point cloud. The score \( \chi \) for \( P \) is computed as \( \chi(P) = \omega_1 \chi_1(P) + \omega_2 \chi_2(P) + \omega_3 \chi_3(P) \), where \( \omega_1, \omega_2, \omega_3 \) are scalar weights defined by the user, and \( \chi_1, \chi_2 \) and \( \chi_3 \) are determined as follows. First, to compute \( \chi_1(P) \), a value of \( k \) is chosen by the user and then a least-squares plane is fit through the point’s \( k \)-nearest neighbors, and the distances of all points in that neighborhood to the plane are calculated. Then, \( \chi_1(P) = d_P(d_P + \bar{d})^{-1} \), where \( d_P \) is the distance from \( P \) to the plane and \( \bar{d} \) is the mean distance of all points in that neighborhood to the plane. Next, \( \chi_2 \) is computed by \( \chi_2(P) = q_P(q_P + 2r/\sqrt{k})^{-1} \), where \( q_P \) is the distance from \( P \) to the center of the sphere, determining the \( k \)-neighborhood and \( r \) is its radius. Lastly, \( \chi_3(P) = N_P k^{-1} \), where \( N_P \) is the number of \( k \)-neighbors, created from the \( k \)-neighbors of \( P \) that do not contain \( P \). Our tests show that this method performs well at identifying noise along smooth surfaces as well as outliers. However, we have noticed that it may treat edges, corners and boundaries between regions with different sampling densities as noise.

One-Sided Hausdorff Outlier Measure: To enhance denoising with respect to the Hausdorff method, we have developed our own denoising method called the one-sided Hausdorff outlier measure. It is based on the idea that outliers should be relatively far from any local fitting of the data. This method depends on a user defined tolerance \( \zeta \), and assigns to each point \( P \) a score \( \xi(P) \). As in the previous method, a simple thresholding of the calculated scores produces the denoised point cloud.

To compute \( \xi(P) \), we do the following with a user defined value of \( k \). For each point \( R \) we consider the set \( N(R) \) of its \( k \)-nearest neighbors and we fit a least-squares plane through this set. For every point \( P \in N(R) \), we consider a sphere with center \( P \) and radius \( \zeta \). If this sphere intersects the plane, we assign \( d_R(P) = 0 \). Otherwise, we set \( d_R(P) \) to be the reciprocal of the number of data points in this sphere. Thus, we assign a number \( d_R(P) \) to each point \( P \), viewed as a point from the \( k \)-nearest neighbors of \( R \). In general, a point \( P \) belongs to several \( k \)-nearest neighbors and has associated to it several numbers \( d_R(P) \). Now, the score \( \xi(P) \) is the average of \( d_R(P) \) taken over all \( k \)-nearest neighborhoods \( N(R) \) that contain \( P \).

This algorithm can be viewed as an extension of the outlier measure method with weights \( (\omega_1, \omega_2, \omega_3) = (1, 0, 0) \). We have found in our tests that it performs similarly to the outlier measure method. However, it is more successful than the outlier measure method when processing corners or edges. It does not treat them as noise, especially when the user defined tolerance \( \zeta \) is bigger than the local sampling rate.

7. Processing Algorithm. In this section, we shall describe our processing algorithm \textsc{Main}. The input to this algorithm is the output data set \( D' \) of \textsc{Init}, a desired target Hausdorff error \( \eta^* \), a tree depth \( \ell \), a numerical tolerance \( \gamma \), such that \( \gamma \leq \frac{1}{\sqrt{3}} \eta^* \), and a user choice of whether the algorithm is to use planar
or quadric surfaces in the subroutine FIT and whether the algorithm is to use the sliding-partition method or k-means in the subroutine CLUSTER. The output is an octree $T$ with depth at most $\ell$. Every node of the tree is adorned with either a local polynomial fit to the data or a flag that says no local fit is available. We define the ‘surface’ $S_{\text{out}}$ to be the union of the local polynomial fits on the leaf nodes of $T$. $S_{\text{out}}$ is typically not the type of surface we see in terrain processing because it is fragmented (primarily due to the use of bounding boxes), discontinuous and is not oriented with definable inside and outside. In §9, we will explain how to obtain smooth oriented surfaces from the output of MAIN.

Each node of $T$ that has a polynomial fit also contains an upper bound $\hat{\eta}_Q$ for the local Hausdorff error. The Hausdorff error $\hat{\eta}_Q$ will be less than our threshold $\eta^*$ for each leaf node $Q$, which contains a polynomial fit, except in the case where further subdivision of the node $Q$ is artificially halted by the user’s choice of $\ell$. Note that the global Hausdorff error $\eta_{S_{\text{out}}}$ between $S_{\text{out}}$ and $D'$, computed with accuracy $\gamma$, is also an output of MAIN.

