Real-Time Object Tracking For Assembly

Echtzeit Objektverfolgung für Konstruktions- und Assemblierungsanwendungen
Bachelor thesis in the field of study “Computational Engineering” by Leon Magnus
Date of submission: March 25, 2022

1. Review: Niklas Funk, M.Sc.
2. Review: Pascal Klink, M.Sc.
3. Review: Prof. Jan Peters, Ph.D.
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Leon Magnus, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 25. März 2022

Leon Magnus

Abstract

In recent years, the importance of object tracking in many technological areas, such as augmented reality, autonomous driving, robotic interaction, or assembly applications, is increasing rapidly as autonomous systems interact more and more in and with the environment. Since most real-world tasks take place in partially observable environments, noise, inaccuracies, and occlusions have a major impact on the tracking of objects. Therefore, many learning- and model-based approaches exist that try to determine reliable poses even under the previously mentioned influences.

In this work, we present a model-based object tracking algorithm based on the Point Set Registration (PSR) method proposed by the FilterReg [1] paper. Hereby, we develop a framework that enables object tracking with PSR algorithms and propose several extensions e.g. additional features to improve the tracking result. Moreover, we test our proposed tracking algorithm on accuracy and real-time capability on a dataset that was recorded in a real robotic assembly setup.

From our results, we conclude that our proposed object tracking algorithm achieves a Euclidean accuracy of $2\mathrm{cm}$ compared to the ground truth, and has a maximum error of $20°$ in the rotation, given in Euler angles, while frame rates of $30\mathrm{Hz} - 76\mathrm{Hz}$ can be achieved. Furthermore, we state that our efficient object tracking algorithm can handle real-time applications, and can be further explored to improve the tracking result.

Zusammenfassung

In den letzten Jahren hat die Bedeutung von Objektverfolgung in vielen technologischen Bereichen wie Augmented Reality, autonomes Fahren, Roboterinteraktion oder Assemblierungsanwendungen rapide zugenommen, da autonome Systeme mehr und mehr in und mit der Umgebung interagieren. Da viele Aufgaben in der realen Welt in teilweise beobachtbaren Umgebungen stattfinden, haben Rauschen, Ungenauigkeiten und Verdeckungen große Auswirkungen auf die Verfolgung von Objekten. Daher existieren viele lern- und modellbasierte Ansätze, die versuchen, auch unter den genannten Einflüssen zuverlässige Posen zu bestimmen.

In dieser Arbeit stellen wir einen modellbasierten Objektverfolgungsalgorithmus vor, der auf der PSR-Methode basiert, die in dem Paper FilterReg [1] vorgeschlagen wurde. Dabei entwickeln wir eine Struktur, die Objektverfolgung mit PSR-Algorithmen ermöglicht und schlagen mehrere Erweiterungen, wie z. B. zusätzliche Merkmale zur Verbesserung des Tracking-Ergebnisses vor. Darüber hinaus testen wir unseren vorgeschlagenen Tracking-Algorithmus auf Genauigkeit und Echtzeitfähigkeit an einem Datensatz, der in einer realen Roboter-Assemblierungsumgebung aufgezeichnet wurde.

Aus unseren Ergebnissen schließen wir, dass der von uns vorgeschlagene Objektverfolgungsalgorithmus eine euklidische Genauigkeit von $2\mathrm{cm}$ im Vergleich zur tatsächlichen Pose des zu verfolgenden Objektes erreicht und einen maximalen Fehler von $20°$ in der Rotation, angegeben in Euler-Winkeln, aufweist, während Wiederholraten von $30\mathrm{Hz} - 76\mathrm{Hz}$ erreicht werden können. Weiterhin folgern wir daraus, dass unser effizienter Objektverfolgungsalgorithmus Echtzeianwendungen verarbeitern kann und weiterentwickelt werden kann, um das Verfolgungsresult weiter zu verbessern.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
CPD	Coherent Point Drift
DDU	Digital Design Unit at TU Darmstadt
EM	Expectation Maximization
GMM	Gaussian Mixture Model
ICP	Iterative Closest Point
PDF	Probability Density Function
PSR	Point Set Registration
SVD	Singular Value Decomposition

1 Introduction & Motivation

Object tracking and pose estimation play a fundamental role in many technological domains, such as Augmented Reality, autonomous driving, robotic interaction, and assembly tasks. In these areas, it is important to reliably track an object despite occlusion, noise, and varying environmental conditions that occur in these partially observable environments. This thesis focuses on the task of robotic assembly. In this context, robotic assembly is defined as the task of moving and stacking parts to create more complex structures. In this very demanding task, the exact pose of all objects involved is required during execution. Therefore, real-time estimation of poses is crucial.

Systems like the motion capturing system Optitrack [2] provide real-time pose estimation of an object, but therefore an inflexible tracking system containing multiple cameras is needed. Furthermore, tracking an object with this system requires markers on the to be tracked object. Hereby, both prerequisites for the use in assembly tasks cannot be guaranteed, since assembly applications are not fixed to one environment, whereby inflexible systems are disadvantageous. In addition, as described before, robotic assembly is about stacking parts to construct more complex structures. Therefore systems that track the object by markers on it e.g. Optitrack [2] and Apriltag [3] either change the structure of the object or the markers are no longer visible due to the assembly of objects. Therefore our goal is to develop an object tracking algorithm that is independent of additional markers on the to be tracked object. Here many approaches attempt to determine the pose of an object using depth cameras. These cameras have the advantage of being compact and therefore very suitable for flexible use in many application areas. To determine poses from depth images, learning- and model-based approaches exist. In the field of learning-based approaches, current research like [4, 5, 6] propose to use neural networks to learn the object representation and then predict the pose of the object in a scene. Hereby, it is beneficial that high process frame rates can be reached. However, these approaches are mostly limited to tracking the object that they were trained on. In contrast, model-based approaches have the benefit to track a wide range of objects. Nevertheless, these

approaches are mostly computationally demanding. Here, similar research to Radowski [7] uses Point Set Registration (PSR) methods to predict the pose of an object in a scene.

In this thesis, we propose an efficient object tracking algorithm based on the PSR algorithm stated in [1] and extend it to be able to track moving objects in partially observable environments. Furthermore, we propose extensions like pre-processing steps or additional color features to reach further improved tracking results. Moreover, we evaluate our object tracking algorithm on a dataset [8], which has been recorded in a robotic assembly setup. Hereby, we focus on the evaluation of the tracking accuracy and especially on real-time capability, which we assume to be given if the tracking algorithm can handle at least an input frame rate of $30\mathrm{Hz}$.

The structure of this thesis is as follows. In Chapter 2, we present related work and foundations on which we base our proposed object tracking algorithm. Here we discuss mainly the Iterative Closest Point (ICP) algorithm and how it relates to our utilized PSR algorithm, which is presented in the FilterReg paper [1]. Then in Chapter 3 we present our object tracking method and describe the experiments, as well as the corresponding setup, with which we evaluate our object tracking algorithm concerning tracking accuracy and real-time capability. Hence, in Chapter 4 we present the experimental results and discuss their impact and meaning. Finally, in Chapter 5 we summarize our results and give an outlook on promising future work.

2 Foundations

In this chapter, we present related work and foundations on which we develop our object tracking algorithm. First, we give an overview of general approaches to object tracking, followed by a deeper insight into PSR methods. Finally, we describe the method presented in [1] on which we establish our object tracking algorithm and provide the mathematical background of it.

2.1 Object Tracking

The task of tracking objects in partially observable environments is a widely spread topic. In this domain, Wu et al. [9] state core challenges like occlusion, scale variation, fast motion, and rotations, which complicate object tracking as information about the environmental state is lost. Common methods that tackle these challenges are based on the Kalman Filter [10, 11, 12], the Unscented Kalman Filter [12, 11, 13], or the Particle Filter. These approaches try to predict the pose of an object, considering the poses of an object in the past. Many object tracking algorithms require depth information to track an object reliably, which is often captured via RGB-D cameras.

Approaches like the Se(3)-TrackNet [4] or PoseRBPF [6] propose neural networks to predict the pose of an object in a scene. These approaches are able to handle highly occluded scenes at real-time frame rates and have produced promising results. Furthermore, model-based approaches exist [7, 12, 14], which register a point cloud of the to be tracked object with a point cloud of the scene provided by a ranged camera and estimate the pose of the object using the register result. State-of-the-art implementations of these PSR methods are provided in the FilterReg [1] and Coherent Point Drift (CPD) [15] papers. In the next section we provide a deeper insight into PSR.

2.2 Point Set Registration

As described in [16], the task of PSR is to find the transformation between two point clouds through matching. A fundamental approach of PSR is the ICP algorithm. This algorithm is an approach to find parameters that describe the transformation between two point sets X and Y. ICP can be separated into two steps:

1. Find corresponding points between both point sets.
2. Compute the optimal transformation with respect to the found correspondences.

In step one, there exist many approaches to find correspondences between both point clouds. The classic ICP algorithm stated in [17] first subsamples both point sets and weights each point correspondence e.g. by the point-to-point or point-to-plane distance. Several methods for sub-sampling the point clouds and extensions like outlier rejection which can improve correspondence results are available. Finding truly correct point-pair correspondences, however, is often not easy, since in partially observable environments noise influences the correspondence decision, leading to poor results. Probabilistic approaches in the correspondence finding try to address this challenge by computing probabilities that express how probable it is for each point in the first point cloud to match the other points in the second point cloud. Using these probabilities, correspondences for each possible point pair are computed. We describe this approach in more detail in Section 2.3.
In the second step, the optimal transformation to align both point sets given the corresponding points from step one is computed. The optimal transformation between both point sets minimizes the squared error

$$E(R,T) = \sum_{(i,j)\in C} \|x_i - Ry_j - T\|^2 \tag{2.1}$$

where x_i describes a point of the first and y_j of the second point cloud. We denote the first point cloud as X and the second as Y. Here $E(R,T)$ is the squared distance error between X and Y, where R and T describe a possible shift of point cloud Y. Here C describes the found point correspondences from the first step through the introduction of index pairs. In the ongoing work, we assume that the pairs of indices have been aligned with each other and thus one index fully describes the correspondence. To find the optimal transformation, equation 2.1 must be minimized, which leads to the minimization problem:

$$(R,t) = argmin \sum_{i=1}^{n} w_i \|(Ry_i + t) - x_i\|^2 \tag{2.2}$$

It is important to mention that here an additional weight w_i is added. This extension enables additional weighting of point correspondences, whereby good point correspondences can be weighted more and bad correspondences less. Gradient-based methods enable us to solve this squared error minimization problem, but these approaches are iterative processes that do not guarantee convergence. An alternative is singular value decomposition as stated in [18]. This approach provides a closed-form solution to the optimization problem. In Section 2.5 we discuss this approach in detail.

2.3 Iterative Closest Point With Expectation Maximization

The classic ICP implementation uses features like the point-to-point or the point-to-plane distance to find one point correspondence for each considered point. Many real-world applications are partial observable environments where inaccuracies and noise play a fundamental role, which is why the point-to-point or the point-to-plane error metric could lead to incorrect point correspondences. These potential misleading assignments will influence the result of the ICP algorithm. A probabilistic view of the correspondence finding problem tries to handle noise and inaccuracies. Therefore for each possible point correspondence, based on given features, e.g. positional or color features, it is computed with which probability a point of point cloud X belongs to a point of point cloud Y. This different approach can help to improve the result since in next steps also locally unlikely points potentially due to noise or inaccuracies are considered, which can lead to globally better results. Against this background, we can state the probabilistic approach as a generalization of the ICP algorithm. [15, 1, 19] utilize the probabilistic view of ICP where each of them induce an own probabilistic model.

One similarity between these approaches is that each of them uses a Gaussian Mixture Model (GMM) to compute the probability of one point corresponding to another. [19] showed that finding the alignment between two point sets using such soft probability assignment is equivalent to Expectation Maximization (EM) for GMMs. EM is an approach that attempts to find the most likely parameters that best describe the given observation. The EM algorithm is used when the given data is incomplete, latent data exist or noise influences the data. The EM algorithm [20, 21]is an iterative concept, which tries to maximizes the Maximum Likelihood estimate. This approach consists of two steps, the expectation or E-step and the maximization or M-step. In the E-Step missing data is calculated, using the current model parameters and the given observation. The M-step maximizes the likelihood function using the data found in the E-Step.

