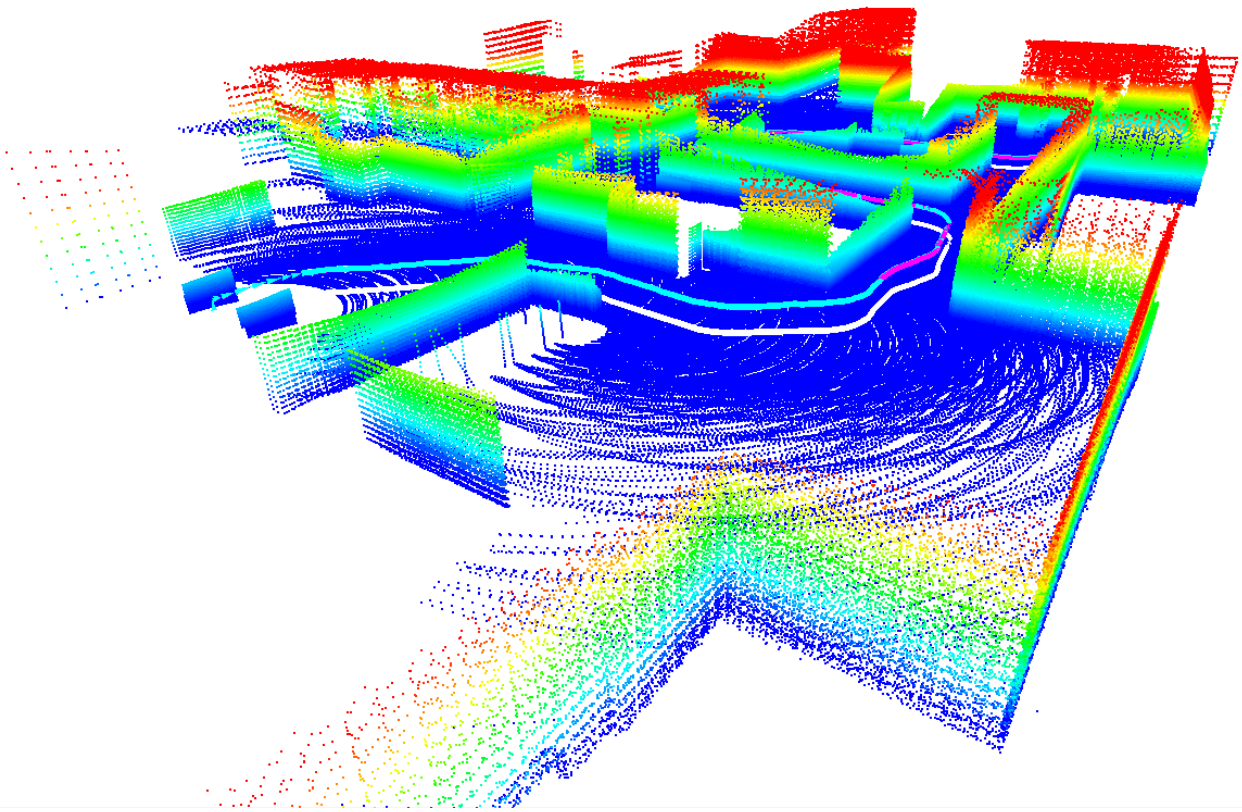
Real-time 3-D Weighted Scan Matching with Octrees

Mustafa Karaalioğlu



Bachelor Informatica

Universiteit van Amsterdam

Real-time 3-D Weighted Scan Matching with Octrees

Mustafa Karaalioğlu

August 24, 2015

Supervisor: Arnoud Visser (UvA)

Signed: Arnoud Visser (UvA)

Abstract

Successful urban search and rescue missions rely on rescue robots accurately and reliably mapping unknown environments while at the same time accurately tracking their pose. To that end, realistic measurement models with inhomogeneous and anisotropic noise can more accurately estimate rigid transformations, but such a model only exists for the 2-D case. The 3-D case is derived here and is combined with an algorithm that takes these noise characteristics into account when estimating rigid transformation. An efficiently encoded octree is introduced boost the performance of the algorithm. The resulting algorithm runs in real-time and is slightly more accurate with potential for larger gains.

Contents

CHAPTER 1

Introduction

More than a a decade ago, the first Urban Search and Rescue (USAR) mission where robots were used for technical search tasks, took place after the attack on the World Trade Center (WTC) Towers [5]. They were tasked with examining areas too small or too dangerous to be examined by human rescue workers, searching for victims and detecting hazards. Robots are expendable, a potential life can be spared for any task a robot can perform instead of a human rescue worker. Unfortunately, robots are not perfect and problems do arise. One of the problems during the USAR at the WTC was a *lack of state of the world* [5], where the map of the environment or the position of the robot were unclear. This left operators and rescue workers in difficult to handle situations, where they would lose the robot.

Dealing with unknown environments in USAR missions using robots is a key challenge. Accurately and reliably measuring the position and orientation of a robot is fundamental in building a map from an unknown environment. Robots are generally equipped with inertial navigation sensors (INS) to estimate relative motion, but as these estimates are not completely accurate, the estimated position of a robot accumulates a significant amount of error over time. This prevents building an accurate map of the environment, which is key in planning and navigation. The error in the estimated position can be reduced by using observations acquired by laser range scan data. Laser range scanners provide accurate range data, often with millimeter precision, which can be transformed to a set of 3-D points. By corresponding sets of 3-D points of a past observation with a current observation, a rigid transformation can be estimated, which can be used to correct the estimated position. Not only does it need to be accurate, it needs to perform in real-time. A search and rescue robot does not have the luxury of time as human life is fragile and every second matters.

Finding the rigid transformation that minimizes the sum of the squared distances between corresponding points is known as the *Procrustes problem* and, as pointed out by Dorst in [6], has long been solved in various fields, ranging from biomechanics [8], satellite control [23] and statistical shape analysis [24] to vision and robotics [2], [9], [10], [26]. The solution provides a rigid transformation with minimal variance if the points have identical and isotropic Gaussian noise. However, as noted in [21] and [20], this model is not realistic as it does not account for the physical phenomena that affect the accuracy of the range measurements, i.e. inhomogeneous and anisotropic sources of noise. The measurement model presented in [21] does take this into account, but is only derived for the 2-D case. The same goes for the rigid transformation estimation, however, this is covered by [20] and [16]. The 3-D measurement model will be derived here and combined with the algorithm in [16] to provide accurate rigid transformation estimates. In order to make this perform in real-time, an efficient octree implementation will be introduced to significantly boost the performance of the otherwise not real-time friendly point correspondence process.

CHAPTER 2

Measurement Model

In order to accurately estimate the rigid transformation of a robot, a more realistic model of the noise affecting the laser range data points and their correspondence between successive poses is required. Let $\{\bar{\mathbf{u}}_i\}$ and $\{\bar{\mathbf{v}}_i\}$ denote the range scan data point sets acquired in subsequent poses. As these data sets are affected by noise, the measurements can be decomposed into the following terms:

$$\bar{\mathbf{u}}_i = \mathbf{u}_i + \delta\mathbf{u}_i + \mathbf{b}_i^u \tag{2.1}$$

$$\bar{\mathbf{v}}_i = \mathbf{v}_i + \delta\mathbf{v}_i + \mathbf{b}_i^v \tag{2.2}$$

where $\mathbf{u}_i$ and $\mathbf{v}_i$ are the "true" Cartesian scan point locations, $\delta\mathbf{u}$ and $\delta\mathbf{v}$ are noise or uncertainty in the measurement process, while $\mathbf{b}_i^u$ and $\mathbf{b}_i^v$ denote the possible range measurement "bias" [21]. In this paper the bias error will be ignored as for many data sets the bias error is often unknown for the used sensor [27] and as demonstrated in [21] it has a negligible impact on the actual position estimate, reducing Eq. 2.1 and Eq. 2.2 to:

$$\bar{\mathbf{u}}_i = \mathbf{u}_i + \delta\mathbf{u}_i \tag{2.3}$$

$$\bar{\mathbf{v}}_i = \mathbf{v}_i + \delta\mathbf{v}_i \tag{2.4}$$

Let $\{(\bar{\mathbf{x}}, \bar{\mathbf{y}})_i\}$ be the set of closest corresponding points between subsequent poses, which are not necessarily the same physical point. The conventional method of determining the rigid transformation between two sets of points is minimizing the following equation:

$$\sum_{i=1}^{n} \|\bar{\mathbf{x}}_i - \mathbf{R}\bar{\mathbf{y}}_i - \mathbf{p}\|^2 \tag{2.5}$$

where $\mathbf{R}$ is a rotation matrix and $\mathbf{p}$ is a translation vector. The error between two corresponding points is:

$$\epsilon_i = \bar{\mathbf{x}}_i - \mathbf{R}\bar{\mathbf{y}}_i - \mathbf{p} \tag{2.6}$$

Substituting Eq. 2.3 and Eq. 2.4 into Eq. 2.6 results in:

$$\epsilon_i = (\mathbf{x}_i - \mathbf{R}\mathbf{y}_i - \mathbf{p}) + (\delta\mathbf{x}_i - \mathbf{R}\delta\mathbf{y}_i) \tag{2.7}$$

If $\mathbf{x}_i$ and $\mathbf{y}_i$ correspond to the exact same physical points, then $\mathbf{x}_i - \mathbf{R}\mathbf{y}_i - \mathbf{p} = \mathbf{0}$. However, $\mathbf{x}_i$ and $\mathbf{y}_i$ generally do not correspond to the exact same physical point as illustrated in Fig. 2.1. Thus the correspondence error $\mathbf{c}_i$ is denoted by

$$\mathbf{c}_i = \mathbf{x}_i - \mathbf{R}\mathbf{y}_i - \mathbf{p} \tag{2.8}$$

The covariance matrices of the measurement noise and the correspondence error will be derived in Section 2.1 and 2.2.

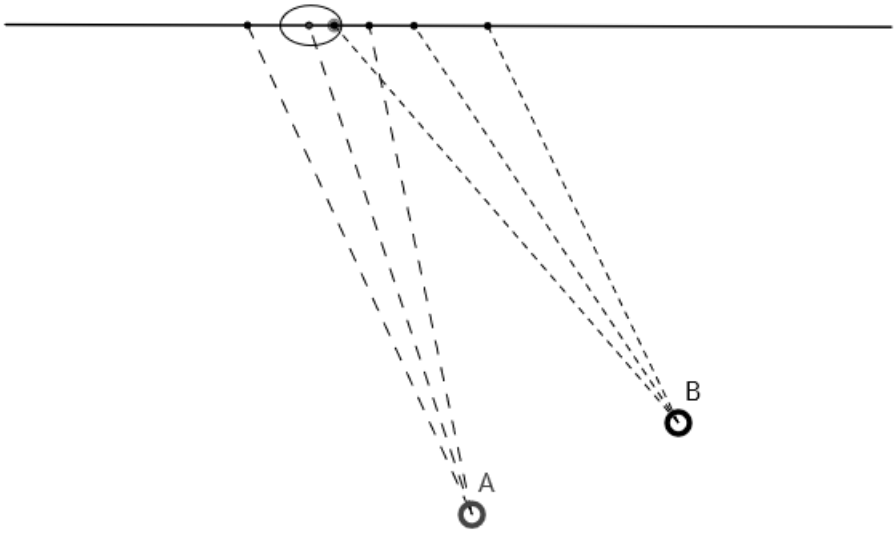


Figure 2.1: Two corresponding points measured in two different poses A and B

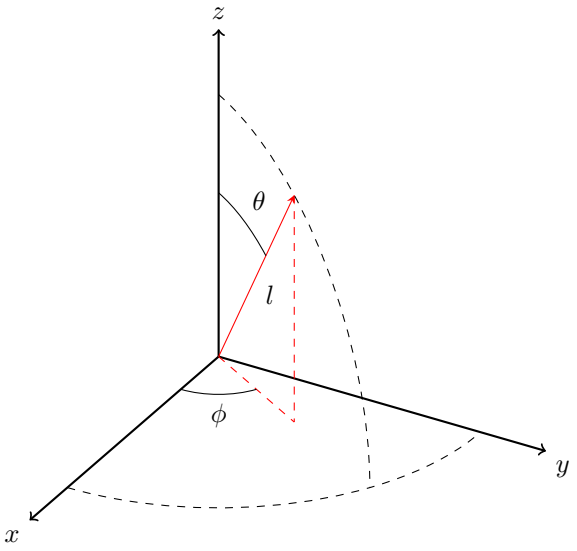


Figure 2.2: Spherical coordinate (l, θ, ϕ)

