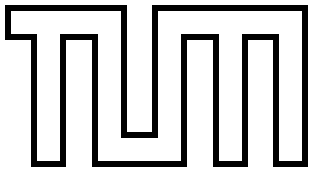


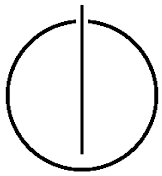
FAKULTÄT FÜR INFORMATIK

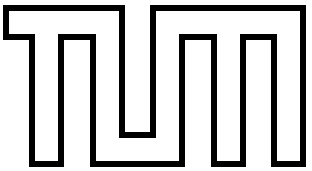
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

3D Object Reconstruction using Point Pair Features

Adrian Haarbach





FAKULTÄT FÜR INFORMATIK

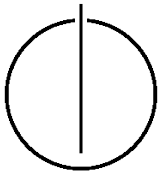
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

3D Object Reconstruction using Point Pair Features

3D Objekt Rekonstruktion mit Point Pair Features

Author:	Adrian Haarbach
Supervisor:	Wadim Kehl
Advisor:	PD Dr. Slobodan Ilic
Date:	March 15, 2015



I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, March 15, 2015 Adrian Haarbach

Acknowledgments

First of all, I would like to thank Prof. Dr. Nassir Navab and my supervisor PD Dr. Slobodan Ilic whose lectures inspired me to dive deeper into the interesting topic of 3D Computer Vision and who provided me with the possibility to pursue my thesis inside the most enjoyable research group at their chair.

Next, a big applause to my advisor Wadim Kehl, who never stopped pushing me forward during the long journey of this thesis. His fun attitude and language skills made working with him a real pleasure. The many suggestions during our fruitful discussions kept me moving and certainly pushed my limits.

Additionally, I would like to thank all the people that I met at Nassir's Computer Aided Medical Procedures chair at TUM, in particular in Slobodan's computer vision research group. I am most grateful that I was joined by Mira during our struggles for thesis perfection. Through her impressive rendering skills in Blender, she provided me with beautiful synthetic sequences. A highlight in our research life was certainly the day we recorded the new real sequences with David, after we nearly failed to pull off the old markers from the board to glue on the new ones. Special thanks to Tolga for giving valuable advice that accelerated my progress in implementing the point pair features after I found out that we had been working on similar topics.

Finally, I would like to thank my family for the support during my studies, my father for proof-reading this thesis, my friends for reminding me of the fun part of life and my beloved Florian for his patience, care and support.

Abstract

This work aims at reconstructing 3D objects by robustly and accurately registering multiple range images of an object from different viewpoints.

An initial alignment between any two overlapping scans is obtained via a voting scheme which matches similar point pair features and thus constrains the relative 6DoF rigid body motion between the poses of two viewpoints. This initial alignment is then refined using pairwise point-to-plane ICP. The result of this step is a tree of relative pose constraints.

In a subsequent global optimization step, we build up a graph of absolute poses, our vertices, from the tree of initial relative pose estimates by adding further edges. We add edges for the k-nearest-neighbors of a vertex, taking the translational difference of the corresponding poses as a distance measure. Constraints between two vertices are added for each closest point correspondence in their respective point clouds. The global point-to-plane energy is then minimized iteratively using the nonlinear least-squares method called Multiview Levenberg-Marquardt ICP.

This refined registration of all the scans used may now be integrated and their corresponding point clouds fused and then meshed to obtain the final reconstructed 3D object mesh.

Abbreviations

2.5D	2.5 dimensions: usually refers to a depth or range image
6DoF	6 degrees of freedom: usually a rigid body motion
ATE	Absolute Trajectory Error
BA	Bundle Adjustment
DoF	Degree(s) of Freedom
GICP	Generalized ICP
ICP	Iterative Closest Point
LM	Levenberg-Marquardt
LM-ICP	Levenberg-Marquardt ICP
NLS	Non-linear Least Squares
PCA	Principal Component Analysis
PPF	Point Pair Feature
RANSAC	RANdom SAmple Consensus
RGB-D	Red Green Blue - Depth: multimodal color and range image
RMS	Root Mean Squared
RPE	Relative Pose Error
SE(3)	Special Euclidean group of rigid body motions
SfM	Structure from Motion, *sometimes also* Structure and Motion
SO(3)	Special orthogonal group of rotation matrices
SLAM	Simultaneous Localization And Mapping
SVD	Singular Value Decomposition
TSDF	Truncated Signed Distance Function
VO	Visual Odometry

Contents

1. Introduction

Figure 1.1.: The different components of the 3D modeling process. [32]

3D Object Reconstruction has been an increasingly active topic of research since the introduction of cheap multimodal color and range (RGB-D) cameras like the Kinect. Those devices provide, together with the color image, a range image of the object. This is sometimes referred to as a depth or 2.5 dimensions (2.5D) image, since it contains depth information, yet only the part of the object that is visible from the corresponding camera location. So to reconstruct the whole object, one needs multiple of these shots from different viewing angles. The challenge in 3D Object Reconstruction is to find correspondences between these views, register them into a global coordinate system, integrate them into a single representation and then reconstruct the object mesh to get an actual 3D model of the object (figure 1.1). One can generally divide algorithms which tackle the problem of 3D Object Reconstruction into multiple categories.

The first category is also the fastest one, since it uses a markerboard on which to place the object, so there is always quite a good estimate of the relative camera location towards the object. However, this approach brings some serious practical limitations: one cannot reconstruct the bottom side of the object, since it is laying on the markerboard. Additionally, once the object is moved on top of the board, the poses between object and board don't correspond anymore. Also, only objects smaller than the markerboard can be reconstructed, since enough markers need to be visible for the pose estimation to work. Finally, some objects are just fixed or too heavy to be put on a board.

The second category uses the 2D color images and pixel-to-pixel correspondences to get an estimate of the relative motion between two overlapping frames. Dense methods calculate a flow field for each pixel and are usually called Visual Odometry (VO). Sparse methods only compute correspondences for a subset of the pixels and are the basis of most (although not all) Simultaneous Localization And Mapping (SLAM) based approaches due to their speed. Either dense or sparse, it is important that the object exhibits enough variation in texture, in the dense setting to avoid the aperture problem, in the sparse setting to get good corner points for feature detection which can then be used together with a feature descriptor for matching.

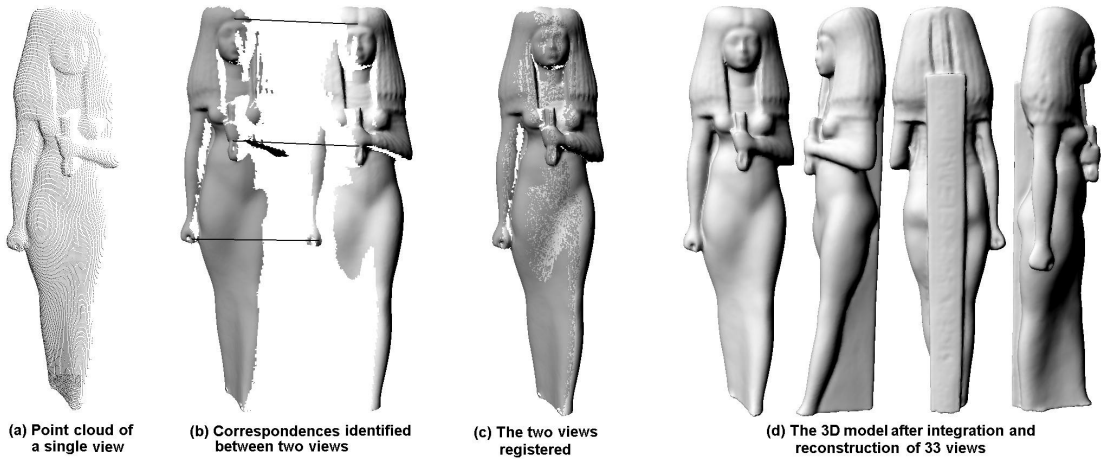


Figure 1.2.: Range image integration. [32]

The third category is range image integration (figure 1.2), which uses the 2.5D depth shots directly. What happens if it is dark, the object to reconstruct is textureless or has been acquired with a range scanning system rather than with an RGB-D camera? In this case, we can drop the useless color image and switch to a purely geometry based approach. These approaches mostly rely on some variant of the Iterative Closest Point (ICP) algorithm to iteratively make the 2.5D point clouds overlap. For this to work, however, there can only be a small motion between consecutive frames, otherwise the ICP algorithm doesn't converge and one loses track of the camera motion. The KinectFusion [35] algorithm is probably the most prominent recent technique in this category, which works good and efficient on live data since frames are acquired and processed in real time at around 30Hz. One must really shake the depth sensor drastically to violate the small motion assumption to lose tracking, so this approach needs a lot of frames to reconstruct one single object. Additionally, it is a Truncated Signed Distance Function (TSDF) volume based approach, meaning that new frames are readily integrated into the global volume as they come in. This is necessary to keep space requirements of such a real time algorithm low, but is not optimal for the most accurate reconstruction of an object, since the model volume necessarily accumulates drift that can't be optimized away at a later stage.

We propose a new method for range image integration that eliminates most of the shortcomings of previous methods while only using a few range images. The only condition they should fulfill is that the object is sufficiently covered and that there exist areas of overlap between pairs of frames. We make no assumption about the relative motion between consecutive frames, although our algorithm is accelerated by telling him to how many of the last frames he should try to match the new frame. We do not use a volumetric model representation to integrate the registered view, but rather just assign each frame its estimated camera pose (like it is done in SLAM approaches), which we further optimize for. This allows us to subsequently reduce the drift in pairwise and multiview refinement stages, based on optimizing the pose graph of the cameras associated to each view, which essentially minimizes the 3D alignment errors defined on the edges in this graph.

2. Previous work

2.1. 3D object recognition

The task in **3D object recognition** is to recognize a known 3D model, usually given as a mesh or a point cloud, in a scene. The more sophisticated algorithms additionally give a 6 degrees of freedom (6DoF) **pose estimate** of the viewpoint from which the scene was taken in relation to the model.

This can be done in various ways, depending on the format of the scene: camera image, range image, or a combination of both called RGB-D image, which is a pair of registered multimodal color and depth images. A nice overview and taxonomy of the different classes of methods is given in the related work section of [19] and briefly summarized and extended here.

Camera image based object detection algorithms can be divided into 2 classes. *Learning* based approaches like the well known Viola Jones Face Detector [24] use a large amount of training data and work well for recognizing object classes like faces, cars or chairs. However, they usually don't give a pose estimate. *Template* based approaches like LINEMOD [18] usually render the high-quality mesh model from different viewpoints with known ground truth and then match these templates with image patches, using some sort of 2D feature descriptor like Scale Invariant Feature Transform (SIFT), Speeded-Up Robust Features (SURF), Oriented FAST and Rotated BRIEF (ORB), Binary Robust Independent Elementary Features (BRIEF) or Histogram of oriented gradients (HOG). These provide a quantized pose estimate.

Range image based object detection and correspondence estimation is reviewed in [32]. The standard approach for object pose estimation, ICP, needs a good initial estimate and is thus not suitable for object detection, but often used as a subsequent pose refinement step after the object is detected. More recent, robust approaches for object detection, which do not require a good initial estimate, all use 3D locally invariant features - based on point configurations, surface normal distributions around a point, surface curvature or relative angles between normals. These approaches have names like spin-image [23], point-pair [33], Point Pair Feature (PPF) [8] and point-pair histograms [40, 47].

RGB-D image based algorithms use the multi-modal information from a registered camera and range image to improve correctness and precision. An extension of LINEMOD [18] to also include range information is given in [19]. In the opposite way, an extension of the PPF [8] to the Color Point Pair Feature is given in [6].

2.2. 3D object reconstruction

Tracking, Mapping, Simultaneous Localization And Mapping (SLAM), Structure from Motion (SfM) and *Visual Odometry (VO)* are all different terms coming from different communities for basically the same problem: maintaining an estimate of the current camera position (tracking / motion) and of the environment (mapping / structure) while moving a camera in space. These ingredients, together with a final surface reconstruction step, are also the recipe for most **3D object reconstruction** algorithms.

The basic steps of the 3D model acquisition pipeline, consisting of view planning, registration and surface reconstruction were first described in great detail in the fundamental work [39] and can be found in nearly all of the follow-up work on 3D object reconstruction. The authors built their own structured light projector and camera setup to capture range images from multiple views and then register them with each other using ICP [50] for alignment. Finally, they merge and render the model. This approach uses only **range images**.

Recently, real-time, dense, solely depth-based reconstruction algorithms like KinectFusion [35] became quite popular. The main drawback of these depth-based approaches is the small motion assumption between consequtive frames, which is needed for ICP to converge. This is usually given if the frames are aquired and processed in real time and the camera is moved smoothly, but not if one wants to reconstruct an object from a few depth images, with no prior knowledge about their initial alignment. However, SLAM++ [41] circumvents this problem by doing a frame to model tracking instead of a frame to frame tracking, for which they first detect known objects in the scene using a variant of [8], and then track each new camera frame against the detected objects.

Another option, which was especially common before the introduction of consumer depth cameras like the Kinect around 2009, is to rely only on the **color images**. The goal of dense methods is to calculate a flow field for each pixels that maximizes the photo-consistency between multiple images brought into reference. A survey of these techniques can be found in [43] and in the Mutliple View Geometry book [16]. Sparse methods are usually faster, but less accurate, and are thus used in most SLAM algorithms, including Parallel Tracking and Mapping (PTAM) [27] and Large-Scale Direct Monocular SLAM (LSD-SLAM), where the latter one is actually semi-dense, a hybrid between sparse and dense. But even with cheap RGB-D cameras, solely image based reconstruction techniques are still quite useful, especially in outdoor environments where structured light sensors fail due to infrared radiation and on mobile devices which don't (yet) have integrated RGB-D cameras. A quite recent work in this area which even runs on mobile CPU's is LSD-SLAM [11]. The main drawback of these photo-consistency based approaches is the assumption of sufficiently textured Lambertian surfaces and good lightning conditions. This can be troublesome in industrial applications where most workpieces are uniformly colored, maybe even out of shiny metal and with no control on the lightning.

As in object recognition, **RGB-D** based approaches have also been introduced in tracking, including dense methods like [45], Dense Visual Odometry (DVO)-SLAM [25] or sparse methods like RGB-D-SLAM[10]. These combine the advantages of the multiple modalities and are thus more stable and accurate in general.

2.3. Multiview refinement

Most SLAM algorithms build up a pose graph of camera positions and landmarks, connected by some sort of constraints, in their main thread. In the background, they usually run loop closure detection and **pose graph optimization** to globally reduce the accumulated drift from the pairwise alignment.

The problem of drift also occurs in 3D object reconstruction based on pairwise alignment, which is why the last step in algorithms that aim for accuracy is to refine the pose graph of the camera pose estimates based on a global registration error formulation. We call this step **multiview refinement**, in which the error function depends on the camera poses only and we try to reduce the 3D registration error of point correspondences. This is related to Bundle Adjustment (BA), in which in addition to the camera poses, the camera parameters are adjusted and the 2D reprojection error in each image pair is minimized instead.

In our approach, we use range images only, for which ICP is the method of choice to reduce pairwise 3D registration errors. But since standard ICP - with its closed form solution - only works for pairwise registration, [12] proposed an extension of ICP for multiple views, called Multiview Levenberg-Marquardt ICP (LM-ICP). Reducing the global registration error, which sums up the registration error from all pairs of frames, is done via Non-linear Least Squares (NLS) optimization.

The relationship between different frames can easily be expressed in graph form, e.g. by an adjacency matrix. Kummerle et al. [28] provides a generic framework for graph based optimization, which is used in most of the previously discussed SLAM Algorithms like [41, 25, 11, 10]. To adapt this framework to our need of solving the multiview refinement problem, one has to define vertices and edges. In our context, a vertex is a camera pose in the global reference frame. An edge is a closest point correspondence - as in ICP - between pairs of overlapping point clouds.

3. Background

3.1. Range images

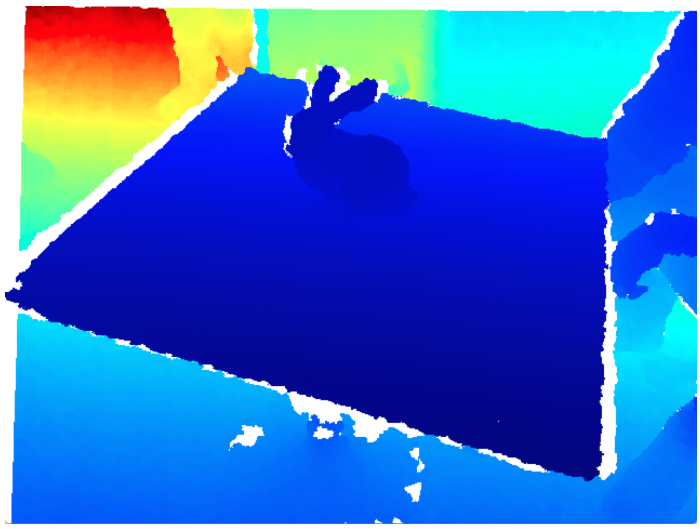


Figure 3.1.: **Range image** acquired with a structured light sensor. The depth at each pixel is color coded where blue means close and red means far away. Note that for the invalid pixels in white, the sensor is unable to obtain a measurement. This happens usually at object boundaries or at reflective or transparent surfaces.

A range image is a 2D image where each pixel denotes the distance between the sensor and a point in the scene. In calibrated systems, this distance can actually be given in metric units. Since a range image provides the partial 3D information of an object from a specific viewpoint, organized in a 2D image grid, these images are sometimes called 2.5D images. The most prominent techniques to acquire such images are stereo triangulation, sheet of light triangulation, structured light or time of flight cameras. In recent years, the structured light sensors got a boost through the introduction of consumer hardware such as Microsoft's Kinect and similar devices with names such as PrimeSense or Asus Xtion. These devices usually combine the depth or range image with a color image, typically encoded in the RGB color space, and are thus called RGB-D cameras. For our purposes, we are only interested in the range images, so the input data can be acquired by any of the techniques mentioned above.

3.2. Pinhole camera model

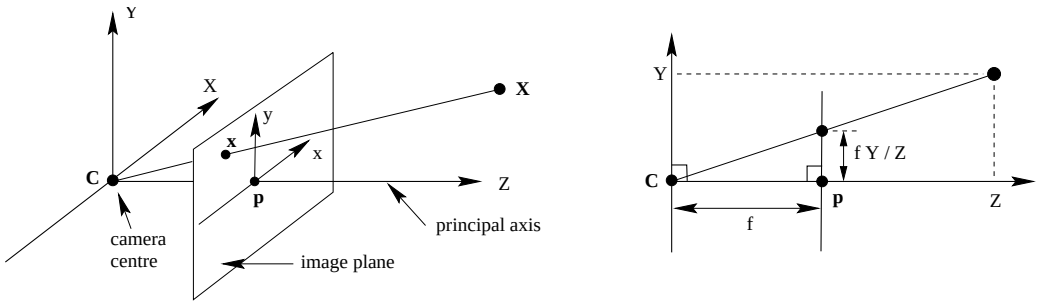


Figure 3.2.: The pinhole camera model [16][p.154]

The process by which the real 3D points of a scene are mapped into the 2D image plane can easily be modeled with the pinhole camera model. This amounts to a central projection of the 3D points onto the image plane, with the camera center **C** as the center of projection. The line from the camera centre perpendicular to the image plane is called the principal axis, and the point where the principal axis meets the image plane is called the principal point **p**. The distance between camera center and image plane is the focal length f. Since pixels need not to be quadratic, two different focal lengths f_x and f_y are used in practice. Usually, the origin of the camera coordinate system (the upper left corner of an image) and the camera center **C** do not coincide, which is why we need the offsets o_x and o_y. Hartley and Zisserman [16][Chapter 6.1] wrote the de facto reference for this topic.

The quadruple (f_x, f_y, o_x, o_y) is called the intrinsic parameters of a camera and can be obtained via standard calibration procedures. In this process, skew as well as radial distortion parameters can also be obtained. We assume that before processing the images, these effects have been removed so that the pinhole camera model assumption holds true.

Multiplication of the camera calibration matrix K, which contains these intrinsics, with a 3D-world point in inhomogeneous coordinates $\mathbf{X} = (X, Y, Z)^T$ yields a 2D-image point in homogeneous coordinates, which has to be divided by its depth Z to get its inhomogenous pixel coordinates $\mathbf{x} = (u, v)^T$:

$$K * \mathbf{X} = \begin{bmatrix} f_x & & o_x \\ & f_y & o_y \\ & & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} f_x X + o_x Z \\ f_y Y + o_y Z \\ Z \end{pmatrix} \equiv \begin{pmatrix} \frac{f_x X}{Z} + o_x \\ \frac{f_y Y}{Z} + o_y \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix} = \mathbf{x} \tag{3.1}$$

Note that this central projection, a mapping $P_K : \mathbb{R}^3 \to \mathbb{R}^2$, is in general not invertible for standard color cameras, since one loses the depth information because of the division through Z. However, range cameras acquire and store these depth values in the range image d at each pixel location $d(u, v) = Z$. With this, we can easily back-project a 2D pixel $\mathbf{x} = (u, v)^T$ into a 3D point **X**, which is a necessary preprocessing step for our algorithm:

$$P_K^{-1}(u, v, Z) = \left(\frac{(u - o_x) * Z}{f_x}, \frac{(v - o_y) * Z}{f_y}, Z \right)^T = (X, Y, Z)^T = \mathbf{X} \tag{3.2}$$

3.3. Rigid body motion

Most of the derivation here follows Ma et al. [30][Chapter 2][1] closely, which we recommend for further details.

A rigid body motion is a transformation in 3D Euclidean space which preserves distance and orientation between any pair of points on the object. It is formalized by the map

$$\begin{aligned} g : \mathbb{R}^3 \to \mathbb{R}^3; x \to g(x) & \\ ||g(p) - g(q)|| = ||p - q|| & \qquad \forall p, q \in \mathbb{R}^3 \\ g(p) \times g(q) = g(p) \times g(q) & \qquad \forall p, q \in \mathbb{R}^3 \end{aligned}$$

The collection of all these transformation forms a Lie Group, called the Special Euclidean group of rigid body motions (SE(3)). Each $g = (R, t) \in SE(3)$ consists of a translational part t and a rotational part R and has 6 degrees of freedom (6DoF) in total. The translational part is easily minimally parameterized by a displacement vector $t \in \mathbb{R}^3$, while there exist multiple parameterizations for the rotational part.

3.3.1. Rotation and transformation matrices

One way is via 3 x 3 rotation matrices $R \in SO(3)$ in the Special orthogonal group of rotation matrices (SO(3)), which must fulfill $RR^T = I$ and $|R| = 1$. From orthogonality, it easily follows that $R^{-1} = R^T$. The whole $SE(3)$ transformation g as well as its inverse g^{-1} can thus be expressed by the 4×4 matrices T and T^{-1}:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}, T^{-1} = \begin{bmatrix} R^T & -R^T t \\ 0 & 1 \end{bmatrix} \tag{3.3}$$

This representation comes in handy when points need to be transformed, since one has to carry out a simple matrix vector multiplication. A point $x \in \mathbb{R}^3$, given as a column vector $x_h = [x_1, x_2, x_3, 1]^T$ in homogeneous coordinates is transformed via $T * x_h$. Moreover, multiple transformations can be composed by simple matrix matrix multiplication of their homogeneous 4×4 matrix representations.