Our processing algorithm is quite simple; it recursively applies a single subroutine called PROCQ. To describe this subroutine, let us first define $\eta := \frac{\eta^*}{\gamma}$ to be our computational tolerance. The subroutine PROCQ takes as input a cube $Q$, the computational tolerance $\eta$, the numerical tolerance $\gamma \leq \frac{1}{\sqrt{3}} \eta^*$ and the specification of linear or quadric fit in FIT. It outputs the current best fit $\hat{S}_{D_Q}$ to the data $D_Q$ on $Q$ and the Hausdorff error $\hat{\eta}_Q$. Thus if $Q$ is not flagged then PROCQ$(Q, \eta^*) = (\hat{S}_{D_Q}, \hat{\eta}_Q)$, where $\hat{S}_{D_Q}$ is either $\hat{S}_Q$, $\hat{S}_{B_Q}$ or $(\hat{S}_{B_Q}^{(1)}, \hat{S}_{B_Q}^{(2)})$. The subroutine is described as follows.

```plaintext
if $\#(D_Q) < K$ then
    Flag $Q$ and exit.
end if
Call FIT$(Q) = \hat{S}_Q$, and HAUS$(\hat{S}_Q, D_Q) = \hat{\eta}_Q$, set $\hat{S}_{D_Q} = \hat{S}_Q$.
if DIST$(\hat{S}_Q, D_Q) > \eta$ and $\text{DIST}^\gamma(D_Q, \hat{S}_Q) \leq \eta$ then
    Call BOX$(Q) = (B_Q, \hat{S}_{B_Q}, \hat{\eta}_Q)$, set $\hat{S}_{D_Q} = \hat{S}_{B_Q}$.
    if $\hat{\eta}_Q > \eta$ then
        Call CLUSTER$(Q) = (B_Q^{(1)}, B_Q^{(2)}, \hat{S}_{B_Q}^{(1)}, \hat{S}_{B_Q}^{(2)}, \hat{\eta}_Q)$, set $\hat{S}_{D_Q} = (B_Q^{(1)}, B_Q^{(2)})$.
    end if
end if
if $\hat{\eta}_Q > \eta$ then
    Set $\hat{S}_{D_Q} = \text{FIT}(Q)$, and $\hat{\eta}_Q = \text{HAUS}^\gamma(\hat{S}_Q, D_Q)$.
end if
```

Notice that if the local fit does not provide Hausdorff error $\leq \eta$ then an attempt is made to fit the data on $Q$ through one or two bounding boxes. However, if these fail and we are not at finest level $\ell$, then we return to the entire cube $Q$ in the further processing. That is, we never subdivide bounding boxes only the entire cube $Q$.

Given the above specification of PROCQ, the main processing algorithm MAIN$(D', \eta^*, \ell, \gamma) = (T, \eta_{S_{\text{out}}})$ maintains a set of cubes to be processed and a set of cubes that have been assigned to the octree. The method is described as follows.

Initialize the cube processing list as $\{\Omega\}$ and the octree $T$ as empty.
for Each cube $Q$ in the processing list do
    Call PROCQ$(Q, \eta^*)$.
    if $Q$ is flagged by PROCQ, then
        Add $Q$ and the flag to $T$.
    else
        Add $Q$ and its adornments $\hat{S}_{D_Q}$ and $\hat{\eta}_Q$ to $T$.
    end if
if $|Q| > 2^{-3\ell}$ and $\hat{\eta}_Q > \eta$ then
    Add the children of $Q$ to the end of the processing list.
end if
Remove $Q$ from the processing list.
end for
Assign \( \eta_{S_{\text{out}}} = 0 \).
for Each leaf cube in \( \mathcal{T} \) do
Assign \( \eta_{S_{\text{out}}} = \max\{ \text{HAUS}\_2(S_{D_Q}, D'), \eta_{S_{\text{out}}} \} \).
end for

We next make some observations on the output \( \mathcal{T} \) of \textbf{MAIN}. We know that each non-flagged terminal node \( Q \) of \( \mathcal{T} \) is adorned by a local polynomial surface produced by \textbf{FIT} whose Hausdorff distance to the data \( D_Q \) in \( Q \) is at most \( \eta_Q \). Generally, we have \( \eta_Q \leq \eta \). The only exception to this is if \( Q \) is at dyadic level \( \ell \) and the subdivision was stopped because of the user imposed level restriction. In the case that no leaf cube is flagged nor artificially stopped by the condition on the maximal depth of \( \mathcal{T} \), then the surface \( S_{\text{out}} \) will have Hausdorff distance \( \eta_{S_{\text{out}}} \) to the point cloud, and \( \eta_{S_{\text{out}}} \leq \eta^* \), which is the original goal of the algorithm. Note that flagged leaf cubes may have fewer than \( K \) points, which is why in \textbf{MAIN} all leaf nodes in \( \mathcal{T} \) are used to compute \( \eta_{S_{\text{out}}} \). We have found that with properly denoised data it is almost always the case that \( \eta_{S_{\text{out}}} \) is close to the target accuracy \( \eta^* \). Indeed, the cube is flagged because it has too few points but it generally has a neighboring cube which is not flagged and therefore has a local polynomial fit on that neighbor. When this is not the case, one could say that the points in the flagged cube are outliers and most of them have been removed by the preprocessing algorithms from §6.

8. Encoding. We anticipate that one of the main applications of our fitting algorithm will be compression and encoding of the point cloud. The output of \textbf{MAIN} is an octree whose nodes are adorned with coefficients of polynomials. This is a common setting in image and surface compression, and there are several approaches to converting such an ornated tree to a bit stream which the user can decode to find the tree and quantized coefficients \[25, 27\]. For our implementation, we have utilized the predictive encoder developed by the Rice group \[31\]. Note that here a simple switch of data structures is necessary, since the the Rice encoder works on an octree whose nodes are adorned with a single polynomial surface. In our case, the terminal nodes may have bounding boxes and may have two polynomials. We therefore have to encode this additional structure. The compression figures given below include the extra bits needed to encode this extra structure but done currently in a rather naive way. With Rice, we are improving on this encoder to include burn in and progressivity and will report on this in a forthcoming paper.

