TECHNISCHE
UNIVERSITÄT
WIEN

DIPLOMARBEIT

Erkennung und 3D Posenschätzung Zusammengesetzter Objekte

ausgeführt am

Institut für
Diskrete Mathematik und Geometrie
TU Wien

unter der Anleitung von

Associate Prof. Mag.rer.nat. Dr.techn. Christian Müller

durch

Florian Wimmer
[redacted]

Wien, am 12.12.2023

TECHNISCHE UNIVERSITÄT WIEN

DIPLOMA THESIS

Composite Object Detection and 3D Pose Estimation

written at the

Institute of
Discrete Mathematics and Geometry
Vienna University of Technology

supervised by

Associate Prof. Mag.rer.nat. Dr.techn. Christian Müller

by

Florian Wimmer
[redacted]

Vienna, 12.12.2023

Kurzfassung

Die orthogonale Gruppe O_n ist definiert als die Gruppe aller regulären $(n \times n)$-Matrizen A, deren transponierte Matrix A^T die Inverse von A ist. Die spezielle orthogonale Gruppe SO_n besteht aus allen orthogonalen $(n \times n)$-Matrizen mit Determinante 1. Sie repräsentiert Rotationen um den Ursprung in $\mathbb{R}^n$. Die spezielle Euklidische Gruppe SE_n besteht aus allen Paaren (R, t), wobei R eine Rotation in SO_n und t ein Vektor in $\mathbb{R}^n$ ist. Die Elemente von SE_n können die Posen von Objekten repräsentieren. Diese drei Untergruppen von GL_n sind differenzierbare Mannigfaltigkeiten.

Faktorgraphen sind bipartite Graphen mit Variablenknoten und Faktorknoten. Sie definieren die Faktorisierung einer Funktion und können die geometrischen Beziehungen verschiedener Objekte zueinander darstellen. Zusätzlich können Faktorgraphen eine probabilistische Struktur tragen.

Eine Retraktion ist eine Abbildung vom Tangentialbündel TM einer glatten Mannigfaltigkeit M auf M, die bestimmte Eigenschaften hat. Auf SO_n und SE_n können mithilfe der Exponentialfunktion für Matrizen Retraktionen definiert werden. Retraktionen ermöglichen die Anwendung iterativer Optimierungsmethoden auf Mannigfaltigkeiten analog zu Vektorräumen.

Im folgenden konkreten Anwendungsszenario werden Relativpositionen teilweise beweglicher Teile geschätzt. Betrachtet man einen Lastwagen als zusammengesetztes Objekt bestehend aus einfacheren Komponenten, wie zum Beispiel den Rädern des Lastwagens, erhält man eine Darstellung dieses zusammengesetzten Objekts als Faktorgraph. Die Variablenknoten des Faktorgraphen repräsentieren die verschiedenen Teile des Lastwagens, und die Faktorknoten die relativen Posen der Teile zueinander. Durch die Einführung eines Sensors, der einzelne Teile des Lastwagens beobachtet, erweitert sich dieser Faktorgraph. Für jeden Zeitschritt wird ein neuer Variablenknoten in den Faktorgraphen eingefügt, der den Sensor zu diesem Zeitpunkt repräsentiert. Die Beobachtungen des Sensors werden durch neue Faktorknoten dargestellt. Die Faktorknoten erhalten Wahrscheinlichkeitsdichten, wodurch die Berechnung einer maximalen a posteriori-Schätzung der Posen X unter gegebenen Beobachtungen Z möglich ist. Dabei wird die zusammengesetzte Wahrscheinlichkeitsfunktion $p(X, Z)$ mithilfe von Optimierung auf Mannigfaltigkeiten maximiert. Man erhält Schätzungen für die genaue Konfiguration des Lastwagens und die Pose des Sensors. Dieser Ansatz zur Posenschätzung zusammengesetzter Objekte kann mit dem Python-Paket GTSAM umgesetzt und getestet werden.

Abstract

The orthogonal group O_n is defined as the group of all invertible $(n \times n)$-matrices A whose transposed matrix A^T is the inverse of A. The special orthogonal group SO_n consists of all orthogonal $(n \times n)$-matrices with a determinant of 1. It represents rotations around the origin in $\mathbb{R}^n$. The special Euclidean group SE_n comprises all pairs (R, t), where R is a rotation in SO_n and t is a translation vector in $\mathbb{R}^n$. An element of SE_n can be used to represent the pose of an object. These three subgroups of GL_n are smooth manifolds.

Factor graphs are bipartite graphs with variable nodes and factor nodes and define the factorization of a function. They can encode geometrical relations among certain objects. Additionally, a factor graph can be equipped with a probabilistic structure.

A retraction is a mapping from the tangent bundle TM of a smooth manifold M to the manifold M that satisfies certain properties, such as the local rigidity condition. By utilizing the exponential map for matrices, retractions can be defined on SO_n and SE_n. Retractions allow simple implementations of iterative optimization techniques on manifolds.