We present a deeper insight to the papers CPD [15] and FilterReg [1], since both methods utilize the EM concept in the context of PSR. For further explanation, we state point cloud X as model and point cloud Y as observation. The distinction between model and observation is very important since we assume that we have one model point cloud that describes a certain structure that we want to align in an observation point cloud. Hereby, the observation can be disturbed by noise or inaccuracies.
In the E-Step the probabilities for each possible point correspondence are computed. Both methods have their underlying model, but both calculations are based on Gaussian mixture models, using a Probability Density Function (PDF) to express the correspondence probability. Next, the M-Step tries to maximize the maximum likelihood of the underlying probability model by finding the optimal transformation between model X and observation Y.

The main difference between CPD and FilterReg is the mentioned underlying probability model. CPD states that model X implies the GMM. Here each model point is taken as a GMM centroid with a given variance. The points of observation Y are treated as data points, on which the GMM centroids should be fitted. On a high level, this method tries to explain the observation given the model. Contrary FilterReg utilizes model X as data points and the observation points as GMM centroids. Here the model data points are matched to the observation centroids. Thereby the FilterReg approach describes the model in the observation. The advantage of this approach is that we only consider parts in the observation which fit the model. The following section provides a deeper insight into the FilterReg method.

2.4 FilterReg

We propose an object tracking algorithm based on the PSR algorithm stated by FilterReg [1]. In this section, we describe the mathematical background in more detail. As stated in Section 2.3 FilterReg [1] uses Y and X as input, including observation points $y_1, ..., y_N$ and model points $x_1, ..., x_M$. Furthermore, as described in Section 2.3 FilterReg [1] views the problem of PSR from a probabilistic perspective. Therefore FilterReg [1] proposes a GMM where each point of the observation Y induces a Gaussian distribution. The mean of each distribution is the 3D position of each point in the environment. To fully describe the GMM a diagonal matrix is used as covariance since it is assumed that all dimensions are independent and do not correlate with each other. Each point of model X is treated as a data point whose position is controlled by the motion parameters θ.

Here these motion parameters θ induce a rigid transformation $T(\theta) \in SE(3)$. The task of point set registration is to find an optimal transformation $T_{opt}(\theta)$ that aligns observation Y and model X. To solve this optimization problem FilterReg [1] suggests maximizing the log-likelihood function,

$$L = \sum_{i=1}^{M} \log \left(\sum_{j=1}^{N+1} P\left(y_j\right) p\left(x_i(\theta) \mid y_j\right) \right) \tag{2.3}$$

which consists of a prior probability $P(y_j)$ that describes how likely any model point corresponds to the observation point y_j and a probability $p\left(x_i(\theta) \mid y_j\right)$ which describes how likely model point x_i correspond to the observation point y_j. The likelihood $p\left(x_i(\theta) \mid y_j\right)$ is calculated by using the previously introduced Gaussian distribution for each observation point y_i, which is here,

$$p\left(x_i(\theta) \mid y_j\right) = \mathcal{N}\left(x_i(\theta); y_j, \Sigma_{xyz}\right) \tag{2.4}$$

whereby Σ_{xyz} is a given covariance matrix. To maximize the log-likelihood 2.3 the EM procedure is used. As mentioned in Section 2.3 the EM procedure consists of two steps. Therefore in the E-Step probabilities for each possible correspondence are computed using the Gaussian distribution stated in Equation 2.4,

E-Step: Compute for each x_i in model X

$$\begin{aligned} M_{x_i}^0 &= \sum_{y_k} \mathcal{N}\left(x_i\left(\theta^{\text{old}}\right); y_k, \Sigma_{xyz}\right) \\ M_{x_i}^1 &= \sum_{y_k} \mathcal{N}\left(x_i\left(\theta^{\text{old}}\right); y_k, \Sigma_{xyz}\right) y_k \end{aligned} \tag{2.5}$$

where $M_{x_i}^0$ is computed for each model point x_i and consists of the summarized results of the PDF $\sum_{y_k} \mathcal{N}\left(x_i\left(\theta^{\text{old}}\right); y_k, \Sigma_{xyz}\right)$. $M_{x_i}^1$ is a vector with the same dimension as the point vectors x_i and y_k. Here $M_{x_i}^1$ describes the sum of each observation point multiplied by the result of the PDF. The next step takes the computed soft correspondences $M_{x_i}^0$ and $M_{x_i}^1$ and tries to minimize the function:

M-Step: Minimize the function

$$\sum_{x_i} \frac{M_{x_i}^0}{M_{x_i}^0 + c} \left(x_i(\theta) - \frac{M_{x_i}^1}{M_{x_i}^0}\right)^T \Sigma_{xyz}^{-1} \left(x_i(\theta) - \frac{M_{x_i}^1}{M_{x_i}^0}\right) \tag{2.6}$$

In equation 2.6 $M_{x_i}^0/(M_{x_i}^0+c)$ is a weighting factor for each model point to detect potential outliers. Here c describes a constant,

$$c = \frac{\omega}{1-\omega}\frac{N}{M} \tag{2.7}$$

where in Equation 2.7 $0 \leqslant \omega \leqslant 1$ is a parameter called outlier ratio, which is adjustable depending on the tracking scenario. This parameter is used to set the weighting factor of the $M_{x_i}^0/(M_{x_i}^0 + c)$ depending on the application. The larger ω is selected, the stronger the weighting factor influences the result of the tracker and thus more points are marked as outliers. In addition, M and N describe the number of points in the model and the observation. Here Σ_{xyz}^{-1} represents the inverse of the given diagonal covariance matrix. Furthermore, both terms $x_i(\theta) - M_{x_i}^1/M_{x_i}^0$ in equation 2.6 form the squared error of each model point to value $M_{x_i}^1/M_{x_i}^0$, which individual components are calculated in the E-Step. Minimizing the squared error in equation 2.6 leads to the optimal transformation $T_{opt}(\theta)$, which maximizes the log-likelihood in equation 2.3. FilterReg [1] proposes a Twist parameterization method that attempts to find $T_{opt}(\theta)$ with gradient based methods. Furthermore, a permutohedral lattice filter is used in E-Step to reduce computation time. Since we do not consider both of these specific implementations, we refer to FilterReg [1] for further insight. In the following Section we describe our implementation for solving the minimization problem in Equation 2.6, which is based on the closed from solution provided by Singular Value Decomposition (SVD).

2.5 Singular Value Decomposition

As mentioned in Section 2.2 [18] proposes SVD to solve the minimization problem

$$(R,t) = argmin \sum_{i=1}^{n} w_i \|(Ry_i + t) - x_i\|^2 \tag{2.8}$$

in closed form. This minimization problem is equivalent to the minimization of Equation 2.6 in the M-Step of FilterReg, which leads to the equation

$$(R,t) = argmin \sum_{i=1}^{n} \frac{M_{x_i}^0}{M_{x_i}^0 + c} \|x_i(\theta) - \frac{M_{x_i}^1}{M_{x_i}^0}\|^2 \tag{2.9}$$

where w_i is replaced by the weighting factor $M_{x_i}^0/(M_{x_i}^0 + c)$. $(Ry_i + t)$ describes the rigid transformation $T_{opt}(\theta)$ that is induced by $x_i(\theta)$ in Equation 2.6. Finally x_i is similar to

$M^1_{x_i}/M^0_{x_i}$. The output of both equations 2.8 and 2.9 is the optimal transformation to align two points given the probabilistic point correspondences. In the next part of the section, we describe SVD using the general Equation 2.8. This procedure can be applied analogously to Equation 2.9.
To find the optimal transformation, first the weighted center of mass for each point set is computed, which is then used to center each point set on the origin.

$$\begin{aligned} \bar{x} &= \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i} \\ \bar{y} &= \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i} \\ p_i &= x_i - \bar{x} \\ q_i &= y_i - \bar{y} \end{aligned} \tag{2.10}$$

Herein x_i , y_i are points of both point sets and w_i is the weighting factor from Equation 2.8. $\bar{x}$,$\bar{y}$ describe the weighted center of mass of both point sets, from which then points p_i and q_i follow. These points represent the weighted centered points of each point in point set X and Y. Next the $d \times d$ covariance matrix

$$S = PWQ^T \tag{2.11}$$

is computed. In equation 2.11 P and Q represent a $d\ x\ n$ matrix having in each column point p_i and point q_i. Matrix W is a $n\ x\ n$ diagonal matrix containing all weights for each point correspondence $w_1, ..., w_n$. From matrix S the singular value decomposition $S = U\Sigma V^T$ is computed. U and V are then used to compute the optimal rotation R,

$$R = V \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & det(VU^T) \end{bmatrix} U^T \tag{2.12}$$

where R consists of the multiplication of V, U and a diagonal matrix with ones as entries. The last entry of this diagonal matrix is the determinant of VU^T, since each rotation matrix must have a determinant of one, which is achieved by using the value $det(VU^T)$, as the last value of the diagonal matrix. One is able to compute the translation t using the resulting rotation R from Equation 2.12, which is given by:

$$t = \bar{x} - R\bar{y} \tag{2.13}$$

Here $\bar{x}$, $\bar{y}$ are the weighted center of mass, which then result using rotation R in the translation t. Each step is described in [18]. Moreover we reference to [18] for the exact derivation, as well as for a deeper insight. The advantage of this solution is that it is a closed-form solution that does not require iterative calculations. After having depicted the foundations we base our work on, we present in the next section how we implement our object tracking algorithm using these foundations.

3 Object Tracking Implementation & Experiments

Our goal is to implement an object tracking algorithm that reliably tracks the pose of an object in a partially observable environment. We focus especially on robotic assembly setups, where objects are manipulated by robots. In this section, we first introduce our environmental setup where our object tracking algorithm is used. Then we present the implementation of our object tracking algorithm and describe the experiments we propose to test the tracking algorithm on several properties.

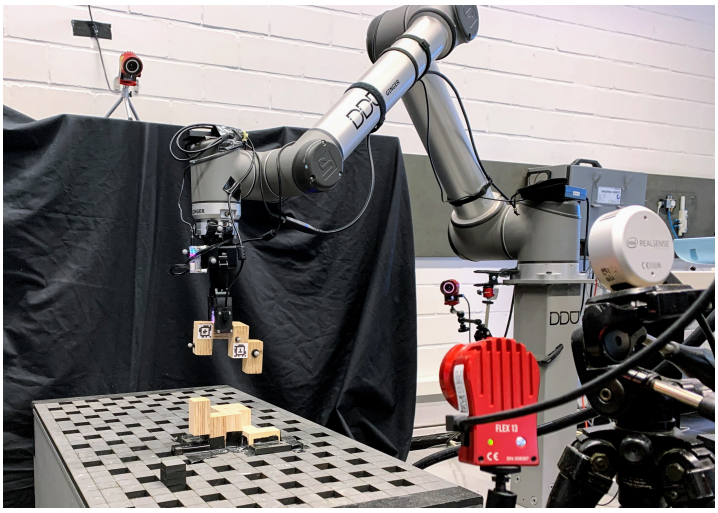


Figure 3.1: Robotic assembly setup at DDU, where the SL-BLock is grasped by the UR10 robot arm [22].The pose of the SL-Block is recorded with Optitrack [2] sensors. The scene is recorded by the camera Intel Realsense L515 LIDAR [23]. Our utilized dataset consists of trajectories where the robot arm moves the SL-Block in the scene.

3.1 Environmental Setup

As stated before our focus is to introduce a robust algorithm for tracking an object in the domain of robotic assembly. Figure 3.1 shows the robotic assembly setup at the Digital Design Unit at TU Darmstadt (DDU). This setup focuses on robotic assembly by using a robot arm with an attached gripper, which consists of a UR10 robot arm [22] and a RH-P12-RN gripper [24], to interact especially with SL-Blocks [25]. These blocks have a unique shape that provides self-interlocking properties, which are beneficial for robotic assembly. In our case, we track this SL-BLock in the scene of Figure 3.1. To receive information about the environment, like RGB-values and depth information, we use an Intel Realsense L515 LIDAR camera [23]. To evaluate the tracking algorithm we need ground truth information about the tracked SL-Block, which is provided by the motion capturing system Optitrack [2]. We present detailed information about our used experiment in Section 3.3.1.

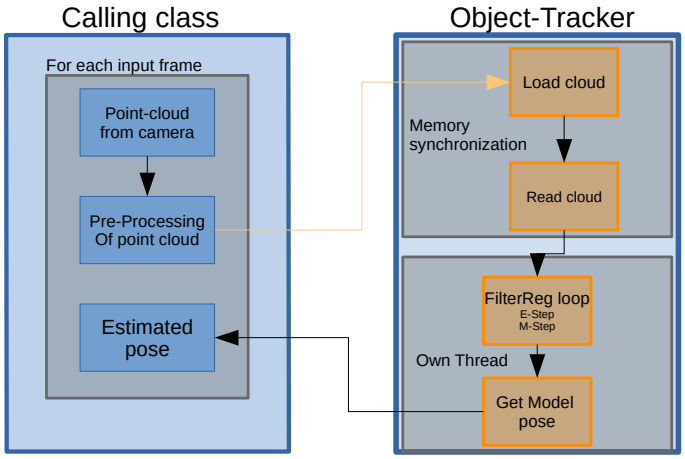


Figure 3.2: Structure of the implemented Object Tracker. The tracker is implemented in one class. The class has 4 main functions. The Object tracker uses an extra thread for the Point Set Algorithm and has therefore simultaneous data access protection.