For large terrain data sets the ranges in \( x \) and \( y \) are typically significantly larger than the range in the vertical direction \( z \). This is the reason we have introduced the integer \( \kappa \) and the initialization step \textbf{INIT}. This step shifts the data in the \( z \) direction and compute the largest nonnegative integer \( \kappa \), such that \( D' \subset [0, 1]^2 \times [0, 2^{-\kappa}] \). This fact was not utilized in \textbf{MAIN} but will be exploited in the encoding. Namely, if \( \kappa \) is large, then there are a lot of cubes in the tree at levels coarser than or equal to \( \kappa \), which have no data but would be encoded if we began the encoding at the root \([0, 1]^3\). For this reason, we will instead encode a forest starting with the occupied cubes at level \( \kappa \). This will improve the encoding because we remove \( \kappa \) levels from the encoding of the single octree \( \mathcal{T} \), including the intermediate fits, and replace those by encoding \( \kappa \) and the root of each of the subtrees. This allows us to restrict the data to at most \( 2^{2\kappa} \) subtrees whose separate encoding results in higher compression rates. This step generally improves the compression rate over simply starting with \( \Omega = [0, 1]^3 \) as the root of an octree. The encoding of \( \kappa \) and the subtree roots is not free, and thus some overhead is required. To ensure that this overhead does produce lower overall compression than simply encoding the original octree, we impose a minimum value \( \kappa_0 \) and require \( \kappa > \kappa_0 \) before the subtrees are considered separately for encoding. In all our experiments, we have \( \kappa_0 = 1 \).

Next, we give some numerical results which tabulate the rate distortion performance of our algorithm \textbf{MAIN} when coupled with the Rice encoder. Note that one can utilize \textbf{MAIN} with different Hausdorff target tolerances \( \eta^* \) in order to create a progressive encoder analogous to wavelet-based image encoding, see for example \[11, 18\]

For our experiments, we first denoise all point cloud data sets, using the Outlier Measure algorithm, described in §6.

\textbf{Test 1}: Our first test is the Maple Mountain data set \[7\], generated by high-altitude terrain scans. The original data contains 62,500 points as 16-bit unsigned integer height values, with a file size of 125,000 bytes.
In Figures 8.1 and 8.2, we display the local plane and quadratic fits, respectively, that ornament the final leaves of the octree $\mathcal{T}$ for various values of the Hausdorff tolerance $\eta^*$. One observes the progressive structure of the octree, in that smaller $\eta^*$ values correspond to more detail on the surface. In Table 8.1, we show various statistics for the corresponding octrees, such as tree depth, number of nodes, number of polynomial fits, boxes and clusters.

<table>
<thead>
<tr>
<th>Maple Mountain</th>
<th>$\eta^*$</th>
<th>Tree Depth</th>
<th># Nodes in Tree</th>
<th># Leaf Nodes</th>
<th># Basic Fits</th>
<th># BOX</th>
<th># CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLANE</td>
<td>0.01</td>
<td>6</td>
<td>817</td>
<td>715</td>
<td>274</td>
<td>46</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
<td>6</td>
<td>273</td>
<td>239</td>
<td>98</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>4</td>
<td>49</td>
<td>43</td>
<td>20</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>QUAD</td>
<td>0.01</td>
<td>6</td>
<td>321</td>
<td>281</td>
<td>101</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
<td>4</td>
<td>97</td>
<td>85</td>
<td>34</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>3</td>
<td>17</td>
<td>15</td>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.1: Tree statistics using $\text{CLUSTER}_\gamma = \text{sliding-partition method}$, with parameters $\gamma = 2^{-10}$, $N = 10$, $K = 10$, and $L = 20$.

In Table 8.2, we present the Hausdorff error between the points in the nodes of the octree $\mathcal{T}$ and the corresponding fit in these nodes (Hausdorff error on fits), and the Hausdorff error between the whole point cloud $D'$ and the resulting fragmented surface $S_{out}$ (Hausdorff error global).
Table 8.2: Maple Mountain. The Hausdorff errors and $\eta^*$ are given with respect to the unit cube.

Notice that in this example, the global Hausdorff error is sometimes smaller than the local Hausdorff error. The reason for this is that the point cloud to surface distance may improve globally by using the portions of the surface on neighboring cubes; similarly for the surface to point cloud distance.

Table 8.3 contains the compression results obtained from the application of our algorithm and the Rice encoder (with parameters $Smoothness = s$, $WindowSize = 16$, and $TaperLevel = TreeDepth$). The computed Hausdorff error is the error between the point cloud and the decoded planar fits. The compression ratios in the last two columns are the results reported by the Rice encoder without and with the additional bits needed for the encoding of the data structure switch, respectively.

<table>
<thead>
<tr>
<th>$\eta^*$</th>
<th>$s$</th>
<th>Encoded Size (bytes)</th>
<th>Hausdorff Error (Rice)</th>
<th>Comp. Ratio (Rice)</th>
<th>Comp. Ratio (Rice &amp; extra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>3</td>
<td>3922.25</td>
<td>0.01209860</td>
<td>52.92125318</td>
<td>35.35067873</td>
</tr>
<tr>
<td>0.02</td>
<td>3</td>
<td>1216.25</td>
<td>0.02352490</td>
<td>176.5536723</td>
<td>111.2099644</td>
</tr>
<tr>
<td>0.05</td>
<td>4</td>
<td>219</td>
<td>0.04855540</td>
<td>892.8571429</td>
<td>578.7637037</td>
</tr>
</tbody>
</table>

Table 8.3: Compression results, Maple Mountain.

The next figure shows a comparison between the surface $S_{out}$, representing the Maple Mountain and its decoded counterpart, after the Rice Encoder compression. Clearly, both Table 8.3 and Figure 8.3 demonstrate the high compression rates without sacrificing the quality of the surface.

Fig. 8.3: Maple Mountain, $\eta^* = 0.01$

**Test 2:** Our next data set is a portion of a building taken from the Eglin data (AFRL/MNG VEAA Data Set #1). It contains typical features present in real world urban terrain, such as windows, doors, corners, thin structures and bushes. Resolving these structures is a challenge for any reconstruction algorithm. Note, that this is a true three dimensional point cloud data, where each point coordinate is represented by a 32-bit floating point value. The denoised data contains 22,058 points and has a file size of 264,696 bytes.
η^* = 0.06

η^* = 0.04

η^* = 0.03

Fig. 8.4: Building reconstruction using **PLANE** for different values of η^*.