2.1 Measurement Noise

Let $\{(l,\theta,\phi)_i\}$ be a set of measurements described in spherical coordinates where l is the range, θ is the inclination and ϕ is the azimuth (see Fig. 2.2). Converting spherical coordinates to Cartesian coordinates gives:

$$\bar{\mathbf{s}}_i = l_i \begin{bmatrix} \sin(\theta_i)\cos(\phi_i) \\ \sin(\theta_i)\sin(\phi_i) \\ \cos(\theta_i) \end{bmatrix} \tag{2.9}$$

The measurements of l_i, θ_i and ϕ_i will be imperfect due to measurement noise. The range measurement l can be decomposed into the following terms:

$$l_i = L_i + \epsilon_{l_i} \tag{2.10}$$

where L_i is the "true" range and ϵ_{l_i} is an additive noise term. The noise ϵ_{l_i} is assumed to be a zero-mean Gaussian random variable with variance $\sigma_{l_i}^2$ [21]. The angle measurements θ_i and ϕ_i can respectively be decomposed into the following terms:

$$\theta_i = \Theta_i + \epsilon_{\theta_i} \tag{2.11}$$

and

$$\phi_i = \Phi_i + \epsilon_{\phi_i} \tag{2.12}$$

where Θ_i and Φ_i are the "true" angles and ϵ_{θ_i}, and ϵ_{ϕ_i} are again zero-mean Gaussian random variables with variances $\sigma_{\theta_i}^2$ and $\sigma_{\phi_i}^2$. The "true" point then becomes:

$$\mathbf{s}_i = (l_i - \epsilon_{l_i}) \begin{bmatrix} \sin(\theta_i - \epsilon_{\theta_i})\cos(\phi_i - \epsilon_{\phi_i}) \\ \sin(\theta_i - \epsilon_{\theta_i})\sin(\phi_i - \epsilon_{\phi_i}) \\ \cos(\theta_i - \epsilon_{\theta_i}) \end{bmatrix} \tag{2.13}$$

Substituting $\bar{\mathbf{s}}_i$ and $\mathbf{s}_i$ into Eq. 2.3 or Eq. 2.4 and making the reasonable assumption that $\epsilon_{l_i} \ll 1$, $\epsilon_{\theta_i} \ll 1$ and $\epsilon_{\phi_i} \ll 1$, the measurement noise $\delta\mathbf{s}_i$ becomes:

$$\delta\mathbf{s}_i = l_i \begin{bmatrix} \epsilon_{\theta_i}\cos(\phi_i)\cos(\theta_i) - \epsilon_{\phi_i}\sin(\phi_i)\sin(\theta_i) \\ \epsilon_{\phi_i}\cos(\phi_i)\sin(\theta_i) + \epsilon_{\theta_i}\cos(\phi_i)\cos(\theta_i) \\ -\epsilon_{\theta_i}\sin(\theta_i) \end{bmatrix} + \epsilon_{l_i} \begin{bmatrix} \cos(\phi_i)\sin(\theta_i) \\ \sin(\phi_i)\sin(\theta_i) \\ \cos(\theta_i) \end{bmatrix} \tag{2.14}$$

where the following approximations are used:

$$\begin{aligned} \sin(\epsilon_{\theta_i}) &\simeq \epsilon_{\theta_i} \\ \sin(\epsilon_{\phi_i}) &\simeq \epsilon_{\phi_i} \\ \cos(\epsilon_{\theta_i}) \simeq \cos(\epsilon_{\phi_i}) &\simeq 1 \\ \epsilon_{l_i}\epsilon_{\theta_i} &\simeq 0 \\ \epsilon_{l_i}\epsilon_{\phi_i} &\simeq 0 \\ \epsilon_{\theta_i}\epsilon_{\phi_i} &\simeq 0 \end{aligned} \tag{2.15}$$

The noise covariance matrix is:

$$\mathbf{P}_i^N = E\left[\delta\mathbf{s}_i(\delta\mathbf{s}_i)^\top\right] \tag{2.16}$$

Assuming that l_i, ϵ_{θ_i} and ϵ_{ϕ_i} are independent, then $\mathbf{P}_i^N = \begin{bmatrix}\mathbf{C}_1 & \mathbf{C}_2 & \mathbf{C}_3\end{bmatrix}$ where

$$\mathbf{C}_1 = \begin{bmatrix} \sigma_l^2\cos(\phi)^2\sin(\theta)^2 + \sigma_\theta^2 l^2\cos(\phi)^2\cos(\theta)^2 + \sigma_\phi^2 l^2\sin(\phi)^2\sin(\theta)^2 + \sigma_\phi^2\sigma_\theta^2 l^2\cos(\theta)^2\sin(\phi)^2 \\ (\sigma_l^2 - \sigma_\phi^2 l^2)\cos(\phi)\sin(\phi)\sin(\theta)^2 + (\sigma_\theta^2 l^2 - \sigma_\phi^2\sigma_\theta^2 l^2)\cos(\phi)\cos(\theta)^2\sin(\phi) \\ (\sigma_l^2 - \sigma_\theta^2 l^2)\cos(\phi)\cos(\theta)\sin(\theta) \end{bmatrix}$$

$$\mathbf{C}_2 = \begin{bmatrix} (\sigma_l^2 - \sigma_\phi^2 l^2)\cos(\phi)\sin(\phi)\sin(\theta)^2 + (\sigma_\theta^2 l^2 - \sigma_\phi^2\sigma_\theta^2 l^2)\cos(\phi)\cos(\theta)^2\sin(\phi) \\ \sigma_l^2\sin(\phi)^2\sin(\theta)^2 + \sigma_\phi^2 l^2\cos(\phi)^2\sin(\theta)^2 + \sigma_\theta^2 l^2\cos(\theta)^2\sin(\phi)^2 + \sigma_\phi^2\sigma_\theta^2 l^2\cos(\phi)^2\cos(\theta)^2 \\ (\sigma_l^2 - \sigma_\theta^2 l^2)\cos(\theta)\sin(\phi)\sin(\theta) \end{bmatrix}$$

$$\mathbf{C}_3 = \begin{bmatrix} (\sigma_l^2 - \sigma_\theta^2 l^2)\cos(\phi)\cos(\theta)\sin(\theta) \\ (\sigma_l^2 - \sigma_\theta^2 l^2)\cos(\theta)\sin(\phi)\sin(\theta) \\ \sigma_l^2\cos(\theta)^2 + \sigma_\theta^2 l^2\sin(\theta)^2 \end{bmatrix}$$

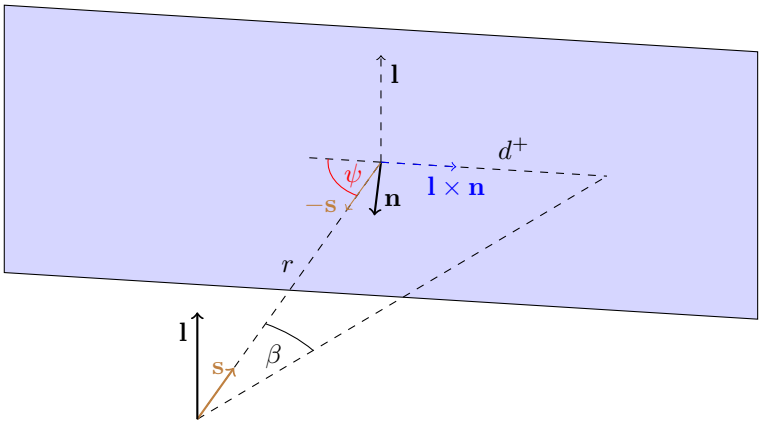


Figure 2.3: Correspondence error

2.2 Correspondence Error

A complete description and derivation of a second order probabilistic approximation to this error is given for the 2-D case in [21]. This section describes the error and derives the 3-D case. Consider the sets of range scan data points $\{\bar{\mathbf{x}}_i\}$ and $\{\bar{\mathbf{y}}_i\}$ of two subsequent poses, x and y, lying on the same plane. The implicit ordering of the sampled points means that $\bar{\mathbf{x}}_{i+1}$ is adjacent to $\bar{\mathbf{x}}_i$, with an angle difference of β. Let the distances between two adjacent points be denoted by:

$$\begin{aligned} d_i^+ &= \|\bar{\mathbf{x}}_{i+1} - \bar{\mathbf{x}}_i\| \\ d_i^- &= \|\bar{\mathbf{x}}_{i-1} - \bar{\mathbf{x}}_i\| \end{aligned} \tag{2.17}$$

Geometric analysis of Fig. 2.3 shows that d_i^+ can be expressed as

$$\begin{aligned} d_i^+ &= \frac{r\sin(\beta)}{\sin(\psi - \beta)} \\ &= \frac{r\sin(\beta)}{\sin(\pi - \cos^{-1}((\mathbf{l}\times\mathbf{n})\cdot\mathbf{s}) - \beta)} \\ &= \frac{r\sin(\beta)}{\sin(\cos^{-1}(\alpha) + \beta)} \end{aligned} \tag{2.18}$$

where $\mathbf{s}$ denotes the normalized direction of a sensor beam striking a surface with normal $\mathbf{n}$, $\mathbf{l}$ denotes the normal of the scanning plane and $\alpha = (\mathbf{l}\times\mathbf{n})\cdot\mathbf{s}$. Similarly, d_i^- can be expressed as

$$\begin{aligned} d_i^- &= \frac{r\sin(\beta)}{\sin(\psi + \beta)} \\ &= \frac{r\sin(\beta)}{\sin(\cos^{-1}(\alpha) - \beta)} \end{aligned} \tag{2.19}$$

For adjacent points on two successive planes with an angle difference of δ, the error distance can be expressed similarly as

$$d_i^{\perp +} = \frac{r\sin(\delta)}{\sin(\cos^{-1}(\gamma) + \delta)} \tag{2.20}$$

$$d_i^{\perp -} = \frac{r\sin(\delta)}{\sin(\cos^{-1}(\gamma) - \delta)} \tag{2.21}$$

where $\gamma = \mathbf{n}\times(\mathbf{l}\times\mathbf{n})\cdot\mathbf{s}$, which is the angle between the binormal of the surface and $\mathbf{s}$.

The maximum error distance at which point $\bar{\mathbf{x}}_i$ can correspond with point $\bar{\mathbf{y}}_j$ is half the minimum distance between adjacent points in pose x or y. Anything larger would cause the point to either correspond with another point or not correspond with any point at all.

Local to the boundary, the correspondence error can be expressed as a linear combination of the boundary's tangent and binormal. Let $\mu_i = \mathbf{c}_i \cdot \mathbf{t_i}$ and $\nu_i = \mathbf{c}_i \cdot \mathbf{b_i}$ be the projections of $\mathbf{c}_i$ onto the boundary's tangent and binormal resulting in the signed quantities μ_i and ν_i. Let $\mu_i(x)$ and $\nu_i(y)$ denote the correspondence error along their respective direction. The expected values of the error in the intervals $x \in [-d_i^-, d_i^+]$ and $y \in [-d_i^{\perp -}, d_i^{\perp +}]$ are:

$$E[\mu_i] = \int_{-d_i^-}^{d_i^+} \mu_i(x) P(x) dx \tag{2.22}$$

$$E[\nu_i] = \int_{-d_i^-}^{d_i^+} \nu_i(y) P(y) dy \tag{2.23}$$

where $P(x)$ and $P(y)$ are the probabilities that $\mathbf{x}_i$ will be located at (x, y) local to the boundary.