However, the rotation matrix R consists of 9 parameters, even though it only has 3 degrees of freedom (DoF). This means we can only freely choose 3 entries that automatically determine the 6 remaining entries through the conditions $RR^T = I$ and $|R| = 1$, which is inconvenient for several reasons and which is why we also use several other parameterizations of rotations.

First of all, we need to store 12 parameters (the last row of T can be omitted, since it is always the same for rigid body motions), while only having 6 free parameters.

3.3.2. Unit quaternions

Secondly, we will need to cluster and average a set of rigid body transformations in our approach. While this is easy for the translational part, since all translation vectors themselves form an Euclidean space with an intuitive definition of distances and averages, this is hard

[1] http://vision.ucla.edu/MASKS/MASKS-ch2.pdf

for rotation matrices $R \in SO(3)$ since we neither have a definition of distance between two rotations matrices nor are they closed under addition, which is needed for calculating averages. For this reason, we will revert to an alternative representation of the rotational part of the rigid body motion with **unit quaternions**.

A quaternion q is basically the extension of a complex number $c = a + bi, i^2 = -1$ from 2 to 4 dimensions: $q = w + xi + yj + zk, i^2 = j^2 = k^2 = ijk = -1$. Just as complex numbers can be used to represent rotations in $\mathbb{R}^2$ (remember Euler's formula $\mathrm{e}^{\mathrm{i}\,\varphi} = \cos(\varphi) + \mathrm{i}\,\sin(\varphi)$), quaternions can be used to represent rotations in $\mathbb{R}^3$.

Formally, a quaternion $q \in \mathbb{H}$ may be represented by a vector $q = [q_w, q_x, q_y, q_z]^T = [q_w, q_{x:z}]^T$ together with the definitions:

$$\begin{aligned} \text{adjoint}: \quad & \bar{q} = [q_w, -q_{x:z}]^T \\ \text{norm}: \quad & ||q|| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} \\ \text{inverse}: \quad & q^{-1} = \frac{\bar{q}}{||q||} \end{aligned}$$

A unit quaternion is a quaternion with unity norm, $||q|| = 1$ and can be used to represent the orientation of a rigid body in 3D Euclidean space. Specifically, a unit quaternion can be retrieved from the axis-angle representation, yet another commonly used representation for rotations, with a rotation ϕ about the normalized rotation axis $n, ||n|| = 1$ via

$$q(\phi, n) = [\cos(0.5\phi), n\sin(0.5\phi)]^T$$

Moreover, there are closed form solutions for converting a unit quaternion into a rotation matrix as well as the other way around which we will use extensively. Since they don't look as neat as the previous formula, we refer to [7] for the details.

Quaternion distance

$$\theta(q_1, q_2) = \cos^{-1}\left(2\langle q_1, q_2\rangle^2 - 1\right) \qquad \in [0°, 180°] \tag{3.4}$$

$$d(q_1, q_2) = 1 - \langle q_1, q_2\rangle^2 \qquad \in [0, 1] \tag{3.5}$$

For pose clustering we need the notion of distance between two unit quaternions q_1 and q_2. It turns out that either one of the above quantities θ, d gives a rough estimate of the distance in orientation between two quaternions, while the angle θ or the dimensionless quantity d are smaller the closer the rotations are to each other.[2]

Quaternion average

For pose cluster averaging we need the notion of an average orientation, given a set of similar orientations. The findings in [15] suggest that the component-wise mean of multiple unit quaternion vectors is indeed a good approximation of the average orientation. This is justified if the underlying orientations are assumed to be drawn from a Gaussian distribution, for which the quaternion mean is the least squares estimate of their linear

[2] http://math.stackexchange.com/questions/90081/quaternion-distance

approximation (subsection 3.3.4). However, before taking the mean, one has to make sure that the quaternions to average lie on the same half-sphere of $\mathbb{H}$, since they double-cover the space of rotations, meaning that q and $-q$ represent the same orientation. Given a set of similar orientations, represented by their quaternions $q_1, q_2, \ldots, q_n$, their average orientation is given by the renormalized mean quaternion, where we had to flip quaternions which live on the other side of the half sphere as the reference quaternion q_1

$$\bar{q}_{gramkow} = \frac{\tilde{q}}{||\tilde{q}||}, \quad \tilde{q} = \frac{1}{n}\sum_{i=1}^{n} q_i * sign(\langle q_1, q_i \rangle) \tag{3.6}$$

Another way to average orientations [31], which circumvents the problems of flipping and renormalization, is based on the eigenvalue decomposition of the symmetric matrix formed by the sum of the outer products of the quaternions. The average quaternion is then just the eigenvector corresponding to the largest eigenvalue of that matrix.

$$\bar{q}_{markley} = e_1(A), \quad A = \frac{1}{n}\sum_{i=1}^{n} q_i * q_i^T, \tag{3.7}$$

$$e_1(A) : \text{eigenvector corresponding to largest eigenvalue } \lambda_1 > \lambda_{2,3,4}$$

It is further possible to incorporate weights into these formulations to compute a weighted mean. Both of the above methods are reviewed and extended in [17].

3.3.3. Lie algebra of twists

Lastly, we will need to optimize over the space of rigid body motions in a numerical minimization scheme. For this, a minimal representation of g would allow us to do unconstrained minimization. While this can in principle also be done over the 7-Degree(s) of Freedom (DoF) combination of a translation vector with a unit quaternion, where we would have to re-normalize the quaternion after each update step, it is much more elegant to do so via the 6DoF twist parameterization.

Each rigid body transformation matrix T in the Lie group $T \in SE(3)$ has a minimal representation as a twist $\hat{\xi}$ in its associated Lie algebra $\hat{\xi} \in se(3)$. Each twist is uniquely defined by its twist coordinates $\xi \in \mathbb{R}^6$:

$$\xi = (\xi_1, \ldots, \xi_6)^T = (u_1, u_2, u_3, \omega_1, \omega_2, \omega_3)^T = (u^T \omega^T)^T$$

where u represents the translational velocity and ω the rotational velocity.

Let us first define the operator $[.]_x$, which is an isomorphism between $\mathbb{R}^3$ and the space $so(3)$ of all 3x3 skew symmetric matrices ($[\omega]_x = -[\omega]_x^T$).

$$[.]_x : \mathbb{R}^3 \to so(3) \subset \mathbb{R}^{3\times 3}; [\omega]_x = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}$$

It allows to express the cross product $p \times q$ with $p, q \in \mathbb{R}^3$ as a matrix-vector multiplication: $p \times q = [p]_x * q$. Furthermore, it allows to convert from twist coordinates to a twist using

the hat operator:

$$\wedge : \mathbb{R}^6 \to se(3) \subset \mathbb{R}^{4\times4}; \hat{\xi} = \xi^{\wedge} = \begin{bmatrix} u \\ \omega \end{bmatrix}^{\wedge} = \begin{bmatrix} [\omega]_x & u \\ 0 & 0 \end{bmatrix}$$

The mapping from the twist in the Lie algebra to the transformation matrix in the Lie group is done by matrix exponentiation:

$$\begin{gathered} exp : se(3) \to SE(3) \\ exp(\hat{\xi}) = exp\left(\begin{bmatrix} [\omega]_x & u \\ 0 & 0 \end{bmatrix} \right) = \begin{bmatrix} e^{[\omega]_x} & Vu \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \end{gathered} \tag{3.8}$$

We can convert the vector $\omega \in \mathbb{R}^3$, which represents rotational velocity, into an axis angle representation by $\phi = ||\omega||, n = \frac{\omega}{\phi}$, to obtain a closed form solution for the Taylor series expansion $e^{[w]_x} = \sum_{i=0}^{\infty} \frac{[\omega]_x^i}{!i}$ using Rodrigues' rotation formula:

$$e^{[w]_x} = e^{[n]_x \phi} = I + sin\phi[n]_x + (1 - cos\phi)[n]_x^2$$

and similarly for V

$$V = I + \frac{1 - cos\phi}{\phi}[n]_x + \frac{\phi - sin\phi}{\phi}[n]_x^2$$

To get from twist coordinates $\xi \in \mathbb{R}^6$ to a transformation matrix $T \in SE(3) \subset \mathbb{R}^{4\times4}$ one first has to apply the hat operator $\wedge$ and then the exponential map exp. The other direction is possible as well, by first applying the inverse of the exponential map, called the logarithmic map log, and then the inverse of the hat operator, called the vee operator $\vee$. For our application however, we only need the forward direction, which is why just summarize the different operators here:

$$\xi \in \mathbb{R}^6 \underset{\vee(vee)}{\overset{\wedge(hat)}{\rightleftarrows}} \hat{\xi} \in se(3) \underset{log}{\overset{exp}{\rightleftarrows}} T \in SE(3)$$

Every Lie group, such as SO(3) and SE(3), is a group that is also a smooth manifold, with the property that the group operations of multiplication and inversion are smooth maps. They can locally be approximated by their corresponding Lie algebras $so(3)$ and $se(3)$, which form the tangent space of the group at the identity. This allows one to do calculus on the elements of the Lie algebra, such as calculating derivatives, which we will need for numerical minimization.

In the Lie algebra, a point $x \in \mathbb{R}^3$ is transformed by a twist via [30][p.32] :

$$\begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} [\omega]_x & u \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} = \begin{bmatrix} & -\omega_3 x_2 & +\omega_2 x_3 & +u_1 \\ \omega_3 x_1 & & -\omega_1 x_3 & +u_2 \\ -\omega_2 x_1 & +\omega_1 x_2 & & +u_3 \\ & & 0 & \end{bmatrix} \Leftrightarrow y = w \times x + u$$

The Jacobian of the twist $\hat{\xi}$ is calculated by partial derivates of $y = [y_1, y_2, y_3]^T$ and the

twist coordinates $\xi = (u_1, u_2, u_3, \omega_1, \omega_2, \omega_3)^T$ [44] :

$$J = \frac{\partial y}{\partial \xi} = \begin{bmatrix} \frac{\partial y_1}{\partial \xi_1} & \cdots & \cdots & \frac{\partial y_1}{\partial \xi_6} \\ \vdots & & & \vdots \\ \frac{\partial y_3}{\partial \xi_1} & \cdots & \cdots & \frac{\partial y_3}{\partial \xi_6} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & x_3 & -x_2 \\ 0 & 1 & 0 & -x_3 & 0 & x_1 \\ 0 & 0 & 1 & x_2 & -x_1 & 0 \end{bmatrix} \tag{3.9}$$

3.3.4. Infinitesimal rotations

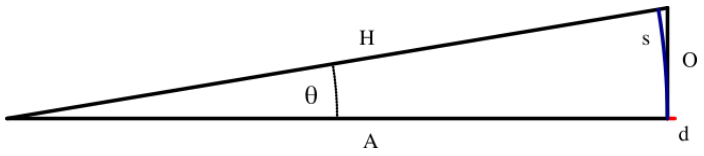


Figure 3.3.: Small angle triangle

Consider the triangle in fig. 3.3, together with the definitions of $sin(\theta) = \frac{O}{H}$, $tan(\theta) = \frac{O}{A}$ and $cos(\theta) = \frac{A}{H}$, where $d = H - A$. For small angles $|\theta| << 1$, it holds that d is quite small, so $H \approx A$ and we can use the following small angle approximation:

$$\sin(\theta) = \frac{O}{H} \approx \frac{O}{H} = \tan(\theta) = \frac{O}{A} \approx \frac{s}{A} = \frac{A * \theta}{A} = \theta$$
$$\cos(\theta) = \frac{A}{H} \approx \frac{A + d}{H} = \frac{H}{H} = 1$$

Analytically, this stems from dropping the second and higher order terms of the Taylor series development of the trigonometric functions around 0 for $|\theta| << 1$:

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - + \cdots, \qquad \cos\theta = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - + \cdots$$

In 2D, a Rotation Matrix which rotates a given point by the small angle $|\theta| << 1$ around the origin can thus be approximated as:

$$R_{2D}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \approx \underbrace{\begin{bmatrix} 1 & -\theta \\ \theta & 1 \end{bmatrix}}_{r_{2D}(\theta)} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{I} + \underbrace{\begin{bmatrix} 0 & -\theta \\ \theta & 0 \end{bmatrix}}_{S}$$

which is what we call an infinitesimal rotation matrix $r_{2D}(\theta)$. They are just the sum of the identity I and a skew-symmetric matrix $S = -S^T$. Note that up to second order, $r_{2D}(\theta)$ is really an orthonormal matrix $|r_{2D}| = 1 + \theta^2, r_{2D}^T \cdot r_{2D} = I + diag(\theta^2)$ and thus defines a rotation.

In contrast to normal rotations, infinitesimal rotations have some favorable characteristics. They are commutative, which means it doesn't matter in which order one applies the different rotations, and the inverse of $r_{2D} = I + S$ is simply given as $r_{2D}^T = I - S$. If one does the corresponding calculations[3], one sees that the appearing θ^2 can be dropped.

[3] http://mathworld.wolfram.com/InfinitesimalRotation.html

This concept generalizes to 3D, where any rotation can be defined by rotation of an angle θ around an axis n (axis angle representation). The approximation one obtains is[4]:

$$R(n,\theta) \approx I + \theta[n]_x \quad (3.10)$$
infinitesimal rotation

Let us now again have a closer look at the rotation group SO(3), a subgroup of SE(3), and its corresponding Lie algebra $so(3)$, which is made up of all skew-symmetric matrices as we saw in the last part. The matrices in the Lie algebra are not themselves rotations; the skew-symmetric matrices are derivatives, proportional differences of rotations. An actual differential rotation, or infinitesimal rotation matrix has the above form $I + \theta[n]_x, |\theta| << 1, n \in \mathbb{R}^3$. This means we just found the derivation of the elements $\theta[n]_x$ in our Lie algebra $so(3)$ as well as the approximation involved. It is important to note is that commutativity also generalizes to 3D. This is the reason why our mean-based quaternion averaging (3.6) of similar rotations can actually work. Infinitesimal rotations play an important role in point cloud registration as we will see later. There, we make extensive use of linearizing the non-linear space of rotations around the identity to get practicable algorithms for minimization.

3.4. Principal component analysis

To describe a set of scalar values $\mathcal{S} = \{s_i\} = \{s_1, s_2, ..., s_{N_\mathcal{S}}\}, s_i \in \mathbb{R}$ more compactly, one can compute statistical measures such as the mean and the variance. The mean $\bar{s} = \frac{1}{N_\mathcal{S}} \sum_{i=1}^{N_\mathcal{S}} s_i$ describes the central value of the distribution of the values, while the variance $Var(\mathcal{S}) = \frac{1}{N_\mathcal{S}} \sum_{i=1}^{N_\mathcal{S}} (s_i - \bar{s})^2$ describes how far the set of numbers are spread out. For higher dimensional data, e.g. a set of 3-dimensional points $\mathcal{P} = \{p_i\} = \{p_1, p_2, ..., p_{N_\mathcal{P}}\}, p_i \in \mathbb{R}^3$, the mean generalizes to the component-wise mean, called the centroid $\bar{p} \in \mathbb{R}^3$ and the variance generalizes to the covariance matrix:

$$Cov(\mathcal{P}) = \frac{1}{N_\mathcal{P}} \sum_{i=1}^{N_\mathcal{P}} (p_i - \bar{p})(p_i - \bar{p})^T = C, C \in \mathbb{R}^{3\times 3}$$

Its 9 entries not only describe how much the points deviate from their centroid, but also the amount of correlation between the different dimensions.

Principal Component Analysis (PCA) is a statistical method which converts a set of correlated variables into a set of linearly uncorrelated ones, using an orthogonal basis transformation. This is done by eigenvalue decomposition of the covariance matrix C. The eigenvectors of the covariance matrix are called principal components and point in the direction of largest variance, measured by their associated eigenvalues. Being the eigenvectors of the symmetric covariance matrix, the principal components are orthogonal and thus uncorrelated to each other. The transformation is defined in such a way that the principal components $e_1, e_2, e_3 \in \mathbb{R}^3$ are ordered by the amount of variance in their direction, i.e. the magnitude of their corresponding eigenvalues $\lambda_1 > \lambda_2 > \lambda_3 \in \mathbb{R}$.

[4] http://rotations.berkeley.edu/?page_id=1682

3.4.1. Normal estimation

Point clouds acquired by range cameras usually represent dense surfaces, so neighboring points can be assumed to lie approximately in a common plane. The PCA of these neighboring points can be used for the estimation of the surface normal at that point. The first two principal components e_1, e_2 span the plane tangent to the point, while the third principal component, e_3 is the normal of that plane. This is due to the fact that its corresponding eigenvalue λ_3 is the smallest of all three, since the variation of points approximately lying in a plane is smallest in the direction of the plane normal. For every tangent plane, there are two possible normals which point in opposite directions. The application of PCA usually outputs normals that lie on both sides of the surface. Our application requires the normals to be oriented consistently over the whole surface, meaning that the angle between any two neighboring normals should be smaller than 90 degrees. On unorganized 3D point clouds, this problem is NP-hard, but can be solved approximately using a normal flipping scheme [20]. Luckily, we are dealing with 2.5D data acquired by range cameras and unprojected into the camera coordinate system, with the optical center as the origin. Because we can only capture points in front of the camera, it holds for every point $\mathbf{X} = (X, Y, Z)^T$ that $Z = d(u, v) > 0$. To orient all normals towards the camera, which corresponds to the outside direction of the object's surface, we have to flip all normals whose angle with the positive z axis in the camera coordinate system is smaller than 90 degrees. This can be done by checking the value of the dot product $(e_{3,x}, e_{3,y}, e_{3,z}) \cdot (0, 0, 1)^T = e_{3,z} > 0$. So we simply multiply all normals that have a positive z component with -1, which gives us consistently oriented normals. For convenience, we normalize the normals to unit norm. All together, the surface normal n of a point set is:

$$n(\{p_i\}) = -sign(e_{3,z}) \frac{e_3}{||e_3||} \tag{3.11}$$

3.4.2. Curvature estimation

Another useful feature of a point lying on a surface is the local surface curvature of the surface at that location, i.e. the amount of convexity or concavity. For analytic, parameterized curves embedded in $\mathbb{R}^2$, this is the norm of the second derivative of the curve with respect to its parameters. For surfaces, which are embedded in $\mathbb{R}^3$, the definition of curvature is more involved, since the local curvature depends on the direction we are taking it in. One way to calculate the directional curvatures is to rotate a plane around the normal of the point, as calculated before, and intersect it with the surface. The intersection is a curve in $\mathbb{R}^2$ for which the curvature can be computed. However, there are an infinite amount of directions and thus directional curvatures. If we choose the ones with the highest and the lowest curvatures, we get the principal curvatures of the surface. To arrive at a scalar valued function for surface curvature, principal curvatures are averaged to yield mean curvature, or their product is taken, giving Gaussian curvature [26]. On point clouds, PCA can again be used to get an estimate of the mean curvature of the underlying surface. In [36], the authors approximate the mean curvature at a point p with what they call the surface variation $\sigma(\{p_i\})$ based on N neighbors. It is defined as the ratio of the largest eigenvalue of the covariance matrix, which quantitatively describes the variation

along the surface normal, to the sum of all eigenvalues:

$$\sigma(\{p_i\}) = \frac{\lambda_1}{\lambda_1 + \lambda_2 + \lambda_3} \tag{3.12}$$

A value of $\sigma = 0$ indicates a flat region, while the maximum value $\sigma = 1/3$ is attained for totally isotropically distributed points.

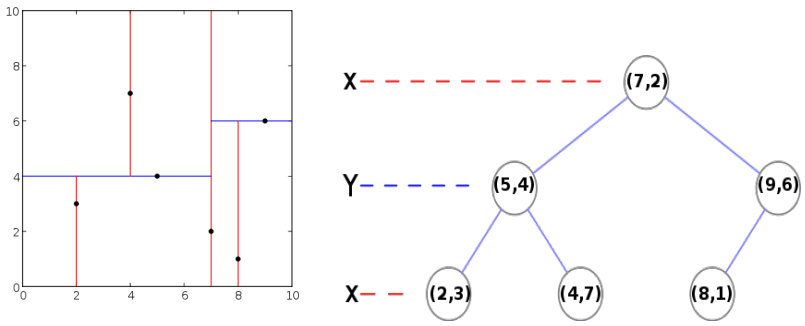
3.5. Nearest neighbor search

The computation of nearest neighbors is an important step in the inner loop of the ICP algorithm (section 3.6). Given a query point a and a point set $\mathcal{B} = \{b_i\} = \{b_1, b_2, ..., b_{N_\mathcal{B}}\}, b_i \in \mathbb{R}^3$ of $N_\mathcal{B}$ points, the problem consists in finding the point $b_i \in \mathcal{B}$ that is closest to a [26]. There are several strategies for this nearest neighbor lookup which differ in their running times and requirements for space and structure of $\mathcal{B}$.

3.5.1. Exhaustive search

This method just calculates all the distances from a to each point in $\mathcal{B}$ and chooses the shortest one. The runtime is $O(N_\mathcal{B})$, but there are no necessary preprocessing steps.

3.5.2. k-d tree



(a) median splits (b) the resulting tree

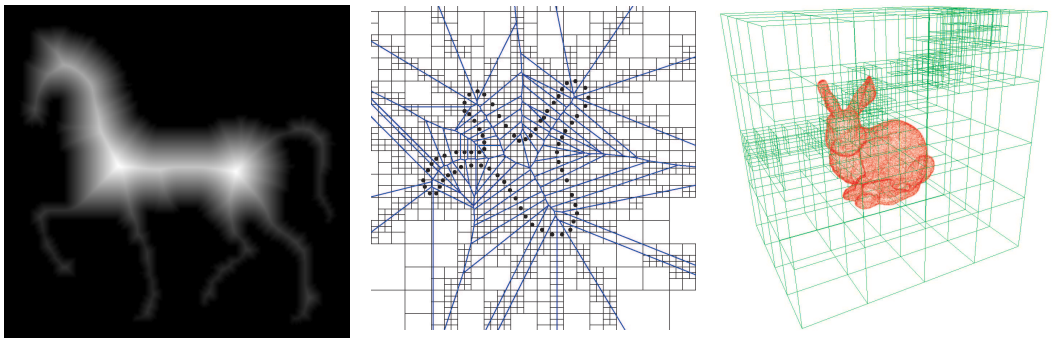
Figure 3.4.: A k-d tree[5] of $(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)$

This *binary space partitioning* method needs to precompute an efficient search structure, called a *k-d tree*, before doing the actual nearest neighbor lookup. It can be thought of as an extension of binary search to multiple dimensions, where each inner node represents a hyperplane that divides the space into two parts. A *k-d tree* is built by first splitting the point set $\mathcal{B}$ along the median of its first dimension, which becomes the root node. Next, the remaining two subsets are split according to their second dimension and so on. Once all dimensions are used up, one starts again with the first dimension. This process generates a balanced binary tree of height $O(\log(N_\mathcal{B}))$ in time $O(N_\mathcal{B} \log(N_\mathcal{B}))$ in the best case.