η^* = 0.06

η^* = 0.04

η^* = 0.03

Fig. 8.5: Building reconstruction using **QUAD** for different values of η^*.

The following tables describe the statistics and results for the Building data set.

<table>
<thead>
<tr>
<th>Building</th>
<th>η^*</th>
<th>Tree Depth</th>
<th># Nodes in Tree</th>
<th># Leaf Nodes</th>
<th># Basic Fits</th>
<th># BOX</th>
<th># CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLANE</td>
<td>0.03</td>
<td>6</td>
<td>521</td>
<td>456</td>
<td>149</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>5</td>
<td>273</td>
<td>239</td>
<td>103</td>
<td>29</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0.06</td>
<td>5</td>
<td>177</td>
<td>155</td>
<td>80</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>QUAD</td>
<td>0.03</td>
<td>6</td>
<td>353</td>
<td>309</td>
<td>108</td>
<td>17</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>5</td>
<td>201</td>
<td>176</td>
<td>82</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0.06</td>
<td>5</td>
<td>97</td>
<td>85</td>
<td>49</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 8.4: Tree statistics using **CLUSTER** = *sliding-partition method*, with parameters γ = 2^{-10}, N = 10, K = 10, and L = 20.

<table>
<thead>
<tr>
<th>η^*</th>
<th>PLANE</th>
<th>QUAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hausdorff Error on Fits</td>
<td>Hausdorff Error Global</td>
</tr>
<tr>
<td>0.03</td>
<td>0.02534980</td>
<td>0.03290050</td>
</tr>
<tr>
<td>0.04</td>
<td>0.03992240</td>
<td>0.03992240</td>
</tr>
<tr>
<td>0.06</td>
<td>0.05418520</td>
<td>0.05013430</td>
</tr>
</tbody>
</table>

Table 8.5: Building. The Hausdorff errors and η^* are given with respect to the unit cube.

Compression results for this example, as well as those that follow, is given in Figure 8.14 at the end of this section.
**Test 3:** Next, we process another portion of the Eglin data: a single light pole and a portion of a vehicle (in the upper right corner). This point cloud is a computational challenge since it contains a thin structure with no discernible interior. The denoised data consists of 5,901 points and has a file size of 70,812 bytes.

![Light pole reconstruction using PLANE for different values of $\eta^*$](image1)

![Light pole reconstruction using QUAD for different values of $\eta^*$](image2)

Fig. 8.6: Light pole reconstruction using **PLANE** for different values of $\eta^*$.

Fig. 8.7: Light pole reconstruction using **QUAD** for different values of $\eta^*$.

Tables 8.6 and 8.7 show the octree statistics and Hausdorff errors for the light pole data. Notice the good approximation of the original point cloud using a succinct representation.

<table>
<thead>
<tr>
<th>Light Pole</th>
<th>$\eta^*$</th>
<th>Tree Depth</th>
<th># Nodes in Tree</th>
<th># Leaf Nodes</th>
<th># Basic Fits</th>
<th># BOX</th>
<th># CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PLANE</strong></td>
<td>0.05</td>
<td>5</td>
<td>49</td>
<td>43</td>
<td>14</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0.06</td>
<td>4</td>
<td>33</td>
<td>29</td>
<td>13</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0.08</td>
<td>4</td>
<td>25</td>
<td>22</td>
<td>11</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td><strong>QUAD</strong></td>
<td>0.05</td>
<td>5</td>
<td>41</td>
<td>36</td>
<td>14</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0.06</td>
<td>4</td>
<td>33</td>
<td>29</td>
<td>13</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0.08</td>
<td>4</td>
<td>25</td>
<td>22</td>
<td>11</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.6: Tree statistics using **CLUSTER** = sliding-partition method, with parameters $\gamma = 2^{-10}$, $N = 10$, $K = 10$, and $L = 20$.

<table>
<thead>
<tr>
<th>$\eta^*$</th>
<th>PLANE</th>
<th>QUAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hausdorff Error on Fits</td>
<td>Hausdorff Error Global</td>
</tr>
<tr>
<td>0.05</td>
<td>0.04897710</td>
<td>0.04897710</td>
</tr>
<tr>
<td>0.06</td>
<td>0.05851370</td>
<td>0.05851370</td>
</tr>
<tr>
<td>0.08</td>
<td>0.07074950</td>
<td>0.07074950</td>
</tr>
</tbody>
</table>

Table 8.7: Light Pole. The Hausdorff errors and $\eta^*$ are given with respect to the unit cube.
Test 4: We consider now a more complicated data set, obtained using simulated LiDAR [7, 13]. It was created from a simulated low-altitude flight through a CAD representation of an actual MOUT site. We present three pairs of images showing the point cloud and reconstructed surface from different vantage points. We have selected these views to emphasize the various urban terrain structures that are present in the data. The denoised data contains 632,448 points and has a file size of 7,589,376 bytes.

Fig. 8.8: MOUT, View 1.

Fig. 8.9: MOUT, View 2.
Point Cloud

Reconstruction using PLANE, \( \eta^* = 0.002 \)

Fig. 8.10: MOUT, View 3.

Tables 8.8 and 8.9 show the octree statistics and Hausdorff errors for the MOUT data.