As the geometry of the environment is unknown, it is not possible to know the probabilistic distribution of the correspondence errors $P(x)$ and $P(y)$. The assumption is made that μ_i and ν_i are a priori uniformly distributed in the intervals $[-d_i^-, d_i^+]$ and $[-d_i^{\perp -}, d_i^{\perp +}]$, thus $P(x) = 1/(d_i^- + d_i^+)$ and $P(y) = 1/(d_i^{\perp -} + d_i^{\perp +})$. Evaluating Eq. 2.22 and Eq. 2.23 yields:

$$\begin{aligned} E[\mu_i] &= \frac{(d_i^+)^2 - (d_i^-)^2}{d_i^+ + d_i^-} = d_i^+ - d_i^- \\ &= \frac{-2r\ \sin^2(\beta)\ \alpha}{\sin^2(\cos^{-1}(\alpha)) - \sin^2(\beta)} \end{aligned} \tag{2.24}$$

$$E[\nu_i] = \frac{-2r\ \sin^2(\delta)\ \gamma}{\sin^2(\cos^{-1}(\gamma)) - \sin^2(\delta)} \tag{2.25}$$

Note that when $\mathbf{s}$ is not parallel to $\mathbf{n}$, α and γ become unequal to 0, at which point the means become non-zero. However, since the means are proportional to $sin^2(\beta)$ and $sin^2(\delta)$, this term is negligible when β and δ are small. Thus the correspondence errors can be considered to have zero-mean. The variances of the correspondence errors μ_i and ν_i can then be computed, assuming the means are zero, as follows:

$$\begin{aligned} E[\mu_i^2] &= \int_{-d_i^-}^{d_i^+} \frac{x^2}{d_i^+ + d_i^-} dx \\ &= \frac{(d_i^+)^3 + (d_i^-)^3}{3(d_i^+ + d_i^-)} \end{aligned} \tag{2.26}$$

$$E[\nu_i^2] = \frac{(d_i^{\perp +})^3 + (d_i^{\perp -})^3}{3(d_i^{\perp +} + d_i^{\perp -})} \tag{2.27}$$

Expressing $\mathbf{c}_i$ in terms of μ_i, ν_i, $\mathbf{t}_i$ and $\mathbf{b}_i$ gives:

$$\begin{aligned} \mathbf{c}_i = \mu_i \mathbf{t}_i + \nu_i \mathbf{b} &= \sqrt{\mu_i^2 + \nu_i^2} \frac{\left(\frac{\mu_i}{|\mu_i|} \mathbf{t} + \frac{\nu_i}{|\nu_i|} \mathbf{b} \right)}{\|\mathbf{t} + \mathbf{b}\|} \\ &= \sqrt{\mu_i^2 + \nu_i^2} \frac{(\pm \mathbf{t} \pm \mathbf{b})}{\sqrt{2}} \end{aligned} \tag{2.28}$$

The covariance matrix of the correspondence error can then be found as:

$$\mathbf{P}_i^C = E[\mathbf{c}_i (\mathbf{c}_i)^\top] = \frac{1}{2} E[\mu_i^2 + \nu_i^2] (\pm \mathbf{b} \pm \mathbf{t})(\pm \mathbf{b} \pm \mathbf{t})^\top \tag{2.29}$$

The covariance matrix of the matching error at the i^{th} point correspondence of two subsequent poses then becomes:

$$\mathbf{P}_i = \mathbf{P}_i^N + \mathbf{P}_i^C \tag{2.30}$$

With these matrices, more accurate pose estimations can be acquired as described in Chapter 4.

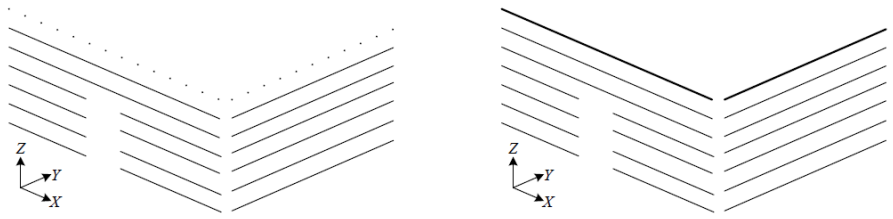
CHAPTER 3

Estimating Surface Normals

The correspondence covariance matrix is a function of the environment's surface normals. As the environment is unknown, the normal vectors have to be estimated by performing surface detection on the laser range data points. Surface detection from point cloud data is a well known and studied topic [17], [11], [1], [14]. But these techniques require dense and high quality data sets [22]. Even though solutions exist that better fit the problem of laser range data points that run in real-time [22] [4], the LINMER algorithm [25] will be used here, for its ease of implementation and fast performance.

By using the implicit ordering of the laser range data scan points, the simple and fast LENCOMP algorithm described in [25] can be used to detect lines along each plane in a scan (see Fig. 3.1). Let $\{\bar{\mathbf{a}}_i\}$ denote the range scan data points on a single plane. Assume that points $\bar{\mathbf{a}}_i$ through $\bar{\mathbf{a}}_j$ form a line, point $\bar{\mathbf{a}}_{j+1}$ also lies on that line if:

$$\frac{\|\bar{\mathbf{a}}_{j+1} - \bar{\mathbf{a}}_i\|}{\sum_{k=i}^{j} \|\bar{\mathbf{a}}_{k+1} - \bar{\mathbf{a}}_k\|} < \epsilon(j) \tag{3.1}$$

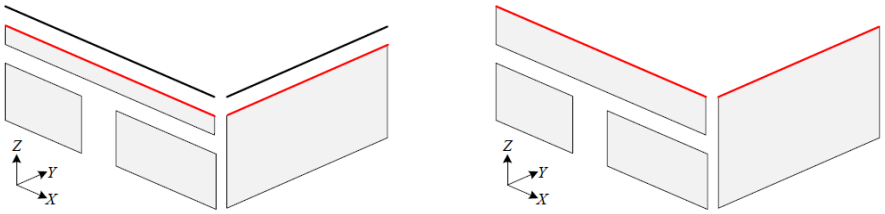


(a) (b)

Figure 3.1: Result of LENCOMP line detection: a) Laser range data scan points on a single scan plane to be processed; b) The two lines that were detected. Figures courtesy of [19].

The lines found in every plane can then be used to detect surfaces with the LINMER [25] algorithm. Assume that the laser range sensor scans from bottom to top. Let $L = \{\}$ be the stored set of lines and $S = \{\}$ be the stored set of surfaces, combined with LENCOMP the LINMER algorithm proceeds as follows:

1. Find a line n with LENCOMP.
2. Check if the line matches any line in L, if a match l is found, n and l are transformed into a surface and added to S. Remove l from L if a match was found.
3. If no matching line is found, check if n might be an extension of a surface in S by matching against the top line of the surface. Replace the top line of the surface with n, resulting in an enlarged surface.



(a) (b)

Figure 3.2: Surface expansion in the LINMER algorithm: a) The red lines mark the top line of surfaces for which a new line match has been found; b) Top lines of surfaces replaced by their matching lines. Figures courtesy of [19].

4. If no matches are found at all, store n in L.

5. Go back to step 1 until no new lines are found.

In order for a line to match another, the following criteria have to be met:

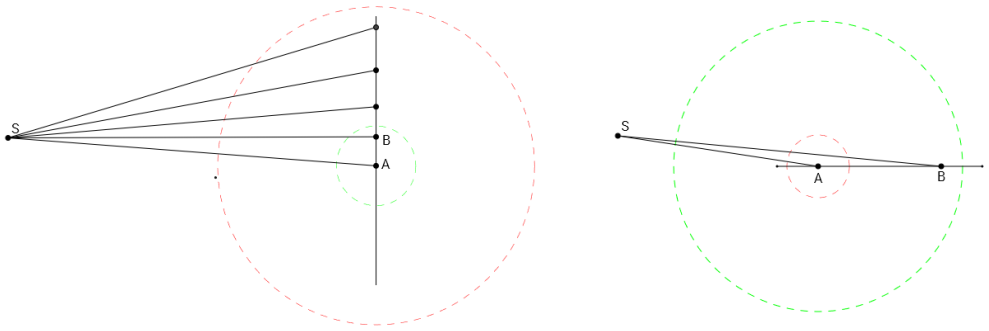
- The angle between two lines has to be smaller than a given angle ω.
- The endpoints of each line must lie within a given distance λ of the corresponding endpoints of the other line.
- The lines must lie roughly in the same plane.

Lower values of the threshold angle ω will result in more lines, but are more sensitive to noise, whereas higher values result in fewer lines at the potential cost of detail [19].

Using a fixed threshold distance λ is not ideal as the distance between endpoints are smaller for lines scanned at close range than lines scanned at long range. Furthermore, using a single value performs a check against a sphere, which is not optimal for all directions as shown in Fig. 3.3.

A better approach is introduced here, one that checks against a bounding box or ellipsoid instead of a sphere. Let $\mathbf{\Lambda}_i$ denote the radii of the bounding box or ellipsoid and let r_i denote the sensor scan ranges, from Fig. 3.3a and 3.3b we can derive values for each of the components of $\mathbf{\Lambda}_i$:

$$\mathbf{\Lambda}_i = r_i \begin{bmatrix} \gamma \\ \tan(\beta) \\ \tan(\alpha) \end{bmatrix} \tag{3.2}$$



(a) (b)

Figure 3.3: Suboptimality of using a fixed distance for line matching: a) Range scans are almost perpendicular to the wall, in this case the radius of the inner circle is desirable; b) Here the range scans are almost parallel to the wall, in this case the radius of the outer circle is desirable.

where γ is a fraction of the range in the direction of the laser, β is the angle between rays on the same scan plane and α is the angle between two scan planes. Let the vector from endpoint $\mathbf{a}_i$ to $\mathbf{b}_j$ be denoted by $\mathbf{d}_i$. In order to match the endpoints, $\mathbf{d}_i$ has to be rotated to line up with the axis aligned $\mathbf{\Lambda}_i$. The rotation matrix can be constructed with the inclination θ_i and azimuth ϕ_i of the measurement:

$$\mathbf{A}_i = \begin{bmatrix} \sin(\theta_i)\cos(\phi_i) & \sin(\theta_i)\cos(\phi_i - \frac{\pi}{2}) & \sin(\theta_i - \frac{\pi}{2})\cos(\phi_i) \\ \sin(\theta_i)\sin(\phi_i) & \sin(\theta_i)\sin(\phi_i - \frac{\pi}{2}) & \sin(\theta_i - \frac{\pi}{2})\sin(\phi_i) \\ \cos(\theta_i) & \cos(\theta_i) & \cos(\theta_i - \frac{\pi}{2}) \end{bmatrix}^{-1} \tag{3.3}$$

The first column is simply the direction in which the point was sampled, the second and third columns are respectively the tangent and normal of the scan plane. The vector $\mathbf{r}_i = \mathbf{A}_i\mathbf{d}_i$ lies within ellipsoid $\mathbf{\Lambda}_i$ under the condition that:

$$\left(\frac{r_i^x}{\Lambda_i^x}\right)^2 + \left(\frac{r_i^y}{\Lambda_i^y}\right)^2 + \left(\frac{r_i^z}{\Lambda_i^z}\right)^2 < 1 \tag{3.4}$$

If $\mathbf{\Lambda}_i$ represents a bounding box instead, checking if a point lies within the bounding box reduces to a trivial comparison of each of the components of $\mathbf{\Lambda}_i$ and $\mathbf{r_i}$

With the LINMER algorithm and the improvement suggested above, the environment's surface normals can be detected and used to calculate the correspondence covariance matrices, which is necessary for the pose estimation algorithm described in Chapter 4.

CHAPTER 4

Pose Estimation

In order to estimate the pose of a robot, the rigid transformation between two point sets must be determined. In the absence of noise, the corresponding points $\mathbf{a}$ and $\mathbf{b}$ satisfy the following constraint:

$$\mathbf{b} = \mathbf{R}\mathbf{a} + \mathbf{t} \tag{4.1}$$

where $\mathbf{R}$ is a rotation matrix and $\mathbf{t}$ is a translation vector. A conventional method of determining the rigid transformation between two sets of points in the presence of noise, is minimizing the following equation:

$$\sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{R}\mathbf{x}_i - \mathbf{t}\|^2 \tag{4.2}$$

where $\{\mathbf{x}_i\}_{i=1}^n$ and $\{\mathbf{y}_i\}_{i=1}^n$ are point sets of two successive scans. Closed-form solutions exists for Eq. 4.2 in which the data points are considered to have homogeneous and isotropic noise. However, in reality the noise is heteroscedastic, that is, inhomogeneous and anisotropic. One algorithm that takes this into account is presented in [20], but is limited to rotation estimation alone. The algorithm presented in [16] simultaneously estimates the translation and rotation and will be used and described here. However, both algorithms assume known correspondence between points. A method to correspond points in an unknown two-dimensional environment is given in [15]. Here a somewhat similar, but more simple approach is taken. Given two points are within distance d of each other, they correspond when they both are their nearest neighbour.