[5] http://en.wikipedia.org/K-d_tree

The actual lookup consists of moving down the tree by comparing the split dimension of each node with that of the query point, saving the leaf node as *current best*, and then moving the tree up again while looking for an even closer point to a in a hypersphere around a with radius equal to the current nearest distance. Since the number of additional points to consider in this branch and bound procedure is constant, the overall lookup time is still bounded by the height of the tree, and thus in $O(\log(N_{\mathcal{B}}))$

3.5.3. Distance transform



(a) Euclidean distances to the border

(b) Indices of the closest point stored in a quadtree(2D) or octree(3D) [34]

Figure 3.5.: distance transform

Nearest neighbor search can also be done in $O(1)$, given one first precomputes a volumetric bounding box of the point set $\mathcal{B}$, which stores the index of the closest point for every voxel, through what is known as the *distance transform*. While this method requires a large amount of preprocessing and is certainly not space efficient, since one has to store the empty space around the point cloud just to fill it with indices, it provides very fast lookup times. This method was introduced in [13]. The memory efficiency issue can be reduced by the use of adaptive grid structures such as octrees [34]. The distance transform is also the basis for the Truncated Signed Distance Function (TSDF), which inspired a whole class of successful volumetric reconstruction algorithms like KinectFusion[35].

3.5.4. Projection-based matching

Another way to do the lookup in $O(1)$ is via an approximate nearest neighbor search, given the two involved point clouds are organized, meaning that they are acquired as range images from depth cameras. This is called *projection-based matching* and requires no preprocessing. It works by projecting the query point a into the image plane of the camera that acquired the point set $\mathcal{B}$, giving some not necessarily integer pixel coordinates (u, v), and then choosing the point b which corresponds to the pixel $\lfloor (u, v) \rfloor$. Note that this method does not return the exact closest point and also sometimes does not return a point at all since some pixels in the depth image are invalid. Due to the constant time lookup, the overall ICP algorithm will still run much faster, even though the correspondences obtained via this method do not offer the best convergence per iteration as stated in [38].

3.6. Iterative Closest Points

The optimal registration of point clouds is an important problem for which an extensive amount of research was done. Given a set of point to point correspondences $\{p_i \rightarrow q_i\}_1^N$, the goal is to find a rigid body transformation g that aligns p_i to q_i, meaning that the distances $g(p_i) - q_i$ are as small as possible. Formally, the optimal alignment is given by minimizing an error function of the following form:

$$E(g) = \sum_{i=1}^{N} l(d(g(p_i), q_i)) * w(g(p_i), q_i) \tag{3.13}$$

$$g : \mathbb{R}^d \rightarrow \mathbb{R}^d,\ g \in \text{SE(3)}$$

$$d : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^d,\ \text{distance function} \tag{3.14}$$

$$l : \mathbb{R}^d \rightarrow \mathbb{R},\ \text{loss function} \tag{3.15}$$

$$w : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R},\ \text{weight function} \tag{3.16}$$

In general, the set of point correspondences $\{p_i \rightarrow q_i\}_1^N$ is not already given, but needs to be obtained before minimizing (3.13). The overall problem of point cloud registration is thus to align a data $\mathcal{A} = \{a_i\} = \{a_1, a_2, ..., a_{N_\mathcal{A}}\}, a_i \in \mathbb{R}^3$ to a model $\mathcal{B} = \{b_i\} = \{b_1, b_2, ..., b_{N_\mathcal{B}}\}, b_i \in \mathbb{R}^3$ point cloud, without knowing any correspondences between their points beforehand. The key idea of the ICP algorithm, introduced by [4, 5, 50], is to use the closest point correspondences ($p_i \in \mathcal{A}, q_i \in \mathcal{B}$) between data and model point clouds. It can be summarized in two steps, which are iteratively repeated until convergence to the desired solution is achieved [42]:

1. compute **correspondences** $\{p_i \rightarrow q_i\}_1^N$ between two scans, i.e. for each data point a_i, transformed by the current estimate g, $p_i = g(a_i)$, find its closest model point $q_i = b_j$.

2. update the current **transformation** estimate g so that it minimizes an error metric defined on these correspondences, i.e. the distance between corresponding points.

Since both steps must reduce the error, convergence to a local minimum is guaranteed [13]. Furthermore, an easy termination criterion is met if the set of correspondences in 1. does not change. This means the previous transformation in 2. brought the energy to a local minimum, so the current transformation would be the identity.

3.6.1. Taxonomy

Over the years, there have been various modifications to the original algorithm. A taxonomy of the different variants is given by [38]:

1. **Selection** of some set of points in one or both input point sets.
2. **Matching** these points to compatible points in the other set.
3. **Weighting** the corresponding pairs appropriately.
4. **Rejecting** certain pairs as outliers.
5. **Error metric** definition on the set of inliers.
6. **Minimization** of the error metric.

Selection

Selection means to consider only a subset of the model and data points, which may be beneficial for the further steps. For instance one may filter out outliers stemming from noise in the input data or do a random or uniform subsampling to reduce computational complexity. In our approach we will use uniform subsampling together with outlier rejection based on small, isolated point sets.

Matching

Matching amounts to actually compute the correspondences $\{p_i \rightarrow q_i\}_1^N$ between the two point clouds $\mathcal{A} = \{a_i\} = \{a_1, a_2, ..., a_{N_\mathcal{A}}\}, a_i \in \mathbb{R}^3$ and $\mathcal{B} = \{b_i\} = \{b_1, b_2, ..., b_{N_\mathcal{B}}\}, b_i \in \mathbb{R}^3$. For each $p_i = g(a_i), a_i \in \mathcal{A}$, this amounts to a nearest neighbor search (section 3.5) in $\mathcal{B}$, so $N_\mathcal{A}$ of these in total. The choice of which search method one uses heavily depends on the application, speed and accuracy requirements as well as on the format of the point clouds. In the ICP part of KinectFusion [35], the authors used projection-based matching (subsection 3.5.4), while in [13], the distance transform was used.

Weighting

Correspondences may be weighted according to their compatibility, which is done by multiplying the loss (3.15) introduced by each correspondence with a factor w (3.16) that depends on p and q. For range images, this can be based on distance $w = 1 - \frac{d(p,q)}{d_{max}}$, normal direction $w = n_p * n_q$ or curvature $w = e^{-(c_p - c_q)^2}$ [26]. For the last two measures, it is necessary to compute additional features for each point p and q like normals (subsection 3.4.1) and curvature (subsection 3.4.2). Since [38] concluded that the the effect of weighting on convergence rate is small and highly data-dependent, we just use a constant weight of $w = 1$ to stay as general as possible.

Rejecting

The purpose of rejecting correspondences is to eliminate outliers, which may have a huge effect on least-squares optimization [38], as they disturb the assumption of Gaussian distributed data samples. A small amount of outliers may drag the final result away from the optimal solution. A simple rejection strategy is to prune correspondences whose point pair distances (in the sense of their L2 norm) are above a certain rejection threshold d_{max}. The choice of d_{max} represents a trade-off between convergence and accuracy. A low value results in bad convergence (the algorithm becomes *short sighted*); while a large value causes incorrect correspondences to pull the final alignment away from the correct value [42].

$$d_{max} \qquad \text{rejection threshold} \tag{3.17}$$

There are more sophisticated methods like rejecting the worst n percent of pairs, but for our purposes the threshold version works fine. Dropping correspondences is the reason why $N < N_{\mathcal{A}}$ in general. Note that rejecting certain pairs is equivalent to setting their weight to zero $w = 0$, which is the formulation used in [13, 42]. In that case $N = N_{\mathcal{A}}$.

Error metric

The error metric used, which is a combination of distance (3.14) and loss function (3.15), is the part in which the different methods differ the most, since the specific choice determines what is actually minimized and how minimization can be carried out.

Choices for loss functions are [16][p. 619]:

$$l(r) = r^2 \qquad \text{squared loss} \tag{3.18}$$

$$l(r) = |r| \qquad \text{absolute loss} \tag{3.19}$$

$$l_\delta(r) = \begin{cases} r^2 & \text{if } |r| < \delta \\ |r|2\delta - \delta^2 & \text{else} \end{cases} \qquad \text{Huber loss} \tag{3.20}$$

Choices for distance functions are (note that $p^* = g(p)$):

$$d_{p-p}(p^*, q) = p^* - q \qquad \text{point to point distance} \tag{3.21}$$

$$d_{p-pl}(p^*, q) = (p^* - q) \cdot n_q \qquad \text{point to plane distance} \tag{3.22}$$

$$d_{pl-pl}(p^*, q) = f(p^*, q, n_p^*, n_q) \qquad \text{plane to plane distance} \tag{3.23}$$

Minimization

Certain combinations of above functions have closed form solutions, either exact or approximate ones. Other combinations can only be solved iteratively. In the following, we will review the most prominent ones. Their energy functionals are obtained from (3.13). In all of them, the weighting function is set to $w = 1$ and the the correspondences $\{p_i \to q_i\}_1^N$ are already retrieved by ordering the matches from $\mathcal{A}_1^{N_{\mathcal{A}}}$ and $\mathcal{B}_1^{N_{\mathcal{B}}}$ and rejecting certain pairs with $d > d_{max}$, which means that $N < N_{\mathcal{A}}$ in general.

3.6.2. Point to point

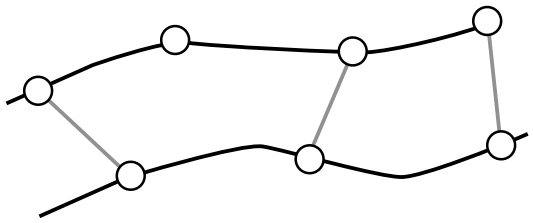


Figure 3.6.: point to point distance

The combination of (3.18) with (3.21) is the original and also most cited version of ICP, introduced by Besl and McKay [4], maybe due to the fact of the extensive examples and performance analysis of the basic algorithm. Almost simultaneously, Zhang [50] describes the same variant of ICP but adds a robust method for outlier rejection as in section 3.6.1, either based on a maximum distance threshold d_{max} which he obtains adaptively in a robust manner by analyzing distance statistics, or based on normal compatibility of the points, similar to what we discussed in section 3.6.1, where he prunes correspondences whose normal's angular difference is above the maximum angle of rotation expected between two frames. The written out form of this energy is:

$$E = \sum_{i=1}^{N} ||Rp_i + t - q_i||^2 \qquad \text{(3.24)}$$
point to point

At least 4 different closed form solutions for the minimization of this least squares problem exist, which are summarized in [9]. The first one is the Singular Value Decomposition (SVD) based approach of Arun et al. [2], which is usually always used in combination with Umeyama [48] to fix the original failure case of reflections. The second one of Horn [21] employs unit quaternions, while the third one [22], also from the same author, uses orthonormal matrices. The fourth approach of Walker et al. [49] uses dual unit quaternions.

While [4] uses the method based on unit quaternions [21] which automatically eliminates reflections ($|R| = -1$), we use the SVD approach [2], in which we just flip the last column of R if $|R| = -1$ as in [48], to ensure it is a rotation matrix and not a reflection. The details of the actual implementation can be found in appendix A.2.

Recently, there has also been an extension to the SVD based approach (Arun et al. [2], Umeyama [48]) to align two differently scaled point clouds in Zinßer et al. [51].

3.6.3. Point to plane

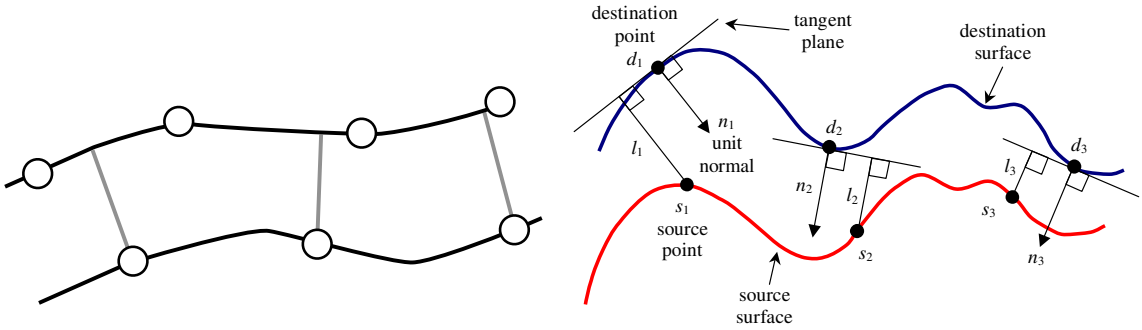


Figure 3.7.: point to plane distance, right one from [29]

The combination of (3.18) with (3.22) was introduced by Chen and Medioni [5], who considered the more specific problem of aligning range data for object modeling, which is thus very relevant for our work. They take advantage of the fact that most range data is locally planar and can thus be supplied with surface normals n_q for each of the model points q:

$$E = \sum_{i=1}^{N} ||(Rp_i + t - q_i) \cdot n_{q_i}||^2 \qquad (3.25)$$
point to plane

What is now minimized is the sum of distances of the data points to the tangent planes of the model points. This lets flat regions slide along each other, which turns out to be very advantageous for the alignment of our subsampled range scans: The location of points in two frames, describing a similar surface, may be different, but the actual surface they describe locally through their normals is still the same.

An approximate closed form solution can be obtained by locally linearizing the rotation matrix R, as we saw in subsection 3.3.4. Due to the iterative nature of ICP, the errors introduced by this linearization become small in late iterations, when rotations are expected to be small. The overdetermined set of equations of the approximation is a standard linear least-squares problem of the form $Ax = b$, which can be solved using the pseudo-inverse $x = A^{\dagger}b$ computed via SVD as in Low [29], or by Cholesky decomposition of the positive semidefinite matrix C, obtained either from the normal equations $Cx = d \Leftrightarrow A^T Ax = A^T b$ or, equivalently, by partial differentiation of the approximation as in Gelfand et al. [14], who additionally interpret C as a covariance matrix that can be used to describe the stability of this algorithm for different kinds of surfaces. The details of our actual implementation can be found in appendix A.3.

3.6.4. Generalized ICP

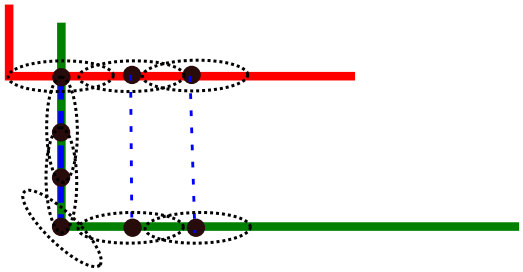


Figure 3.8.: plane to plane distance [42]

The distance metric (3.23) is the symmetric extension of (3.22), a function $f(p^*, q, n_p^*, n_q)$ of both points and both normals. The idea is to compute distances between tangent planes. However, this is not as trivial as one might think, because the distance between two planes in $\mathbb{R}^3$ is zero, unless they have exactly the same normal direction. To cope with this issue, the quite recent work of Segal et al. [42] introduced Generalized ICP (GICP), which puts the whole problem onto a probabilistic framework that additionally uses the covariance matrices C_p, C_q of model and data point, respectively. The idea is that we know the position of a point along its normal with high confidence $\epsilon < 1$, while we are unsure about its location in the plane. The covariance matrices $C_{p,q}$ are obtained by rotating this confidence estimate into the coordinate system defined by the normals via

$$C_{p,q} = R_{n_p,n_q} \cdot diag(\epsilon, 1, 1) \cdot R_{n_p,n_q}^T$$

where R_x denotes a rotation matrix that aligns the first unit axis with x: $R_x \cdot [1,0,0]^T = x$. The complete function to minimize is given as:

$$E = \sum_{i=1}^{N} d_{p-p}(Tp, q)^T (C_{p_i} + TC_{q_i}T^T)^{-1} d_{p-p}(Tp, q)^T \quad \text{plane to plane} \tag{3.26}$$

where T is the matrix form of the transformation g. This formula resembles the Mahalanobis distance, only the square root is missing. There is no closed form solution for the minimization of this energy, the authors used the iterative conjugate gradients.

GICP is considered as a unifying ICP variant, because it can also model the other two distance functions. Point to point is retrieved via

$$C_p = 0, C_q = I \tag{3.27}$$

and point to plane via

$$C_p = 0, C_q = P^{-1} \tag{3.28}$$

where P_i is the projection onto the span of the surface normal at q_i.

3.6.5. Levenberg-Marquardt ICP

Levenberg-Marquardt ICP (LM-ICP), introduced by Fitzgibbon [13], abandons the closed-form solutions for the inner loop of ICP and instead employs a standard iterative non-linear optimizer, the Levenberg-Marquardt (LM) algorithm. At no significant loss of speed, robust loss functions may now be easily incorporated, but also, more importantly, the minimization can be generalized to the multiview setting. Because the original paper deals with the two-frame case exclusively, we will review it here. The multiview extension will be given in 4.4. The basic idea is to minimize (3.13) via Non-linear Least Squares (NLS) optimization. The LM algorithm is an optimization procedure that is best suited to functions that are expressed as a sum of squared residuals, but it also works for robust loss functions as long as they are smooth [13]. Note that in the inner ICP loop the correspondences $\{p_i \rightarrow q_i\}$ are fixed, the only thing that changes is the transformation g. This means that (3.13) depends only on g. Using the squared loss, it can be rewritten as:

$$E(g) = \sum_{i=1}^{N} ||d(g(p_i), q_i)||^2 \tag{3.29}$$

Lets introduce the vector error function $e_i = d(g(p_i), q_i) \in \mathbb{R}^3$, which gives the distance vector of each correspondence, where the data point is transformed by the current estimate g. Rewriting the squared L2 vector norm $||.||^2$ using the dot product, the above can be rewritten as

$$E(g) = \sum_{i=1}^{N} e_i^T e_i \Leftrightarrow e^T e \tag{3.30}$$

The right side is obtained by stacking all of the N e_i's into e (3.32), the vector of residuals [13]. The LM algorithm combines gradient descent and Gauss-Newton approaches to function minimization. The goal at each iteration k is to choose an update $x = \Delta g$ to the current estimate g_k, so that the new estimate $g_{k+1} = g_k + x$ reduces the error $E(g)$. The idea is to approximate the vector error functions e_i by their first order Taylor series expansion:

$$e_i(g + x) \approx e_i(g) + \nabla e_i(g)x = e_i + J_i x \tag{3.31}$$

Here, $J_i = \frac{\partial e_i}{\partial g} = \nabla e_i(g) \in \mathbb{R}^{3\times p}$ is the Jacobian of e_i, computed in g, where p is the number of parameters by which g is parameterized (6 at a minimum, since g has 6DoF). Similarly to stacking the error vectors e_i into e, we can stack the Jacobians J_i into $J = \frac{\partial e}{\partial g} = \nabla e(g) \in \mathbb{R}^{3N\times p}$

$$e = \begin{bmatrix} e_i \\ \vdots \\ e_N \end{bmatrix}_{3N\times 1} \tag{3.32}$$

$$J = \begin{bmatrix} J_i \\ \vdots \\ J_N \end{bmatrix}_{3N\times p} \tag{3.33}$$

With the approximation (3.31), the sum of squared residuals (3.30) becomes:

$$\begin{aligned} E(g+x) &= e(g+x)^T e(g+x) \\ &\approx (e+Jx)^T (e+Jx) \\ &= e^T e + e^T Jx + (Jx)^T e + (Jx)^T (Jx) \\ &= e^T + 2x^T J^T e + x^T J^T Jx \end{aligned}$$

At each iteration, the task is to find an update step x which will minimize $E(g+x)$. Differentiating the above expression with respect to x and equate with zero gives:

$$\nabla_x E(g+x) = \underbrace{2J^T e}_{b} + \underbrace{2J^T J}_{H} x = 0 \tag{3.34}$$

This is an expression involving, b the gradient, and H, the Gauss-Newton approximation to the Hessian. The useful property of this approximation of the Hessian is that it only requires first order derivatives, not second order ones like the true Hessian. Solving the linear system $Hx = -b$ for x yields the Gauss-Newton update

$$x = -(J^T J)^{-1} J^T e \tag{3.35}$$
Gauss-Newton step

Whether this step actually reduces the error $E(g+x) < E(g)$ depends on the accuracy of the Taylor series expansion at g and the validity of the Gauss-Newton approximation. This is usually the case when near the minimum. On the other hand, a simple gradient descent approach with

$$x = -\lambda^{-1} J^T e \tag{3.36}$$
gradient descent step

guarantees to reduce E, provided that λ is sufficiently large (and thus the step size λ^{-1} small), but its convergence near the minimum is very slow. The LM algorithm combines both approaches in quite a simple way:

$$x = -(J^T J + \lambda I)^{-1} J^T e \tag{3.37}$$
Levenberg step

Large λ correspond to small, safe gradient descent steps, which are sure to decrease the error, while small λ allow fast convergence near the minimum, but might otherwise increase the error. This update step can also be obtained by minimizing the damped version, called the augmented normal equations

$$(H + \lambda I)x = -b \tag{3.38}$$

instead of $Hx = -b$ [28]. The scaled identity matrix summand λI adds like a regularizer, meaning that even if H is close to singular, Cholesky decomposition of $H + \lambda I$ to solve the above would still work. The art of a good LM implementation is to tune λ after each iteration to allow for fast convergence: If the new error is lower than the previous one, the

update x is accepted and λ is decreased. If the new error is larger than the previous one, the update x is discarded and λ is increased. Our Matlab implementation is given in appendix A.4. Marquardt suggested to solve the augmented normal equations $H + \lambda diag(H)x = b$, which scales the step size in each dimension according to the difference to the minimum, useful for faster minimization of long, narrow valleys. However, we found this version to be less stable.