<table>
<thead>
<tr>
<th>MOUT</th>
<th>( \eta^* )</th>
<th>Tree Depth</th>
<th># Nodes in Subtrees</th>
<th># Leaf Nodes</th>
<th># Basic Fits</th>
<th># BOX</th>
<th># CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLANE</td>
<td>0.002</td>
<td>10</td>
<td>15718</td>
<td>1357</td>
<td>3056</td>
<td>1096</td>
<td>407</td>
</tr>
<tr>
<td></td>
<td>0.004</td>
<td>9</td>
<td>6782</td>
<td>5938</td>
<td>1744</td>
<td>686</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>0.006</td>
<td>9</td>
<td>4086</td>
<td>3579</td>
<td>1162</td>
<td>437</td>
<td>107</td>
</tr>
<tr>
<td>QUAD</td>
<td>0.002</td>
<td>10</td>
<td>14142</td>
<td>12378</td>
<td>2607</td>
<td>938</td>
<td>390</td>
</tr>
<tr>
<td></td>
<td>0.004</td>
<td>9</td>
<td>5478</td>
<td>4797</td>
<td>1209</td>
<td>599</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>0.006</td>
<td>9</td>
<td>3230</td>
<td>2830</td>
<td>837</td>
<td>385</td>
<td>116</td>
</tr>
</tbody>
</table>

Table 8.8: Tree statistics using CLUSTER = sliding-partition method, with parameters \( \gamma = 2^{-12} \), \( N = 10 \), \( K = 10 \), and \( L = 20 \).

<table>
<thead>
<tr>
<th></th>
<th>PLANE</th>
<th>QUAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \eta^* )</td>
<td>Hausdorff Error on Fits</td>
<td>Hausdorff Error Global</td>
</tr>
<tr>
<td>0.002</td>
<td>0.00199971</td>
<td>0.01965250</td>
</tr>
<tr>
<td>0.004</td>
<td>0.00399864</td>
<td>0.01402490</td>
</tr>
<tr>
<td>0.006</td>
<td>0.00599457</td>
<td>0.01574560</td>
</tr>
</tbody>
</table>

Table 8.9: MOUT. The Hausdorff errors and \( \eta^* \) are given with respect to the unit cube.

Let us note that, in this example and some of the ones that follow, the global Hausdorff distance is larger than the local distance. The reason for this is the appearance of flagged cubes which contain points but do not have a local surface fit. The points in these flagged cubes have to be included when computing the global distance but were of course not included in the computation of the local distance.

**Test 5:** We now consider a larger portion of the Eglin data set containing both thin structures and buildings. The purpose of this test is to demonstrate that no special tuning of the algorithm is needed to handle both thin structures and buildings. The denoised data contains 382,143 points and has a file size of 4,585,716 bytes.
Point Cloud

Reconstruction using PLANE, $\eta^* = 0.006$

Fig. 8.11: Eglin1.

Tables 8.10 and 8.11 show the octree statistics and Hausdorff errors for the Eglin1 data.

<table>
<thead>
<tr>
<th>Eglin1</th>
<th>$\eta^*$</th>
<th>Tree Depth</th>
<th># Nodes in Subtrees</th>
<th># Leaf Nodes</th>
<th># Basic Fits</th>
<th># BOX</th>
<th># CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLANE</td>
<td>0.006</td>
<td>8</td>
<td>2801</td>
<td>2456</td>
<td>678</td>
<td>199</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>0.008</td>
<td>8</td>
<td>1777</td>
<td>1560</td>
<td>483</td>
<td>139</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>0.010</td>
<td>8</td>
<td>1201</td>
<td>1056</td>
<td>328</td>
<td>114</td>
<td>47</td>
</tr>
<tr>
<td>QUAD</td>
<td>0.006</td>
<td>8</td>
<td>2273</td>
<td>1994</td>
<td>505</td>
<td>184</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>0.008</td>
<td>8</td>
<td>1457</td>
<td>1280</td>
<td>356</td>
<td>131</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>0.010</td>
<td>8</td>
<td>977</td>
<td>860</td>
<td>253</td>
<td>101</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 8.10: Tree statistics using CLUSTER = sliding-partition method, with parameters $\gamma = 2^{-12}$, $N = 10$, $K = 10$, and $L = 20$.

<table>
<thead>
<tr>
<th>$\eta^*$</th>
<th>PLANE</th>
<th>QUAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.006</td>
<td>0.005999877</td>
<td>0.01640190</td>
</tr>
<tr>
<td>0.008</td>
<td>0.007999094</td>
<td>0.00814113</td>
</tr>
<tr>
<td>0.010</td>
<td>0.009996485</td>
<td>0.010655530</td>
</tr>
</tbody>
</table>

Table 8.11: Eglin1. The Hausdorff errors and $\eta^*$ are given with respect to the unit cube.

**Test 6:** We now consider second portion of the Eglin data set consisting of several portions of multiple buildings. The denoised data contains 192,021 points and has a file size of 2,304,252 bytes.
Tables 8.12 and 8.13 show the octree statistics and Hausdorff errors for the Eglin2 data.

Table 8.12: Tree statistics using $\text{CLUSTER} = \text{sliding-partition method}$, with parameters $\gamma = 2^{-12}$, $N = 10$, $K = 10$, and $L = 20$.

![Point Cloud](image1)

Reconstruction using $\text{PLANE}$, $\eta^* = 0.006$

![Reconstruction](image2)

Fig. 8.12: Eglin2.

Table 8.13: Eglin2. The Hausdorff errors and $\eta^*$ are given with respect to the unit cube.