Any rotation or sequence thereof in three-dimensional space about a fixed point is parameterizable by an angle θ and a unit vector $\mathbf{l}$ indicating the direction of the axis of rotation. A quaternion is a four-dimensional vector that can encode this *axis-angle representation* and can be used to apply the corresponding rotation to a three-dimensional point. The quaternion $\mathbf{q}$ that encodes the rotation of θ about $\mathbf{l}$ is defined as

$$\mathbf{q} = \left[\cos\tfrac{\theta}{2}, \mathbf{l}\sin\tfrac{\theta}{2}\right]^\top \tag{4.3}$$

Note that the norm of $\mathbf{q}$ is by definition 1, thus $\mathbf{q}$ is a unit quaternion. The constraint in Eq. 4.1 is equivalent to [16] [20]:

$$\mathbf{M}\mathbf{q} + \chi = \mathbf{0} \tag{4.4}$$

where $\mathbf{M}$ is a 3×4 matrix defined as

$$\begin{bmatrix} b_x - a_x & 0 & -b_z - a_z & b_y + a_y \\ b_y - a_y & b_z + a_z & 0 & -b_x - a_x \\ b_z - a_z & -b_y - a_y & b_x + a_x & 0 \end{bmatrix} \tag{4.5}$$

and

$$\chi = \mathbf{Q}\mathbf{t}, \qquad \mathbf{Q} = \begin{bmatrix} -q_0 & -q_3 & q_2 \\ q_3 & -q_0 & -q_1 \\ -q_2 & q_1 & -q_0 \end{bmatrix}. \tag{4.6}$$

Note that the determinant of the matrix $\mathbf{Q}$ is equal to $-q_0$, thus the matrix is invertible as long as $q_0 \neq 0$. It is therefor not recommended to apply the algorithm for cases where the rotation angle is close to $\pm\pi$.

Let the rows of matrix $\mathbf{M}$ be denoted by $\mathbf{m}_k^\top$, $k = 1, 2, 3$. The matrices $\mathbf{\Gamma}_{kl}$ are defined as the covariance matrices between the vectors $\mathbf{m}_k$ and $\mathbf{m}_l$, $k, l = 1, 2, 3$, which can be expressed in terms of the covariance matrices ${}^a\mathbf{P}$ and ${}^b\mathbf{P}$ derived in Chapter 2. By definition, $\mathbf{\Gamma}_{11} = E\left[(\mathbf{m}_1 - E\left[\mathbf{m}_1\right])(\mathbf{m}_1 - E\left[\mathbf{m}_1\right])^\top\right]$. Assuming measurements are uncorrelated

$$\begin{aligned}
\mathbf{\Gamma}_{11}(1,1) &= E\left[(b_x + u_x - E\left[b_x\right] - E\left[a_x\right])^2\right] \\
&= E\left[(b_x - E\left[b_x\right])^2\right] + E\left[(a_x - E\left[a_x\right])^2\right] \\
&\quad + 2E\left[(b_x - E\left[b_x\right])(a_x - E\left[a_x\right])\right] \\
&= E\left[(b_x - E\left[b_x\right])^2\right] + E\left[(a_x - E\left[a_x\right])^2\right] \\
&= {}^b\mathbf{P}(1,1) + {}^a\mathbf{P}(1,1)
\end{aligned} \tag{4.7}$$

Defining $\mathbf{S} = {}^b\mathbf{P} + {}^a\mathbf{P}$ and $\mathbf{D} = {}^b\mathbf{P} - {}^a\mathbf{P}$ and repeating this for all elements of $\mathbf{\Gamma}_{11}$ results in

$$\mathbf{\Gamma}_{11} = \begin{bmatrix} \mathbf{S}(1,1) & 0 & -\mathbf{D}(1,3) & \mathbf{D}(1,2) \\ 0 & 0 & 0 & 0 \\ -\mathbf{D}(1,3) & 0 & \mathbf{S}(3,3) & -\mathbf{S}(2,3) \\ \mathbf{D}(1,2) & 0 & -\mathbf{S}(2,3) & \mathbf{S}(2,2) \end{bmatrix} \tag{4.8}$$

The rest of the $\mathbf{\Gamma}$ matrices are computed the same way (see Appendix A). The algorithm to estimate the rigid motion proceeds as follows:

1. Obtain an initial solution $\hat{\mathbf{q}}$ and $\hat{\mathbf{t}}$, where $\hat{\mathbf{t}}$ is the estimated translation. Here the odometry estimates are used.

2. Find the set of corresponding points while taking $\hat{\mathbf{q}}$ and $\hat{\mathbf{t}}$ into account.

3. Compute the 3×3 matrices $\mathbf{\Sigma}_i$, $i = 1, ..., n$ for all measurements, having the kl-th element defined by $\hat{\mathbf{q}}^\top \mathbf{\Gamma}_{kli} \hat{\mathbf{q}}$.

4. Compute the weighted "centroid" matrix $\tilde{\mathbf{M}}$

$$\tilde{\mathbf{M}} = \left(\sum_{i=1}^{n} \mathbf{\Sigma}_i^{\#}\right)^{\#} \left(\sum_{i=1}^{n} \mathbf{\Sigma}_i^{\#} \mathbf{M_i}\right) \tag{4.9}$$

   where $\mathbf{A}^{\#}$ stand for the pseudoinverse of matrix $\mathbf{A}$, which can be computed using the *Singular Value Decomposition* of matrix $\mathbf{A}$

$$\mathbf{A} = \mathbf{U}\mathbf{E}\mathbf{V}^\top, \tag{4.10}$$

   inverting all non 0 singular values in $\mathbf{E}$, the pseudoinverse is then

$$\mathbf{A}^{\#} = \mathbf{U}\mathbf{E}'\mathbf{V}^\top. \tag{4.11}$$

   Note that for computational stability, a small value of ϵ should be used instead of 0, where all singular values smaller than ϵ are set to 0.

5. Compute the "scatter" $\mathbf{S}(\hat{\mathbf{q}})$ relative to $\tilde{\mathbf{M}}$

$$\mathbf{S}(\hat{\mathbf{q}}) = \sum_{i=1}^{n} \left(\mathbf{M}_i - \tilde{\mathbf{M}}\right)^\top \mathbf{\Sigma}_i^{\#} \left(\mathbf{M}_i - \tilde{\mathbf{M}}\right) \tag{4.12}$$

   and

$$\mathbf{C}(\hat{\mathbf{q}}) = \sum_{i=1}^{n} \sum_{k,l=1}^{3} \eta_{ki} \eta_{li} \mathbf{\Gamma}_{kli} \tag{4.13}$$

   where

$$\boldsymbol{\eta}_i = \mathbf{\Sigma}_i^{\#} \left(\mathbf{M}_i - \tilde{\mathbf{M}}\right) \hat{\mathbf{q}} \tag{4.14}$$

   Note that the dependency of $\mathbf{S}$ and $\mathbf{C}$ on $\hat{\mathbf{q}}$ was made explicit.

6. Update the quaternion estimate as the solution of the generalized eigenproblem

$$\mathbf{S}(\hat{\mathbf{q}})\hat{\mathbf{q}} = \mathbf{C}(\hat{\mathbf{q}})\hat{\mathbf{q}} \tag{4.15}$$

corresponding to the smallest eigenvalue. Solving Eq. 4.15 with the generalized singular value decomposition (GSVD) could be used, which has better numerical behavior [13].

7. Iterate through Steps 2 and 6 until the value of the smallest eigenvalue becomes one, up to a tolerance.

8. Estimate the translation as $\hat{\mathbf{t}} = \hat{\mathbf{Q}}^{-1}\tilde{\mathbf{M}}\hat{\mathbf{q}}$

With this, the new pose can be estimated by applying the estimated rigid transformation to the current pose.

CHAPTER 5

Octree

Finding point correspondence relies heavily on nearest neighbour search. A naïve implementation would have to visit every point in order to determine the nearest one, resulting in a complexity of $O(n)$ in the number of points. Having to do this for every point leads to quadratic complexity, which has serious impact in terms of performance. This complexity can be reduced by utilizing octrees.

An octree is a tree data structure in which every node represents a cube shaped volume. A node contains eight children that subdivide the volume in to eight smaller cubes, called octants. Octrees are the three-dimensional equivalents of the two-dimensional quadtrees where nodes represent a square area. Having this recursively subdivided tree structure allows for the fast search algorithms similar to binary trees.

A naïve implementation of an octree would look similar to Fig. 5.1. The size of this structure would range from 56 to 112 bytes depending on whether it is a *32-bit* or *64-bit* machine and on the size of the used floating point types, which are usually 8 bytes for increased precision in scientific applications. This is not very efficient memory wise, nor does it fit in a typical *cache line*, making it more costly to traverse.

There is a lot of redundant information stored in this structure, some of which is completely unnecessary and others that could be inferred during tree traversal. The structure always contains eight children, therefore only a single pointer is needed that points to the eight children laid out contiguously in memory, this saves upto 56 bytes, reducing the size in half. Similarly, the number of points and the pointer to the contained points can be combined into a single pointer. The center and radius are only really needed in the root node, while traversing the tree these can be calculated on the fly. This comes with the trade off that you can only traverse down the tree, unless you store which octant a child node is. However, there is no trade off here as traversing up the tree will not be necessary. This saves up another 32 bytes, reducing the total amount of bytes from 112 to 16 bytes composed out of two pointers, which is a lot more memory efficient and also fits four times into a single cache line of 64 bytes.

Apart from the data structure, there is another aspect of octrees that involves redundancy, namely the fixed amount of child nodes. In a non-homogeneously distributed point cloud, most nodes will be void of points. That is, if you have a point cloud representation of some objects, only their surfaces will be covered with points, the entire volume will be empty. An octree data structure that stores only the

```
struct Octree {
    Octree* children[8];
    int pointCount;
    real** points;
    real[3] center;
    real radius;
};
```

Figure 5.1: Implementation of a naive octree model with eight pointers to its children and a pointer to the points it contains.

leaf	children	valid
1	23	8

(a)

leaf	first point	count
1	23 - 31	8 - 0

(b)

Figure 5.2: Efficient 4 byte encoded octree structure: The leaf bit indicates whether the block of memory should be interpreted as either: a) Parent node containing the total number of children for efficient tree traversal and the 8 bits in *valid* indicate which child octants are valid; b) Leaf containing an index to the first point and the number of points in the leaf. Number of point indices vs. points per leaf is adjustable.

nodes that contain points, could thus greatly reduce the amount of nodes required. These types of octrees are called sparse, as opposed to dense octrees. Even though sparse octrees require extra bookkeeping on knowing which octants are present and which are not, efficient encoded octree structures exist that are only 8 bytes in size [12] [7].

5.1 Memory Efficient Encoded Octree

The size of an octree data structure can be made even smaller with a serialized pointer-free encoding where only leaves containing points are laid out linearly in memory, in which case there is no need to store the actual octree structure. Leaves are generally stored in the The Morton order [18] where multi-dimensional data is mapped to one dimension while preserving locality. For octrees this is accomplished by first partitioning points in one dimension relative to the center of a node, into two partitions. Repeating this in another dimension for each partition, and then again for the final dimension results in eight partitions. When done recursively for every node, the result is a linear array of points in depth-first order. As partitioning a set into two subsets has a complexity of $O(n)$ in the number of points, the octree can be generated in $O(n \log n)$.

However, modifying such serialized trees by adding or removing points involves shifting the entire region before or after the point and traversing them cannot be performed in the classical sense [7] as there is no actual structure to traverse. By adding minimal structure data, a 4 byte encoding with efficient tree traversal is devised here, supporting approximately 8 million child nodes per octant of the root node. Fig. 5.2 and 5.3 show the resulting octree data structure and implementation.

```
union EncodedOctree {
    struct
    {
        unsigned32 leaf : 1;
        unsigned32 children : 23;
        unsigned32 valid : 8;
    };
    struct
    {
        unsigned32 leaf : 1;
        unsigned32 firstPoint : 24;
        unsigned32 count : 7;
    };
};
```

Figure 5.3: Implementation of an efficient 4 byte encoded octree model. Note that the two structs share the same bytes and that the three variables within them are so called *bitfields*, occupying the amount of bits specified after the colon.

Unlike [12] and [7], a parent contains no knowledge of whether a child node is a leaf or not. Every node contains a bit indicating if it is a leaf or not, saving seven bits in parent nodes at the cost of one in leaf nodes, thus leaving 23 bits to be used for efficient tree traversal. For an efficient nearest neighbour algorithm, there must be a way to skip a branch and continue on a neighbouring node. Storing the number of child nodes in every parent node is a means to that end, as it is the amount of nodes that need to be skipped in the array to get to a neighbouring node.