Algorithm 1: LM-ICP

Input: $\{p_i \rightarrow q_i\}$ point correspondences
Output: g rigid body transformation

$\lambda \leftarrow \lambda_0$ `// initialize lambda, see [37][p.684]`
$g \leftarrow g_0$ `// usually identity transform`
while *not converged i.e. λ is small* **do**
  $e \leftarrow e(g)$ `// residual vector`
  $J \leftarrow \nabla e(g)$ `// Jacobian`
  $x \leftarrow -(J^T J + \lambda I)^{-1} J^T e$ `// Levenberg step`
  if $||e(g+x)||^2 < ||e(g)||^2$ **then** `// error reduced`
    $g \leftarrow g + x$
    increase λ
  else
    decrease λ
return g

4. Approach

Algorithm 2: 3D OBJECT RECONSTRUCTION USING POINT PAIR FEATURES

Input: $\mathcal{D} = \{d_1, d_2, .., d_N\}$ range images
Output: $\mathcal{P} = \{P_1, P_2, ...P_N\}$ registered poses
Params: $d_{dist}, n_{angle}, nFrames, knn$

1 $\mathcal{C} \leftarrow \{C_1, C_2, .., C_N\} \leftarrow list()$ // Point clouds
2 $\mathcal{P} \leftarrow \{P_1, P_2, .., P_N\} \leftarrow list()$ // Camera poses
3 **for** $i \leftarrow 1$ ***to*** N **do**
4 $C_i \leftarrow Preprocessing(d_i, d_{dist})$ // 4.1
5 $F_i \leftarrow Learning(C_i, d_{dist}, n_{angle})$ // algorithm 3, 4.2.2
6 $K_i \leftarrow kdTreeBuild(C_i)$ // 3.5.2
7 **for** $i \leftarrow 2$ ***to*** N **do** // pairwise alignment and refinement
8 $v_{max} \leftarrow -1$
9 $k \leftarrow -1$
10 **for** $j \leftarrow i - 1$ ***to*** $max(1, i - nFrames)$ **do**
11 $\mathcal{Q} \leftarrow Matching(C_j, C_i, F_j, F_i, n_{angle})$ // algorithm 4, 4.2.3
12 $(P_{est}, v_{sum}) \leftarrow PoseClusteringAndAveraging(\mathcal{Q})$ // algorithm 5, 4.2.4
13 **if** $v_{sum} > v_{max}$ **then**
14 $v_{max} \leftarrow v_{sum}$
15 $k \leftarrow j$
16 $\mathcal{P}_{k \to i} \leftarrow P_{est}$ // update of best matching frame
17 $P_{k \to i} \leftarrow pairWiseRefinement(C_i, C_k, P_{k \to i}, K_k, d_{dist})$ // algorithm 6, 4.3
18 $P_i \leftarrow P_k * P_{k \to i}$ // need absolute pose for multiview refinement
19 $\mathcal{P} \leftarrow multiviewRefinement(\mathcal{C}, \mathcal{P}, d_{dist}, knn)$ // algorithm 7, 4.4
20 **return** $\mathcal{P}$

We implemented each step of the 3D modeling process (fig. 1.1), starting with multiple range images and ending with integrated views. All code is available in our github repository[1]. An overview of our approach can be given by splitting it up into its major components, which we describe in their respective sections:

[1] https://github.com/adrelino/ppf-reconstruction

A novelty of our method versus previous methods is that we don't just do a frame by frame alignment, but instead we find the best matching frame from the last $nFrames$ frames and align to that one. This is possible because our matching algorithm actually outputs a pose together with a score, which is the number of votes v_{sum} that it received in our voting process. In turn, this means that the concept of a trajectory generalizes to that of a tree in a pose graph. Before we run the final multiview refinement stage, the relative poses in this tree are converted to absolute poses in a pose graph.

4.1. Preprocessing

4.1.1. Backprojection

The first step is to transform our range images into point clouds by inverting the camera projection in the pinhole camera model using (3.2). For this, we just loop over all pixels in the range image d, and for each valid pixel with $Z = d(u, v) > 0$ (0 means no measurement which often happens at object boundaries) we add its back-projected 3D point to a list $\mathcal{P}$. We call the list $\mathcal{P} = \{p_i\} = \{p_1, p_2, ..., p_{N_\mathcal{P}}\}, p_i \in \mathbb{R}^3$ a point cloud.

4.1.2. Segmentation

Since we just want to reconstruct the object of interest and not a whole room, we need to do a two-region segmentation into foreground and background. Because our real world models were all acquired while standing on a table, we use this knowledge to mask out all objects not standing on the table as well as the table itself. We employ RANdom SAmple Consensus (RANSAC) to find the algebraic model parameters $ax + by + cz + d = 0$ of the largest plane in the point cloud which we assume to be the table's surface:

1. randomly select 3 linearly independent points $p_i, p_j, p_k \in \mathcal{P}$
2. these span a plane, calculate its parameters $ax + by + cz + d = 0$
3. calculate the distances d of all points $p \in \mathcal{P}$ to the plane (a, b, c, d)
4. add $p \in \mathcal{P}$ for which $|d| \leq |d_t|$ holds to the consensus set, where d_t is a threshold to set by the user.

The algorithm terminates whenever the size of the consensus set is sufficiently large. Having the plane, we can now construct a polygonal prism with height h, specified by the user, on top of it, for which we know that the object of interest must lie inside it. We now just mask out all the 3D points whose coordinates lie outside the prism. For convenience, we can now project the segmented 3D object points into the 2D image plane using (3.1) and save the resulting binary mask alongside the depth images. This is useful so we don't have to recompute the segmentation over and over again if we run the algorithm multiple times. For the synthetic sequences generated by Blender[2] the segmentation is already given, since background pixels are already set to 0 automatically.

[2] http://www.blender.org/

4.1.3. Downsampling

Our pose estimation algorithm [8] requires the point cloud to be regularly subsampled, so that all points have a minimum distance of d_{dist} (4.9), the distance quantization step. This is important, since point pairs which are too close together would give badly conditioned pose estimates. Additionally, downsampling reduces the number of points and thus the computational complexity of our learning (subsection 4.2.2) and matching (subsection 4.2.3) phase, since the number of point pair features is quadratic in the number of points. One of the simplest methods to do this is to employ a grid based subsampling technique. In 3D, grid cells are called voxels. Following [8][Results Section], we just take d_{dist} as our voxel size to fulfill the above criteria. Instead of the conventional three dimensional array, we use a hash map $h = \{(key, bucket)\}$ to represent a regular volume of voxels, so we don't have to store any empty cells and are independent of the actual location of the object in front of the camera. For each point in the original point cloud we calculate its key via

$$k = (\lfloor \frac{p_x}{d_{dist}} \rfloor, \lfloor \frac{p_y}{d_{dist}} \rfloor, \lfloor \frac{p_z}{d_{dist}} \rfloor)$$

and insert it into its corresponding bucket in the hash map. Now all of our original points are clustered into buckets corresponding to the voxels they belong to. All in all we just performed a clustering of our original cloud into multiple point sets where the Manhattan distance of a point to the voxel center was used as a metric.

The new, subsampled point cloud can now be easily obtained by taking the centroid of the point sets from each bucket. However, we wait with this step and keep the buckets for now since they are used in our denoising and normal and curvature estimation algorithms anyway.

4.1.4. Denoising

We just remove the buckets with less than $max(3, pts_{min})$ original points to eliminates spurious, isolated point clusters coming from an imperfect segmentation (subsection 4.1.2) as well as small stripes at the object boundary where the regular grid was unfavorable to the actual geometry of the object. Having at least 3 points per bucket is especially important for the estimation of normals (subsection 4.1.5), since one can imagine that normal estimates with less than 3 points don't make any sense and are even more robust the more points one uses.

4.1.5. Normal and curvature estimation

In our method, the downsampling, denoising and normal and curvature estimation algorithms are actually tied together because we use the original points for estimating them. As we saw in 3.4, surface normals (3.11) and curvature (3.12) can be estimated using principal component analysis of point sets. Usually, one uses the neighbors of each point in a sphere centered at that point as input to this method. We, however, use the original points that fell into the same voxel to estimate surface normal and curvature of the centroid of that voxel. This is not as accurate as using a sphere centered at the centroid, but much faster and works well for our purposes. Our implementation of a combined point set PCA is given in appendix A.1.

4.2. Pairwise coarse alignment

For the initial coarse pairwise alignment between any two overlapping point clouds we adapt a method known from object recognition in 3D point clouds [8]. An object is detected and simultaneously localized in 6DoF via the accumulation of votes in a parameter space as in the Generalized Hough Transform [3]. A vote is cast for every correspondence between two Point Pair Features (PPFs). In the following subsections, we first define the Point pair feature 4.2.1 before discussing the different parts of the algorithm: Learning 4.2.2, Matching 4.2.3, Pose clustering and averaging 4.2.4.

4.2.1. Point pair feature

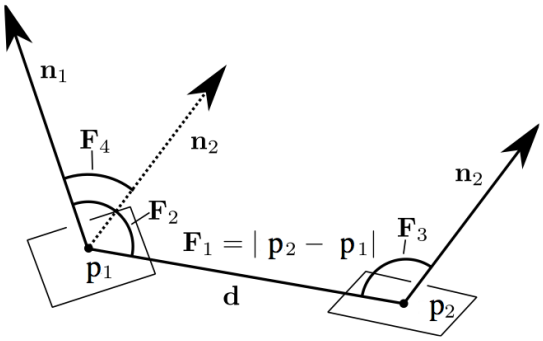


Figure 4.1.: A point pair feature PPF of two oriented points. [8]

A point pair feature is a 4D signature of a configuration of two oriented points which describes their relative position and orientation on the surface of an object. It is depicted in Figure 4.1. For two points $p_1, p_2 \in \mathbb{R}^3$ with normals $n_1, n_2 \in \mathbb{R}^3$, we define the *point pair feature* $F \in \mathbb{R}^4$ as:

$$\begin{aligned} F &= PPF(p_1, p_2, n_1, n_2) \\ &= (||d||_2, \angle(n_1, d), \angle(n_2, d), \angle(n_1, n_2)) \\ &= (F_1, F_2, F_3, F_4) \in \mathbb{R}^4 \end{aligned} \tag{4.1}$$

where

$$d = p_2 - p_1 \tag{4.2}$$

is the vector connecting the two points and

$$\angle(v_1, v_2) = acos\left(\left(\frac{v_1}{||v_1||}\right)^T \left(\frac{v_2}{||v_2||}\right)\right) \tag{4.3}$$

is the angle between two vectors, calculated via the $acos$ of the dot product between the normalized vectors. So, F_1 represents the Euclidean distance between the two surface points, F_2 and F_3 are the angles between the normal vectors n_1, n_2 of the points p_1, p_2 and their connecting vector d and F_4 is the angle between the two normal vectors n_1, n_2.

This asymmetric feature effectively encodes geometric constraints of point cloud sur-

faces so that efficient matching between two clouds is possible, especially when they contain rich surface normal variations [6]. It is used both for building the model description as well as for finding the object of interest in the scene [8].

4.2.2. Learning

Algorithm 3: LEARNING

Input: $\mathcal{C} = \{(p,n)_1, \cdots, (p,n)_N\}$ point cloud with normals
Output: $\mathcal{F} = \{(k, \alpha^*, i)_1, \cdots, (k, \alpha^*, i)_{N\cdot(N-1)}\}$ learned cloud
Params: d_{dist}, n_{angle} discretization parameters

$d_{angle} = 2\Pi / n_{angle}$ // (4.10)
$\mathcal{F} \leftarrow tupleList(N \cdot (N-1))$
$l \leftarrow 1$
for $i \leftarrow 1$ ***to*** N **do** // reference point
  for $j \leftarrow 1$ ***to*** N **do** // referred point
    if $i \neq j$ **then**
      $F \leftarrow PPF(p_i, p_j, n_i, n_j)$ // (4.1)
      $\tilde{F} \leftarrow quantizePPF(F, d_{dist}, d_{angle})$ // (4.4)
      $k \leftarrow bitEncode(\tilde{F})$ // (4.11)
      $\alpha^* \leftarrow planarRotAngle(p_i, n_i, p_j)$ // Listing 4.2
      $\mathcal{F}(l) \leftarrow (k, \alpha^*, i)$
      $l \leftarrow l + 1$
$\mathcal{F} \leftarrow sort(\mathcal{F})$ // sort by k
return $\mathcal{F}$

In the learning phase, the $N \times (N-1)$ point pair features $F \in \mathbb{R}^4$ for every pair of different points $(p_i, p_j), i \neq j$ in the point cloud $\{p_1, ..., p_N\}$ are computed. We denote p_i as the *reference* point and p_j as the *referred* point. These feature vectors are then grouped together based on their similarity, which is done by checking for equality of their discretized versions $\tilde{F} \in \mathbb{Z}^4$. We discretize by quantizing each component of the feature vector appropriately:

$$\begin{aligned} \tilde{F} &= quantizePPF(F, d_{dist}, d_{angle}) \\ &= \left(\lfloor \frac{F_1}{d_{dist}} \rfloor, \lfloor \frac{F_2}{d_{angle}} \rfloor, \lfloor \frac{F_3}{d_{angle}} \rfloor, \lfloor \frac{F_4}{d_{angle}} \rfloor \right) \\ &= (\tilde{F}_1, \tilde{F}_2, \tilde{F}_3, \tilde{F}_4) \in \mathbb{Z}^4 \end{aligned} \tag{4.4}$$

It is important to set the quantization levels $d_{dist}, d_{angle} \in \mathbb{R}$ for distances and angles carefully. If they are too small compared to the point cloud resolution, we will most likely not find any correspondences in the voting step due to noise. If they are too big, very dissimilar features will still be matched together. Thus, a good value for these parameters is important to allow for small deviations in the normal directions and the point pair distances, which arise naturally in real data due to the voxel-based subsampling as well as noise in the input data from the depth sensor. In our experiments, the default values of the

parameters are:

$$diam(M) = 0.15m \quad \text{model diameter} \tag{4.5}$$

$$\tau_d = 0.10 \quad \text{distance sampling rate} \tag{4.6}$$

$$n_{angle} = 30 \quad \text{number of angle buckets} \tag{4.7}$$

$$n_{dist} = 1/\tau_d = 10 \quad \text{number of distance buckets} \tag{4.8}$$

$$d_{dist} = \tau_d \cdot diam(M) = 1.5cm \quad \text{distance quantization step} \tag{4.9}$$

$$d_{angle} = 2\Pi/n_{angle} \hat{=} 12^\circ \quad \text{angle quantization step} \tag{4.10}$$

Note that only the first 3 parameters are free and determine the last 3 derived parameters. It turned out to be sufficient that the user specifies the model diameter $diam(M)$ (4.5) correctly. The standard values of the other free parameters n_{angle} and τ_d worked well in general. In particular, the default angle quantization step of 12 degrees is totally independent of the used model and works well for our estimated normal vectors.

In the original implementation [8], the authors insert $\tilde{F}$ into a hash table for later constant time expected lookup. However, the performance of the hash table strongly depends on the used hash function. We observed that lots of point pair features had the same discretized versions due to self-similar regions on an object. This means that our hash table would have only a few non-empty buckets, but the size of these buckets would be large. It takes a lot of time to construct such a hash table due to the non-locality of the memory accesses and furthermore, such an unbalanced hash-table does not guarantee constant time lookup. So, instead of using a hash table, we follow the idea of [41] to store $\tilde{F}$ in a simple array data structure, and then sort this array. The original idea here is to lookup a feature via binary search in expected logarithmic time. Since each component of $\tilde{F}$ usually fits in one byte ($n_{angle}, n_{dist} \leq 255$) and because $\tilde{F}$ has 4 components, we can encode it as a 32-bit unsigned integer k using bitwise concatenation:

$$\begin{aligned} \tilde{F}_1 &\in [0, n_{dist}] \subset [0, 255] = [0, 2^8 - 1] \\ \tilde{F}_{2,3,4} &\in [0, n_{angle}] \subset [0, 255] = [0, 2^8 - 1] \\ k = bitEncode(\tilde{F}) &= (\tilde{F}_1 | \tilde{F}_2 \ll 8 | \tilde{F}_3 \ll 16 | \tilde{F}_4 \ll 24) \in [0, 2^{32} - 1] \end{aligned} \tag{4.11}$$

Sorting an array of discretized point pair features is then as simple as sorting an array of unsigned integers. Additionally to the key k by which we sort the features, we also store the *reference* point index i as well as the planar rotation angle α^* in the array, which we need to efficiently calculate the 6DoF transformation between two matched point pair features later on (4.14)

To summarize, a point cloud $\mathcal{C}$ is learned and saved as an array $\mathcal{F}$ of 3-tuples (k, α^*, i), sorted by $k : k_1 \leq k_2 \leq \cdots \leq k_{N \cdot (N-1)}$

$$\mathcal{F} = \{(k, \alpha^*, i)_1, \cdots, (k, \alpha^*, i)_{N \cdot (N-1)}\} \tag{4.12}$$

4.2.3. Matching

Algorithm 4: MATCHING

Input: $\mathcal{F}^m = \{(k^m, \alpha^m, i^m)_1, \cdots, (k^m, \alpha^m, i^m)_{N_m \cdot (N_m - 1)}\}$ learnt model
$\mathcal{F}^s = \{(k^s, \alpha^s, i^s)_1, \cdots, (k^s, \alpha^s, i^s)_{N_s \cdot (N_s - 1)}\}$ learnt scene
$\mathcal{C}^m = \{(p^m, n^m)_1, \cdots, (p^m, n^m)_{N_m}\}$ model cloud
$\mathcal{C}^s = \{(p^s, n^s)_1, \cdots, (p^s, n^s)_{N_s}\}$ scene cloud

Params: n_{angle}

Output: $\mathcal{P} = \{(P_1, v_1), ..., (P_{N_s}, v_{N_s})\}$ pose estimates with scores

$d_{angle} = 2\Pi / n_{angle}$ // (4.10)
$\mathcal{A} \leftarrow \{0_1^{N_m \times n_{angle})}, \cdots, 0_{N_s}^{N_m \times n_{angle}}\}$ // list of N_s zero matrices
$a \leftarrow 1, b \leftarrow 1$
while $a \leq N_m(N_m - 1)$ ***and*** $b \leq N_s(N_s - 1)$ **do**
  if $k_a^m < k_b^s$ **then**
    $a \leftarrow a + 1$
  else if $k_a^m > k_b^s$ **then**
    $b \leftarrow b + 1$
  else // both keys are equal
    $c \leftarrow a$
    while $k_c^m == k_b^s$ **do**
      $\alpha \leftarrow \alpha_c^m - \alpha_b^s$ // (4.14)
      $i^\alpha \leftarrow \lfloor \frac{\alpha}{d_{angle}} \rfloor$
      $\mathcal{A}(i_b^s)[i_c^m, i^\alpha] \leftarrow \mathcal{A}(i_b^s)[i_c^m, i^\alpha] + 1$
      $c \leftarrow c + 1$
    $a \leftarrow a + 1$
    $b \leftarrow b + 1$
for $i^s \leftarrow 1$ ***to*** N_s **do**
  $v \leftarrow max(\mathcal{A}(i^s))$
  $i^m, i^\alpha \leftarrow argmax(\mathcal{A}(i^s))$
  $\alpha \leftarrow (i^\alpha + 0.5) * d_{angle}$
  $T_{s \to g} \leftarrow$ alignToOriginAndXAxis$(p_{i^s}^s, n_{i^s}^s)$ // Listing 4.1
  $T_{m \to g} \leftarrow$ alignToOriginAndXAxis$(p_{i^m}^m, n_{i^m}^m)$ // Listing 4.1
  $R_x(\alpha) \leftarrow$ angleAxis$(\alpha, [1, 0, 0]^T)$
  $T_{m \to s} = T_{s \to g}^{-1} R_x(\alpha) T_{m \to g}$ // (4.13)
  $\mathcal{P}(i^s) \leftarrow (T_{m \to s}, v)$
return $\mathcal{P}$

The original paper [8] and follow-up work using point-pair features [41, 6] deal exclusively with object detection. This means all point pair features in the model are learned once in an offline phase in the beginning and stored in hash tables or in sorted arrays as seen in the last section. In the online phase, a subset of all possible point pair features in the scene, whose size is given by the scene sampling rate, which is $1/5$ in [8], is usually considered. Each one of them is looked up in the learnt model description - either via hash

lookup or binary search, and if a match is found, a vote for this correspondence is cast.

In our application of object reconstruction, however, we don't have an offline model learning phase. Instead, we do a pairwise matching between any two frames which take on the roles of model as well as scene. This means that we have to calculate all possible point pair features in each frame anyway since we cannot expect to find correspondences between randomly selected subsets of them. In this context, we propose to merge the computation of point pair features from learning and matching phases. For each frame, we calculate all point pair features, store their discretized keys as well as additional information in arrays, and sort them as in subsection 4.2.2.

Then for each pair of frames that are to be matched, we employ an algorithm which is based on the idea of finding the intersection of two sorted arrays. It is implemented as the outer while loop starting on line 4 in algorithm 4. Let the lengths of the arrays be a and b respectively. It is important to note that the running time of such an algorithm, which uses the fact that both arrays are sorted, is in $O(a+b)$, while an algorithm which just assumes that the first array of length a is sorted is in $O(b \cdot \log(a))$, using b binary searches for the a sorted elements.

Intermediate coordinate system

Following the original vocabulary, the goal is to coarsely estimate the pose of the object, which means to align the model $\{p_i^m\}_1^{N_m}$ to the scene $\{p_i^s\}_1^{N_s}$ point cloud. Consider we found a correct match between point pair features on model $F^m = PPF(p_{i'}^m, p_{j'}^m, n_{i'}^m, n_{j'}^m)$ and scene $F^s = PPF(p_i^s, p_j^s, n_i^s, n_j^s)$. Because of the discriminative nature of the point pair feature, this match constrains enough DoF to compute the 6DoF rigid body transformation to align F^m to F^s. Let us first consider only the upper part of Figure 4.2 with the blue arrows. The alignment of the two *reference* points $p_{i'}^m \rightarrow p_i^s$ constrains 3DoF translation and the alignment of the two *reference* normal vectors $n_{i'}^m \rightarrow n_i^s$ constrains 2DoF rotation. The remaining 1DoF rotation is the rotation of the transformed *referred* point $p_{j'}^m$ around n_i^s to align it with p_j^s. The whole transformation can be more easily computed with the help of an intermediate coordinate system, as seen in the lower part of fig. 4.2.

The Transformation $T_{s\rightarrow g}$ moves the scene *reference* point p_i^s into the origin and aligns its normal n_i^s with the x-axis. The same is done to the model *reference* point and normal via $T_{m\rightarrow g}$. In the intermediate coordinate system, the images of scene and model *reference* points $p_i^s = p_{i'}^m = [0,0,0]^T$ and normals $n_i^s = n_{i'}^m = e_1 = [1,0,0]^T$ are aligned, but the images of the *referred* points $p_j^s, p_{j'}^m$ are still misaligned. Due to the definition of the point pair features, it is clear that for these images, it holds that $||p_{j'}^m||_2 = ||p_j^s||_2 = F_1$ and $\angle(p_{j'}^m, e_1) = \angle(p_j^s, e_1) = F_3$ up to the error introduced by discretization. In fact, this means that the images of the referred points have about the same x coordinates as well as about the same distance from the x-axis. Thus, they can coarsely be aligned by a rotation of an angle α around the x-axis, denoted as $R_x(\alpha)$.