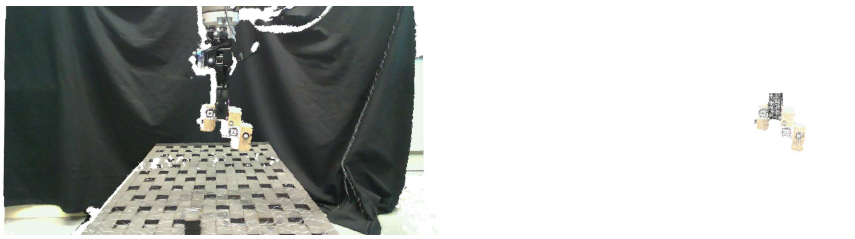
3.2 Implementation

We implement an object tracking algorithm based on the PSR method presented by the FilterReg paper [1]. To receive the pose of an object in an environment our implementation requires a point cloud from the environment and a point cloud of the object which should be tracked. For each new input, both point clouds are matched and the computed rigid transformation between both clouds is then returned as the pose of the object in the scene. We state the input point cloud of the RGB-D camera as observation point cloud and the to be tracked object point cloud as model point cloud, whereby the nomination here is similar as in Section 2.4.

Figure 3.2 shows the structure of our object tracking algorithm. As the computation speed of our PSR method relies on the number of points in the point cloud we filter the observation point cloud. Our filtering methods are described in Section 3.2.1. As visualized in Figure 3.2 the pre-processing of the observation point cloud is done in the calling class. Next, we use the PSR approach of FilterReg [1] to estimate the rigid transformation which matches the model point cloud on the observation point cloud. To guarantee constant transformation results we run this algorithm in an endless and thread separated loop. In Figure 3.2 this loop is called FilterReg loop. The load and read cloud functions in Figure 3.2 contain memory synchronization methods, since as mentioned before we run the actual tracking loop in a separated thread. We have chosen this structure of our code to provide it as a black-box implementation. As a result, the algorithm can be easily applied to different tracking scenarios in many tracking domains. We present each step of the algorithm in the next sections.

3.2.1 Pre-Processing

The complexity of the presented E-step in Section 2.5 depends on the number of points of the observation and the model, which is why pre-processing of these point clouds is important. We propose two steps to minimize the number of points in both point clouds. In general, ICP algorithms rely on a good initial pose of the object to be tracked. Therefore, we propose to use a region-based filter for the given observation. In our implementation, we use the given initial pose of the block, which is described by a reference point near the center of mass of the block, and filter out points that are further away than $10\ \mathrm{cm}$ in each spatial dimension. In our setup, we assume this region-based filter is valid, since the dimension of the SL-Block is $12\mathrm{cm} \times 6\mathrm{cm} \times 9\mathrm{cm}$. As a reasonable filter range depends on



(a) Before filtering (b) After filtering

Figure 3.3: Observation point cloud before filter usage and point cloud after usage of region based filter and voxelization filter

the given observation and tracking scenario it can be adapted. To further decrease the number of points in the point cloud, we voxelize the observation point cloud. Therefore we use an already implemented function provided by the Point Cloud Library [26]. For voxelization, a three dimensional grid is put over the point cloud and all points lying in one cell are used to compute one new point, which coordinates and color values are the mean of all points in this cell. In the further part of this work, we name the size of one grid cell in each spatial dimension voxel grid size.

Figure 3.3 visualizes our proposed filtering methods, with which we can reduce the number of points in the point cloud from nearly 200,000 to 2,000 points, which is a reduction factor of 100. Since voxelization reduces the number of points by summarizing points to one new point shape information of the SL-Block is lost. Therefore we test our object tracking algorithm on several voxel grid sizes. In Figure 3.3 we use a cell size of three millimeters in each spatial dimension. Furthermore, we convert the point cloud from the data structure provided by the Point Cloud Library [26] into our structure, which is based on an Eigen Matrix [27]. Here each column represents one point and the rows consist of XYZ coordinates and potential features in addition. For example, we propose additional color features, which we describe in Section 3.2.2.

3.2.2 FilterReg Implementation

In this section, we describe one iteration of the FilterReg loop, in which the observation point cloud and the model point cloud are matched. Our implementation is based on the foundations stated in Sections 2.4 and 2.5. As described in these sections FilterReg [1] views the PSR problem as an EM problem. Hereby, the approach consists of the E-Step and the M-Step.

E-Step

For our implementation, we use the Equations 2.5 of the E-Step from the FilterReg [1] paper and extend these equations to a general form which is able to use as input not only a vector containing XYZ-coordinates, but also a general feature vector.

$$\begin{aligned} M_{x_i}^0 &= \sum_{y_k} \mathcal{N}(f(x_i(\theta^{old})); f_{y_k}, \Sigma_f) \\ M_{x_i}^1 &= \sum_{y_k} \mathcal{N}(f(x_i(\theta^{old})); f_{y_k}, \Sigma_f) f_{y_k} \end{aligned} \tag{3.1}$$

Hereby Equation 3.1 is equivalent to Equation 2.5, however with the extension to general features. That explains the vectors f_{y_k} and f_{x_i} because they not only describe the XYZ position but in our case extend it with color features, from which we expect more robustness. We describe these features in Section 3.2.2. In our stated point cloud data structure, which we described in Section 3.2.1, these vectors correspond to one column of each point cloud matrix.
As each operation in the E-step is very time consuming, we implement the computation of the probability $\mathcal{N}(f(x_i(\theta^{old})); f_{y_k}, \Sigma_f)$ manually. $M_{x_i}^1$ is a $3 \times n$ matrix and $M_{x_i}^0$ is a $n \times 1$ vector. We implement two functions with different input feature vectors. The first function takes a 4×1 feature vector and a 4×4 co-variance matrix. These inputs represent XYZ-feature with an additional feature value for potential outlier recognition. The second function takes a 6×1 feature vector and a 6×6 covariance matrix, which consists of the first four dimensions from the first function but extends these feature values by two color features. These feature values are based on the HSV-color space and are described in the next section.

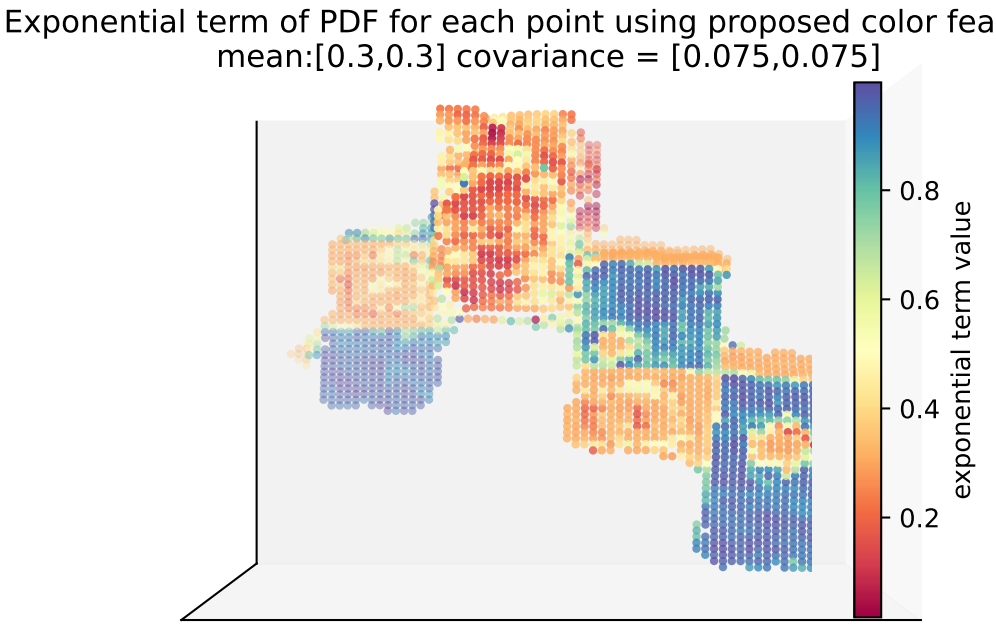


Figure 3.4: Visualisation of the exponential term of the PDF for our proposed color features. A value of 1 describes the mean and a value a of 0 an infinitely distance to mean.

Features

As stated in Section 3.2.2 we extend the 4×1 feature vector. The observation point cloud provides RGB values for each point, which is visualized in Figure 3.3. In the domain of robotic assembly illumination changes is a challenge for object tracking. Therefore we propose to transform the RGB values into HSV-color space, where the v-component describes how color appears if pure white light shines on it. This representation of color is beneficial for us since we can decouple this value from our features from which we expect a more robust tracking regardless of lighting conditions. To receive illumination independent color features we ignore the v-value color component. The remaining h- and s-color values describe a circular plane, where each color is described in polar coordinates. Here the h-value describes the angle and the s-value describes the distance from origin to the desired color. As this representation of color is nonlinear we transform this polar

coordinate representation into a Cartesian representation

$$\begin{aligned} color_1 &= S \cdot \cos(H) \\ color_2 &= S \cdot \sin(H) \end{aligned} \tag{3.2}$$

and use $color_1$ and $color_2$ as entry five and six in our 6×1 feature vector.

A visualisation of our proposed color features can be seen in Figure 4.2. In this figure, the filtered observation point cloud from Figure 3.3b is shown, where the color of each point describes the exponential term of the PDF computed in the E-Step, stated in Equation 3.1. Here we only consider our proposed color features in the PDF. As mean we choose for both $color_1$ and $color_2$ features the value 0.3. Furthermore, we assume for both color features a variance of 0.075, which leads to a 2×2 diagonal matrix with 0.075 as the entry for both diagonal elements. However, we received the mean values by evaluating the tracking algorithm on various color feature means. Based on Figure 4.2 we assume our found color features as valid since the wooden parts of the block are very likely to our proposed means. Nevertheless, at the upper left and the lower right, there are locations where the exponential term is between 0.2, and 0.4, which derives from Apriltag [3] and Optitrack [2] markers on the SL-Block. We assume that these locations disturb our object tracking algorithm, and for a more uniform SL-Block even better results can be achieved. Furthermore, the points which describe the gripper in the scene are recognized and classified as not likely to the means. We derived the covariance matrix entries by computing the variance of the color features in the filtered observation matrix.

3.2.3 Adapting Variance

Since in our method each point in the observation point cloud induces a Gaussian distribution a covariance matrix is needed. This matrix is a hyperparameter, which has to be tuned manually. To reach better results FilterReg [1] proposes an adapting covariance matrix from which we state the computation

$$\begin{aligned} \sigma^2 = \left(\Sigma_i \frac{M_{x_i}^0 f(x_i)^T f(x_i) - 2 f(x_i)^T M_{x_i}^1 + M_{x_i}^2}{M_{x_i}^0 + c} \right) / \left(\Sigma_i \frac{M_{x_i}^0}{M_{x_i}^0 + c} \right) \\ M_{x_i}^2 = \sum_{f_{y_k}} \mathcal{N}\left(x_i^{\text{old}} \mid f_{y_k}, \Sigma_{xyz} \right) f_{y_k}^T f_{y_k} \end{aligned} \tag{3.3}$$

for the adapting variance. Hereby we receive $M_{x_i}^0$ and $M_{x_i}^1$ from the E-Step computation, depicted in Equation 3.1. Next, f_{y_k} and $f(x_i)$ describe again the feature vector for each

point in every point cloud. We further add the computation of $M_{x_i}^2$ to the E-Step. The computation of a new covariance matrix in each step, should lead to a smoother likelihood function, stated in Equation 2.3. Therefore less local optima should exist and the tracking results should be improved.