In the following specific application scenario, the relative positions of partially movable components are estimated. Considering a truck as a composite object composed of simpler components, such as its wheels, leads to a representation of the truck as a factor graph. Variable nodes in the factor graph represent different parts of the truck, while factor nodes represent the relative poses of these parts to each other. Introducing a sensor observing specific parts of the truck expands the factor graph by adding variable nodes for the sensor at each time step and factor nodes for the observations. Equipping factor nodes with probability densities enables the computation of the maximum a posteriori estimate of some state X given observations Z by maximizing the joint probability function $p(X, Z)$ through optimization on manifolds. This approach provides estimates for the configuration of the truck and the pose of the sensor. Implementation and testing of this pose estimation method for composite objects can be achieved using the Python package GTSAM.

Acknowledgement

This thesis was developed in collaboration with the Austrian Institute of Technology (AIT) as part of the project *AWARD – All Weather Autonomous Real logistics operations and Demonstrations*. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101006817, see `https://award-h2020.eu/`. I am grateful for the opportunity to gain insights into applied research. I extend my thanks to my AIT supervisor, Markus Murschitz, and also to Katharina Ölsböck, Matthias Schörghuber, and all colleagues of the Assistive and Autonomous Systems research group at AIT.

A special expression of appreciation goes to Michael Schwingshackl, who invested countless hours in assisting me with every software problem, brainstorming new ideas, and generating data according to my needs. I would also like to acknowledge Philipp Schiller, who provided both moral and mathematical support during my time at AIT.

I wish to express my gratitude to Prof. Christian Müller, not only for supervising my thesis but also for his numerous outstanding lectures that fueled my interest in various fields of geometry.

Throughout my academic journey, I had the privilege of meeting a lot of amazing people. We spent countless hours studying, solving exercises, and admiring the beauty of mathematics. Thank you, Johanna, Christoph, Christian, Paul, Moritz, Konstantin, and others who exemplified the true value of teamwork. Together, we not only solved problems that none of us could understand individually, but we also made mathematics much more fun.

Lastly, I would like to thank my parents and my brother. From the very beginning, they sparked and nurtured my interest in mathematics and have been persistent in their support throughout my studies.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 12.12.2023

Contents

1. Introduction

Modern robot systems need robust and time-efficient techniques for detecting and locating objects in their environment. In this thesis, we discuss a novel technique for the estimation of the pose of composite objects, adapting existing methods used for related problems in robotics like Simultaneous Localization and Mapping (SLAM). This new technique includes the representation of composite objects as factor graphs and optimization on manifolds. In cooperation with the Austrian Institute of Technology (AIT), this approach was implemented and tested in an automated truck-loading scenario. Furthermore, we describe and implement an edge detection algorithm to detect the loading edge of a truck. Dellaert and Kaess present in [13] methods for modeling and solving problems in robotics with factor graphs. Here, we examine some mathematical background, especially the geometric aspects of the pose estimation problem.

The pose of an object in $\mathbb{R}^3$ is a distinguishable, static state of this object and can be represented by a matrix T in the special Euclidean group SE_3. It is commonly referred to as the position and orientation of this object. A matrix $T \in \mathrm{SE}_3$ has the form

$$T = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \in \mathbb{R}^{4\times 4}$$

for a vector $t \in \mathbb{R}^3$ indicating the position and a rotation matrix $R \in \mathrm{SO}_3$ indicating the orientation. The special orthogonal group SO_3 is the subgroup of all matrices in O_3 with determinant 1. The orthogonal group O_3 is the group of all matrices $A \in \mathbb{R}^{3\times 3}$ with

$$AA^T = A^T A = I_3$$

where I_3 denotes the identity matrix in $\mathbb{R}^{3\times 3}$.

The estimation of the pose of an object can result in an optimization problem. Given a measurement $z \in \mathbb{R}^n$ and an estimation function $h\colon \mathrm{SE}_3 \to \mathbb{R}^n$ that predicts measurements for given poses, we search for the matrix $T \in \mathrm{SE}_3$ that best approximates the measurements z under the function h. Hence, we have to solve

$$\operatorname*{arg\,min}_{T \in \mathrm{SE}_3} \|h(T) - z\|.$$

For this optimization problem, simple iterative optimization techniques like gradient descent fail. They rely on the updating rule

$$x^{(t+1)} = x^{(t)} + \alpha\delta^{(t)}$$

leading from the estimate $x^{(t)}$ in the time step t to an improved estimate $x^{(t+1)}$ in the next time step by taking a step in the direction of $\delta^{(t)}$. The sum

$$\begin{pmatrix} R_1 & t_1 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} R_2 & t_2 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R_1 + R_2 & t_1 + t_2 \\ 0 & 2 \end{pmatrix}$$

of two matrices T_1 and T_2 in SE_3 is not in SE_3 anymore. Even the sum $R_1 + R_2$ of two rotation matrices $R_1, R_2 \in \mathrm{SO}_3$ is in general not in SO_3. Therefore, we cannot expect to receive a new valid estimate $T^{(t+1)} \in \mathrm{SE}_3$ by adding some matrix $\delta^{(t)}$ to $T^{(t)} \in \mathrm{SE}_3$. To work around this problem, we will exploit the structure of SO_3 and SE_3 as smooth manifolds and use retractions. With the exponential map

$$e^A = \sum_{k\geq 0} \frac{A^k}{k!}$$

for quadratic matrices A, we will define a retraction $\mathcal{R}$ that brings certain matrices back onto the manifolds SO_3 respectively SE_3.