The whole 6DoF transformation $T_{m\rightarrow s} \in SE(3)$ which aligns F^m to F^s can now be obtained via:

$$T_{m\rightarrow s} = T_{s\rightarrow g}^{-1} R_x(\alpha) T_{m\rightarrow g} \tag{4.13}$$

Note that due to the definition of the point pair feature, the *referred* normals $n_j^s, n_{j'}^m$ will be aligned by this transformation automatically, meaning we don't need to consider them

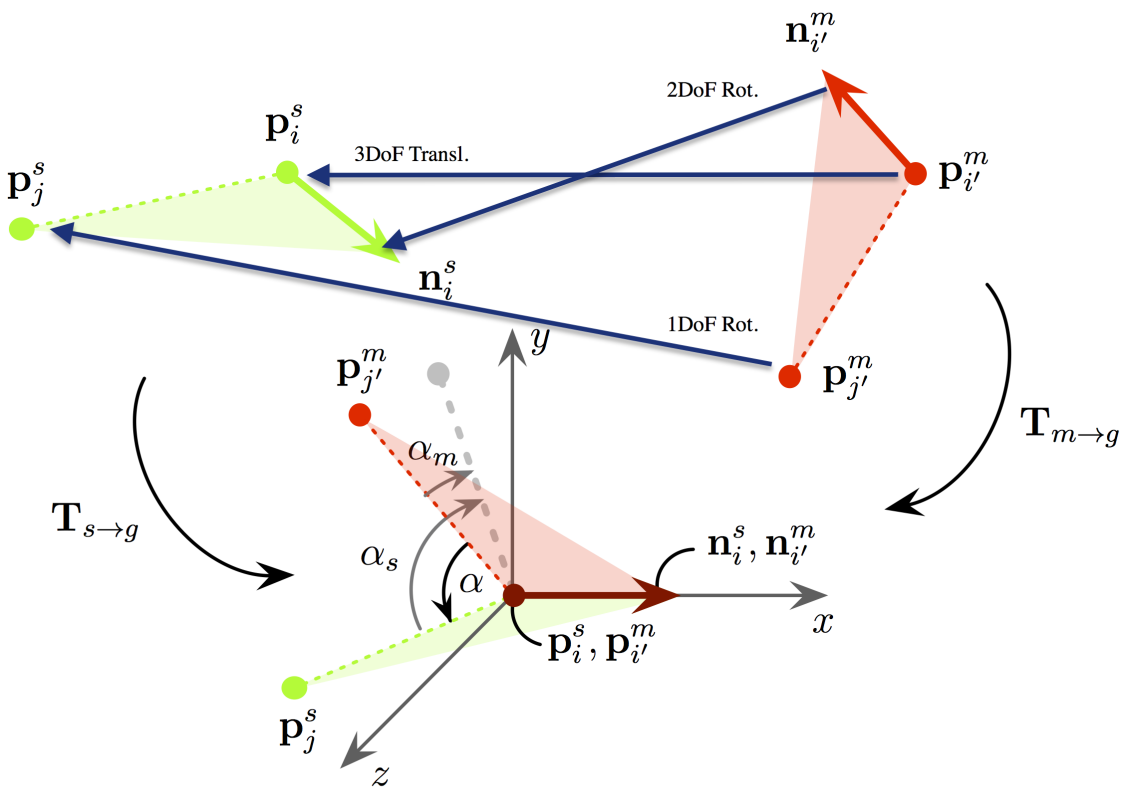


Figure 4.2.: Aligning two PPFs in the **intermediate coordinate system** [6]. In the upper part, p and n denote the original points and normals, while in the lower part, the same notation is used for their images.

here. Hence they are not even drawn in Figure 4.2.

Since the devil is in the detail, we directly provide our implementation of computing the transformations $T_{(m/s)\to g}$

```
Isometry3f alignToOriginAndXAxis(Vector3f p, Vector3f n){
    Vector3f xAxis = Vector3f::UnitX();
    double angle = acos(xAxis.dot(n));
    Vector3f axis = (n.cross(xAxis)).normalized();
    //if n parallel to x axis, cross product is [0,0,0]
    if(n.y()==0 && n.z()==0) axis=Vector3f::UnitY();
    Translation3f tra(-p);
    return Isometry3f( AngleAxisf(angle, axis) * tra);
}
```

Listing 4.1: **alignToOriginAndXAxis**(p, n): Computes the transformation $T_{(m/s)\to g}$ which aligns $p^{(m/s)}$ with the origin and $n^{(m/s)}$ with the x-Axis. *(Eigen/C++)*

Voting

Unfortunately, the aforementioned assumption of a correct match between two point-pair features is not always valid, because they only encode geometric constraints locally. In reality, most objects are at least partly self-similar, so there is a nontrivial amount of locally similar geometric surfaces which produce the same quantized point pair features. To overcome this issue a majority voting scheme, similar to the Generalized Hough Transform [3], is used.

For each scene reference point p_i^s, the transformation $T_{m\to s}$ can be parameterized by its matched model reference point $p_{i'}^m$ and the rotation angle α around the x-axis. In the sense of the Generalized Hough Transform, the parameter space for each scene reference point has dimensions $N_m \times n_{angle}$, where N_m is the number of points in the model point cloud and n_{angle} the number of angle bins. We use a 2D accumulator matrix $A_{p_i^s} = 0_{N_m, n_{angle}}$ to represent this parameter space.

Since we have more than just 1 scene reference point, but are using all the scene points as scene reference points as explained before, we actually have a list $\mathcal{A} = \{A_{p_1^s}, \cdots, A_{p_{N_s}^s}\}$ of N_s 2D accumulator matrices. For each of our point pair feature matches, we can obtain the scene and model *reference* point indices r^s, r^m from (4.12). Then, the angle α can be obtained from (4.13). This angle needs to be discretized $\alpha_{discr} = \frac{\alpha}{d_{angle}}$. For each match, we increment $\mathcal{A}$ at the correct 3D indices: $A(r^s)[r^m, \alpha_{discr}]$. In each of the N_s 2D accumulator spaces, we find the maximum score. The argmax gives us the model *reference* point index r_{best}^m, and the quantized alignment angle i_{best}^{α} corresponding to the optimal local transformation $T_{m\to s}$ again obtained via (4.13). Since it received the highest number of votes, it is a quantized candidate for the correct global transformation of the model to the scene point cloud.

Efficient Voting Loop The computation of α can actually be speeded up by precomputing the two components α^m, α^s in the learning phase. The component α^m is given as the angle between the vector $p_{j'}^m - p_{i'}^m$ and the upper xy half-plane, and similarly for α^s in the scene point cloud. Then, α is computed by a cheap minus operation:

$$\alpha = \alpha^m - \alpha^s \tag{4.14}$$

The intermediate angles are computed via the function planarRotAngle, for which we provide our implementation:

```
1 float planarRotAngle(Vector3f p_i, Vector3f n_i, Vector3f p_j){
2     Isometry3f T_ms_g=alignToOriginAndXAxis(p_i,n_i);
3     Vector3f p_j_image=T_ms_g*p_j;
4     //can ignore x coordinate, since we rotate around x axis
5     return atan2f(p_j_image.z(), p_j_image.y());
6 }
```

Listing 4.2: **planarRotAngle(p_i, n_i, p_j)**: Computes the angle α^* between the image of p_j and the upper xy half-plane. *(Eigen/C++)*

4.2.4. Pose clustering and averaging

Algorithm 5: POSE CLUSTERING AND AVERAGING

Input: $\mathcal{P} = \{(P_1, v_1), ..., (P_{N_s}, v_{N_s})\}$ pose estimates with scores
Params: t_{tra}, t_{rot} thresholds for translation and rotation
Output: (P_{est}, v_{sum}) best averaged pose estimate with score

1 $\mathcal{P} \leftarrow sortDecr(\mathcal{P})$ // sort decreasing by v
2 $\mathcal{K} \leftarrow \{\{\mathcal{P}(1)\}, \cdots, \{\mathcal{P}(N_s)\}\}$ // initialize pose clusters
3 **for** $i \leftarrow 1$ ***to*** N_s **do**
4 **for** $j \leftarrow 1$ ***to*** N_s **do**
5 **if** $i \neq j$ **then**
6 **if** $\forall (P_i, P_j) \in \mathcal{K}(i)_{1st} \times \mathcal{K}(j)_{1st}$:
7 *isPoseSimilar*$(P_i, P_j, t_{rot}, t_{tra})$ **then** // Listing 4.3
8 $\mathcal{K}(i) \leftarrow \mathcal{K}(i) \cup \mathcal{K}(j)$
9 $\mathcal{K} \leftarrow \mathcal{K} \setminus \mathcal{K}(j)$
10 $Q \leftarrow$ poseWithScoreList(size($\mathcal{K}$))
11 **for** $i \leftarrow 1$ ***to*** *size*($\mathcal{K}$) **do**
12 $P_{est} \leftarrow averagePose(\mathcal{K}(i)_{1st})$ // uses quaternion mean (3.6)
13 $v_{sum} \leftarrow sum(\mathcal{K}(i)_{2nd})$
14 $Q(i) \leftarrow (P_{est}, v_{sum})$
15 $\mathcal{Q} \leftarrow sortDecr(\mathcal{Q})$ // sort decreasing by v
16 **return** $\mathcal{Q}(1)$

The N_s peaks in the list of accumulator matrices correspond to the most likely alignment of quantized scene and model point pair features. Thus, the pose estimates computed with these alignments are quantized as well. However, due to the subsampling and noise in the input data, peaks in the list from different scene reference points might actually describe the same transformation, even though the computed pose estimates differ slightly. This can be seen as having multiple noisy measurements in the space of 6DoF rigid body motions. To get an even better initial estimate, one has to cluster and then average similar poses. From a statistical point of view, the averaging step of similar poses can be seen as the ML estimate of the most likely pose, given that they are the disturbed versions of the true pose, produced by Gaussian noise.

First, we sort the poses by the number of votes they received. This ensures that the most likely poses are clustered together. We employ an agglomerative clustering algorithm, meaning that each pose estimate starts in its own cluster (line 2 in algorithm 5). Subsequently, pairs of clusters are merged if all the combinations of poses between these two clusters do not differ in rotation and translation for more than the predefined thresholds t_{tra}, t_{rot}. This is effectively the *complete linkage clustering* criterium, meaning that the distance between clusters is a function of the pairwise distances between the poses they contain:

$$d(\mathcal{K}(i), \mathcal{K}(j)) = max\{d(\mathcal{P}_i, \mathcal{P}_j), \mathcal{P}_i \in \mathcal{K}(i), \mathcal{P}_j \in \mathcal{K}(j)\}$$

In the for loop in line 11, the poses in the remaining clusters are averaged and their votes

summed up per cluster. Lastly, we sort the pose estimates again by decreasing number of votes and finally return the averaged pose that received the highest sum of votes.

To reduce the number of needed parameters, we just set the translation and rotation thresholds as follows per default and never touch them again:

$$t_{tra} = 0.05 * diam(M) \quad \text{translation threshold} \tag{4.15}$$

$$t_{rot} = 15^{\circ} \quad \text{rotation threshold} \tag{4.16}$$

This was the last step of our coarse pairwise alignment. In the next section, we further refine this initial estimate using ICP.

```
bool isPoseSimilar(Isometry3f P_i, Isometry3f P_j, float t_rot, float t_tra){
    //Translation
    float d_tra=(P_i.translation()-P_j.translation()).norm();
    //Rotation
    float d = Quaternionf(P_i.linear()).dot(Quaternionf(P_j.linear()));
    float d_rot = rad2degM(acos(2*d*d - 1));
    return d_rot <= t_rot && d_tra <= t_tra;
}
```

Listing 4.3: **isPoseSimilar(**$P_i, P_j, t_{rot}, t_{tra}$**)**: checks if two poses are similar. Quaternion distance measured as in (3.4) *(Eigen/C++)*

4.3. Pairwise refinement

Algorithm 6: PAIRWISE REFINEMENT

Input: $\mathcal{A} = \{a\}_1^{N_a}$ data, $\mathcal{B} = \{(b, n_b)\}_1^{N_b}$ model cloud with normals
$P_{\mathcal{A}\to\mathcal{B}}$ initial relative pose estimate
$K_\mathcal{B}$ k-d tree of the model cloud
Output: $P_{\mathcal{A}\to\mathcal{B}}$ refined relative pose estimate
Params: d_{max} rejection threshold (3.17), $pointPlane$ boolean flag

1 **while** *not converged* **do**
2 $i \leftarrow 1$
3 $p, q, n_q \leftarrow growableList()$
4 **for** $j \leftarrow 1$ ***to*** N_a **do**
5 $a_j^{inB} \leftarrow P_{\mathcal{A}\to\mathcal{B}} * a_j$
6 $d, k \leftarrow kdTreeLookup(K_\mathcal{B}, a_j^{inB})$ `// distance and index`
7 **if** $d < d_{max}$ **then**
8 $p_i \leftarrow a_j^{inB}$
9 $q_i \leftarrow b_k$
10 $n_{q_i} \leftarrow n_{b_k}$
11 $i \leftarrow i + 1$
12 **if** $pointPlane$ **then** `// true per default`
13 $P_{upd} \leftarrow pointToPlane(p, q, n_q)$ `// A.3`
14 **else**
15 $P_{upd} \leftarrow pointToPoint(p, q)$ `// A.2`
16 $P_{\mathcal{A}\to\mathcal{B}} \leftarrow P_{upd} * P_{\mathcal{A}\to\mathcal{B}}$
17 **return** $P_{\mathcal{A}\to\mathcal{B}}$

Once two point clouds are roughly aligned, we can use ICP to further refine this coarse initial alignment using algorithm 6. As outlined in section 3.6, ICP consists of two steps that are iteratively applied until convergence: computing **correspondences** (line 4 to 11) and computing and applying the **transformation** which minimizes the distance between these correspondences (line 12 to line 16).

4.3.1. Correspondences

We speed up the computation of correspondences (in line 6) using a precomputed k-d tree as explained in (subsection 3.5.2). Projection-based matching (subsection 3.5.4) is not possible, since downsampling destroyed the organized structure of our point cloud. We also did not use the distance transform (subsection 3.5.3), because we wanted to avoid to compute and store bounding volumes for each of our point clouds. We use the nanoflann[3] library for k-d tree construction and exact $O(\log(n))$ nearest neighbor searches. The most expensive operation is the initial construction of the tree, which is in $O(n \log(n))$ in the

[3] https://github.com/jlblancoc/nanoflann

best case. This means that it is best if we only have to construct the tree just once in the beginning, and then leave the points of each point cloud untouched and don't project them during the ICP iterations. This can actually be achieved by leaving the points in their respective camera coordinate system, computing the tree once $K_{\mathcal{B}}$ initially in that coordinate system, and then just updating the pose of the camera (line 16) instead of projecting the points into the global coordinate system. Now, for a nearest neighbor lookup (line 6), we just have to transform the point a_j for which we want to find a nearest neighbor into the camera coordinate system of the model cloud $\mathcal{B}$ (line 5), and then look up its nearest neighbor in the precomputed k-d tree $T_{\mathcal{B}}$ of the model cloud (line 6).

If the distance d is below the rejection threshold d_{max} (3.17) (in line 7), we found a correspondence $a_j^{inB} \rightarrow b_k$ and save it as $\{p_i \rightarrow q_i\}$. The arrays p and q are now indexed by their correspondences and outlier pairs have been removed, as needed for the next step in ICP, updating the transformation.

4.3.2. Transformation

In general, it doesn't matter in which coordinate system the correspondences $\{p_i \rightarrow q_i\}$ are defined, since all we do in the end is to apply an incremental transformation update P_{upd} (line 16) to the relative pose estimate $P_{\mathcal{A}\rightarrow\mathcal{B}}$. A trivial choice is the camera coordinate system of $\mathcal{B}$. For minimizing the distances defined on these correspondences, we observed that point to plane (3.25) gives better accuracy than point to point (3.24), which is most likely due to the fact that we are dealing with range images, for which point to plane was actually designed. While the first version additionally needs the normals of the model cloud, the second version gets along without them. Our actual implementations of both function calls in lines 13 and 15 can be found in appendix A.3 and A.2, respectively.

The coarse pose estimates we got from the point pair feature alignment turned out to be good enough for ICP to converge well in general. This means that we can set the rejection threshold d_{max} (3.17), used in the rejection step (in line 7), rather small to allow for more accuracy at the sake of stability. Per default, we set it equal to the distance quantization step d_{dist} (4.9):

$$d_{max} = d_{dist} \tag{4.17}$$

Alternatively, we could use a robust loss function like the Huber loss (3.20) to minimize the impact of outliers. But then, we would not have nice closed form linear least-squares solutions anymore for either of the above choices.

4.4. Multiview refinement

Once all the pairwise alignment and refinement is finished, we can even further optimize the refined pose estimates by optimizing them all together. We do this by extending the LM-ICP algorithm from subsection 3.6.5 to the simultaneous alignment of more than two views, which we call **Multiview LM-ICP**, as introduced by Fantoni et al. [12]. In our overall algorithm(algorithm 2) we first compute the absolute pose of each camera from the relative pose constraint to its neighbor from the pairwise alignment and refinement. With this absolute pose, the topology of the correspondences between views can be modeled as a graph and the resulting global registration error reduced via **pose graph optimization**, for which specialized frameworks have been developed.

Algorithm 7: MULTIVIEW REFINEMENT

Input: $\mathcal{C} = \{C_1, C_2, ...C_M\}$ point clouds
$\mathcal{P} = \{P_1, P_2, ..., P_M\}$ camera poses
Output: $\mathcal{P} = \{P_1, P_2, ..., P_M\}$ refined camera poses
Params: d_{max} rejection threshold (3.17), knn, $n_{GraphUpdates}$

1 **for** $i \leftarrow 1$ ***to*** $n_{GraphUpdates}$ **do** // $n_{GraphUpdates} = 1$ per default
2 $optimizer \leftarrow g2oGraphOptimizer()$
3 $optimizer.addNodes(\mathcal{P})$
4 **for** $h \leftarrow 1$ ***to*** M **do**
5 $\{k_1, \cdots, k_{knn}\} \leftarrow knnPoseNeighbours(P_h, \mathcal{P})$
6 **for** $j \leftarrow 1$ ***to*** knn **do**
7 $e_{h,k_j} \leftarrow pairwiseCorrespondences(C_h, C_{k_j}, d_{max})$
8 $optimizer.addDirectedEdges(e_{h,k_j}, P_h, P_{k_j})$
9 **while** *not converged* **do**
10 $\mathcal{X} = \{x_1, x_2, \cdots, x_M\} \leftarrow optimizer.computePoseUpdates()$
11 **if** *error reduced* **then**
12 $\mathcal{P} \leftarrow \mathcal{P} + \mathcal{X}$
13 **return** $\mathcal{P}$

The overall structure of ICP (section 3.6) is preserved: computing **correspondences** in each iteration of the outer loop (line 1) and computing and applying the **transformation** which minimizes the distance between these correspondences (line 9). The only difference is that these transformation updates are now applied iteratively in the inner loop.

4.4.1. Global registration error

Let $C_1, \ldots C_M$ be the set of point clouds that are brought to be in alignment. Remember the alignment error from the pairwise setting, where g stood for the transformation aligning the model cloud to the data cloud:

$$E(g) = \sum_{i=1}^{N} ||d(g(p_i), q_i)||^2 \tag{3.29}$$

In the multiview setting, the roles of model and data cloud are no longer fixed. To generalize and formalize the notation of which cloud gets registered to which other cloud, we can encode these relations as a directed graph, with the adjacency matrix $A \in \{0,1\}^{M \times M}$, such that $A(h,k) = 1$ if cloud C_h can be registered to cloud C_k. Let $g_1, \dots g_M$ be the rigid transformations that are applied to each view in the global reference frame. The alignment error between two clouds C_h and C_k then is:

$$E(g_h, g_k) = A(h,k) \sum_{i=1}^{N_h} ||d(g_h(p_i^h), g_k(q_i^h))||^2 \tag{4.18}$$

where $\{p_i^h \rightarrow q_i^h\}$ are the N_h closest point correspondences obtained from the the clouds C_h and C_k (line 7). Since we still only update the camera poses, we can obtain them the same way as in the pairwise refinement (section 4.3) and even reuse the k-d-trees from before. The pairwise formulation (3.29) can be obtained by setting $g = g_k * g_h^{-1}$. The overall alignment error, which we want to minimize at this stage, is obtained by summing up the contribution of every pair of overlapping views:

$$E(g_1, ...g_M) = \sum_{h=1}^{M} \sum_{k=1}^{M} A(h,k) \sum_{i=1}^{N_h} ||d(g_h(p_i^h), g_k(q_i^h))||^2 \tag{4.19}$$

In practice, we assume that clouds are overlapping and can be registered to each other if the distance between the translational components of their poses is small. Because it is hard to specify an absolute threshold that works for different scenarios, we compute the knn_{Poses} nearest neighbors in the Euclidean space of their translation vectors (line 5). Even if this assumption is invalid for some of the pose pairs (i.e. if the rotational components are totally different, meaning that the cameras look in opposite directions), it doesn't hurt our algorithm, because only point to point correspondences smaller than d_{max} (line 7) are considered. We use a neighbor list of length knn_{Poses} for each pose instead of an adjacency matrix A, because it allows faster lookup of neighbours.

4.4.2. Sparsity structure of the linearized system

The error vector for one closest point correspondence between two point clouds is $e_{h,k,i} = d(g_h(p_i^h), g_k(q_i^h))$. Since the error function of a constraint depends only on the values of two nodes, namely start and end node of that edge, the Jacobian

$$J = \frac{\partial(e_{h,k,i})}{\partial(g_1 \dots g_M)} \tag{4.20}$$

has a regular sparse block structure. Let $S = \{(h,k) : A(h,k) = 1\}$ be the set of overlapping views, equivalent to the set of directed edges in our pose graph, where an edge is added if there are correspondences between the two camera nodes. Given a view pair (h,k) whose index is s, the block row

$$J^s = [0 \cdots 0 \; J^{s,h} \; 0 \cdots 0 \; J^{s,k} \; 0 \cdots 0] \tag{4.21}$$

has only two non-zero blocks, where the block column indices h, k correspond to the camera poses g_h, g_k whose clouds are connected by the constraint imposed through their closest point correspondences. A camera pose g_h is thus only affected by the alignment if its block column contains some non-zero blocks.

In LM-ICP, we introduced the vector of residuals $e = [e_i^T, \cdots, e_N^T]^T \in \mathbb{R}^{3N}$ (3.32). In the multiview setting, each pairwise correspondence yields such a vector $e = e_{h,k}$. These can be stacked as well into

$$\tilde{e} = (e_{1,1}^T, \ldots, e_{1,M}^T), (e_{2,1}^T, \ldots, e_{2,M}^T), \ldots \ (e_{M,1}^T, \ldots, e_{M,M}^T) \tag{4.22}$$

Most of these elements are zero and can be dropped, we only have $|S|$ non-zero entries $e_{h,k}$. The only requirement the stacked vector $\tilde{e}$ must fulfill is that its elements are ordered in the same way as the block rows in J to which they correspond. In that way, we can retrieve the gradient $b = J'\tilde{e}$ as well as the approximate Hessian $H = J'J$. No matter how many correspondences we actually used to compute them, their block structure looks always the same, its dimension does only depend on the number of cameras M. That means b has exactly M block columns and one block row, while H consists of $M \times M$ blocks. These blocks are exactly the pairwise stacked Jacobians $J_{h,k}$ as in (3.33). Interestingly, the block structure of H corresponds to our graph adjacency matrix A, meaning that every non-zero block in H, which is the Jacobian $J_{h,k}$, corresponds to a $A(h, k) = 1$.