5.2 Nearest Neighbour

Algorithm 5.1 shows the nearest neighbour search algorithm, which uses the FindNearestLeaves() function described in Algorithm 5.2. The algorithm is straight forward, it searches for all leaves that overlap the region around the point. From all the points contained in the leaves, the nearest point is returned.

The nearest leaves are found by doing an axis-aligned bounding box intersection test for all octants in a node with the region of interest. To be able to go from one octant to a neighbouring octant, the number of children is used to skip the right amount of nodes. Doing this recursively for all nodes that intersect results in a list of all nearest leaves.

As traversing down the tree is done in $O(n)$ and the number of nodes that will intersect with the region of interest is practically constant, the nearest neighbour search is performed in $O(n)$.

Algorithm 5.1: Nearest Neighbour Search

```
NearestNeighbour(point, maxDistance to point) {
    pointAABB = AABB around point with maxDist as radius
    leaves = FindNearestLeaves(root, pointAABB)

    for all leaves {
        for all points in leaf {
            if nearest point {
                store as nearestPoint
            }
        }
    }

    return nearestPoint
}
```

Algorithm 5.2: Nearest Leaves Search

```
FindNearestLeaves(node to search in, region of interest) {
    if node is leaf {
        return node
    }

    for all octants in node {
        if valid {
            if AABBIntersect(octantAABB, region) {
                leaves += FindNearestLeaves(octant, region)
            }
            if octant not leaf {
                skip children to get to next octant
            }
        }
    }

    return leaves
}
```

CHAPTER 6

Experiments

Every experiment described in this chapter was performed on an Intel(R) Core(TM) i5-4200u 1.60 GHz CPU with 8 GB of ram running at 1600 MHz. Although there are two cores and four logical processors available on the CPU, the implementation used here was written in a single threaded manner, thus utilizing only a single core. The implementation was done in *C++* and compiled using the *Microsoft Visual Studio 2015* compiler. For the linear algebra parts of the implementation the *Eigen 3.2.5* and *Armadillo 5.200* libraries were used, where the latter was only used to solve the generalized eigenproblem as the Eigen library's implementation was not complete.

Virtual robot simulations were performed using *UDK 2014* and *USARSim*. Collection of three-dimensional sensor data was done using a *P3AT* virtual robot with a custom 3-D laser range sensor script (see Appendix B), with a maximum range of 20 meters. *UvARescue 2014* was used to control the virtual robot and retrieve the sensor data.

6.1 Surface Detection

The surfaces depicted in this section are all from a single environment, see Fig. 6.1. Fig. 6.2 shows the results of the traditional surface detection algorithm of a single pose. For the same pose, Fig. 6.3 shows the results of the newly introduced methods applied. Another pose is shown in Fig. 6.4 and Fig. 6.5. Note that back facing surfaces are not displayed. The points in the figures are color coded according to their elevation, where *blue* points are at ground level and *red* points are at ceiling level.

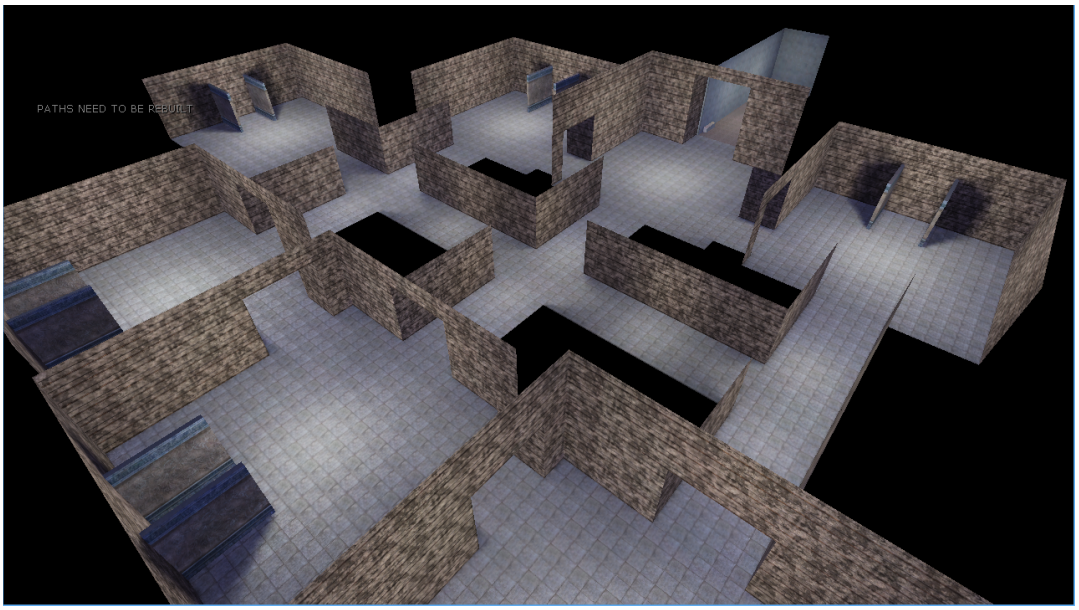
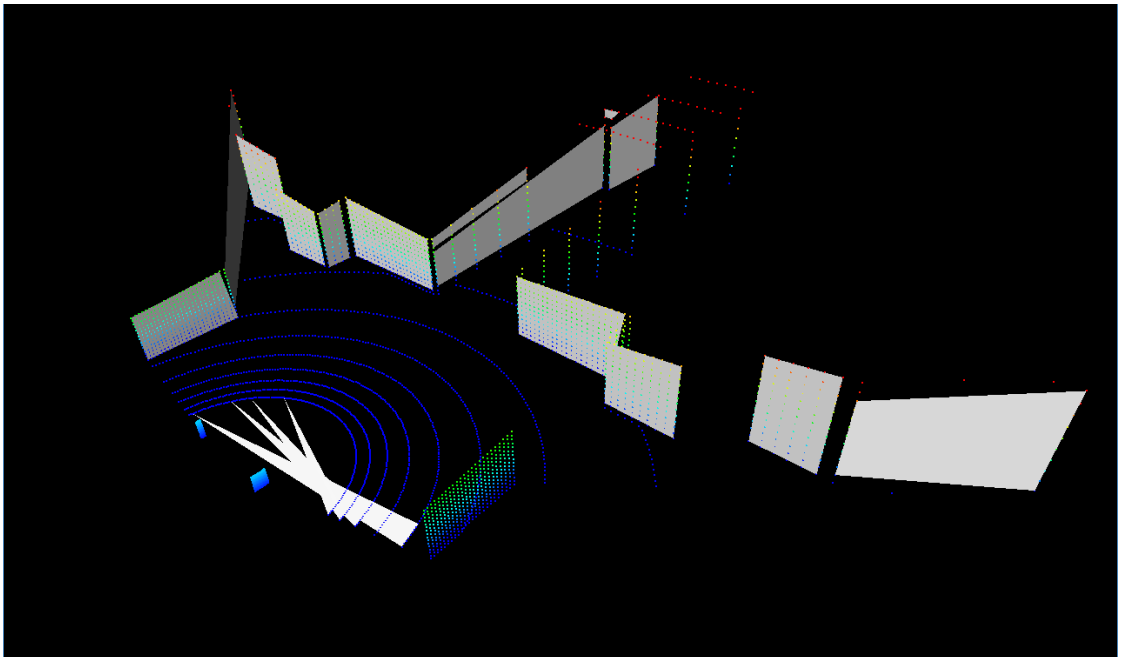
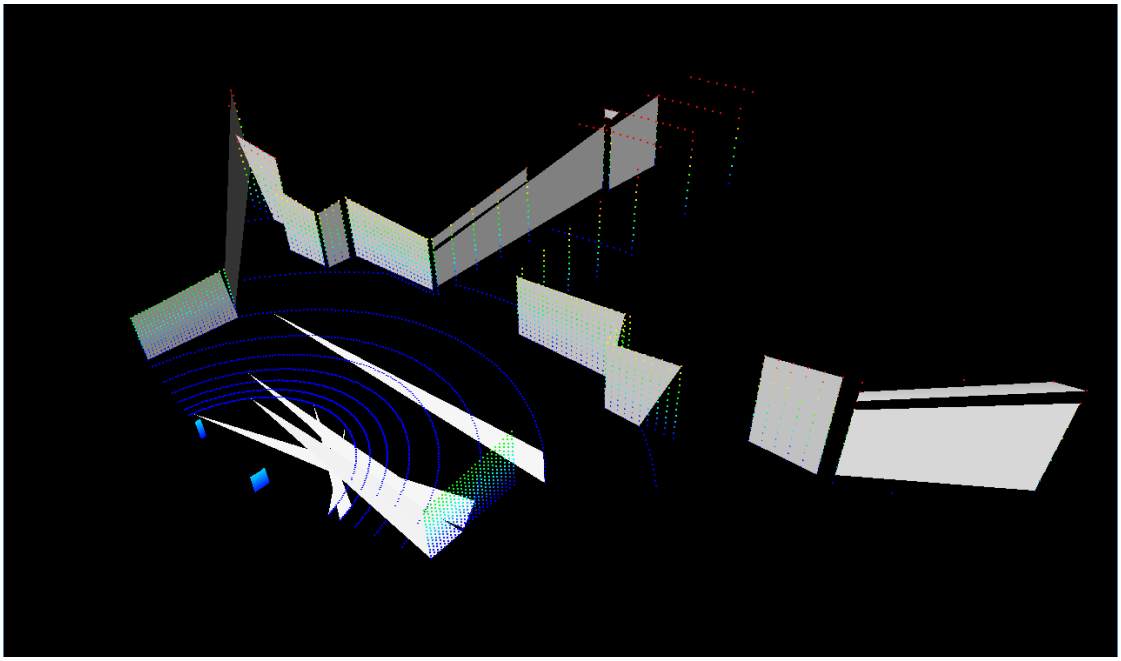


Figure 6.1: Virtual environment used for experimenting with the algorithms.

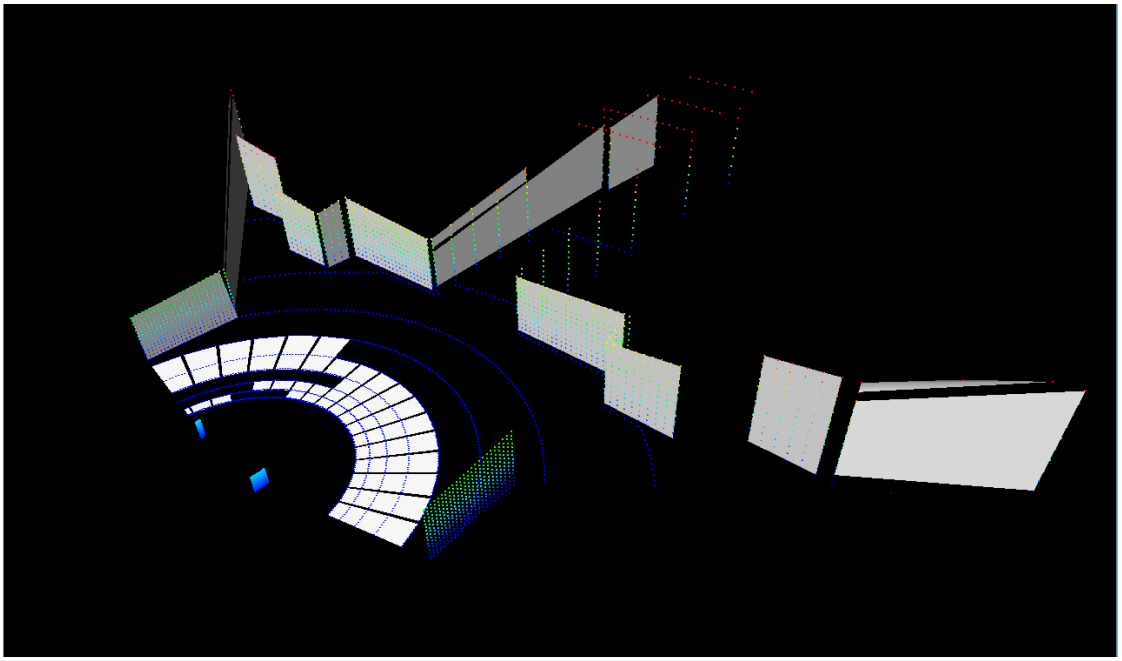


(a)