3.2.4 M-Step

In the E-Step we compute the correspondences for each model point to the observation point. In the M-Step, our objective is to find the optimal transformation between the current model point cloud pose and the computed corresponding points from the E-Step. This procedure maximizes the log-likelihood function, stated in Equation 2.3. Maximizing the log-likelihood function is equivalent to minimizing the function

$$\sum_{x_i} \frac{M_{x_i}^0}{M_{x_i}^0 + c} \left(x_i(\theta) - \frac{M_{x_i}^1}{M_{x_i}^0} \right)^T \Sigma_f^{-1} \left(x_i(\theta) - \frac{M_{x_i}^1}{M_{x_i}^0} \right) \tag{3.4}$$

which we presented in Section 2.4. We solve this minimization problem equivalent to the procedure stated in Section 2.5, which leads to the equation

$$(R, t) = argmin \sum_{i=1}^{n} \frac{M_{x_i}^0}{M_{x_i}^0 + c} \| (Rx_i + t) - \frac{M_{x_i}^1}{M_{x_i}^0} \|^2 \tag{3.5}$$

where the optimal motion parameters $\theta_{optimal}$ are described by a rotation R and a translation t. As described in section 2.5 we propose SVD to solve this equation.

3.2.5 Memory Synchronization

As stated in Section 3.2 we implement the FilterReg loop in a separated thread, which allows a constant output of estimated poses that is independent of the specific arrival time of new observation point clouds, since until new observations are read the computation is done on the old observation. This implementation is beneficial as in real object tracking applications sensors do not provide new point clouds at a constant rate. A major challenge of this implementation is to synchronize the shared memory on which the observation point cloud is stored.

Algorithm 1: Algorithm to load a new observation into the tracking algorithm

```
1: function LOAD_CLOUD(point_cloud)
2:     lock shared memory
3:     next_observation ← point_cloud
4:     next_written ← true
5:     unlock shared memory
6:     return
```

Algorithm 2: Algorithm to read the newest observation into the tracking loop

```
1: function READ_CLOUD( )
2:     if next_written is true then
           lock shared memory;
           tmp ← next_observation;
           next_observation ← current_observation;
           current_observation ← tmp;
           next_written ← false;
           unlock shared memory
           return current_observation
```

To prevent simultaneous access on the observation point cloud by both threads we propose a memory synchronization method stated by Algorithm 1 and 2. These algorithms show how we implement the $load\_cloud()$ and $read\_cloud()$ functions from Figure 3.2. The $load\_cloud$ function gets called by an outer thread and its purpose is to load new observation point clouds into the object tracking class. The $read\_cloud()$ function is called at the beginning of each iteration of the FilterReg loop. Here the newest point cloud that is fully written into memory is loaded, which is then used for estimating the model pose

in the observation.

To synchronize both functions we propose the usage of pointers where each of them points to one point cloud. The $current\_observation$ pointer points to the observation point cloud which is currently used by the FilterReg loop. The $next\_observation$ pointer is mainly used in the $load\_cloud()$ function, where for each new observation cloud $next\_observation$ is set to point to the new observation. Algorithm 1 shows the code of the $load\_cloud()$ function, which consists of three main steps. First, we lock the memory where the point cloud is stored, then we set the pointer $next\_observation$ to the new observation which is given by the function call. After this step, we set the Boolean $next\_written$ to true, which indicates that the new observation point cloud is referenced by pointer $next\_observation$ and unlock the shared memory.

As laid out before the $read\_cloud()$ function is called at the beginning of each iteration of the FilterReg loop. Here Algorithm 2 shows the structure of this function. First, the function checks if a new point cloud is completely loaded into the tracking algorithm by checking the Boolean $next\_written$. If $next\_written$ is true, then the shared memory is locked, which prevents simultaneous access. Next, the addresses of both pointers are switched and $next\_written$ is set to false. Finally the $current\_observation$ pointer is returned. Using this implementation we ensure that only if a new observation point cloud is fully loaded, the point cloud is used in the FilterReg loop. If there is no new observation point cloud the current point cloud is returned by the $read\_cloud()$ function.

With describing the memory synchronization we stated the structure and core implementations of our object tracking algorithm. In the next section, we give an overview of the setup we use to evaluate our object tracking algorithm and which experiments we perform.

3.3 Experimental Setup

To guarantee safe interactions between robots and objects the error of the estimated pose of the object can only be in a narrow range depending on the application. Furthermore, to deal with applications in real world, the computation must be real-time capable. Therefore we test the proposed algorithm for pose accuracy and computation speeds. To evaluate our algorithm on these properties we test our object tracking algorithm in the environment

presented in section 3.1 Therefore we utilize a dataset, which was created in a previous Robot Learning Integrated project [8]. We present this dataset in the following section.

3.3.1 Dataset

The stated dataset [8] consists of trajectories where a robot arm moves an SL-Block. Hereby, 12 different trajectories, which differ in translation and rotation are provided. Furthermore, some trajectories include environmental changes, like background variation, illumination changes, and occlusion. The dataset provides RGB-/depth-images, pcd-files, robot joint states, and the ground truth pose of the SL-Block, where the ground truth is recorded by the motion capturing system Optitrack [2]. Therefore, the ground truth is represented as a reference point on the SL-BLock, which consists of a translational and a rotational part.

The core challenge in this dataset [8] is that the Optitrack [2] system and the lidar camera RealSense L515 [23] interfere with each other, which leads to bad depth values and prohibits to run both sensing modalities in parallel. To solve this problem each trajectory has been recorded twice, where firstly only the lidar camera records the trajectory and secondly Optitrack [2] records the ground truth pose of the SL-BLock. Here the largest error component of the provided ground truth consists of the error of replaying the same trajectory twice. We call the error resulting from this double recording *reproduction error*. In [8] it is stated that this reproduction error for most trajectories is in the magnitude of millimeters. We assume this error to be tolerable but it should be kept in mind for evaluating our proposed algorithm. To give a better understanding of our used data, videos of each trajectory with additional visualisation of the ground truth can be found here. For a deeper evaluation of the dataset we refer to [8].
To test our object tracking algorithm we focus on three trajectories of the dataset,

- Spiral
- Square
- Random-Bridge

as each of them differs in movement and structural visibility of the SL-BLock.

Experiments	Color features	Adaptive variance	Frame-rate [Hz]	Initial pose diff [m]	Voxelization	Outlier ratio ω [m]	Section
Color Feature Comparison	✓/-	✓	-	0	0.003	0	4.1
Variance Comparison	✓	✓/ -	-	0	0.003	0	4.2
Initialization Pose	✓	✓	-	0.05	0.003	0	4.3
Outlier Ratio Comparison	✓	✓	-	0	0.003	$0.05, 0.25$	4.4
Voxelization Comparison	✓	✓	-	0	$[0.001 - 0.005]$	0	4.5
Frame Rate Comparison	✓	✓	$40, 100$	0	0.003	0	4.6

Table 3.1: Overview of which parameters are used for each experiment. The color features column indicates if additional color features are used or not. If adaptive variance is set an adapting covariance matrix is used by the tracking algorithm. Frame-rate is important for testing the multi-thread implementation and specifies in which time intervals new observations are loaded into the tracking algorithm. Initial pose difference indicates if an initialization pose is used which diverges from the ground truth pose at the beginning. Voxelization states which voxel grid size is used to down-sample the observation point cloud. The outlier ratio ω is a parameter which is used to detect potential outliers. Section references the location in the Results chapter where these parameters are used.

3.3.2 Experiments

To evaluate our proposed object tracking algorithm sufficiently we test it in six different experiments, where we evaluate the tracker on accuracy and run-time performance of the tracking loop. To evaluate our tracking approach on accuracy we run two EM iterations of our proposed method for each incoming observation. Hereby, we store the resulting estimated pose of the second run and use the provided time of each observation input as the timestamp. Therefore, we ignore the actual calculation time of the algorithm and only consider the accuracy of our tracker for each observation input. The tracking results are then compared to the ground truth provided by the dataset presented in Section 3.3.1.

Furthermore, we evaluate in another experiment the run-time performance of one EM iteration by computing the mean and standard deviation for all iterations of the tracking algorithm. Since the run-time performance of the computation depends on the used hardware, we state that we used a setup with an Intel(R) Core(TM) i7-6700K CPU and 16GB RAM. These experiments are done with the mentioned trajectories Spiral, Square and Random-Bridge.

Table 3.1 depicts our experiments by which we test our object tracking algorithm and

the parameters used during these experiments. We focus on evaluating our extensions, as proposed in Section 3.2. Therefore, in each experiment we evaluate one parameter, leaving all other parameters constant.

As laid out in Section 3.2.2, we propose to use additional color features from the HSV-space. To test our approach we evaluate for each trajectory the accuracy of the tracking algorithm with XYZ-features and XYZ-features plus color features. Furthermore, we name the algorithm with XYZ-features implementation as without color features and the algorithm with XYZ-feature plus color feature as with color features.

In Section 3.2.3 we state an additional adapting variance, which should lead to better results. We test our object tracking algorithm on this property and run for each trajectory the tracking algorithm with and without adapting variance.

As ICP related point set registration approaches mostly rely heavily on good initial poses of the to be tracked object, we evaluate our tracking algorithm by varying this initialization. Since we assume that an initial pose can be provided that is at most $5\mathrm{cm}$ away from the actual ground truth pose, we test the behavior of our tracker at this distance from the initial pose. We vary the translational part of the initial pose by sampling randomly selected positions around the actual initial pose. Therefore, we consider a sphere with a radius of $5\mathrm{cm}$ (see. Table 3.1) around the initial pose. We then randomly select a position from the surface of this sphere and assume this position as the new translational part of the initial pose. Furthermore, we sample 20 different positions for each radius and evaluate the mean and the standard deviation of the results. To consider rotational variations, we compute a random initial rotation and complement with this new rotation the used initial pose of the object.

As stated in Section 2.4, the M-Step (see Equation 2.6) includes the weighting factor $M^0_{x_i}/(M^0_{x_i} + c)$ that is used to detect potential outliers. Since in the computation of the constant c, which is shown in Equation 2.7, the outlier ratio ω must be provided, we evaluate our algorithm for different values of ω. As stated in Table 3.1, we use for all experiments the outlier ratio of $\omega = 0$ except in Section 4.4. Setting the outlier ratio to $\omega = 0$ causes the weighting factor $M^0_{x_i}/(M^0_{x_i} + c)$ to become 1 and thus no outlier detection is used. However, in Section 4.4 we evaluate the influence of the outlier ratios $\omega = 0.05$ and $\omega = 0.25$ on the tracking result, and if it potentially can increase the accuracy of the tracking algorithm.

We voxelize the observation point cloud to provide real-time results. Because voxelization reduces the number of points in the observation point cloud information about the environmental state is lost. To keep a balance between fast computation speed and accuracy

we analyze our object tracking algorithm with different voxel grid sizes of the voxelization method. Here we consider voxel grid sizes between 1mm and 5mm(see Table 3.1).

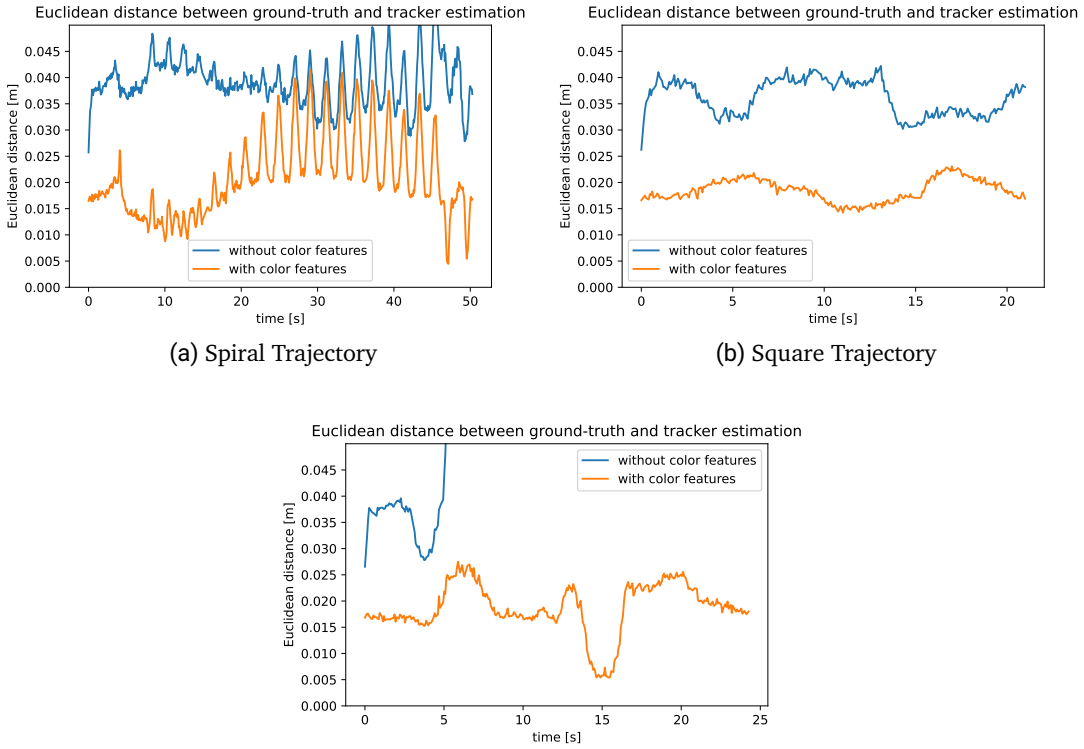
In all previous experiments, we use the implementation of the tracking algorithm where we compute two iterations of the tracking algorithm for each new input point cloud, whereby the results are independent of the computation time of the tracking algorithm. This implementation is beneficial to evaluate the tracking algorithm for accuracy but does not analyze the threaded implementation stated in Section 3.2.5. Since our tracking algorithm has a certain computation time we expect the tracking results should be slightly shifted to the ground truth. To analyze this potential offset we evaluate our proposed threaded implementation. As our object tracking algorithm should be able to deal with real-time input speeds we test our algorithm on new input point cloud frame-rates at 40Hz and 100Hz., which is stated in Table 3.1. Hereby, we simulate the input frame rates of 40Hz and 100Hz by loading new observation point clouds at a frame rate of $40\text{Hz}/100\text{Hz}$ into the object tracking algorithm. Furthermore, it is important to mention that the observation point clouds of the dataset were not recorded at a frame rate of $40\text{Hz}/100\text{Hz}$.