Next, we show the compression ratios for the smallest values of $\eta^*$ for all the above data. The values are produced by the same method used to create Table 8.3 (with parameters $\text{Smoothness} = s$, $\text{WindowSize} = 16$, and $\text{TaperLevel} = \text{TreeDepth}$). One can obtain higher compression ratios if larger values of $\eta^*$ are used but this would result in less accurate surfaces. The reported Hausdorff errors in Table 8.14 are the Hausdorff errors between the corresponding point clouds $D$ and the decoded surfaces that represent them.
<table>
<thead>
<tr>
<th>Data</th>
<th>$\eta^*$</th>
<th>$s$</th>
<th>Encoded Size (bytes)</th>
<th>Hausdorff Error</th>
<th>Comp. Ratio (Rice)</th>
<th>Comp. Ratio (Rice &amp; extra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maple</td>
<td>0.01</td>
<td>3</td>
<td>3922.25</td>
<td>0.01209860</td>
<td>52.92125318</td>
<td>35.35067873</td>
</tr>
<tr>
<td>Building</td>
<td>0.03</td>
<td>4</td>
<td>2165.375</td>
<td>0.03149410</td>
<td>162.1911765</td>
<td>133.9554656</td>
</tr>
<tr>
<td>Light Pole</td>
<td>0.05</td>
<td>5</td>
<td>317.5</td>
<td>0.05292810</td>
<td>345.4243902</td>
<td>219.2321981</td>
</tr>
<tr>
<td>MOUT</td>
<td>0.002</td>
<td>5</td>
<td>85129.625</td>
<td>0.01996640</td>
<td>137.8583158</td>
<td>101.2456777</td>
</tr>
<tr>
<td>Eglin1</td>
<td>0.006</td>
<td>6</td>
<td>16675.5</td>
<td>0.01620250</td>
<td>420.3222731</td>
<td>303.0075237</td>
</tr>
<tr>
<td>Eglin2</td>
<td>0.006</td>
<td>3</td>
<td>30037</td>
<td>0.04627640</td>
<td>146.0791175</td>
<td>96.99802543</td>
</tr>
</tbody>
</table>

Table 8.14: Compression results.

Remark. Note that the local Hausdorff error on a single cube and the global Hausdorff error may be quite different since the first one heavily depends on the structure of the octree. When a cube has been chosen for subdivision, the choice of how to create the child cubes is automatic and does not take completely into account the structure of the point cloud within that cube. This could lead to the introduction of a false noise in the child cubes. A similar problem occurs when the octree creates nodes that contain too few points for a fit to be computed. The latter often happens when the sampling density of the point cloud vary substantially from region to region. We illustrate this phenomenon in Figure 8.13, produced using the MOUT data with $\eta^* = 0.002$. The red planes in this figure correspond to fits for cubes that do not meet the error threshold $\eta^*$ and if subdivided further would have produced all children each with fewer than $K$ points.

Fig. 8.13: Planes colored according to their Hausdorff error $\eta^*$: green $\eta^* <= 0.001$, yellow $\eta^* <= 0.0015$, brown $\eta^* <= 0.002$ and red $\eta^* > 0.002$.

9. Implicit Surface Representations. The surface $S_{\text{out}}$, generated by the output of MAIN, and representing the point cloud $D$ is a piecewise (discontinuous) polynomial surface. Direct display of $S_{\text{out}}$ is possible, but not appealing (see the figures in §8) due to the large number of discontinuities and missing regions. In this section, we present alternative methods that take $S_{\text{out}}$ and create a more appealing mathematical surface $S_{\text{math}}$. Our main goal is to retain the fidelity of the representation but to have $S_{\text{math}}$ be a smoother connected surface. We shall discuss our method for generating an $S_{\text{math}}$ that can be displayed using standard graphics hardware.

The surface $S_{\text{math}}$ that we generate is given implicitly as the solution set of $F = h$, where $F$ is a function defined on $[0,1]^3$ and $h$ is a real number. There are many possible choices for such a function $F$. The typical implicit representations of surfaces use signed or unsigned distance functions. The main drawback of these methods is the fact that the associated parameter $\varepsilon$ is too much data dependent, and the resulting surface either lacks detail (if $\varepsilon$ is too big) or has artificial holes (if $\varepsilon$ is too small). For this reason, we shall use an alternative method based on multiscale (wavelet) decompositions, originally developed in [23] for laser scan data. This method views the surface as boundary of a three dimensional body $M$. Notice that the surface is then the level set of the indicator function of $M$, $\chi_M$. The method takes the wavelet decomposition of $\chi_M$.
and a truncation $F$ of this decomposition. Then $S_{\text{math}}$ is a level set of $F$ (typically we use $F = 0.5$ in our algorithms).

The wavelet based methods require knowledge of the surface orientation (the identification of the inside and outside of the surface) in order to define $M$. This orientation is generally not available to us. Determining the surface orientation is a well studied topic in geometric modeling and is typically derived from the point cloud data directly, see [15, 16, 20]. The existing surface orientation methods are successful when the sensor produces a point cloud that either includes normals, the point cloud has such high resolution that normals can be accurately numerically computed, or the surface to be reconstructed is very smooth. These ingredients are usually not available for terrain point clouds. Therefore, we shall develop a surface orientation algorithm that is based only on the concise tree representation $T$ and without any additional information. We discuss this orientation algorithm in the following subsection and then discuss how we find $F$ in subsequent sections.

### 9.1. Identifying the Orientation of the Terrain Surface.

In this section, we propose the subroutine ORIENT that orients $S_{\text{out}}$ without the knowledge of the normals associated to the point cloud data $D$, or any sensor-related information such as position or orientation. The algorithm uses a progressive coarse to fine prediction and voting scheme that utilizes the local polynomial fits from each level of the octree to determine a globally consistent orientation. The algorithm ORIENT has nothing to do with the encoding algorithm of the preceding section. The encoding proceeds without it. We employ ORIENT as a post processing algorithm only when we want to visualize the output of MAIN. We shall only describe ORIENT in the case when the polynomial fits are planes.

We first preprocess the octree $T$, produced from MAIN, using the subroutine CUT. This subroutine replaces with flagged nodes the terminal leaves $Q$ of $T$ that contain planes $S_{DQ}$ obtained from BOX, or CLUSTER, and outputs the resulting tree $T_e$ along with the list $\mathcal{N}$ of replaced nodes.