This sparsity structure can and must be exploited to arrive at practicable algorithms to compute the Levenberg update step (line 10) at each inner iteration:

$$x = -(J^T J + \lambda I)^{-1} J^T \tilde{e} \tag{3.37}$$

Remember that x was computed by solving the augmented normal equations $(J^T J + \lambda I)x = J^T\tilde{e} \Leftrightarrow (H + \lambda I)x = -b$ (3.38). Direct sparse Cholesky solvers allow to solve the left side of the equation efficiently, which are applicable because $H + \lambda I$ is a symmetric positive definite matrix, while the right side, the gradient b, is obtained by a sparse matrix vector multiplication.

4.4.3. Pose graph optimization

In our implementation, we use the g2o framework[4] of Kummerle et al. [28], which helps in assembling the Jacobian and the residual vector from a simple graph structure. Furthermore, it already includes implementations of the Levenberg algorithm as well as of sparse Cholesky solvers for efficiently computing the Levenberg update step x at each iteration.

Additionally, there is a sample implementation of GICP (subsection 3.6.4) Vertices and Edges[5], which we can use directly. We don't need to worry about a big sparse equation system, all we have to do is to add each camera pose $P \in SE(3)$ as a node (line 3) and each point to point correspondence's distance vector $e_{h,k,i}$ as an edge between two nodes (line 8). We used the point-to-plane incarnation of GICP (3.28), because the more sophisticated plane-to-plane implementation seemed to be buggy. The residual vector and the

[4] http://openslam.org/g2o.html

[5] https://github.com/RainerKuemmerle/g2o/blob/master/g2o/types/icp/types_icp.cpp

Jacobian are then computed automatically.

The space of parameters $P \in SE(3)$ is not Euclidean, because the rotational part spans over the non-Euclidean $SO(3)$ group. Above procedures only work if the space of parameters is Euclidean. However, $SE(3)$ is a manifold, and can thus be locally approximated by its tangent space around the identity. The above implementation uses the axis of a normalized quaternion as a minimal representation for the rotational part of the update steps x. After each update, the quaternion has to be renormalized. Another and more elegant way is to use the Lie algebra $se(3)$ to represent the updates. We did this in our Matlab LM-ICP implementation A.4 and also in our Matlab Multiview-LM-ICP implementation,[6] which works well for small problem sizes. In future work, we plan to port this code to C++.

The increments $\mathcal{P} \leftarrow \mathcal{P} + \mathcal{X}$ (line 12), meaning $P_i \leftarrow P_i + x_i \; \forall i \in [0, M]$, are applied by first converting x_i through a non-linear operator to the same representation as P_i, and then left multiplying this transformation update by standard motion composition, denoted as $\oplus$ in [28]. The nonlinear operator, denoted as $\boxplus$ in [28] is the quaternion to rotation matrix operator or the exponential map, depending on which parametrization of $SE(3)$ is used.

The g2o framework additionally gives us the freedom to switch the nonlinear optimizer through one line of code. Experimentally, we found out that the pose graph was optimized faster using Powell's Dogleg method than the more robust Levenberg-Marquardt algorithm. The first one converged because most of our pairwise refined poses were already quite close to the optimum. Because of this, we suggest that the user chooses between the faster Powell's Dogleg algorithm or the more robust Levenberg-Marquardt algorithm, which has a wider basin of convergence, based on the quality of the result of the pairwise refinement.

[6] https://github.com/adrelino/ppf-reconstruction/blob/master/matlab/mv_lm_icp_step.m

5. Evaluation

We evaluate our approach on both real and synthetic data and give qualitative as well as quantitative results concerning accuracy as well as speed. In the following, we encode the different stages of our algorithm as:

0	A	B	C
preprocessing	pairwise alignment	pairwise refinement	multiview refinement

5.1. Datasets

We use a subset of the real and synthetic datasets which were made by Slavcheva [44]. For some of the larger sequences, we use only every $step'th$ frame. This works well for us since we don't need to rely on the small motion assumption. In general, using around 50 frames that sufficiently cover the object from all sides is totally enough for a decent reconstruction of objects of the sizes we considered ($10 - 40cm$ in diameter).

Real dataset Real data was acquired with a PrimeSense sensor from objects placed on a marker-board (only used to get ground truth), which sits on a turntable. The trajectories obtained by rotating the table are circular and sometimes very large, up to 500 frames, so we only use every $step'th$ frame.

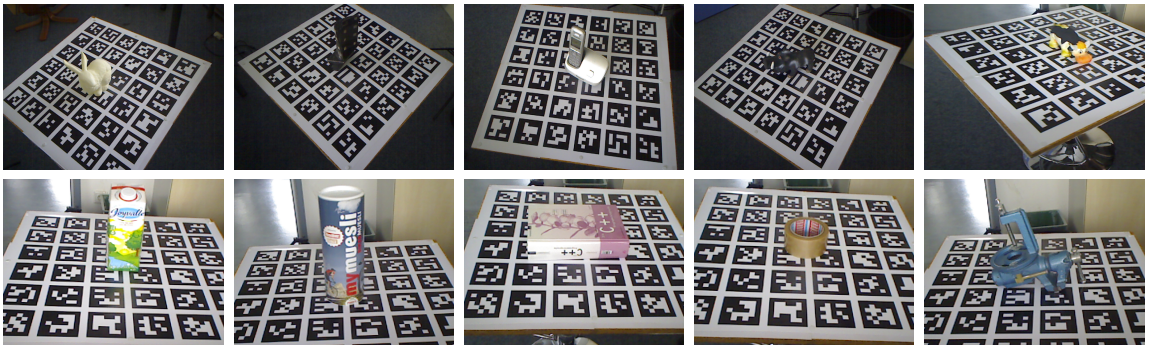


Figure 5.1.: Real models: *bunny, blade, phone, Pumba, cow, milk box, muesli box, book, tape, bench vise* [44]

Synthetic dataset Slavcheva [44] uses Blender[1] to generate sequences of RGB-D images from 9 different models. Each sequence contains exactly 120 frames, sampled from 3 different kinds of trajectories. The ground truth we use to evaluate our approach are the actual camera locations as specified in Blender. We only use the range images and discard

[1] http://www.blender.org/

the color images. The point clouds obtained from each range image are virtually free from any noise and sampling artifacts.

Figure 5.2.: Synthetic models: *bunny, Kenny, teddy, cow, tram, leopard, juice box, tea box, turbine* [44]

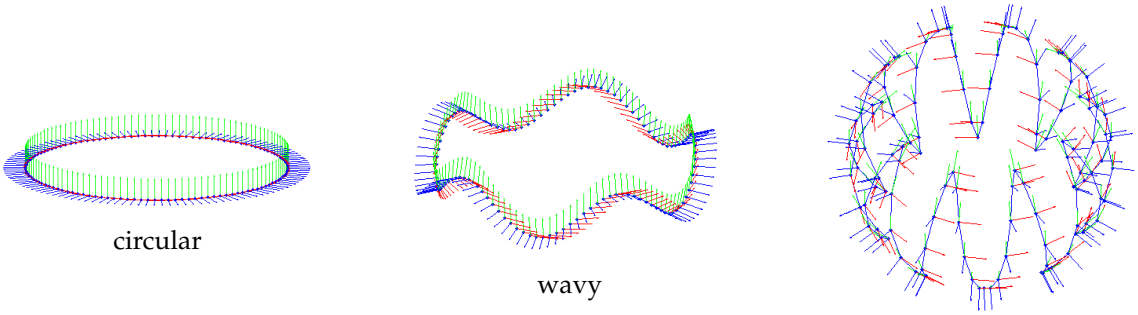


Figure 5.3.: Synthetic trajectories [44]

More bunnies We love the Stanford bunny.[2] It usually gives really good results[3] because of its distinctive geometry, which is why we used it from the very beginning, before acquir-

[2] http://graphics.stanford.edu/data/3Dscanrep/
[3] http://www.cc.gatech.edu/~turk/bunny/bunny.html

ing the above datasets, to test and develop different aspects of our method. So in addition to the above, we have two more bunny sequences.

The first one is from real data and a smaller marker-board and was thus captured at a closer range than the one from the dataset above. The segmentation was also better, resulting in less noise compared to the dataset. It is a circular trajectory of 36 frames. We love it and thus just call it **bunny2**.

The second one is synthetically generated, but not by any of the three trajectories from above. Instead, 41 positions are sampled on a unit sphere around the model, some of which have large displacements in between. This sequence inspired us to develop the pairwise matching method which matches to the last $nFrames$ and chooses the match with the highest vote, resulting in a relative pose dependency tree. This one is called **bunny sphere**.

5.2. Qualitative results

Fused point clouds as well as the 3D trajectories are used to visually compare the results. We fuse the original, non-subsampled point clouds for better visibility of the fine-scale alignment, even though they were not even used by our algorithm. We try to choose the best viewing angle for the 2D projection of the trajectory and the point cloud to highlight the important details. In the following figures, the columns (A | B | C) stand for the results of the different stages of our algorithm. The first three examples are from real data, while the last five are from synthetic data.

5.2.1. Discussion

It was possible to find a right set of parameters to reconstruct all the objects with distinctive geometry. In most examples, each step improved the visual result. The multiview refinement improved details such as tail and ears of bunny2 (figure 5.4) or the legs of the cow (figure 5.8). The noise visible in the original real data, especially in (figure 5.6), was no problem, since it was filtered out by our voxel-based denoising (subsection 4.1.4). Pumba's (figure 5.5) legs are spread out, the estimated trajectory is bent towards the viewer at the top and the bottom because there is little point cloud overlap at the right and left side of the object due to its axial symmetry and long shape. In this case, our multiview refinement even worsened the error. Notice the relative pose dependency tree in bunny sphere (figure 5.7), to which we add further edges before the global refinement, after which it looks like a neatly triangulated sphere. Similarly in teddy abrupt (figure 5.11).

Our only failure case in these examples is the symmetrical juice box (figure 5.9). The pairwise alignment and refinement works well for half of the trajectory, but then suddenly it does a 180 degrees turn and stitches the top of the juice box to the bottom.

Interestingly, our pairwise alignment fails for Kenny (figure 5.10), which exhibits reflectional (left and right half when looking from front) as well as partial rotational symmetry (especially its helmet). But the pairwise refinement is still able to recover from these bad pose estimates.