4 Results & Discussion

After presenting each experimental setup in Section 3.3.2, we visualize and discuss the results of each experiment in this section. The structure of this section follows the structure of Table 3.1, which shows all parameters chosen for each experimental setup. At first, we evaluate our in Section 3.2.2 stated color features. Next, we evaluate the influence of the additional adapting variance discussed in Section 3.2.3 during the tracking. Then we evaluate the influence of shifted initial poses followed by analyzing the influence of different voxel grid sizes. Thereupon, we evaluate the importance of the weighting factor to detect potential outliers. Finally, we evaluate the multi-thread implementation of our object tracking algorithm, which is also our proposed implementation of the tracking algorithm.

4.1 Color Features

In Section 3.2.2 we introduced additional color features to increase the robustness of the object tracking algorithm. Firstly, we evaluate the accuracy by comparing the Euclidean distance for each tracking implementation to the ground truth pose and then compare the rotational difference for each Euler angle between ground truth and estimated pose. Furthermore, we take a closer look at how the Euclidean error is composed, for which reason we analyze the course of each XYZ coordinate. Finally, we evaluate the computation time needed for both implementations regarding real-time capability. An overview of the used parameters can be seen in the row Color Feature Comparison of Table 3.1.



(a) Spiral Trajectory

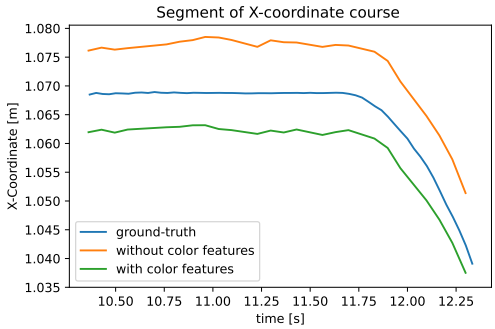
(b) Square Trajectory

(c) Random Bridge Trajectory

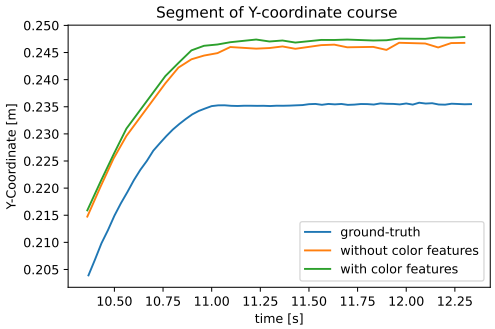
Figure 4.1: Euclidean Distances between the estimated pose of our object tracking algorithm and the given ground truth pose by the dataset for each trajectory

4.1.1 Accuracy

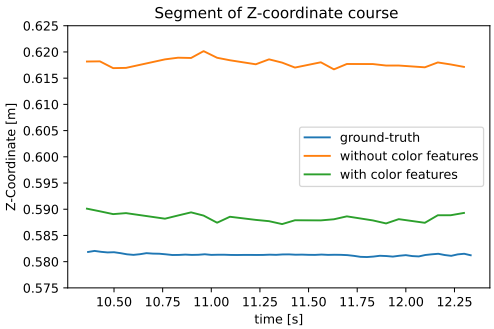
Figure 4.1 shows the Euclidean distance between the ground truth position and the pose estimated by our proposed object tracking algorithm plotted against each time step of the viewed trajectory. In each trajectory, our proposed color feature implementation outperforms the standard XYZ-feature implementation. Figure 4.1b depicts that the estimated pose of the color feature implementation is about 2cm closer to the ground truth pose, which is a reduction by half compared to the standard XYZ-feature implementation. This improvement can nearly be seen for each trajectory. In Figure 4.1a between the timestamps $20\text{s} - 40\text{s}$ the color feature implementation performs as good as the XYZ



(a) X-coordinate course of square trajectory



(b) Y-coordinate course of square trajectory

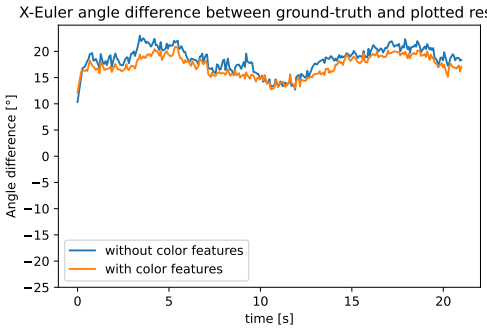


(c) Z-coordinate course of square trajectory

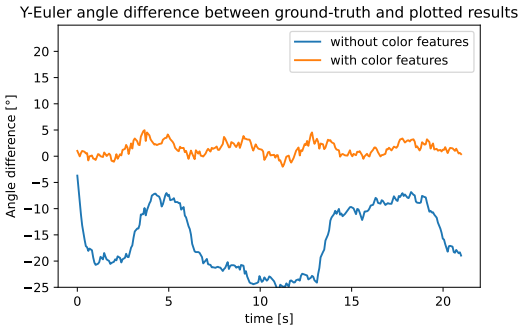
Figure 4.2: Segment of the square trajectory coordinate course of the groundtruth as well as the estimated pose from the color feature and XYZ-feature implementation.

features. However, between $0\text{s} - 20\text{s}$ and $45\text{s} - 50\text{s}$ the color feature implementation outperforms the without color implementation. In Figure 4.1c it is depicted that without the color features the object tracking algorithm is unable to follow the SL-Block and loses it after 5s. In this context, only with our proposed color feature extension, the algorithm can track the object.

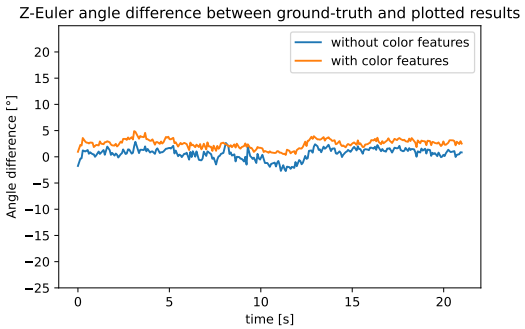
Next, we compare the course of each dimension of the ground truth to the estimated pose by the tracker, to understand how the distance error is composed. In Figure 4.2 the ground truth and estimated course of both tracking implementations are displayed. The Y-coordinate course of both estimations shown in Figure 4.2b is nearly identical,



(a) X-Euler angle of square trajectory



(b) Y-Euler angle of square trajectory



(c) Z-Euler angle of square trajectory

Figure 4.3: Euler angle differences between estimated Euler angles of object tracking algorithm and given ground truth Euler angles by the dataset for each trajectory

where both are around 2cm off from the ground truth Y-coordinate course. This offset is near constant over time and could derive from the fact that the global Y coordinate is in the same direction as the depth values of the depth camera. For this reason, only the front side of the SL-Block is visible and all other geometric properties behind the front of the block are hidden, which could be the reason why the model in this dimension cannot be perfectly fitted to the block in the observation. Comparing the results of the X-coordinate courses shown in Figure 4.2a, we can observe that between 10.5s and 12.0s, both implementations have an offset of about 6mm. In this respect, we observe that the color feature implementation underestimates the ground truth X-coordinate course and the XYZ-feature implementation overestimates it. Nevertheless, after 12s the color feature

implementation aligns better with the ground truth course. The main difference between both implementations can be seen in Figure 4.2c, where the Z-coordinate course is visualized. Here the color feature implementation outperforms the XYZ-feature implementation by over 2cm. This behavior is seen in almost every trajectory indicating that the color features manage to stay closer to the actual block and are less irritated by the gripper hanging over it.

After comparing the translational part of the estimated pose of the object tracking algorithm in the next step we compare the rotational difference between the ground truth pose and the estimated pose. As difference metric, we use the Euler angles, given that the meaning of this metric is intuitive to understand. The results of this comparison are depicted in Figure 4.3. Looking at the X-Euler difference between the ground truth and the estimated results depicted in Figure 4.3a we observe for both implementations an error between $15°$ and $20°$, which let us derive that both implementations perform equally. In Figure 4.3b, the Y-Euler angle difference is stated. Here we can see that the color feature implementation outperforms the XYZ-feature implementation as the maximum error of the color features is about $5°$ and the maximum error of the XYZ-feature implementation is about $25°$. Comparing the Z-Euler angle difference of both implementations (See. Figure 4.3c), no different behavior of both implementations can be observed. and the error of both estimations is about $5°$.
Evaluating these results we conclude that our proposed color features enhance the tracking result since for some trajectories the tracking algorithm is only able to track the SL-Block with additional color features e.g. in Figure 4.1c. Even for trajectories that can be tracked with both implementations, we can observe an improvement in the result using our proposed color features, which is displayed in Figure 4.1b. This improvement is mainly due to the z part of the trajectory, where the color features implementation significantly outperforms the XYZ implementation. We can also observe a strong improvement in the estimation of the Y-Euler angle of the SL-Block. These results suggest that our proposed color features provide a more robust and accurate tracking result, which makes them a useful extension. Moreover, we see additional potential to improve the color features to describe the SL-Block even more accurately. In addition, our proposed color features are perturbed by Apriltag [3] and Optitrack [2] markers on the SL-Block, which indicates that the tracker result for an unmarked SL-Block could enhance the tracking performance. Considering, these results we conclude that the color feature implementation is valid and an important improvement to the standard XYZ features.

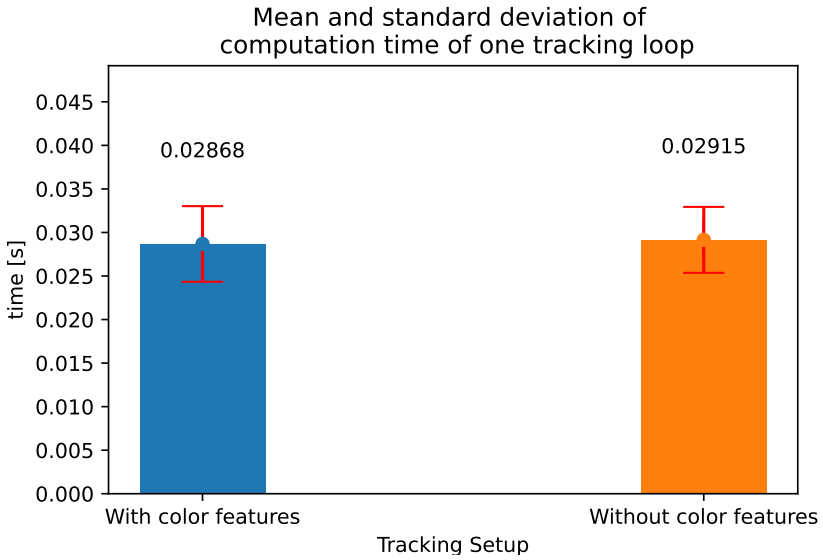


Figure 4.4: Computation time comparison between color-features and only xyz-features following the Square trajectory. The bar visualizes the mean computation time and the error bar visualizes the standard deviation.