The subroutine ORIENT takes the output $T_e$ from CUT and the integer $\kappa$ from INIT, and creates an octree $\tilde{T}$, that is ORIENT($T_e$, $\kappa$) = $\tilde{T}$, which has the same structure as $T_e$, and in addition has the values $-1$, 1 or 0 assigned to the vertices of each of its cubes, as well as each of their children; $-1$ meaning inside the surface, 1 outside the surface and 0 on the surface. Notice that the children of a cube $Q$ in $T_e$ are not in $T_e$ when $Q$ is a terminal leaf, but we will still define a value to the vertices of all these children. Let us denote by $V$ the set of all vertices of all cubes $Q$ such that either $Q$ is in $T_e$ or $Q$ is a child of a cube in $T_e$.

Next, we describe how to assign values to the vertices in $V$ as we march through the levels of the octree from coarse to fine. Given an integer $m \geq \kappa$, we denote by $T_m$ the truncation of $T_e$ to this level. Given any cube $Q$ we denote by $V_Q$ the set of all vertices in $V \cap Q$ that are either vertices of $Q$ or vertices of one of its children. We say that a vertex has level $m$ if it is a vertex of a cube of dyadic level $m$ (i.e. side length $2^{-m}$) but not of a coarser cube. We define $V_m$ to be the set of all vertices in $V$ of level $m$ and $W_m$ the set of all vertices in $V$ of level $\leq m$. To start our labeling, we take advantage of the fact that $D' \subset [0,1]^2 \times [0,2^\kappa]$ and assign values $s_m(v) \in \{ \pm 1, 0 \}$ to all $v \in V$ as follows. We assign $-1$ to all the vertices $v = (x, y, 0) \in V$ which are of level $\leq \kappa$ and assign $+1$ to all other vertices $v \in V$ of level $\leq \kappa$. All other vertices $v \in V$ are initially assigned the value $s_m(v) = 0$.

Let us now assume that we have assigned values $s_m(v)$ to all vertices $v \in V$ and explain how we determine the updated values $s_{m+1}(v)$. Let $Q$ be any leaf cube from $T_m$. Notice that this cube can be of any level $\leq m$. If $Q$ is flagged and $Q$ has level $q$, then for any vertex $v \in W_{q+1} \cap Q$, we define $s_{m+1}(v, Q) := \text{sign} \left( \sum_{w \in V_Q} s_m(w) \right)$, where $\text{sign}(a)$ is the function that gives the sign of the number $a$ with $\text{sign}(0) = 0$. Notice that in this case, all the vertices in $V_Q$ are assigned the same label. If $Q$ is not flagged, then it has a fit $\hat{S}_{DQ}$ that separates the vertices $v \in V_Q$ into two groups, $V_+$ and $V_-$, corresponding to the intersection of $V_Q$ with the closed half spaces generated by the plane $S_{DQ}$. If $v \in V_+$, and does not lie on $\hat{S}_{DQ}$, then we define $s_{m+1}(v, Q) = \text{sign} \left( \sum_{w \in V_+} s_m(w) \right)$. We make the corresponding assignment when $v \in V_-$ and does not lie on $\hat{S}_{DQ}$. If $v \in \hat{S}_{DQ}$, we assign the value $s_{m+1}(v, Q) = 0$.

Finally, we shall assign the value $s_{m+1}(v)$. Given any vertex $v \in V_Q$ of a cube $Q$ of dyadic level $\leq m$, we
define \( s_{m+1}(v) := \text{sign} \left( \sum_{v \in V_R, R \in T_m} s_{m+1}(v, R) \right) \). For any vertex \( v \in V \) whose value has not been updated, we define \( s_{m+1}(v) := s_m(v) \). The entire process halts when we have \( m = \ell \) the finest allowable level. In this case, we define the final values \( s(v) := s_{\ell+1}(v) \) for all \( v \in V \).

The subroutine \textsc{ORIENT}, performing the above mentioned procedure, is as follows.

\begin{algorithm}
\begin{algorithmic}
\Function{ORIENT}{}
\ForEach{vertex \( v = (x, y, z) \in V \)}
\If{Level of \( v \) \leq \( \kappa \)}
\If{\( z > 0 \)}
\State \( s_{\kappa}(v) = 1 \).
\Else
\State \( s_{\kappa}(v) = -1 \).
\EndIf
\Else
\State \( s_{\kappa}(v) = 0 \).
\EndIf
\EndIf
\EndFor
\For{\( m = \kappa, \ldots, \ell \)}
\For{each vertex \( v \in V \)}
\State \( s_{m+1}(v) = s_m(v) \).
\EndFor
\For{each leaf cube \( Q \) of \( T_m \)}
\If{\( Q \) is flagged}
\For{each vertex \( v \in V_Q \)}
\State \( s_{m+1}(v, Q) = \text{sign} \left( \sum_{w \in V_Q} s_m(w) \right) \).
\EndFor
\Else
Find \( V_+ \) and \( V_- \).
\For{each vertex \( v \in V_Q \)}
\If{\( v \in \hat{S}_D_Q \)}
\State \( s_{m+1}(v, Q) = 0 \).
\ElseIf{\( v \in V_+ \)}
\State \( s_{m+1}(v, Q) = \text{sign} \left( \sum_{w \in V_+} s_m(w) \right) \).
\Else
\State \( s_{m+1}(v, Q) = \text{sign} \left( \sum_{w \in V_-} s_m(w) \right) \).
\EndIf
\EndIf
\EndFor
\EndIf
\EndFor
\For{each vertex \( v \in W_{m+1} \)}
\State \( s_{m+1}(v) = \text{sign} \left( \sum_{v \in V_R, R \in T_m} s_{m+1}(v, R) \right) \).
\EndFor
\EndFor
\For{each \( v \in V \)}
\State \( s(v) = s_{\ell+1}(v) \).
\EndFor
\EndFunction
\end{algorithm}
\end{algorithm}