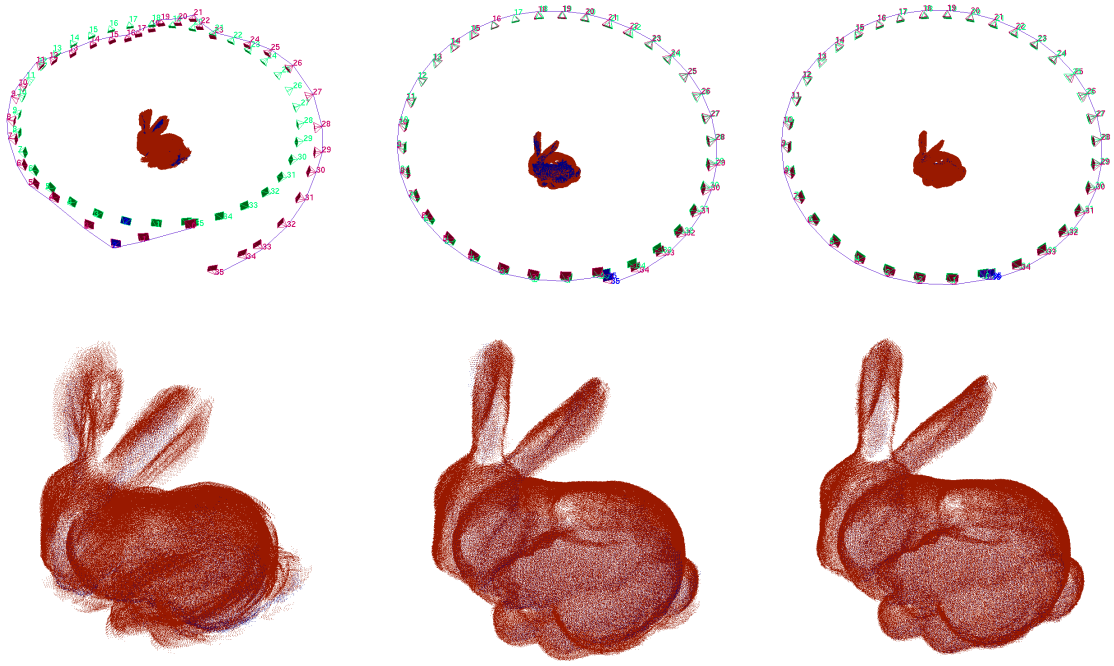


Figure 5.4.: **bunny2**

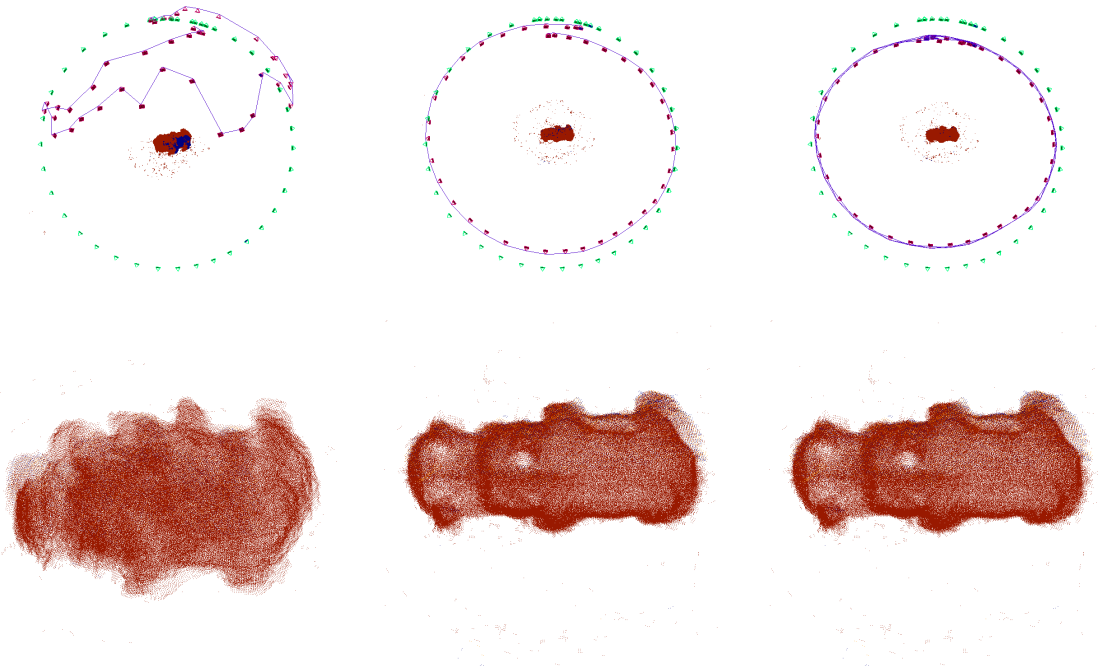


Figure 5.5.: **Pumba** -diamM 0.07 -step 15 -knn 4 -dmax 0.05

Figure 5.6.: **bench vise** -step 15

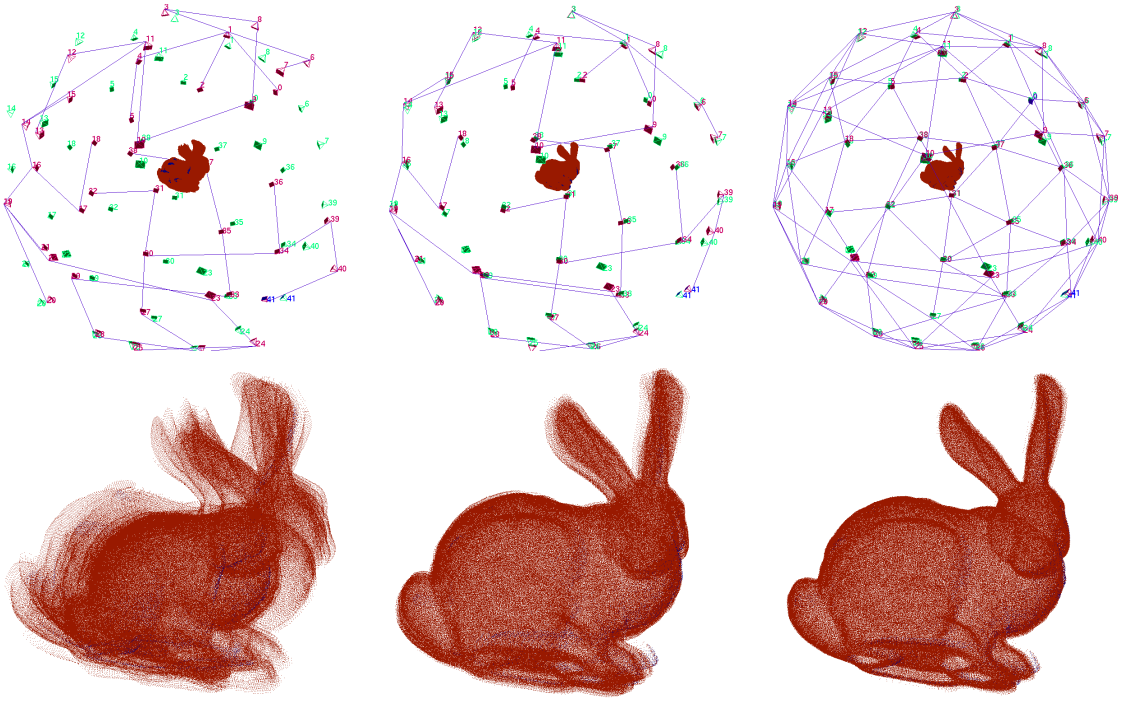


Figure 5.7.: **bunny sphere** -knn 5 -nFrames 5

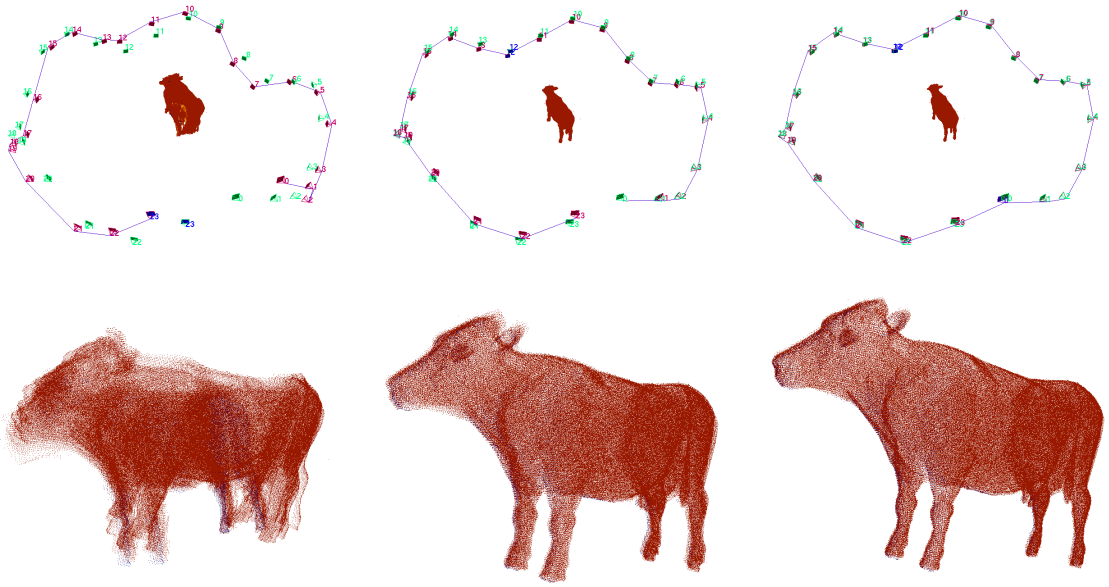


Figure 5.8.: **cow wavy** -step 5

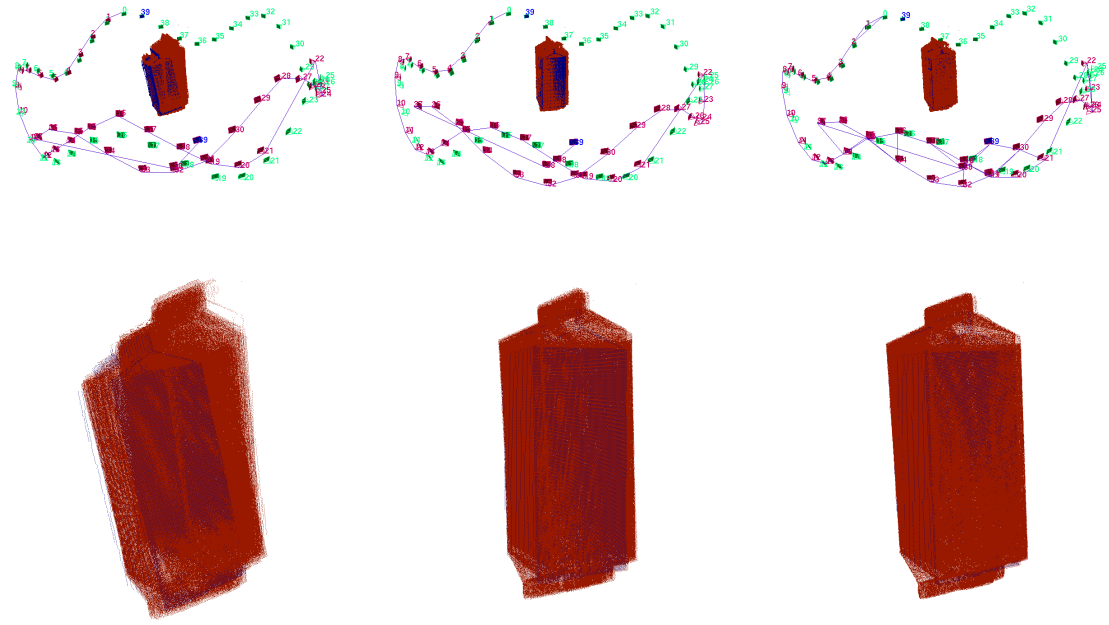


Figure 5.9.: **juice box wavy** -step 3

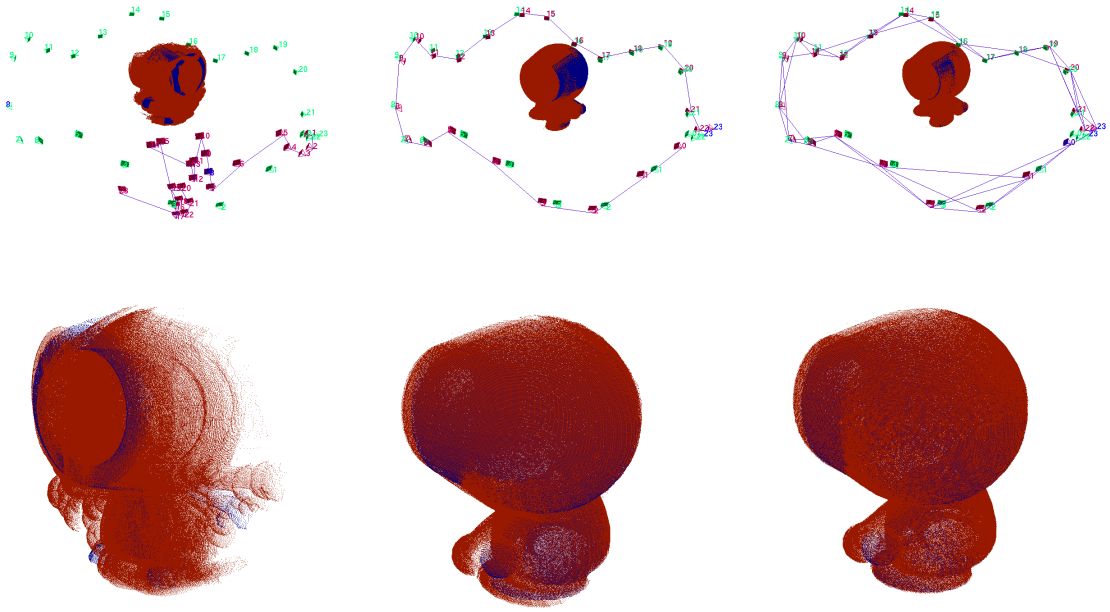


Figure 5.10.: **Kenny wavy** -step 5 -diamM 0.4 -knn 4

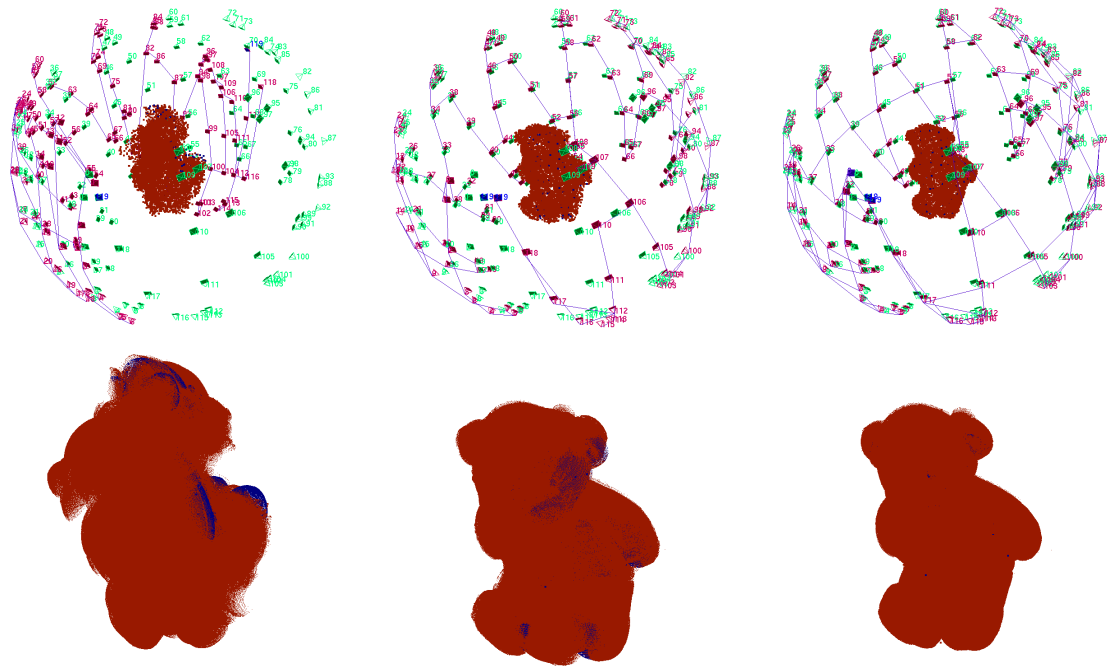


Figure 5.11.: **teddy abrupt** -nFrames 5 -knn 4 -diamM 0.5

5.3. Quantitative results

The accuracy of our approach may also be described in numbers, because we have some sort of ground truth for both synthetic as well as real data and can thus compare our estimated poses to them. Formally, for a sequence of n frames, we have

$$\mathcal{P} = \{P_1, \cdots, P_n\} \in SE(3) \quad \text{pose estimates} \tag{5.1}$$
$$\mathcal{Q} = \{Q_1, \cdots, Q_n\} \in SE(3) \quad \text{ground truth poses} \tag{5.2}$$

that need to be compared. To derive meaningful **evaluation metrics** on the difference between rigid body motions, we follow the approach of Sturm et al. [46].

Relative Pose Error (RPE) The relative pose error RPE at time step i with interval width Δ is defined as:

$$E_{i,\Delta} := (Q_i^{-1} Q_{i+\Delta})^{-1} (P_i^{-1} P_{i+\Delta})^{-1} \tag{5.3}$$

The relative pose error measures the local accuracy of the trajectory over a fixed interval Δ. It corresponds to the drift of the trajectory, which is in particular useful for the evaluation of Visual Odometry (VO) systems [46]. From a sequence of n camera poses, one obtains $m = n - \Delta$ individual relative pose errors along the sequence. For visual odometry systems that match consecutive frames, such as our pairwise alignment and refinement algorithm if we set $nFrames = 1$, $\Delta = 1$ is an intuitive choice, giving the drift per frame. In this case, we obtain $m = n - 1$ individual RPEs, corresponding to pairs of consecutive frames.

Absolute Trajectory Error (ATE) The absolute trajectory error ATE at time step i is defined as:

$$F_i := Q_i^{-1} S P_i \tag{5.4}$$

The absolute trajectory error measures the global consistency of the estimated trajectory, which can be evaluated by comparing the absolute distances between the estimated and the ground truth trajectory. Therefore, it is a useful metric for the global consistency of our reconstructed model, especially after the multiview refinement step. Because both trajectories can live in different coordinate systems, we additionally need the rigid body motion S that aligns them in the least-squares sense. It is obtained by registering all the translational parts of Q to the ones of P using Arun et al. [2], which is exactly the formula we implemented for our point to point ICP step (A.2). We obtain n individual ATEs, corresponding to pairs of ground truth and estimated pose for every frame.

Translation encodes rotation The above measures at time step i are the poses $E_i, F_i \in SE(3)$, respectively. For a sequence of such rigid body motions $g = (R, t) \in SE(3)$, we only need to consider the translational component t, because it encodes the rotational component [46]. In RPE, rotational errors show up as translational errors when the camera is moved, thus the authors found that comparing translational errors only is sufficient. In ATE, rotational errors typically also manifest themselves in wrong translations and are thus indirectly also captured.

Statistics For a trajectory of length n, we compute $n-1$ RPEs and n ATEs, represented by the norms e_i of their translational components $t_i \in \mathbb{R}^3$, so $e_i = ||t_i||_2 = \sqrt{t_{i,x}^2 + t_{i,y}^2 + t_{i,z}^2} \in \mathbb{R}$. Now we can use some simple statistics to more compactly describe a set of scalar errors of length n (similarly for $n-1$). A popular choice in our domain seems to be the Root Mean Squared (RMS) error:

$$rms = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^2} \tag{5.5}$$

Additionally to the **rms**, we compute standard statistics like the **mean** that gives less weight to outliers, the **min** and the **max** as well as the standard deviation **std**.

5.3.1. Discussion

Table 5.1 and Table 5.2 show the **accuracy** on real and synthetic data after each stage of our algorithm as well as the improvements between the stages. It can be observed that the pairwise refinement reduced both RPE and ATE errors by about 90 percent, except for our failure case of the symmetric juice box. If this initial alignment was good enough (for all except Pumba and juice box), the multiview refinement further reduced RPE as well as ATE, but the percentual improvement in ATE was usually twice that of RPE. This makes sense, considering that ATE measures global consistency, which is what our multiview refinement tries to improve.

Further we conducted experiments of **speed vs. accuracy** while changing different parameters and leaving other parameters fixed. The preprocessing running times were not affected by the parameters. The pairwise refinement converges blazingly fast (under 1 second for the whole sequence) because we use k-d-trees and an efficient direct implementation to minimize the ICP point to plane energy, as given in section A.3.

Table 5.3: Accuracy can be improved by lowering the distance sampling rate τ_d (4.6). A good balance between speed and accuracy is achieved with our default value of 0.10. A higher value accelerates the coarse pair-wise alignment, but lets the multiview refinement converge slower and to a worse solution when fed with these coarser initial estimates.

Table 5.4: Increasing the number of frames $nFrames$ to which one tries to match the current frame will of course increase the running time of the coarse pairwise alignment step while leaving the other running times untouched. But for some sequences, such as bunny sphere, this is absolutely necessary, since it happens that consecutive frames don't overlap. We did not include the test with -nFrames 02 in the plot, because there, the final accuracy is 0.1 for RPE and 0.2 for ATE, a factor of 10 worse than for -nFrames 03, while the running time is about the same as for nFrames 03, because the multiview refinement runs longer when given such bad initial estimates.

Table 5.5: Increasing the number of nearest neighbors knn to which we add edges before the multiview refinement step only affect the runtime of this last step. The accuracy really depends on the actual topology of the underlying graph. For bunny sphere, the best accuracy is achieved if each vertex has around 5-6 neighbors, which corresponds to the nice triangulation of this sphere as seen in the qualitative section.

sequence params	S	RPE [mm/%]					ATE [mm/%]				
		rms	mean	std	min	max	rms	mean	std	min	max
bunny2	A	32.311	28.996	14.254	4.932	68.382	74.097	68.679	27.812	20.907	141.886
	↓	88	89	85	92	88	91	92	89	91	89
	B	3.936	3.330	2.098	0.412	8.365	6.610	5.803	3.165	1.893	14.910
	↓	15	14	17	-59	17	21	15	47	-11	32
	C	3.344	2.859	1.735	0.655	6.962	5.212	4.932	1.685	2.110	10.070
bunny -step 20 -knn 5	A	70.769	60.943	35.975	10.246	154.445	305.968	281.249	120.479	95.815	555.003
	↓	83	82	83	86	83	64	66	52	71	61
	B	12.325	10.728	6.068	1.449	26.618	111.651	95.818	57.313	27.865	215.305
	↓	15	16	10	-16	-1	53	50	64	44	57
	C	10.494	8.964	5.457	1.684	27.011	52.448	48.233	20.601	15.683	91.872
blade -step 15	A	90.248	70.332	56.551	2.542	239.625	288.147	253.961	136.134	43.080	494.748
	↓	62	72	50	52	47	58	61	50	77	52
	B	34.706	19.963	28.390	1.231	126.032	120.684	99.421	68.411	10.033	238.690
	↓	9	22	3	19	-0	56	56	55	4	35
	C	31.632	15.613	27.511	0.997	126.368	53.064	43.293	30.684	9.674	155.886
phone -step 20 -knn 5	A	108.270	90.488	59.449	22.374	312.704	479.117	468.869	98.564	230.952	756.462
	↓	80	79	81	86	83	70	73	24	87	61
	B	21.926	18.739	11.383	3.023	52.713	145.851	125.348	74.568	29.053	297.482
	↓	22	24	18	70	21	49	48	56	31	49
	C	17.001	14.200	9.349	0.918	41.631	73.658	65.768	33.167	19.903	151.880
Pumba -diamM 0.07 -step 15 -knn 4 -dmax 0.05	A	111.926	90.269	66.172	19.144	283.182	367.296	337.233	145.534	116.170	702.401
	↓	83	84	81	92	79	74	72	82	56	80
	B	19.295	14.810	12.367	1.475	58.741	96.967	93.496	25.710	51.569	143.857
	↓	-12	-8	-17	19	-6	-16	-17	-14	-38	-19
	C	21.519	15.982	14.411	1.189	62.218	112.917	109.038	29.342	71.380	171.405
cow -step 15 -diamM 0.1	A	118.987	95.633	70.796	7.854	322.196	428.768	375.221	207.489	81.748	867.979
	↓	87	87	87	92	85	81	81	82	69	82
	B	15.478	12.315	9.377	0.652	47.395	80.812	71.985	36.725	25.639	159.731
	↓	2	4	-0	23	-4	-6	-6	-9	29	1
	C	15.150	11.881	9.401	0.501	49.189	85.912	76.066	39.936	18.306	158.271
book -step 15 -diamM 0.2	A	71.394	56.598	43.518	14.760	170.864	467.909	430.521	183.278	139.108	867.919
	↓	73	82	63	98	46	93	93	93	90	89
	B	19.024	10.046	16.155	0.300	92.019	33.938	31.092	13.604	14.147	96.956
	↓	3	12	-0	-65	1	33	34	26	73	21
	C	18.451	8.834	16.199	0.496	90.800	22.814	20.475	10.062	3.803	76.508
bench vise -step 15	A	72.998	67.519	27.747	13.559	138.257	215.475	184.514	111.283	41.895	493.878
	↓	83	85	74	90	78	76	75	78	72	82
	B	12.287	9.979	7.168	1.303	30.414	52.579	46.602	24.346	11.843	90.784
	↓	26	23	34	-4	41	59	58	62	45	55
	C	9.031	7.676	4.758	1.361	17.924	21.737	19.662	9.270	6.549	40.583

Table 5.1.: Accuracy on real datasets: *A,B,C stand for the stage of the Algorithm. ↓ indicates the relative improvements between those stages in %. Passed parameters appear below the sequence name, otherwise the default values are used*

sequence params	S	RPE [mm/%]					ATE [mm/%]				
		rms	mean	std	min	max	rms	mean	std	min	max
bunny sphere -knn 5 -nFrames 5	A	65.799	54.217	37.283	3.377	171.165	125.697	108.222	63.937	11.194	322.437
	↓	69	69	70	41	72	76	76	78	42	78
	B	20.212	16.833	11.189	1.992	48.419	29.737	26.353	13.776	6.541	70.087
	↓	34	35	33	76	44	58	57	61	48	65
	C	13.278	10.929	7.540	0.482	27.300	12.445	11.256	5.309	3.381	24.489
cow wavy -step 5	A	38.893	32.589	21.228	4.434	110.011	78.353	71.332	32.418	23.160	146.695
	↓	87	88	86	95	90	86	86	87	95	86
	B	4.915	3.947	2.929	0.211	10.695	10.674	9.814	4.196	1.260	20.307
	↓	18	29	2	-17	-5	50	51	46	26	39
	C	4.021	2.803	2.883	0.247	11.260	5.301	4.789	2.273	0.938	12.344
leopard wavy -step 5 -diamM 0.3 -knn 4	A	36.106	33.840	12.590	14.916	69.583	70.558	62.355	33.019	14.593	157.004
	↓	82	83	74	89	80	87	87	88	93	89
	B	6.569	5.671	3.314	1.567	14.246	9.335	8.415	4.041	1.048	17.257
	↓	15	28	-15	86	-18	43	43	43	-148	30
	C	5.594	4.109	3.796	0.226	16.851	5.280	4.756	2.292	2.599	12.005
juice box wavy -step 3	A	124.981	39.776	118.482	1.076	549.358	520.406	442.792	273.418	159.439	934.931
	↓	3	21	1	57	0	2	1	4	17	2
	B	121.612	31.256	117.526	0.462	547.635	511.973	438.984	263.457	132.380	914.523
	↓	-1	-0	-1	14	-0	0	1	-2	-16	-1
	C	122.356	31.375	118.265	0.398	549.396	509.811	433.165	268.841	153.275	920.450
Kenny wavy -step 5 -diamM 0.4 -knn 4	A	146.957	137.339	52.292	18.787	211.494	478.807	467.906	101.589	239.615	709.235
	↓	86	89	76	87	72	90	92	74	97	87
	B	20.004	15.523	12.618	2.400	58.923	46.516	38.589	25.973	7.942	92.715
	↓	12	15	7	42	15	26	25	28	58	28
	C	17.619	13.126	11.753	1.393	50.151	34.490	29.064	18.571	3.316	66.753
teddy circle -diamM 0.3 -step 4 -knn 4	A	77.922	71.636	30.661	10.509	152.740	366.251	356.196	85.228	240.767	573.667
	↓	94	95	91	91	92	95	96	89	99	94
	B	4.712	3.887	2.664	0.923	11.906	17.075	14.112	9.614	3.076	34.742
	↓	41	37	49	63	37	45	47	41	70	46
	C	2.789	2.429	1.369	0.341	7.509	9.331	7.435	5.638	0.935	18.771
teddy wavy -step 4 -diamM 0.4	A	84.573	75.949	37.206	13.004	168.541	333.451	317.047	103.300	151.981	626.474
	↓	88	89	85	89	85	90	90	88	94	90
	B	10.052	8.336	5.617	1.438	24.686	32.705	30.174	12.615	9.685	64.118
	↓	21	19	25	59	17	50	48	62	20	59
	C	7.963	6.754	4.218	0.596	20.374	16.495	15.790	4.771	7.737	26.207
teddy abrupt -nFrames 5 -knn 4 -diamM 0.5	A	73.390	59.036	43.599	8.475	221.957	274.761	255.011	102.287	58.764	600.277
	↓	82	81	84	83	85	86	86	82	83	84
	B	13.423	11.397	7.091	1.456	32.225	39.428	35.027	18.102	10.071	93.824
	↓	33	30	40	48	37	50	48	56	86	57
	C	9.028	7.976	4.230	0.754	20.436	19.715	18.059	7.908	1.377	40.025

Table 5.2.: Accuracy on synthetic datasets: *A,B,C stand for the stage of the Algorithm. ↓ indicates the relative improvements between those stages in %. Passed parameters appear below the sequence name, otherwise the default values are used*

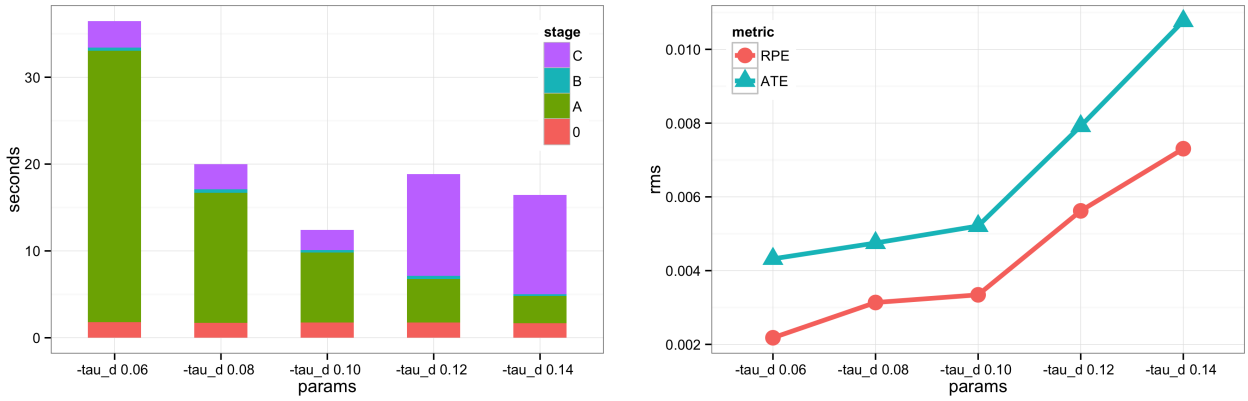


Table 5.3.: Speed vs. accuracy on **bunny2** : *The distance sampling rate* τ_d (4.6) *is modified*

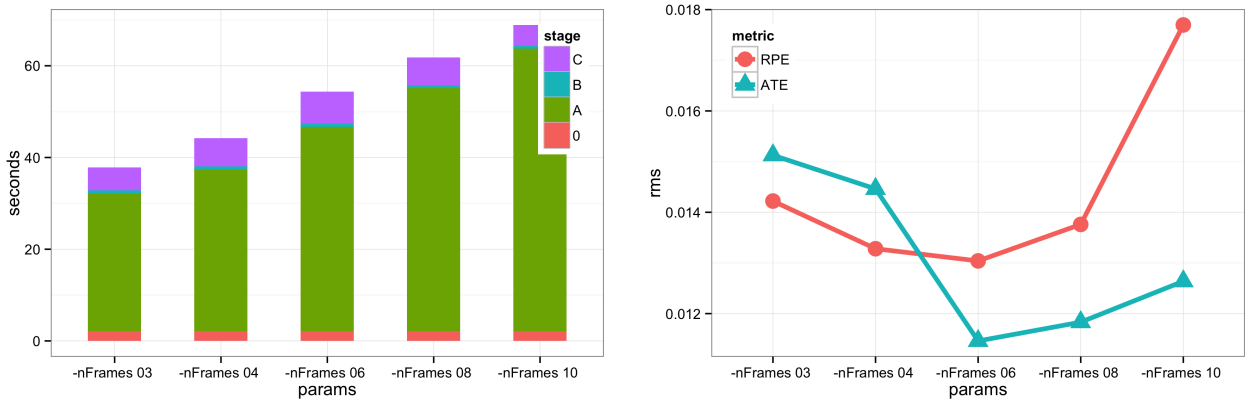


Table 5.4.: Speed vs. accuracy on **bunny sphere** -knn 06 : *The number* $nFrames$ *of frames to match to during the coarse pairwise alignment is modified*

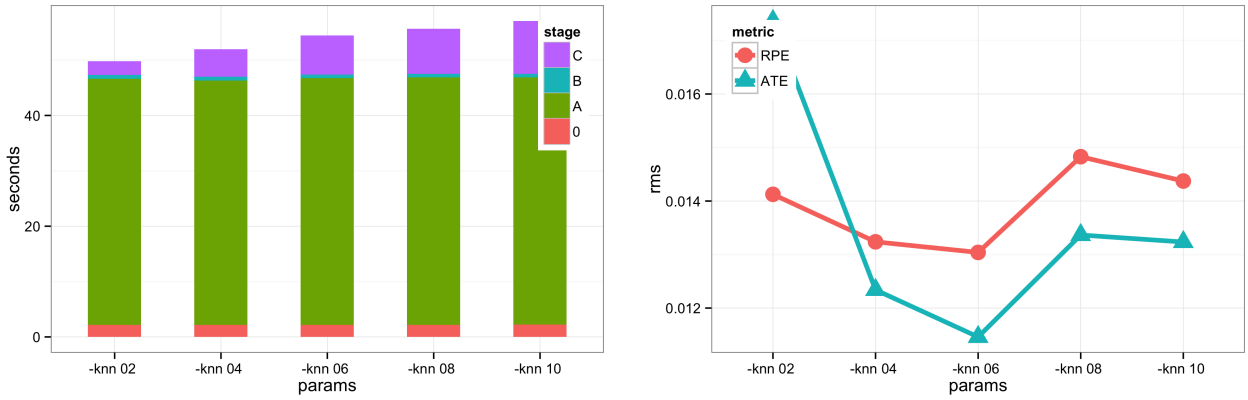


Table 5.5.: Speed vs. accuracy on **bunny sphere** -nFrames 06 : *The number of* knn *nearest neighbors to add to the pose graph before multiview refinement is modified*

6. Summary and outlook

6.1. Conclusion

We developed a method that is able to reconstruct small objects from range images using only geometric information. Our method works well for objects that exhibit a certain amount of variation in geometry. We are able to reconstruct most objects with decent accuracy in about one minute. Failure cases are self-similar, symmetric or planar objects, due to the non-discriminative power that our point pair feature and point to plane ICP exhibit in these cases. The multiview refinement step converges if the initialization is sufficiently close to the minimum and if the pose graph structure actually represent the underlying topology of the sequence.

6.2. Future work

While implementing the whole pipeline, consisting of preprocessing (**0**), coarse pairwise matching and alignment (**A**), pairwise refinement (**B**) and multiview refinement (**C**), we identified many subproblems that could need some further investigation.