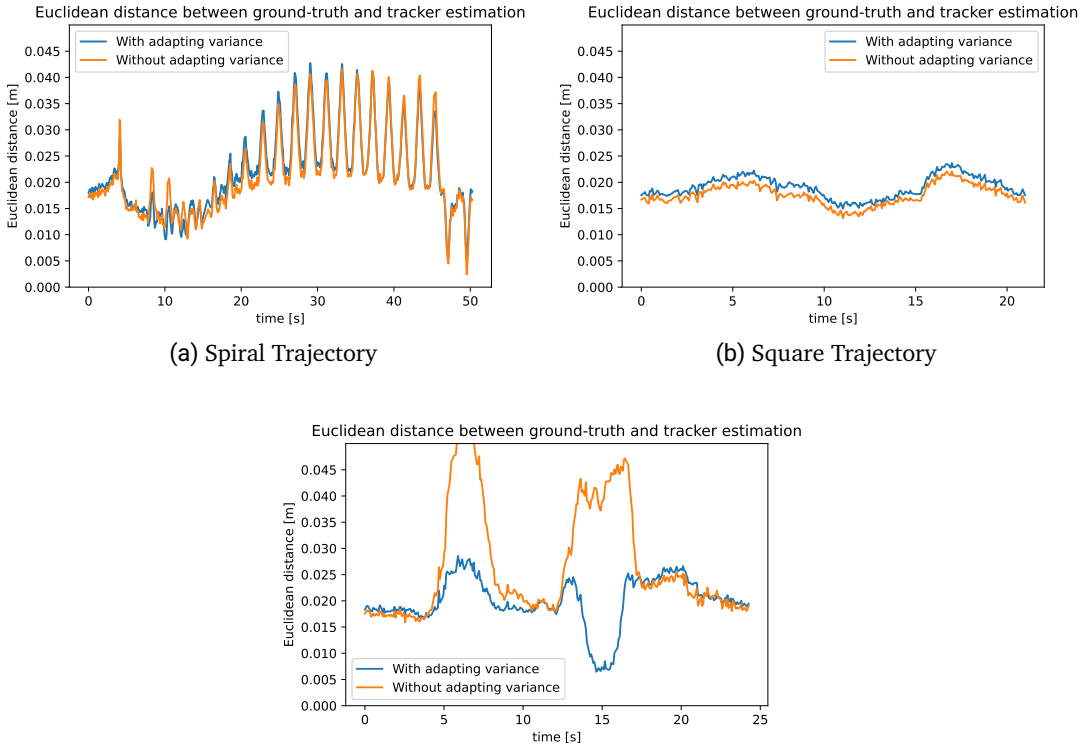
4.1.2 Computation Time

In this section, we compare the mean computation time of one iteration of the tracking algorithm for the color feature and the XYZ-feature implementation. In Figure 4.4 the mean computation times for color features and XYZ-features are shown. We cannot observe a major difference between both implementations since both mean computation times are about $0.029s$. Furthermore, the standard deviation of implementations is also quite similar. We observe this behavior for all trajectories where both implementations can track the SL-Block.

These results let us conclude, that there are no significant computation time variations, from which follows that our color feature implementation leads to a more accurate and robust result, without having to deal with a longer runtime of one iteration.

4.2 Variance Comparison

In Section 3.2.3, we proposed to update the covariance matrix for each iteration of the tracking loop, which should lead to a more smooth log-likelihood function. This smoothing should then lead to a better computation result. We check this assumption by comparing the results of the object tracking algorithm with and without updating the covariance matrix. The experimental setup is stated in the row Variance Comparison in Table 3.1.



(a) Spiral Trajectory

(b) Square Trajectory

(c) Random Bridge Trajectory

Figure 4.5: Comparison of Euclidean distance to groundtruth with updating co-variance matrix and without updating covariance matrix

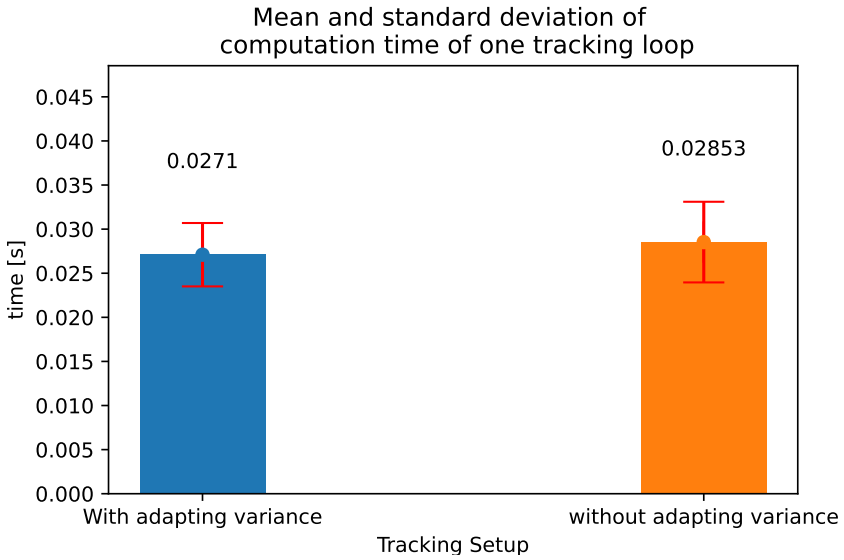


Figure 4.6: Computation time comparison between updating co-variance matrix and constant co-variance matrix

4.2.1 Accuracy

Figure 4.5 shows the resulting Euclidean distance of both implementations. Looking at the trajectories Spiral (Figure 4.5a), and Square (Figure 4.5b) we cannot observe any improvement due to the additional adapting covariance matrix. Both implementations have nearly the same error course, only with some slight deviations. However, we can observe a strong improvement of the result between $5s - 10s$ and $12s - 18s$ at the Random Bridge trajectory, as depicted in Figure 4.5c. Hereby the Euclidean distance error is reduced in the first interval from nearly 5cm to 2.75cm and in the second interval from about 4.5cm to under 1cm.

We conclude that the adapting variance extension can improve the tracking results, depending on the viewed trajectory. We infer that the strong improvement in the Random Bridge trajectory could derive from the unique course of this trajectory. In addition, we conclude that the results are not worsened by the additional adaption of the covariance matrix and that it can lead to a better result for some trajectories. Therefore, we find that this extension is reasonable especially if it does not lead to significantly longer computation times, which we evaluate in the next section.

4.2.2 Computation Time

As stated before it is important to evaluate which influence the adapting variance has on the computation time of one iteration of the tracking loop. Figure 4.6 visualizes the mean computation time and the standard deviation for each tracking implementation. We observe that the adapting variance implementation is about $0.0015s$ faster than without adapting variance.

We reason that this deviation is negligibly small. Thus, we conclude that the additional adapting covariance matrix does not have a significant effect on the computation time, which makes this extension of the tracking algorithm very useful as it can potentially improve the estimation of the pose for some trajectories without increasing the computation time.

4.3 Initial Pose Variation

Good results of standard ICP-related algorithms are bound to a good initial pose of the object in the scene. We evaluate our tracking algorithm on this property by running the tracking setup as displayed in Table 3.1 20 times with randomly selected initial poses as described in Section 3.3.2, and evaluate the mean Euclidean distance and standard deviation of it. Figure 4.7 depicts the mean and standard deviation of the Euclidean error,

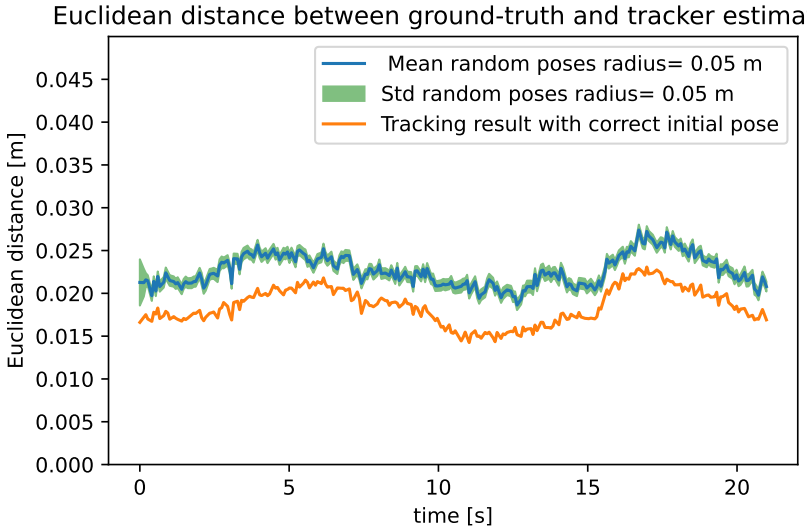
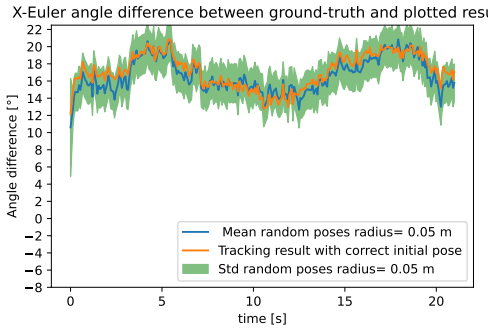
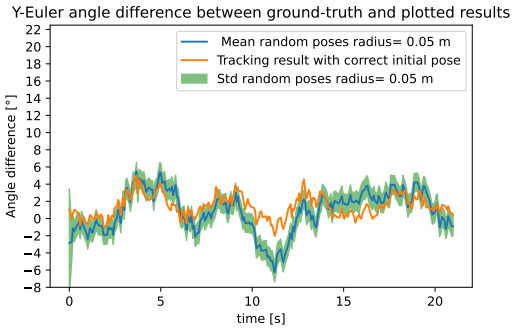


Figure 4.7: Mean Euclidean distance of 20 randomly selected poses to the ground truth of the Square trajectory. The initial translational shift is sampled from a sphere around the initial point with 5cm radius. Furthermore, a random initial rotation of the block was chosen

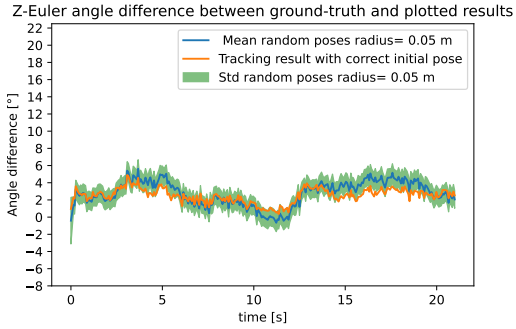
which derives from the randomly selected initial poses. In Figure 4.7 the Euclidean error for a correct initial pose is visualized, which we use as a baseline to evaluate the result of the randomly selected initial poses. We observe that the mean Euclidean distance error is about 5mm higher compared to the baseline. We also state that the standard deviation of the random initial poses is quite small. Furthermore, it is important to point out that in this plot the initial error of 5cm is not visualized, since in this plot we only visualize the tracking result after the first tracking loop is executed.
In the next step, we also evaluate the influence on the rotational side of the computed



(a) X-Euler angle



(b) Y-Euler angle



(c) Z-Euler angle

Figure 4.8: Euler angle differences between mean of 20 randomly selected initial poses and the ground truth. The base-line are the Euler angle differences between the tracking result with correct initial pose and ground truth

poses. Therefore, we view the Euler angle difference between the ground truth and the random poses similar to the color feature experiment in Section 4.1 which is shown in Figure 4.8. Here for each Euler angle difference, the mean and the standard deviation of the 20 viewed initial poses are visualized. Also, we visualize the tracking result with a correct initial pose as a baseline. In Figure 4.8a we observe that the X-Euler angle average of the random initial pose experiments is similar to the baseline. Furthermore, the standard deviation is between $2°$ and $3°$. For the Y-Euler angle visualized in Figure 4.8b the mean between $10s$ and $12.5s$ is at most $6°$ off the baseline and the standard deviation is between $1°$ and $2°$. Figure 4.8c shows the Z-Euler angle difference, where the mean

is nearly similar to the baseline with slight divergences over the course. The standard deviation is again between $1°$ and $2°$.

From these results, we conclude that a Euclidean error degradation of $5\mathrm{mm}$ in the positional part of the estimated pose is acceptable but not negligible. Further, the X-Euler and Z-Euler angles seem to be robust against varying initial poses. On the contrary, we observe a maximum offset of $6°$ at the Y-Euler angle. The results for other trajectories support the hypothesis that the Y-Euler angle is most likely to be affected by the random initial pose. However, we assume a maximum offset of $6°$ due to a random initial rotation of the SL-Block as relatively low. In summary, the initial pose can affect the tracking behavior, but within an acceptable range, and the tracking algorithm can track the SL-BLock despite this random initial position.