22
9.2. The solid region $M$. In this section, we shall describe how we generate a solid $M$ from the output of MAIN. In order to define a solid whose boundary is the surface which represents the data. Given the tree $T$ which is part of the output of MAIN, we apply ORIENT and receive a labeling of the vertices in $V$. If $Q$ is a terminal cube of $T$, we now describe how we define the portion $M_Q = M \cap Q$ of $M$ on $Q$. The cube $Q$ is one of three types. If $Q$ is not flagged in $T$, then $Q$ has a linear fit $\tilde{S}_{D_Q}$ on $Q$ which separates $Q$ into two regions $Q', Q''$. We examine the $s(v), v \in Q' \cap V$, and determine that $Q'$ is in $M$ if $\text{sign} \left( \sum_{v \in Q' \cap V} s(v) \right) \leq 0$; otherwise $Q'$ is outside $M$. We do the same test for $Q''$. We then define $M_Q$ as the union of the regions that are inside. Typically, only one of $Q', Q''$ is inside $M$ but it could happen that both are labeled inside or both are labeled outside. The second possible case is that $Q$ is flagged in $T$ because it had too few data points. In this case, if $\text{sign} \left( \sum_{v \in Q' \cap V} s(v) \right) \leq 0$, we define $M_Q := Q$; otherwise $M_Q := \emptyset$.

The third and last possibility is that $Q$ is flagged in $T$ because it has bounding boxes (more precisely, $Q$ is an element of the output $N$ from CUT). In this case, we define $M_{B_Q}$ as the union of the $\tilde{S}_{B_Q}$ that live in $Q$ (there will be one or two of these). We then define $M_Q$ as the union of the $M_{B_Q}$ associated to $Q$. This method for defining $M_Q$ in the case of bounding boxes tends to preserve thin (lower dimensional) structures such as guy-wires, cables, poles, fences etc.

Note that since the point clouds we work with may be incomplete, contain holes or occlusions and possibly be of lower dimension, finding a globally consistent orientation is challenging and sometimes impossible. However, our algorithm, when tested on various point cloud data, demonstrates robustness and typically produces reasonable solutions to quite complicated data.

9.3. Wavelet Display. In this section, we discuss the wavelet method for creating $S_{\text{math}}$. First, we introduce some basic facts about wavelets and refer the reader to [12] for details on this topic. We use standard construction of three dimensional wavelet bases and the notation $\psi_0 = \varphi$ for the scaling function $\varphi$, $\psi^1 = \varphi$ for the corresponding wavelet, $E'$ for the set of vertices of the cube $[0, 1]^3$, and $E$ for the set of vertices excluding the origin (i.e. $E = E' \setminus \{(0, 0, 0)\}$). For each $e = (e_1, e_2, e_3) \in E'$, $j \in \mathbb{N}$, and $k = (k_1, k_2, k_3)$, we define the $(L_2$ normalized) wavelet,

$$
\psi_{j,k}(x) = 2^{3j/2} \psi_{1}(2^j x - k_1) \psi_{2}(2^j y - k_2) \psi_{3}(2^j z - k_3).
$$

It is known that every locally integrable function $f$ has the expansion

$$
f(x) = \sum_{k \in \mathbb{Z}^3} \sum_{j \in \mathbb{N}} \sum_{k \in \mathbb{Z}^3} \sum_{e \in E} c_{j,k}^e \psi_{j,k}(x),
$$

(9.1)

with coefficients

$$
c_{j,k}^e = \int_{\mathbb{R}^3} f(x) \psi_{j,k}(x) \, dx.
$$

In numerical implementation, one has to take a finite portion of the wavelet sum appearing in (9.1) by retaining only the terms corresponding to $0 \leq j \leq d$. Here $d$ is a user defined parameter, but in our numerical experiments we have always chosen $d$ to be the same as the finest level of the octree $T$. We choose a smoothing parameter $t$, which we take as $2^{-d}$ in all of our experiments, and $M(t)$ as the set of all points whose signed distance from $M$ is $t$. We then construct the wavelet expansion of the function $f = \chi_{M(t)}$ and $F_d$ which is the partial sum of (9.1) obtained by selecting only those summands for which $j \leq d$. We output and display the surface $S_d$, which is the level set $F_d = 0.5$.

Several computational issues arise in implementing the above display method. The first is to find a numerical approximation to $M(t)$. For this, we take a fine voxelization and define a piece-wise constant distance function whose value at each voxel is the signed distance of the center of the voxel to $M$. We then
subtract $t$ from this distance function and use marching cubes to find the zero set. Next, we need to compute the wavelet coefficients

$$2^{3j/2} \int_{M(t_d)} \psi^{e_1}(2^j x - k_1) \psi^{e_2}(2^j y - k_2) \psi^{e_3}(2^j z - k_3) \, dx dy dz. \quad (9.2)$$

of the series (9.1). In this paper, we use the idea from [23], where the divergence theorem is used to transfer the volume integral (9.2) to a surface integral. Another computational issue is how to find the level set $F_d = 0.5$. For this we use the method described [24]. In this way we obtain our numerical display of the level set $S_d$ as shown in the figures that follow.

Note that since the processed point cloud $D$ comes from terrain that may contain holes, occlusions, or simply be incomplete, we remove the portions of $S_d$ that are unreliable and have been filled in by our algorithm in order to create a globally consistent surface. To perform this operation, we introduce the subroutine TRIM that simply removes portions from $S_d$ that are further than $2\eta^*$ from all fits $\hat{S}_D$ in the leaf nodes of $T$. The images that follow show the resulting surfaces for the data sets in §8 when using the Haar wavelets for $\psi$.

![Maple Mountain](image1)
![Building](image2)
![Light Pole](image3)

Fig. 9.1: Wavelet processed decoded surfaces.

![View 1](image4)
![View 2](image5)
![View 3](image6)

Fig. 9.2: Wavelet processed decoded surfaces from the MOUT data.
Fig. 9.3: Wavelet processed decoded surfaces from the Eglin data.

REFERENCES