The first step in (**0**), the segmentation, is based on the assumption of a planar support surface. It would be nice to develop a segmentation algorithm that allows in-hand scanning, meaning that the object is moved freely in front of the camera in someones hand. Segmentation could then be done based on skin color or based on the shape of a hand. This would allow to reconstruct the bottom side of the object, which is normally not reconstructed because it sits on the surface.

Our complete approach (**0,A,B,C**) needs a lot of parameters that need to be set correctly, depending on the data, so that our algorithm converges. We try to leave most of them in their default values or make them dependent on other parameters, so we don't have to set too many of them. The accuracy of the final result is very parameter dependent, as seen in the last section, but the results don't necessarily get better the more time one invests. Thus, it would be really nice if we could determine the best parameters for a given sequence by ourselves or even adapt them while the algorithm is running. Another option would be to allow for more interactivity, i.e. asking the user at intermediate stages if the current alignment looks already good to him or not. Depending on his answer, we could adapt certain parameters or try to match to a frame that the user specifies instead of blindly trying to match to the last $nFrames$.

Apart from these rather practical problems, we should really improve the running time of our point pair feature matching algorithm (**A**), which is the slowest part of the whole pipeline. A real problem is that it is quadratic in the size of points.

Range images Let us first only consider possible extensions that use only range images and no additional information. In our algorithm, we downsampled the point clouds in the beginning and then only worked on these downsampled versions. A straightforward extension to improve the accuracy is thus to use all the available data instead. The loss in speed can be counteracted by the use of a coarse-to-fine scheme. However, a certain degree of downsampling is necessary for our point-pair features to work robustly.

Our normals were obtained from the eigenvectors of a point set PCA. But this PCA contains even more information through its three eigenvalues in the form of curvature information that we already computed, but did not use. This information could be used to augment the point pair feature or prune wrong correspondences. Additionally, curvature information can be integrated into ICP in various ways. Though this will not help for totally planar objects, curvature can be used to effectively reduce the size of points to the most stable ones that usually have high curvature. This geometrically stable sampling strategy was already done for ICP in [14]. It can also be applied to the matching step of our point pair features and has the nice side effect to improve its performance, since we have to consider fewer points, i.e. only the more interesting ones which have high curvature.

Color images A straightforward extension, which became popular with the advent of the Kinect, is to additionally consider the color images. One may incorporate them into our point-based geometry approach for selection, matching and pruning of point correspondences. For Point Pair Features (PPFs), this has been done successfully in [6]. Concerning ICP, there have been various extension to the standard one as well as to GICP to also include color compatibility, usually based on some color space that is more invariant to illumination changes than RGB such as HUV or LAB.

Nevertheless, the most successful approaches work on 2D color images directly. These pixel-based approaches employ an either sparse or dense matching of image patches or pixels and reduce a photometrical reprojection error. These methods are fast because they can exploit the organized grid structure of the 2D images. It would be interesting to extend this idea of organized grid structures from color to range images to improve matching speeds, meaning that we skip the whole backprojection and downsampling steps and work on the organized range images directly.

Multiview refinement The running times of our multiview refinement stage (**C**) are very dependent on the initialization. During the inner Levenberg-Marquardt (LM) iterations, one has to find a value of λ which decreases the error. The LM implementation in g2o sometimes took a really long time for these inner iterations, and some other times it was quite fast. In future work, we are interested to port our own Multiview LM-ICP algorithm written in Matlab, which works well on small problem sizes, to C++ using the Ceres Solver[1] of Agarwal et al. [1], developed originally at Google for the solution of sparse Bundle Adjustment (BA) problems to reconstruct scenes in Google Street View. Since it is used in production code, it has better documentation, a cleaner and more stable API than g2o and nice features such as automatic derivatives, which allow to switch the parametrization of x seamlessly without introducing new errors through wrong Jacobians.

[1] http://ceres-solver.org

A. Core Algorithms

A.1. Point set PCA

Given a set of 3 or more points, which should correspond to the neighborhood of a point on a surface, one can easily calculate its centroid as well as estimate surface normals and curvature using Principal Component Analysis (PCA) of point sets as explained in section 3.4.

```
1  void pointSetPCA(const vector<Vector3f>& pts, Vector3f& centroid, Vector3f&
       normal, float& curvature){
2
3      assert(pts.size()>=3); //otherwise normals are undetermined
4      Map<const Matrix3Xf> P(&pts[0].x(),3,pts.size());
5
6      centroid = P.rowwise().mean();
7      MatrixXf centered = P.colwise() - centroid;
8      Matrix3f cov = centered * centered.transpose();
9
10     //eigvecs sorted in increasing order of eigvals
11     SelfAdjointEigenSolver<Matrix3f> eig(cov);
12     normal = eig.eigenvectors().col(0); //is already normalized
13     if (normal(2) > 0) normal = -normal; //flip towards camera
14     Vector3f eigVals = eig.eigenvalues();
15     curvature = eigVals(0) / eigVals.sum();
16 }
```

Listing A.1: Point set PCA *(Eigen/C++)*

A.2. Point to point ICP

$$E = \sum_{i=1}^{N} ||Rp_i + t - q_i||^2 \tag{3.24}$$

R is a 3 x 3 orthonormal matrix and t is a 3 x 1 translation vector. Following [9][3.1][1] and [26][Appendix C], minimizing E can be done using SVD. First, one calculates the centroids of the point sets. $\bar{p} = \frac{1}{N}\sum_{i=1}^{N} p_i$ and $\bar{q} = \frac{1}{N}\sum_{i=1}^{N} q_i$. Then, we need to center each point in the sets by subtracting their centroids $\tilde{p}_i = p_i - \bar{p}$ and $\tilde{q}_i = q_i - \bar{q}\ \forall i \in [1, N]$. We define the correlation matrix K of the centered point sets as the sum of their outer products, which can also compactly be written as a matrix product of the matrices whose columns are made up of the centered points:

$$K = \sum_{i=1}^{N} \tilde{q}_i \tilde{p}_i{}^T = \tilde{Q}\tilde{P}^T, \qquad \tilde{P} = \begin{bmatrix} | & \dots & | \\ \tilde{p}_1 & \dots & \tilde{p}_N \\ | & \dots & | \end{bmatrix}$$

The singular value decomposition of K is given by $K = U\Sigma V^T$. Then, the optimal rotation matrix is given as $R = UV^T$ and the optimal translation vector as $t = \bar{q} - R \cdot \bar{p}$. If R was actually a reflection ($|R| < 0$), we need to flip its last column to obtain a rotation [48]. The basic algorithm was implemented in less then 20 lines of Matlab or C++ Code, which is provided here.

[1] http://graphics.stanford.edu/~smr/ICP/comparison/eggert_comparison_mva97.pdf

```
function [ T ] = pointToPoint(src,dst)
    N = size(src,1); assert(size(dst,1) == N);
    ps = src';
    qs = dst'; %ps and qs have 3 rows, N columns
    p_dash = mean(ps,2);
    q_dash = mean(qs,2);
    ps_centered = ps - repmat(p_dash,1,size(ps,2));
    qs_centered = qs - repmat(q_dash,1,size(qs,2));
    K = qs_centered * ps_centered';
    [U,~,V] = svd(K);
    R = U * V';
    if(det(R)<0)
        R(:,3) = R(:,3) * (-1);
    end
    T = eye(4);
    T(1:3,1:3) = R;
    T(1:3,4) = q_dash - R*p_dash;
end
```

Listing A.2: ICP point to point step *(Matlab)*

```
Isometry3f pointToPoint(vector<Vector3f>&src,vector<Vector3f>&dst){
    int N = src.size(); assert(N==dst.size());
    Map<Matrix3Xf> ps(&src[0].x(),3,N); //maps vector<Vector3f>
    Map<Matrix3Xf> qs(&dst[0].x(),3,N); //to Matrix3Nf columnwise
    Vector3f p_dash = ps.rowwise().mean();
    Vector3f q_dash = qs.rowwise().mean();
    Matrix3Xf ps_centered = ps.colwise() - p_dash;
    Matrix3Xf qs_centered = qs.colwise() - q_dash;
    Matrix3f K = qs_centered * ps_centered.transpose();
    JacobiSVD<Matrix3f> svd(K, ComputeFullU | ComputeFullV);
    Matrix3f R = svd.matrixU()*svd.matrixV().transpose();
    if(R.determinant()<0){
        R.col(2) *= -1;
    }
    Isometry3f T = Isometry3f::Identity();
    T.linear() = R;
    T.translation() = q_dash - R*p_dash; return T;
}
```

Listing A.3: ICP point to point step *(Eigen/C++)*

A.3. Point to plane ICP

$$E = \sum_{i=1}^{N} ||(Rp_i + t - q_i) \cdot n_{q_i}||^2 \tag{3.25}$$

An approximate closed-form solution, based on the assumption of an infinitesimal rotational part (subsection 3.3.4), can be obtained by linearizing R. This approximation only makes sense for small angles, as are expected in late iterations of the ICP algorithm [26][Appendix D]. Then E can be rewritten as:

$$E = \sum_{i=1}^{N} ||(p_i - q_i) \cdot n_{q_i} + r \cdot (p_i \times n_{q_i}) + t \cdot n_{q_i}||^2, \quad r = [r_x, r_y, r_z]^T$$

We can arrange all the summands into a matrix expression [29] $Ax = b$, which is an over-constrained system of linear equations, given $N > 6$ pairs of point correspondences.

$$A = \begin{bmatrix} (p_1 \times n_{q_1})^T & n_{q_1}^T \\ (p_2 \times n_{q_2})^T & n_{q_2}^T \\ \vdots & \vdots \end{bmatrix}_{N \times 6}, \; x = \begin{bmatrix} r \\ t \end{bmatrix}_{6 \times 1}, \; b = - \begin{bmatrix} (p_1 - q_1)^T n_{q_1} \\ (p_2 - q_2)^T n_{q_2} \\ \vdots \end{bmatrix}_{N \times 1}$$

This is a standard linear least-squares problem. Finding the x which minimizes $|Ax - b|^2$ can be done using the pseudo-inverse $A^\dagger$ of A. Then, the least-squares solution is simply given by $x = (A^T A)^{-1} A^T b = A^\dagger b$. For numerical stability, the pseudo-inverse $A^\dagger = V \Sigma^+ U^T$ should be computed from the SVD of $A = U \Sigma V^T$, where the diagonal matrix Σ^+ is the matrix formed by taking the inverse of the non-zero elements of the diagonal matrix Σ and leaving the zero elements unchanged [29].[2]

Another way to solve this linear least-squares problem is to solve the normal equations $A^T Ax = A^T b$ directly. This is even faster, since we don't need to compute an SVD, and also numerically stable. We define $C_{6\times6} = A^T A, d_{6\times1} = A^T b$ and simply solve

$$Cx = d \Leftrightarrow A^T Ax = A^T b$$

via Cholesky decomposition of the positive semidefinite matrix C. Interestingly, this result for C can also be derived by setting the partial derivatives of C with respect to the six degrees of freedom $[t, r]^T$ to zero.[3] Using symbolic calculations, we checked that C, d are of exactly the same form as in the lengthly derivation in [26][Appendix D]. Our Eigen/C++ implementation of this method follows.

[2] https://www.comp.nus.edu.sg/~lowkl/publications/lowk_point-to-plane_icp_techrep.pdf

[3] http://www.cs.princeton.edu/~smr/papers/icpstability.pdf

```
1  Isometry3f pointToPlane(vector<Vector3f> &src,vector<Vector3f> &dst,vector<
       Vector3f> &nor){
2      assert(src.size()==dst.size() && src.size()==nor.size());
3      Matrix<float,6,6> C; C.setZero();
4      Matrix<float,6,1> d; d.setZero();
5
6      for(uint i=0;i<src.size();++i){
7          Vector3f cro = src[i].cross(nor[i]);
8          C.block<3,3>(0,0) += cro*cro.transpose();
9          C.block<3,3>(0,3) += nor[i]*cro.transpose();
10         C.block<3,3>(3,3) += nor[i]*nor[i].transpose();
11         float sum = (src[i]-dst[i]).dot(nor[i]);
12         d.head(3) -= cro*sum;
13         d.tail(3) -= nor[i]*sum;
14     }
15     C.block<3,3>(3,0) = C.block<3,3>(0,3);
16
17     Matrix<float,6,1> x = C.ldlt().solve(d);
18     Isometry3f T = Isometry3f::Identity();
19     T.linear() = (   AngleAxisf(x(0), Vector3f::UnitX())
20                    * AngleAxisf(x(1), Vector3f::UnitY())
21                    * AngleAxisf(x(2), Vector3f::UnitZ())
22                  ).toRotationMatrix();
23     T.translation() = x.block(3,0,3,1);
24     return T;
25 }
```

Listing A.4: ICP point to plane step *(Eigen/C++)*

A.4. LM-ICP

We implemented Levenberg-Marquardt ICP (LM-ICP) to solve the point to point distance metric with the squared loss, although other distance and loss functions can be easily integrated.

$$E = \sum_{i=1}^{N} ||Rp_i + t - q_i||^2 \tag{3.24}$$

At each iteration, we need to compute the Levenberg update step to the transformation estimate:

$$x = -(J^T J + \lambda I)^{-1} J^T e \tag{3.37}$$

For these transformation updates x inside the inner loop, we use a minimal representation of SE(3) given by its twist coordinates ξ. These also have the advantage that their jacobians (3.9) have quite a nice form.

Fortunately, matrix exponentiation exp (3.8) is already implemented in Matlab as the expm function. That means we only have to implement the hat operator, which turns twist coordinates $\xi \in \mathbb{R}^6$, into a twist $\hat{\xi} \in se(3)$, and then call expm on that matrix to get the transformation matrix $T = exp(\hat{\xi}) \in SE(3)$ which we need to transform the points.

```
function [ T ] = se3Exp( xi )
    xi_hat = [0 -xi(6) xi(5) xi(1);
              xi(6) 0 -xi(4) xi(2);
              -xi(5) xi(4) 0 xi(3);
              0      0     0    0];
    T = expm(xi_hat);
end
```

Listing A.5: Twist coordinates to transformation matrix *(Matlab)*

We initialize λ with 1 and adapt it at each step by multiplying or dividing it with a factor of 10. Convergence is achieved when λ gets large, or equivalently, when the gradient gets small. There are also other possible convergence criteria based on the change in error, but the one above worked fine and convergence was achieved in about 5 steps in general, much less than the maximum iteration threshold of 100. The residual vector as well as the jacobian are directly computed in their stacked form. The transformation matrices we obtained were approximately the same as in the closed form solution A.2, up to differences of 10^{-7}.

```
function T = pointToPoint_lm_twists(src, dst)
    N = size(src,1); assert(size(dst,1) == N);
    T = eye(4);  x_hat=src;   lambda=1;

for i = 1:100
    %stack distance vectors into residual vector
    e = reshape((x_hat - dst)',[3*N,1]);

    %stack Jacobian
    J = zeros(3*N,6); O = zeros(N,1); I = ones(N,1);
    x1 = x_hat(:,1); x2 = x_hat(:,2); x3 = x_hat(:,3);
    J(1:3:3*N,:) = [I O O   O  x3 -x2];
    J(2:3:3*N,:) = [O I O -x3   O  x1];
    J(3:3:3*N,:) = [O O I  x2  -x1  O];

    b = (J' * e); %gradient

    if(max(abs(b))< 0.00001)
        break; %convergence in gradient, lambda is big
    end

    %Levenberg step
    upd = -(J' * J + lambda*eye(6))^(-1) * b;

    %compute new residual vector
    T_new = se3Exp(upd) * T; %exponential map left multiply
    R = T_new(1:3,1:3); t = T_new(1:3,4);
    x_hat_new = (R*src' + repmat(t,1,N))'; %project
    e_test = reshape((x_hat_new - dst)',[3*N,1]);

    %test if we reduced the error (sum of squared residuals)
    if(e_test'*e_test < e'*e) %error reduced
        T = T_new; x_hat = x_hat_new; %accept update
        lambda = lambda / 10; %towards Gauss-Newton
    else %error increased, discard update
        lambda = lambda * 10; %towards gradient descent
    end
end
```

Listing A.6: LM-ICP point to point step *(Matlab)*

Bibliography

[1] Sameer Agarwal, Keir Mierle, and Others. Ceres solver. http://ceres-solver.org.

[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] Dana H Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.

[4] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics, 1992.

[5] Yang Chen and Gérard Medioni. Object modeling by registration of multiple range images. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 2724–2729. IEEE, 1991.

[6] Changhyun Choi and Henrik I Christensen. 3d pose estimation of daily objects using an rgb-d camera. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 3342–3349. IEEE, 2012.

[7] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58:15–16, 2006.

[8] Bertram Drost, Markus Ulrich, Nassir Navab, and Slobodan Ilic. Model globally, match locally: Efficient and robust 3d object recognition. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 998–1005. IEEE, 2010.

[9] David W Eggert, Adele Lorusso, and Robert B Fisher. Estimating 3-d rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications*, 9 (5-6):272–290, 1997.

[10] Felix Endres, Jürgen Hess, Nikolas Engelhard, Jürgen Sturm, Daniel Cremers, and Wolfram Burgard. An evaluation of the rgb-d slam system. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1691–1696. IEEE, 2012.

[11] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *Computer Vision–ECCV 2014*, pages 834–849. Springer, 2014.

[12] Simone Fantoni, Umberto Castellani, and Andrea Fusiello. Accurate and automatic alignment of range surfaces. In *3DIMPVT*, pages 73–80. Citeseer, 2012.

[13] Andrew W Fitzgibbon. Robust registration of 2d and 3d point sets. *Image and Vision Computing*, 21(13):1145–1153, 2003.

[14] Natasha Gelfand, Leslie Ikemoto, Szymon Rusinkiewicz, and Marc Levoy. Geometrically stable sampling for the icp algorithm. In *3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings. Fourth International Conference on*, pages 260–267. IEEE, 2003.

[15] Claus Gramkow. On averaging rotations. *Journal of Mathematical Imaging and Vision*, 15(1-2):7–16, 2001.

[16] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[17] Richard Hartley, Jochen Trumpf, Yuchao Dai, and Hongdong Li. Rotation averaging. *International journal of computer vision*, 103(3):267–305, 2013.

[18] Stefan Hinterstoisser, Stefan Holzer, Cedric Cagniart, Slobodan Ilic, Kurt Konolige, Nassir Navab, and Vincent Lepetit. Multimodal templates for real-time detection of texture-less objects in heavily cluttered scenes. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 858–865. IEEE, 2011.

[19] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Gary Bradski, Kurt Konolige, and Nassir Navab. Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes. In *Computer Vision–ACCV 2012*, pages 548–562. Springer Berlin Heidelberg, 2013.

[20] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *COMPUTER GRAPHICS (SIGGRAPH '92 PROCEEDINGS)*, volume 26, pages 71–78. ACM, 1992.

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

[22] 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.

[23] Andrew E. Johnson and Martial Hebert. Using spin images for efficient object recognition in cluttered 3d scenes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 21(5):433–449, 1999.

[24] Michael Jones and Paul Viola. Fast multi-view face detection. *Mitsubishi Electric Research Lab TR-20003-96*, 3:14, 2003.

[25] Christian Kerl, Jürgen Sturm, and Daniel Cremers. Dense visual slam for rgb-d cameras. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 2100–2106. IEEE, 2013.

[26] Hans Martin Kjer and Jakob Wilm. Evaluation of surface registration algorithms for pet motion correction. Bachelor's thesis, Technical University of Denmark, 2010.

[27] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234. IEEE, 2007.

[28] Rainer Kummerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. g 2 o: A general framework for graph optimization. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3607–3613. IEEE, 2011.

[29] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. *Chapel Hill, University of North Carolina*, 2004.

[30] Y. Ma, S. Soatto, J. Kosecká, and S.S. Sastry. *An Invitation to 3-D Vision: From Images to Geometric Models*. Interdisciplinary Applied Mathematics. Springer New York, 2005. ISBN 9780387008936.

[31] F Landis Markley, Yang Cheng, John Lucas Crassidis, and Yaakov Oshman. Averaging quaternions. *Journal of Guidance, Control, and Dynamics*, 30(4):1193–1197, 2007.

[32] Ajmal S Mian, Mohammed Bennamoun, and Robyn A Owens. Automatic correspondence for 3d modeling: an extensive review. *International Journal of Shape Modeling*, 11 (02):253–291, 2005.

[33] Ajmal S Mian, Mohammed Bennamoun, and Robyn Owens. Three-dimensional model-based object recognition and segmentation in cluttered scenes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(10):1584–1601, 2006.

[34] Niloy J Mitra, Natasha Gelfand, Helmut Pottmann, and Leonidas Guibas. Registration of point cloud data from a geometric optimization perspective. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 22–31. ACM, 2004.

[35] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011.

[36] Mark Pauly, Markus Gross, and Leif P Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings of the conference on Visualization'02*, pages 163–170. IEEE Computer Society, 2002.

[37] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[38] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pages 145–152. IEEE, 2001.

[39] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3d model acquisition. In *ACM Transactions on Graphics (TOG)*, pages 438–446. ACM, 2002.

[40] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast point feature histograms (fpfh) for 3d registration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3212–3217. IEEE, 2009.

[41] Renato F Salas-Moreno, Richard A Newcombe, Hauke Strasdat, Paul HJ Kelly, and Andrew J Davison. Slam++: Simultaneous localisation and mapping at the level of objects. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 1352–1359. IEEE, 2013.

[42] Aleksandr Segal, Dirk Haehnel, and Sebastian Thrun. Generalized-icp. In *Robotics: Science and Systems*, volume 2, 2009.

[43] Steven M Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Computer vision and pattern recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 519–528. IEEE, 2006.

[44] Miroslava Slavcheva. Unified pipeline for 3d reconstruction from rgb-d images using coloured truncated signed distance fields. Master's thesis, Technische Universität München.

[45] Frank Steinbrucker, Jürgen Sturm, and Daniel Cremers. Real-time visual odometry from dense rgb-d images. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 719–722. IEEE, 2011.

[46] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 573–580. IEEE, 2012.

[47] Federico Tombari, Samuele Salti, and Luigi Di Stefano. Unique signatures of histograms for local surface description. In *Computer Vision–ECCV 2010*, pages 356–369. Springer, 2010.

[48] Shinji Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Transactions on pattern analysis and machine intelligence*, 13(4): 376–380, 1991.

[49] Michael W Walker, Lejun Shao, and Richard A Volz. Estimating 3-d location parameters using dual number quaternions. *CVGIP: image understanding*, 54(3):358–367, 1991.

[50] Zhengyou Zhang. Iterative point matching for registration of free-form curves and surfaces. *International journal of computer vision*, 13(2):119–152, 1994.

[51] Timo Zinßer, Jochen Schmidt, and Heinrich Niemann. Point set registration with integrated scale estimation. In *Vortrag: Eighth International Conference on Pattern Recognition and Image Processing, Belarusian State University of Informatics and Radioelectronics, International Association for Pattern Recognition, Minsk, Belarus*, volume 18, 2005.