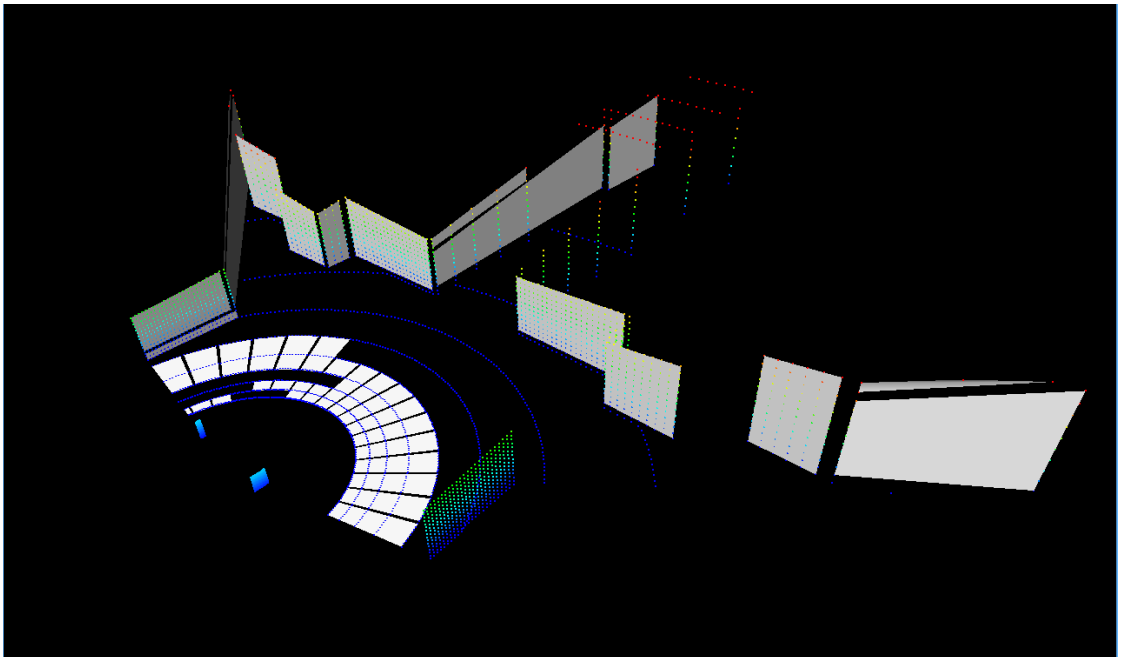


(b)

Figure 6.2: Surface detection using the traditional spherical check: a) Radius of 0.5 meter; b) Radius of 1 meter.

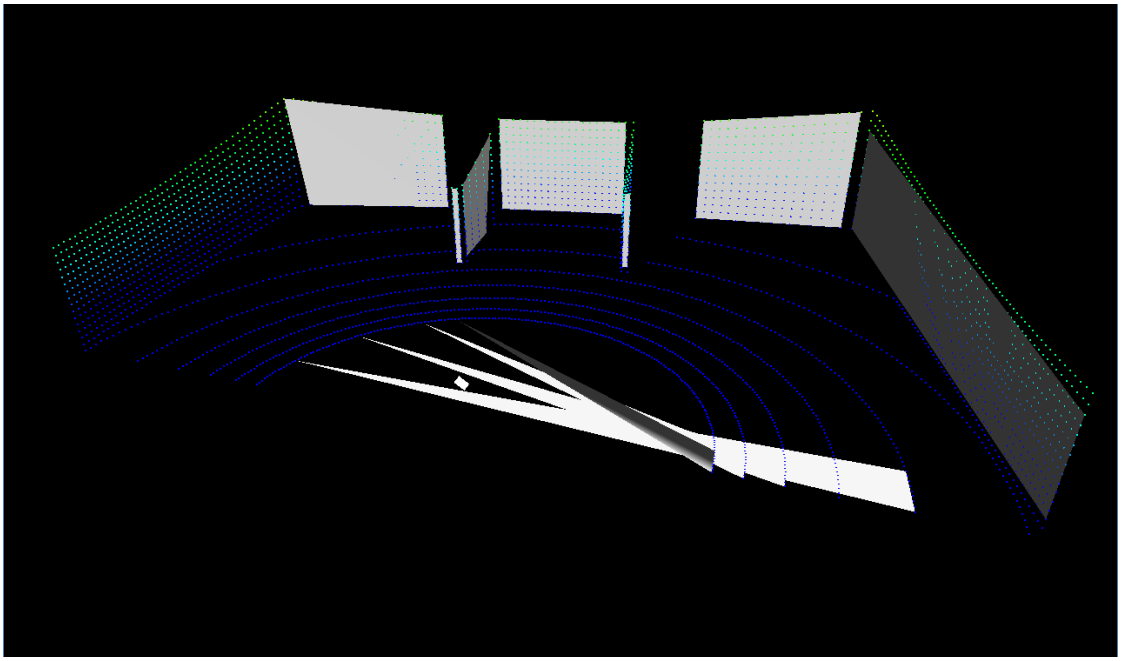


(a)

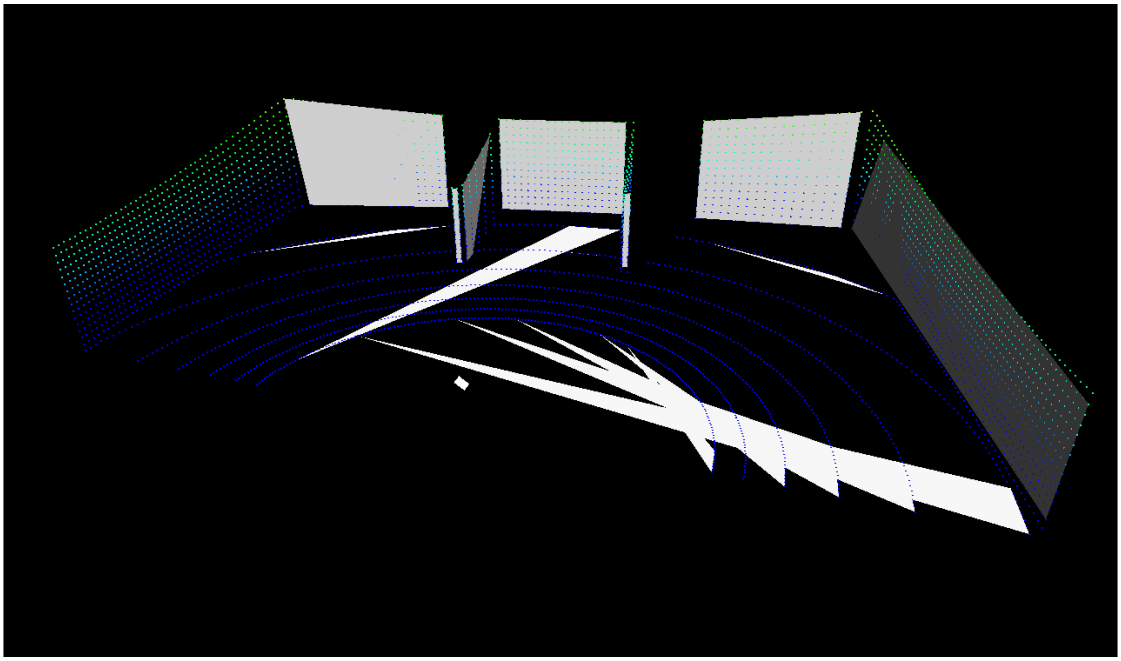


(b)

Figure 6.3: Surface detection using the newly introduced methods: a) bounding box; b) ellipsoid.

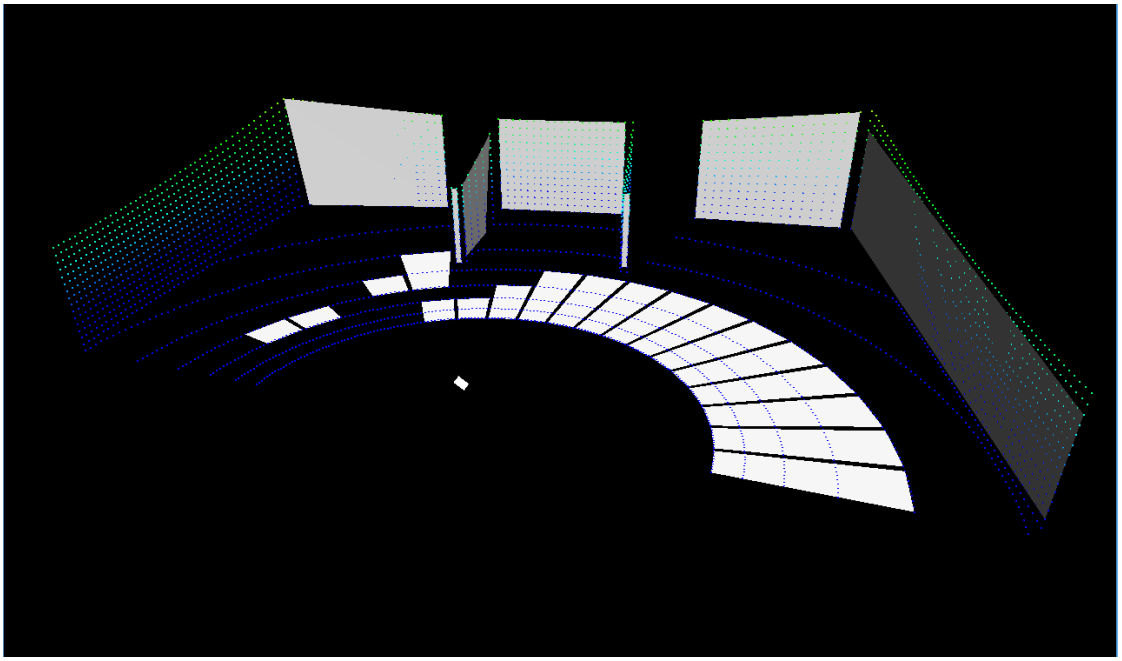


(a)

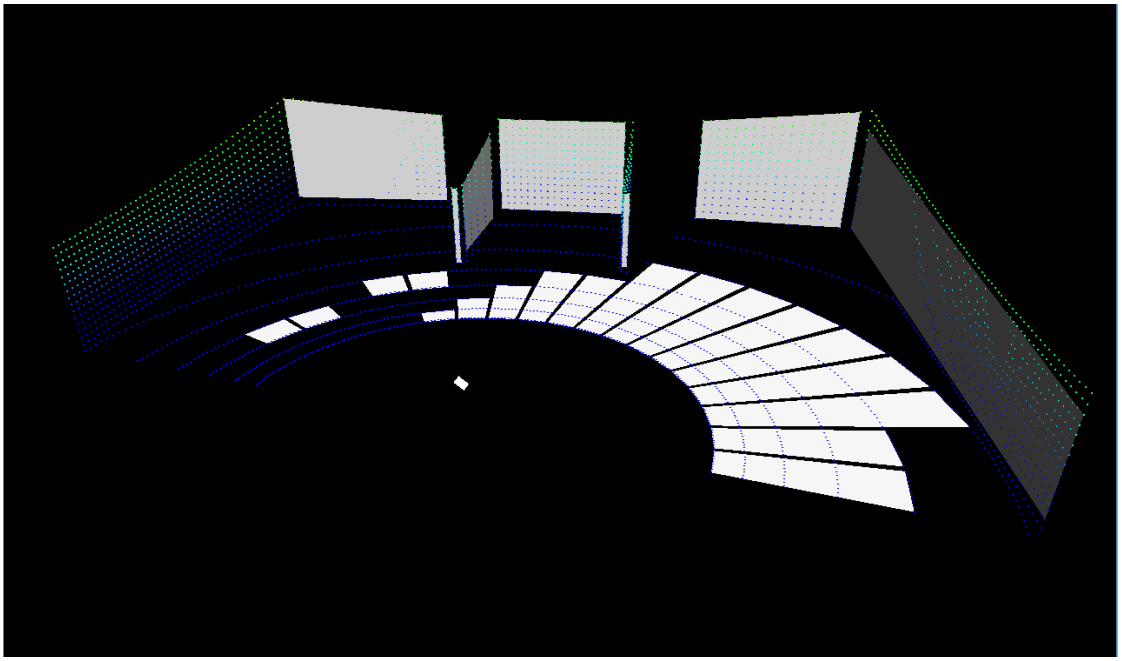


(b)

Figure 6.4: Surface detection using the traditional spherical check: a) Radius of 0.5 meter; b) Radius of 1 meter.