4.4 Outlier Ratio Comparison

As stated in Section 2.4, the outlier ratio ω determines the constant $c = \omega/(1-\omega)\ N/M$, which then determines the weighting factor $M^0_{x_i}/(M^0_{x_i}+c)$. Since ω is a hyperparameter, it is required to evaluate the influence of it on the tracking behaviour. Figure 4.9

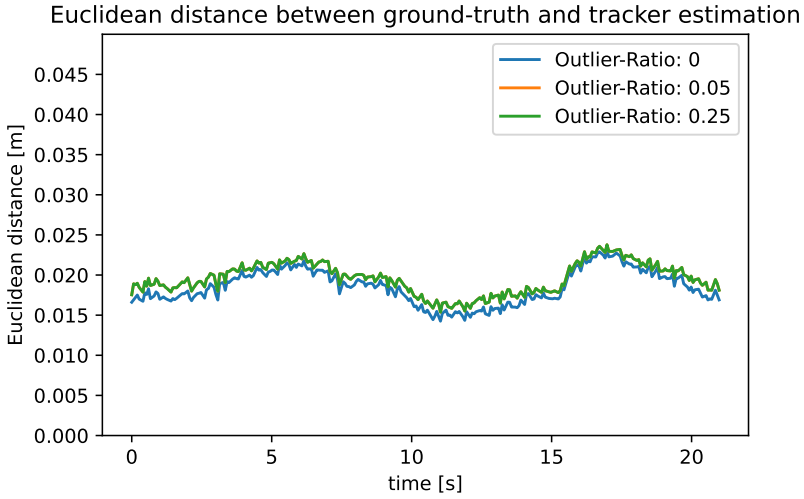
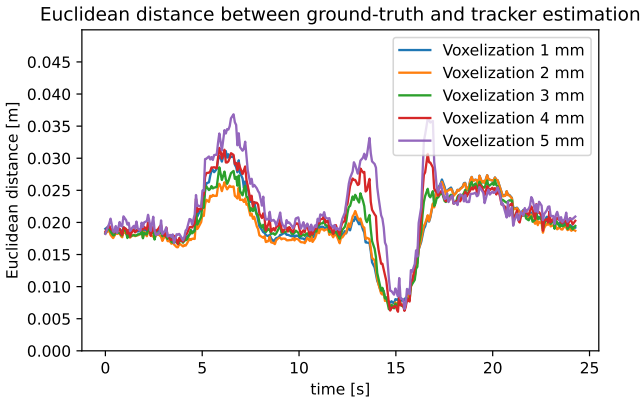


Figure 4.9: Comparison of different outlier ratios. Outlier ratio 0 is equivalent to a weighting factor of 1.

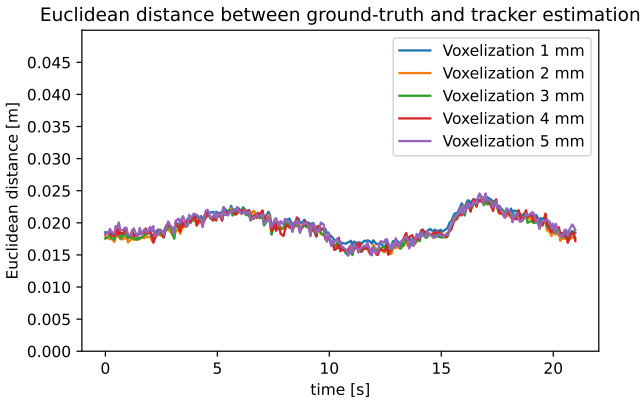
visualizes the Euclidean distance error between the ground truth and the estimate of tracker implementations with varying outlier ratios. We observe that between outlier-ratio 0.05 and outlier-ratio 0.25 no difference is identifiable, which is why the 0.05 outlier-ratio implementation is not visible in Figure 4.9. A minor difference between the outlier-ratio of 0 and both other outlier-ratio implementations can be noted. From this result, we conclude that the outlier ratio does not influence our object tracking scenario. We derive this behavior from the fact that we filter out all points around the last estimated pose that are further away than 10cm as described in Section 3.2.1. This filtering filters out many potential outliers and leaves only points that are considered similar for the tracking scenario, which results in values for the weighting factor that is close to 1. Therefore, the algorithm returns an almost identical result as if all weighting factors are set to 1. Since the weighting factors for the 0.05 and 0.25 outlier-ratio implementations are only close to 1, both of them diverge slightly compared to the 0 outlier-ratio implementation.

4.5 Voxelization

In our proposed tracking setup we reduce the number of points in the observation point cloud by voxelizing it. The resulting number of points depends on the cell size of the grid which is placed over the point cloud. The larger the cell size the more points are joined reducing the total number of points. It is important to find a trade-off where not too much information is lost by aggregating points, while still reducing the overall amount of points. Therefore, we test our implementation with different grid sizes on accuracy and computation time to find this trade-off.



(a) Random Bridge Trajectory



(b) Square Trajectory

Figure 4.10: Comparison of Euclidean distance accuracy with different voxel grid sizes

4.5.1 Accuracy

We evaluate the accuracy of the voxelization distances, visualized in Figure 4.10 by comparing the Euclidean distance error between each voxelization setup and the ground truth pose of the SL-Block. In Figure 4.10a we can state that the implementation of the tracking algorithm with a voxel size of $5\mathrm{mm}$ reaches a maximum error of $3.5\mathrm{cm}$. In general, we observe that with increasing voxel size the error also increases. Hereby, we witness at around $14.5s$ the highest divergence between the various voxel implementations. Here the error difference between the $5\mathrm{mm}$ and the $1\mathrm{mm}$ voxel implementation is about $2\mathrm{cm}$, even if only for a short time. Viewing the Square trajectory visualized in Figure 4.10b we cannot state a difference between any of these implementations.

As a result, we conclude that the influence of the voxel size depends on the viewed trajectory. We expected the behavior of the tracking algorithm for the Random Bridge trajectory as with a higher voxel size more information about the environment is lost because more points are summarized to one new point and with that, the number of points is further decreased. However, if a smaller cell size is chosen, a better estimation result can be expected. To find an acceptable trade-off between tracking accuracy and run-time performance, we evaluate in the next section the computation time for one iteration of the tracking algorithm for each voxel implementation.

4.5.2 Computation Time

As laid out in Section 4.5 the computation time depends on the number of points in the point cloud. This dependency makes it important to evaluate the computation duration for different voxel grid cell sizes. Therefore we evaluate the average duration of one iteration of the tracking loop.

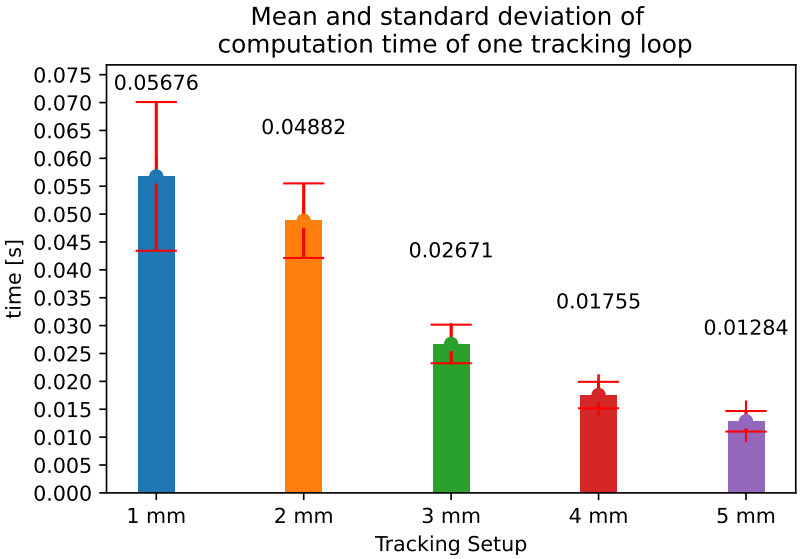


Figure 4.11: Computation time comparison between different voxel grid cell sizes in trajectory Square

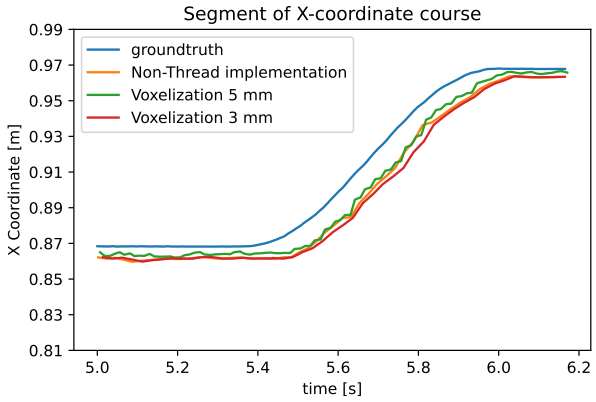
Figure 4.11 depicts the average computation time and the standard deviation of it. It is important to point out that the computation time results rely strongly on the used hardware. As mentioned in Section 3.3.2, we use a setup with an Intel(R) Core(TM) i7-6700K CPU and $16GB$ RAM. In our work, we observe in Figure 4.11 a steady decrease in computation time as voxel sizes become larger. Furthermore, the standard deviation is also reduced for larger voxel sizes.

We evaluate these results with particular attention to the ability to achieve real-time tracking results. By real-time, we define the ability to process input observation point cloud at least with a frame rate of 30Hz. With our hardware setup, we achieve this frame-rate at a voxel size of 3mm as depicted in Figure 4.11 the algorithm takes on average about $0.027s$ for one tracking iteration. This results in a frame-rate of $1/0.027\ \text{Hz} \approx 37\text{Hz}$.

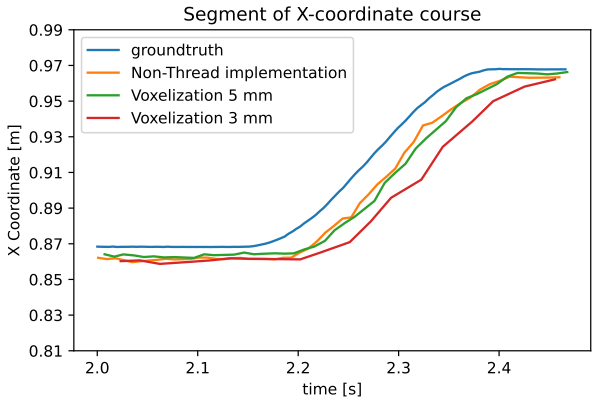
Moreover, we can reach a frame-rate of $1/0.013\ \mathrm{Hz} \approx 76\mathrm{Hz}$ using a voxel grid size of $5\mathrm{mm}$. Therefore, we conclude that our object tracking algorithm can handle real-time tracking applications in our experimental setup. Furthermore, we state that the voxel cell size parameter is a well suited parameter to tune the tracking algorithm in the context of accuracy and run-time performance.

4.6 Multi-Thread Implementation

In this section, we test our final implementation where the tracking algorithm runs in an infinite loop in its thread. As stated in Section 4.5, each loop iteration takes an amount of time to return an estimated pose dependent on the grid size of the voxelization. The parameters for the multi-thread implementation are depicted in Table 3.1.



(a) Input Frame-Rate 40Hz



(b) Input Frame-Rate 100Hz

Figure 4.12: Multithread implementation of object tracker following the Square trajectory at 40Hz/100Hz input frame-rate of new observation point clouds compared to ground truth and Non-Thread version.

Figures 4.12a and 4.12b visualize the results for two versions of our final object tracking algorithm with different voxel grid sizes. The difference between both figures is the different input frame-rates of new observations. Hereby in Figure 4.12a an input frame-rate of 40Hz is used. In this figure, we state that both final tracking algorithm implementations are capable to perform nearly as well as the Non-Thread version, which we used to evaluate the accuracy of the tracking algorithm in the previous sections. We also observe that the 5mm voxel implementation provides more results than the 3mm voxel implementation, which is understandable from the analysis in Section 4.5 because the tracking loop with 5mm voxel size needs about half of the calculation time compared to the 3mm voxel size implementation. In Figure 4.12b an input frame-rate of 100Hz is used. Hereby it stands out that after 2.2s the 3mm voxel size implementation lags behind the non-thread implementation. The 5mm voxel size implementation reaches decent results, although an input frame-rate of 100Hz is used.