(a)



(b)

Figure 6.5: Surface detection using the newly introduced methods: a) bounding box; b) ellipsoid.

6.2 Pose Estimation

To test the accuracy of the pose estimation, a virtual robot was steered through the environment seen in Fig. 6.1. On its path, 450 poses were recorded whereby the true position, odometry data and the laser range sensor data was recorded. For every pose, approximately 3800 laser ranges were recorded. The poses resulting from the odometry data were then processed by the pose estimation algorithm. The estimations of the algorithm were only taken into account if at least 70% of the points corresponded between poses. Lower values would result in too many faulty correspondences from which the algorithm could not recover. The average distance from the unprocessed poses to the true poses was 0.415 meter, for the processed poses this was 0.399, a reduction of 3.9%. Processing the poses took an average of 265 milliseconds and the algorithm usually converges within 10 iterations. The paths are visualized in Figures 6.6 through 6.9.

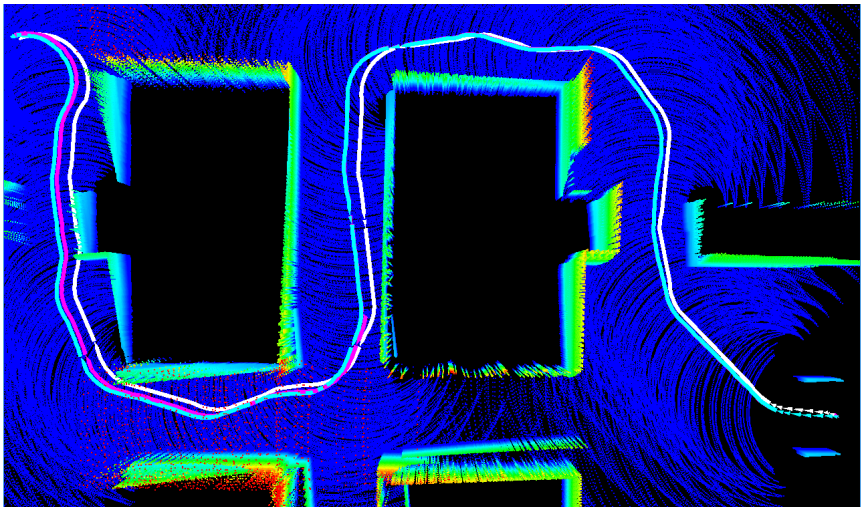


Figure 6.6: Path of the virtual robot starting at the right, working its way to the left. The white path is the true path, cyan is from odometry data and magenta is the processed path. Note that the magenta path is hidden under the cyan path at the beginning, as it is practically similar during that time.

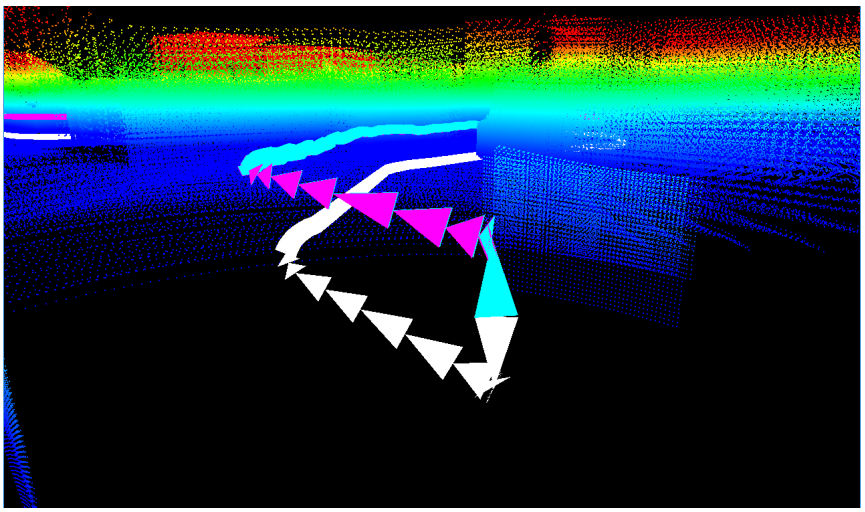


Figure 6.7: Path of the virtual robot right at the start as it is 'dropped' into the world.

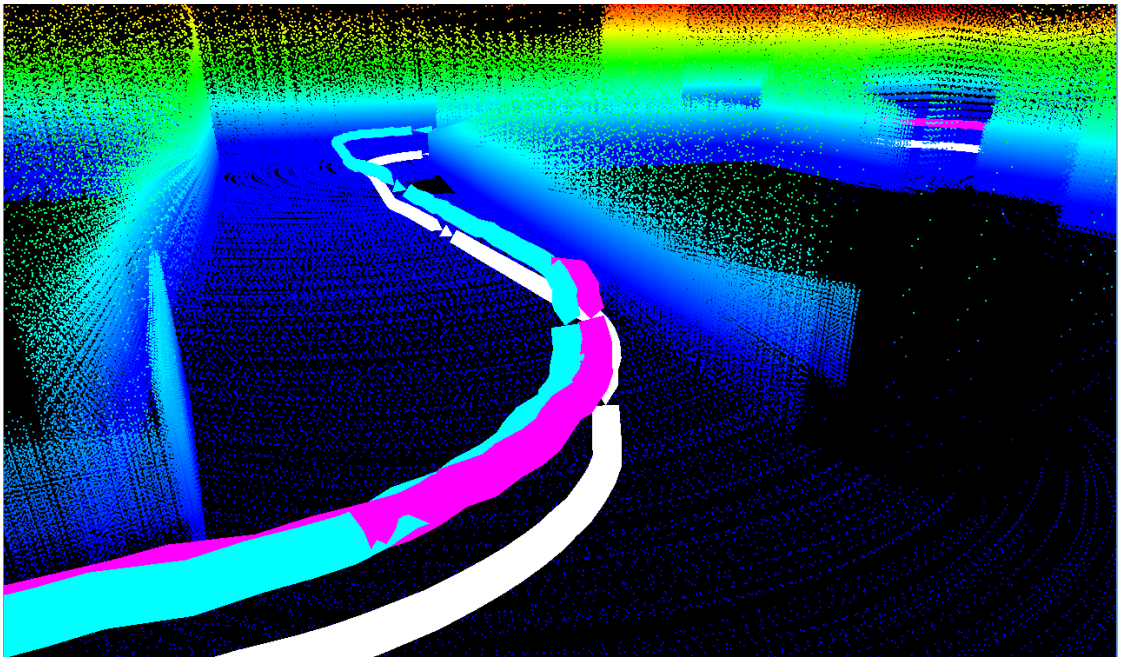


Figure 6.8: Path of the virtual robot where the processed path gets closer to the true path right before turning around the corner.

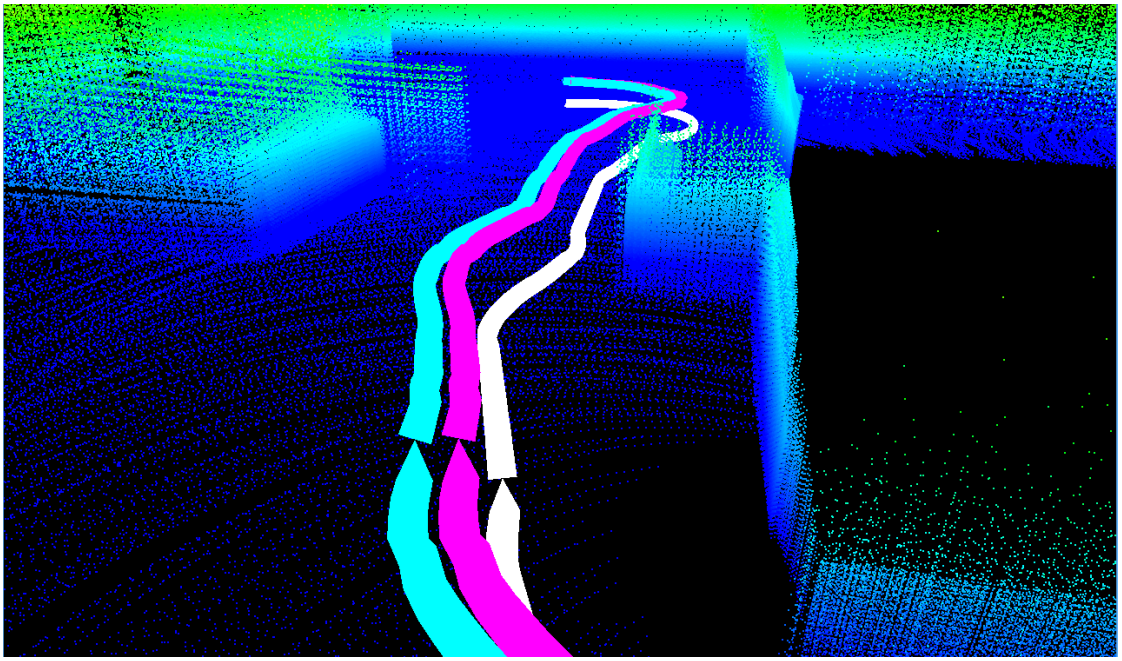


Figure 6.9: Path of the virtual robot where the processed path stays closer to the true path with slight turning after a corner.

6.3 Octree Efficiency

For the octree memory requirement and performance experiments, the publicly available Thermocolorlab data set [3] was used. The data set contains a total of 8 poses, where every pose is constructed out of 9 scans. A single pose holds approximately half a million points. Fig. 6.10 shows a visualization of the points in the first pose, for which the experiment results are listed in Table 6.1. The first four rows in the table are randomly sampled points from all points in that pose. The three rows after that are all points in some of the scans, while the last row is for all of the points in the pose.

					Nearest Neighbour (ms)	
# Points	# Nodes	# Leaves	Mem. size	Constr. time (ms)	Max dist. 1 cm	5 cm
3535	237	189	980 b	0.23	2.3	3.6
7070	411	325	1.68 kB	0.44	4.7	9.3
14140	930	741	3.75 kB	0.86	10.3	21.1
28280	1711	1363	6.87 kB	1.9	24.6	58.7
56560	3880	3130	15.56 kB	3.4	54.9	162
112817	6908	5503	27.67 kB	6.7	99.1	255
227149	14090	11339	56.39 kB	15.0	210	532
516110	32014	25634	128.1 kB	42.3	563	1146

Table 6.1: Memory requirements and performance of the octree data structure for varying amount of points using the Thermocolorlab data set of Jacobs University Bremen [3]. Leaves are also included in the number of nodes. The last two columns show the total time of searching a nearest neighbour for every point in the tree at different maximum distances.

Figure 6.10: Visualization of the points in the first pose of the Thermocolorlab data set.

CHAPTER 7

Conclusion

Accurately mapping the environment and estimating the robot pose is of utmost importance to USAR missions. To do this accurately, a realistic measurement model needs to be used that accounts for the inhomogeneous and anisotropic nature of measurement noise. A derivation of such model is presented here, based on an existing model for the 2-D case. Real-time performance matters as every second matters in saving human lives.

As a result of the very fast octree nearest neighbour search, the algorithm performs in real-time with an average of 265 milliseconds per pose. Furthermore, a very slight increase in accuracy is achieved by the pose estimation algorithm. On average the processed poses are 3.9% closer to the true poses when compared to the poses estimated from odometry data. However, the odometry data returned by the USARSim simulation is unrealistic, as the resulting noise error does not accumulate over time. Considering that and the fact that the estimated poses are on average 4 meters off from the true poses, the accuracy increase is poor. Good point correspondence is absolutely paramount to correct pose estimation. Unfortunately, a simple nearest neighbour search is insufficient to find the right correspondences. More often than not, the pose estimation can not be improved upon due to a lack of point correspondences. The potential accuracy increase could be much higher.

Part of the measurement model is adjusting for correspondence error. For this the surface normals need to be detected. Visual inspection of the surface detection methods shows that the bounding box and ellipsoid method outperform the traditional method. Wall surfaces are almost similar, but cover slightly more area in the bounding box and ellipsoid method. The biggest improvement comes from detecting surfaces on the floor, which the traditional method seems to have trouble with. Correspondence errors are largest for points on surfaces nearly parallel to the laser's incidence vector, meaning these points should be less of a determining factor in pose estimation. With more surfaces detected by the bounding box and ellipsoid method, more accurate covariance matrices for the correspondence errors can be computed, thus lowering the effect of points on the floor on pose estimation, resulting in more accurate estimations.

The algorithm runs in real-time as it is, but leaves a lot to be gained from fully utilizing every core of the CPU. This could open up room to increase the amount of points per scan, perform more complex surface detection algorithms or more complex point correspondence. Although surface detection could be improved, the biggest hurdle lies with point correspondence. It is at the root of accurate pose estimation as the algorithm heavily relies on it. Improving this might not only yield better results, the algorithm might require less iterations to converge, possibly resulting in even better performance. Finally, to get a better understanding of how well the algorithm performs under realistic conditions, the simulated noise should reflect those conditions. Either that, or experiments should be performed with real robots and scanners.

Appendices

APPENDIX A

Covariance matrices

In this appendix the $\mathbf{\Gamma}_{kl}$ matrices described in Chapter 4 are provided.

$$\mathbf{\Gamma}_{11} = \begin{bmatrix} \mathbf{S}(1,1) & 0 & -\mathbf{D}(1,3) & \mathbf{D}(1,2) \\ 0 & 0 & 0 & 0 \\ -\mathbf{D}(1,3) & 0 & \mathbf{S}(3,3) & -\mathbf{S}(2,3) \\ \mathbf{D}(1,2) & 0 & -\mathbf{S}(2,3) & \mathbf{S}(2,2) \end{bmatrix} \tag{A.1}$$

$$\mathbf{\Gamma}_{12} = \begin{bmatrix} \mathbf{S}(1,2) & \mathbf{S}(1,3) & 0 & -\mathbf{D}(1,1) \\ 0 & 0 & 0 & 0 \\ -\mathbf{D}(2,3) & -\mathbf{S}(3,3) & 0 & \mathbf{S}(1,3) \\ \mathbf{D}(2,2) & \mathbf{S}(2,3) & 0 & -\mathbf{S}(2,1) \end{bmatrix} \tag{A.2}$$

$$\mathbf{\Gamma}_{13} = \begin{bmatrix} \mathbf{S}(1,3) & -\mathbf{D}(2,2) & \mathbf{D}(1,1) & 0 \\ 0 & 0 & 0 & 0 \\ -\mathbf{D}(3,3) & \mathbf{S}(3,2) & -\mathbf{S}(3,1) & 0 \\ \mathbf{D}(2,3) & -\mathbf{S}(2,2) & \mathbf{S}(2,1) & 0 \end{bmatrix} \tag{A.3}$$

$$\mathbf{\Gamma}_{22} = \begin{bmatrix} \mathbf{S}(2,2) & \mathbf{D}(2,3) & 0 & -\mathbf{D}(2,1) \\ \mathbf{D}(3,2) & \mathbf{S}(3,3) & 0 & -\mathbf{S}(3,1) \\ 0 & 0 & 0 & 0 \\ -\mathbf{D}(1,2) & -\mathbf{S}(1,3) & 0 & \mathbf{S}(1,1) \end{bmatrix} \tag{A.4}$$

$$\mathbf{\Gamma}_{23} = \begin{bmatrix} \mathbf{S}(2,3) & -\mathbf{D}(2,2) & \mathbf{D}(2,1) & 0 \\ \mathbf{D}(3,3) & -\mathbf{S}(3,2) & \mathbf{S}(3,1) & 0 \\ 0 & 0 & 0 & 0 \\ -\mathbf{D}(1,3) & \mathbf{S}(1,2) & -\mathbf{S}(1,1) & 0 \end{bmatrix} \tag{A.5}$$

$$\mathbf{\Gamma}_{33} = \begin{bmatrix} \mathbf{S}(3,3) & -\mathbf{D}(3,2) & \mathbf{D}(3,1) & 0 \\ -\mathbf{D}(2,3) & \mathbf{S}(2,2) & -\mathbf{S}(2,1) & 0 \\ \mathbf{D}(1,3) & -\mathbf{S}(1,2) & \mathbf{S}(1,1) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.6}$$

Due to symmetry $\mathbf{\Gamma}_{21} = \mathbf{\Gamma}_{12}$, $\mathbf{\Gamma}_{31} = \mathbf{\Gamma}_{13}$ and $\mathbf{\Gamma}_{32} = \mathbf{\Gamma}_{23}$.

APPENDIX B

3-D Laser Range Scanner

The implementation of the 3-D laser range scanner used to scan the environment is shown in Listing B.1.

Listing B.1: 3-D sensor implementation for USARSim.

```
class SICK3D extends RangeScanner config (USAR);

var vector dir;
//  Retrieves the range data using trace and reports this range in UU or meters
//  depending on presence of converter.
//  The Trace method traces a line to point of first collision.
//  Takes actor calling trace collision properties into account.
//  Returns first hit actor, level if hit level, or none if hit nothing
function float GetRange()
{
    local vector StartLocation;
    local vector HitLocation, HitNormal;
    local Actor HitActor;
    local SmokeInterface smoke;
    local float range, curRange, maxRangeRemaining;

    StartLocation = Location;
    HitActor = self;
    curRange = 0.0;
    maxRangeRemaining = MaxRange;

    while(maxRangeRemaining > 0.0)
    {
        HitActor = HitActor.Trace(HitLocation, HitNormal,
            StartLocation + maxRangeRemaining * dir, StartLocation, true);
        smoke = SmokeInterface(HitActor);
        range = VSize(HitLocation - StartLocation);

        if (HitActor == None)
        {
            range = curRange + maxRangeRemaining;
            range = class'UnitsConverter'.static.LengthFromUU(range);
            return range;
        }

        // No smoke, so a normal object. Smoke that uses particles always block.
        if (smoke == None || smoke.SmokeAlwaysBlock())
        {
            range = curRange + VSize(HitLocation - StartLocation);
            range = class'UnitsConverter'.static.LengthFromUU(range);
            return range;
        }

        StartLocation = HitLocation;
        maxRangeRemaining = maxRangeRemaining - range;
        curRange = curRange + range;
    }

    curRange = class'UnitsConverter'.static.LengthFromUU(curRange);
    return curRange;
}

function String VecToStr(vector v)
{
    return v.X $ "," $ v.Y $ "," $ v.Z;
```

```
}

function String GetData()
{
    local vector ray, look, right, up;
    local String rangeData;
    local float a, i, range;
    local float res;
    local float minInclination;
    local float maxInclination;
    local float maxAzimuth;

    res = class'UnitsConverter'.static.AngleFromUU(Resolution);
    minInclination = 0.5*Pi - Pi / 18.0;
    maxInclination = 0.5*Pi + Pi / 18.0;
    maxAzimuth = class'UnitsConverter'.static.AngleFromUU(ScanFov) / 2.0;

    time = WorldInfo.TimeSeconds;

    GetAxes(Rotation, look, right, up);

    for (i = minInclination; i <= maxInclination; i += res)
    {
        // from right to left
        for (a = maxAzimuth; a >= -maxAzimuth; a -= res)
        {
            ray.X = Sin(i) * Cos(a);
            ray.Y = Sin(i) * Sin(a);
            ray.Z = Cos(i);

            // rotate direction vector by current orientation
            dir.X = look.X * ray.X + right.X * ray.Y + up.X * ray.Z;
            dir.Y = look.Y * ray.X + right.Y * ray.Y + up.Y * ray.Z;
            dir.Z = look.Z * ray.X + right.Z * ray.Y + up.Z * ray.Z;

            range = GetRange();
            if (rangeData == "")
                rangeData = class'UnitsConverter'.static.FloatString(range, 4);
            else
                rangeData = rangeData $ "," $
                class'UnitsConverter'.static.FloatString(range, 4);
        }
    }

    return "{Name " $ ItemName $ "} {Resolution " $
        class'UnitsConverter'.static.Str_AngleFromUU(Resolution) $ "} {FOV " $
        class'UnitsConverter'.static.Str_AngleFromUU(ScanFov) $ "} {MaxRange " $
        class'UnitsConverter'.static.LengthFromUU(MaxRange) $
        "} {Range " $ rangeData $ "}";
}

defaultproperties
{
    BlockRigidBody=true
    bCollideActors=true
    bBlockActors=false
    bProjTarget=true
    bCollideWhenPlacing=true
    bCollideWorld=true

    Begin Object Name=StaticMeshComponent0
        StaticMesh=StaticMesh'SICKSensor.lms200.Sensor'
        CollideActors=true
        BlockActors=false
        BlockRigidBody=true
        BlockZeroExtent=true
        BlockNonZeroExtent=true
    End Object
}
```

Bibliography

[1] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 415–421. ACM, 1998.

[2] K Somani Arun, Thomas S Huang, and Steven D Blostein. Least-squares fitting of two 3-d point sets. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (5):698–700, 1987.

[3] Dorit Borrmann. Thermocolorlab. `http://kos.informatik.uni-osnabrueck.de/3Dscans/`. Accessed: 2015-08-15.

[4] Dorit Borrmann, Jan Elseberg, Kai Lingemann, and Andreas Nüchter. The 3d hough transform for plane detection in point clouds: A review and a new accumulator design. *3D Research*, 2(2):1–13, 2011.

[5] Jennifer Casper and Robin Roberson Murphy. Human-robot interactions during the robot-assisted urban search and rescue response at the world trade center. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 33(3):367–385, 2003.

[6] Leo Dorst. First order error propagation of the procrustes method for 3d attitude estimation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(2):221–229, 2005.

[7] Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. One billion points in the cloud–an octree for efficient processing of 3d laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing*, 76:76–88, 2013.

[8] Bert F Green. The orthogonal approximation of an oblique structure in factor analysis. *Psychometrika*, 17(4):429–440, 1952.

[9] Berthold KP Horn. Closed-form solution of absolute orientation using unit quaternions. *JOSA A*, 4(4):629–642, 1987.

[10] Berthold KP Horn, Hugh M Hilden, and Shahriar Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *JOSA A*, 5(7):1127–1135, 1988.

[11] Jan Klein and Gabriel Zachmann. Proximity graphs for defining surfaces over point clouds. In *Eurographics Symposium on Point-Based Grahics (SPBG04)*, pages 131–138, 2004.

[12] Samuli Laine and Tero Karras. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation*, 2, 2010.

[13] Yoram Leedan and Peter Meer. Estimation with bilinear constraints in computer vision. In *Computer Vision, 1998. Sixth International Conference on*, pages 733–738. IEEE, 1998.

[14] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.

[15] Feng Lu and Evangelos Milios. Robot pose estimation in unknown environments by matching 2d range scans. *Journal of Intelligent and Robotic Systems*, 18(3):249–275, 1997.

[16] Bogdan Matei and Peter Meer. Optimal rigid motion estimation and performance evaluation with bootstrap. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 1. IEEE, 1999.

[17] Niloy J Mitra, An Nguyen, and Leonidas Guibas. Estimating surface normals in noisy point cloud data. *International Journal of Computational Geometry & Applications*, 14(04n05):261–276, 2004.

[18] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[19] Peter Nelson. 3d mapping for robotic search and rescue. 4th year project report, University of Oxford, 2011.

[20] Naiya Ohta and Kenichi Kanatani. Optimal estimation of three-dimensional rotation and reliability evaluation. *IEICE TRANSACTIONS on Information and Systems*, 81(11):1247–1252, 1998.

[21] Sam T Pfister, Kristo L Kriechbaum, Stergios Roumeliotis, Joel W Burdick, et al. Weighted range sensor matching algorithms for mobile robot displacement estimation. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 2, pages 1667–1674. IEEE, 2002.

[22] Jann Poppinga, Narunas Vaskevicius, Andreas Birk, and Kaustubh Pathak. Fast plane detection and polygonalization in noisy 3d range images. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 3378–3383. IEEE, 2008.

[23] Peter H Schönemann. A generalized solution of the orthogonal procrustes problem. *Psychometrika*, 31(1):1–10, 1966.

[24] Robin Sibson. Studies in the robustness of multidimensional scaling: Procrustes statistics. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 234–238, 1978.

[25] Hartmut Surmann, Kai Lingemann, Andreas Nüchter, and Joachim Hertzberg. Fast acquiring and analysis of three dimensional laser range data. In *VMV*, pages 59–66, 2001.

[26] Shinji Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (4):376–380, 1991.

[27] Arnoud Visser, Bayu A Slamet, and Max Pfingsthorn. Robust weighted scan matching with quadtrees. In *Proc. of the Fifth International Workshop on Synthetic Simulation and Robotics to Mitigate Earthquake Disaster (SRMED 2009)*, 2009.