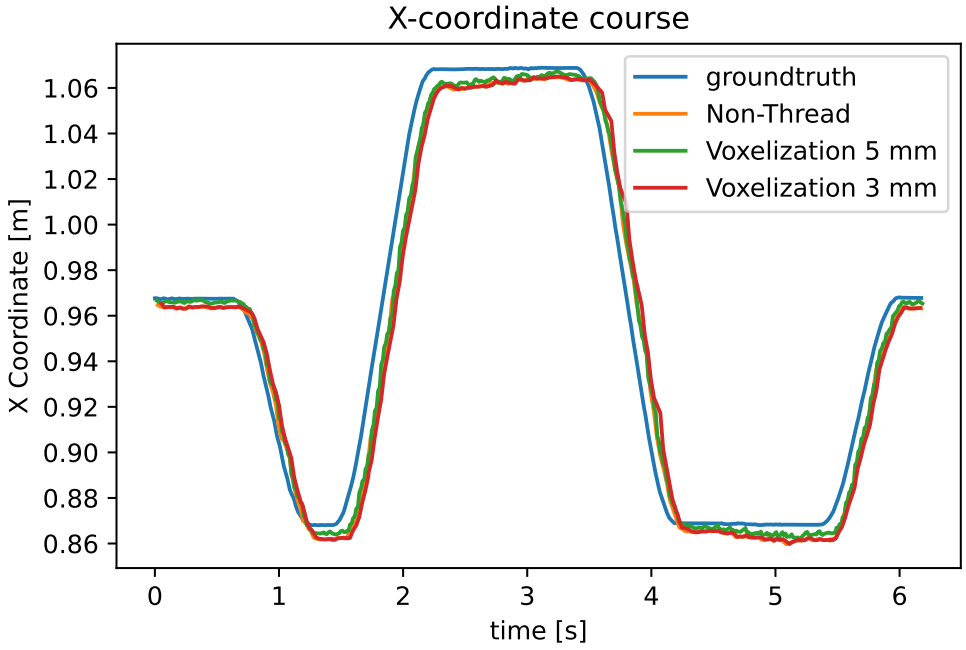


Figure 4.13: Multithread implementation of object tracker following the Square trajectory $at 40\text{Hz}$ input frame-rate of new observation point clouds compared to ground truth and Non-Thread version.

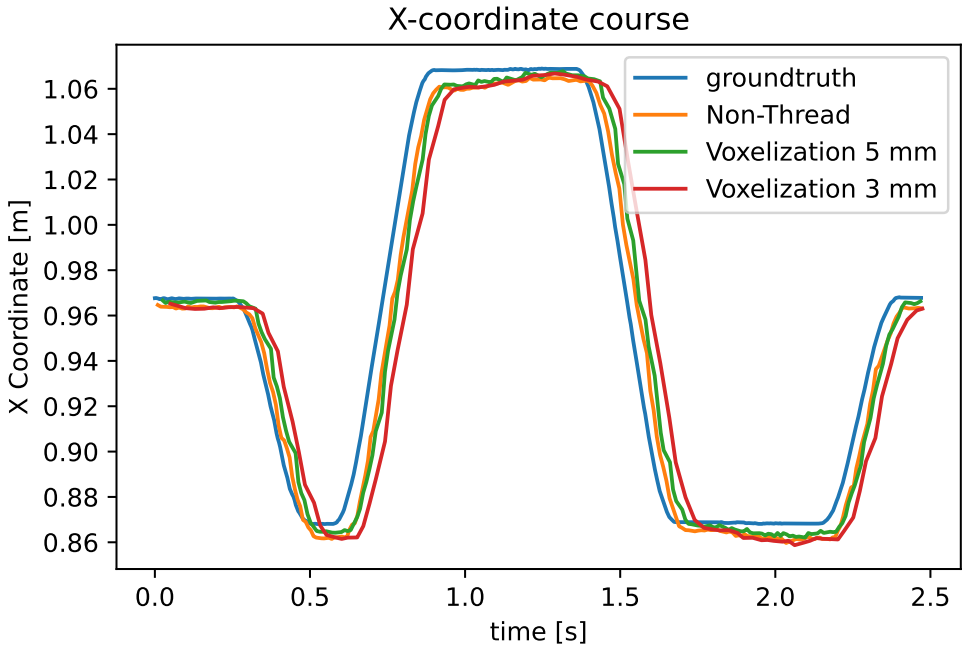


Figure 4.14: Multithread implementation of object tracker following the Square trajectory at 100Hz input frame-rate of new observation point clouds compared to ground truth and Non-Thread version.

In Figure 4.13 and 4.14 the estimation over the whole Square trajectory of both implementations are shown. The behavior of the tracking algorithms shown in Figure 4.13 verify that both object tracking implementations are capable to track the SL-BLock at a 40Hz input frame-rate. Furthermore, in Figure 4.14 we observe the same lag for 3mm voxel grid size implementation stated Figure 4.12a, though this offset is constant over time. Moreover, in Figure 4.14 we can verify the decent results of the 5mm voxel grid size implementation, since a small offset in parts of the trajectory is observable, though it is acceptable for an input frame-rate of 100Hz.

These experiments confirm our assumption that the object tracking algorithm has a delay due to its computation time compared to the non-threaded implementation. Depending on the choice of the voxel grid size the positional error due to the delay is narrower or wider. Since this offset is unavoidable, it can be argued to keep the computation time as small as possible to prevent the positional error from becoming too large. From this, we conclude that depending on the application, it must be considered how temporally accurate the

estimated pose must be and, if necessary, the resulting delay due to the computation time must be taken into account. We have evaluated our object tracking algorithm on several extensions we provide. Especially we focused on evaluating the accuracy and the computation time of the object tracking algorithm. In the final section, we summarize our results and give an outlook on potential future work.

5 Outlook

In this thesis, we first presented the foundations on which we build our object tracking algorithm. Next, we explained how we structured our approach to ensure reliable object tracking. Finally, we tested our algorithm on a dataset [8] which describes tasks of robotic assembly in a real robotics setup. Our presented implementation results in a reasonable tracking performance with a mean translational error of $2\mathrm{cm}$ and a maximum rotational error of $20°$, viewing the Euler angles. To reach this accuracy we proposed additional color features. These color features either reduced the error of the tracking result by almost half or allowed to reliably track the object in the scene. Furthermore, we evaluated a proposed adapting variance in [1] to improve the tracking result, which evidenced that for some trajectories improved tracking results could be reached at the same computation time. In addition, we concluded that the accuracy of the provided initial pose affects the tracking result but in an acceptable magnitude. We also stated that the computation time for one iteration of the object tracking algorithm relies on the number of points in the observation point cloud. Therefore we evaluated the voxelization in the pre-processing in more detail. Hereby, we concluded that the cell size of the voxel grid is a well-suited parameter to tune accuracy and computation time for each potential scope of application. Furthermore, we derived that our object tracking algorithm can process real-time frame-rates. Finally, we tested our multithread implementation of the object tracking algorithm, in which we expected a delay of the results due to the computation time of one iteration of the tracking algorithm. Therefore, we tested the tracking algorithm on two simulated input frame-rates of $40\mathrm{Hz}$ and $100\mathrm{Hz}$. At $40\mathrm{Hz}$ we could not observe a major delay due to its computation time, which let us conclude that our multithreaded tracking algorithm can handle real-time input frames. Furthermore, we also concluded that higher input frame-rates are also processable for our proposed object tracking algorithm if therefore the pre-processing is adjusted. Finally, we suggested keeping the emerging delay in mind and depending on the application adjusting the computation time, for example via the voxel grid size to obtain real-time pose estimation results.

Our results indicate that there is further potential to improve the tracking accuracy. As laid out in Section 3.2.2, our used SL-Block has markers on it that do not map well to our color features. Therefore, we conclude that either adjusting the color features in these areas or using a uniformly colored block further improves the tracking result. Furthermore, our color features can be also fine-tuned to reach better results, or other features could help to track the object even more robustly and accurately. For example potentially learned features could further boost the tracking accuracy, which is an extension that would be worthwhile to investigate in future work. Furthermore, we currently achieve the calculation speed by filtering the point cloud in the pre-processing step. To avoid positional filters, a KD-tree could be used to achieve real-time results in a scene. Herein, KD-trees can find points of interest in the scene which then can be used to compute the pose of the object. Such an extension to the existing baseline is also an interesting path to further explore in future work.

Finally, our object tracking algorithm is a model-based approach, which is why it can be easily transferred to other application areas. This property is very useful in the domain of robotic assembly and general pose estimation tasks and should be investigated in the future. Furthermore, as stated in Chapter 1 the focus of our object tracking algorithm is on the calculation of estimated poses in real-time, which we were able to achieve since we can reach a decent accuracy of 2cm and can handle frame-rates near 40Hz. In addition, higher frame rates are also possible to handle with our tracking algorithm, depending on the application. Finally, we conclude that our proposed tracking algorithm indicates promising results to reliably track an object in partially observable environments, which can be taken as a baseline and further elaborated. The next step could be to implement our object tracking algorithm on a setup and evaluate its long-term performance. To this end, a link to ROS would be very useful, allowing universal tracking in a wide range of scenarios.

Bibliography

[1] W. Gao and R. Tedrake, "Filterreg: Robust and efficient probabilistic point-set registration using gaussian filter and twist parameterization," 2019.

[2] "Optitrack." `https://optitrack.com/`. (Accessed on 03/22/2022).

[3] "Apriltag." `https://april.eecs.umich.edu/software/apriltag`. (Accessed on 03/21/2022).

[4] B. Wen, C. Mitash, B. Ren, and K. E. Bekris, "se(3)-tracknet: Data-driven 6d pose tracking by calibrating image residuals in synthetic domains," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10367–10373, 2020.

[5] Y. Xiang, T. Schmidt, V. Narayanan, and D. Fox, "Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes," *CoRR*, vol. abs/1711.00199, 2017.

[6] X. Deng, A. Mousavian, Y. Xiang, F. Xia, T. Bretl, and D. Fox, "Poserbpf: A rao-blackwellized particle filter for 6d object pose tracking," in *Robotics: Science and Systems (RSS)*, 2019.

[7] R. Radkowski, "Object Tracking With a Range Camera for Augmented Reality Assembly Assistance," *Journal of Computing and Information Science in Engineering*, vol. 16, 01 2016. 011004.

[8] L. Magnus, S. Menzenbach, M. Siebenborn, B. Belousov, and N. Funk, "Creating a dataset for object tracking in robotic assembly," 2021.

[9] Y. Wu, J. Lim, and M. Yang, "Object tracking benchmark," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 9, pp. 1834–1848, 2015.

[10] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[11] "Github - rlabbe/kalman-and-bayesian-filters-in-python: Kalman filter book using jupyter notebook. focuses on building intuition and experience, not formal proofs. includes kalman filters,extended kalman filters, unscented kalman filters, particle filters, and more. all exercises include solutions.." `https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python`, 10 2020. (Accessed on 03/22/2022).

[12] M. Wüthrich, P. Pastor, M. Kalakrishnan, J. Bohg, and S. Schaal, "Probabilistic object tracking using a range camera," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3195–3202, IEEE, Nov. 2013.

[13] S. Thrun, *Probabilistic robotics*. Communications of the ACM, 2002.

[14] J. Isaac, *Depth-Based Object Tracking Using a Robust Gaussian Filter*. IEEE International Conference on Robotics and Automation (ICRA), 2016.

[15] A. Myronenko and X. Song, "Point set registration: Coherent point drift," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, p. 2262–2275, Dec 2010.

[16] X. Huang, G. Mei, J. Zhang, and R. Abbas, "A comprehensive survey on point cloud registration," *CoRR*, vol. abs/2103.02690, 2021.

[17] F. Wang and Z. Zhao, "A survey of iterative closest point algorithm," in *2017 Chinese Automation Congress (CAC)*, pp. 4395–4399, 2017.

[18] O. Sorkine-Hornung and M. Rabinovich, "Least-squares rigid motion using svd," 2016. Technical note.

[19] H. Chui and A. Rangarajan, "A feature registration framework using mixture models," in *Proceedings IEEE Workshop on Mathematical Methods in Biomedical Image Analysis. MMBIA-2000 (Cat. No.PR00737)*, pp. 190–197, 2000.

[20] S. Borman, "The expectation maximization algorithm-a short tutorial," *Submitted for publication*, vol. 41, 2004.

[21] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[22] U. Robot, "Technical specifications ur10." https://www.universal-robots.com/media/50880/ur10_bz.pdf. (Accessed on 03/24/2022).

[23] "Intel realsense l515." https://www.intelrealsense.com/lidar-camera-l515/. (Accessed on 03/22/2022).

[24] "Robotis hand rh-p12-rn." https://www.robotis.us/robotis-hand-rh-p12-rn/. (Accessed on 03/22/2022).

[25] S.-G. Shih, "The art and mathematics of self-interlocking sl blocks," in *Proceedings of Bridges 2018: Mathematics, Art, Music, Architecture, Education, Culture* (E. Torrence, B. Torrence, C. Séquin, and K. Fenyvesi, eds.), (Phoenix, Arizona), pp. 107–114, Tessellations Publishing, 2018. Available online at http://archive.bridgesmathart.org/2018/bridges2018-107.pdf.

[26] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9-13 2011.

[27] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." https://eigen.tuxfamily.org/index.php?title=Main$_{P}age$, 2010.