To estimate the poses of different parts of a composite object simultaneously while considering the geometrical relations of the different parts to one another, we will represent the composite object as a factor graph. A factor graph is a bipartite graph with variable nodes and factor nodes that defines a factorization of a function. The variable nodes represent the different parts of the composite object and a factor node connected to two variable nodes represents the geometric transformation between the respective parts. The variable nodes define variables and the factor nodes are equipped with functions dependent on the variables of the nodes they are connected to. The functions of the factor nodes will define probability densities dependent on variables X and observations Z. Consequently, the pose estimation problem for a composite object represented with a factor graph reads as a maximum a posteriori estimation problem of the joint probability function $p(X, Z)$, which can be reformulated as an optimization problem on the manifold SE_3.

Dellaert and various contributors realized this factor graph and manifold optimization in the Python and C++ package GTSAM [11]. We use GTSAM for the implementation of the pose estimation of a truck viewed as a composite object.

This work consists of two main parts. In Chapter 2, we discuss the mathematical backgrounds for the implemented solutions of the loading edge detection and the pose estimation problem described in Chapter 3. The essential parts of the codes for our solutions to the loading edge detection problem and the pose estimation problem can be found in Appendix A and Appendix B.

2. Mathematical Foundations

On the surface, pose estimation as described and implemented in Section 3.2 looks like a software engineering problem. However, various mathematical concepts and considerations are necessary to enforce robust real-time pose estimation algorithms.

Here, the main mathematical concepts – either used implicitly as assumptions and foundations for programming and utilized Python packages, or explicitly as geometrical relations and algorithms in the implementation of this specific pose estimation problem – will be established and summarized.

First and foremost, we take in Section 2.1 a look at the geometric aspects of this problem. We define poses and pose spaces, mathematical groups and spaces related to this problem, and smooth manifolds. Throughout this Section, we investigate the rotation group SO_3 from several perspectives.

Next, an important class of graphs will be introduced in Section 2.2. Factor graphs are the main idea of this pose estimation approach. Also, some common applications of factor graphs will be briefly described.

In Section 2.3, the employed optimization techniques are presented. On the one hand, we optimize on manifolds, in particular on SO_3. This raises the issue of moving on the manifold to reach better solutions. On the other hand, we will see how to optimize a factor graph.

Finally, some standard algorithms and algorithmic concepts like RANSAC and point cloud manipulation as well as geometric data structures used in this project are described and analyzed in Section 2.4.

2.1. Geometry

The detection of objects in 3D space using traditional methods raises various geometric issues. It starts with the reconstruction of a 3D scene from multiple images taken by one or several cameras. Hartley and Zisserman describe in [29, Chapter 18] a few methods for solving these problems such as bundle adjustment. The data for the loading edge detection algorithm described in Section 3.1 and the pose estimation algorithm of a composite object described in Section 3.2 was obtained by using a special stereo camera which uses bundle adjustment to compute depth information.

Another issue is the processing of point clouds. Some aspects are covered in [16] and their realizations in the C++ and Python library Open3D are briefly described in [64]. In Section 3.1 we apply and discuss some point cloud processing techniques on the loading edge detection problem.

To detect an object, for example in a point cloud, it can be useful to view it in a simplified way as a geometric 3D shape that can easily be described mathematically. As a result, we can exploit the well-known geometric properties in detection algorithms. For instance, the wheels of a truck as well as tree trunks resemble a right circular cylinder.

Going deeper into differential geometry, Dellaert and Kaess describe in [13, Chapter 6] methods for optimization on manifolds. Considering not only the position of an object in 3D space as a vector in $\mathbb{R}^3$, but also its orientation, raises the problem of how to search the space of possible solutions efficiently. The geometric foundation for this issue will be addressed in Section 2.1.2 and the actual optimization in Section 2.3.

These are just some of a variety of geometrical problems that arise in the surroundings of the tasks of object detection and pose estimation. In this Section, we start in 2.1.1 with the descriptions of mathematical groups like O_n, SO_n, and SE_n. Different notions for describing the position and orientation of objects (especially in $\mathbb{R}^3$) are reviewed. Then in Section 2.1.2, we will dive into differential geometry to establish the basics for a geometric understanding and structure for SO_n, discussed in 2.1.3. Furthermore, the exponential map for quadratic matrices is introduced and analyzed in 2.1.4, as we need it for optimization on SO_n.

2.1.1. Matrix Groups and Poses

Representing an object's position and orientation is a crucial starting point for real-life geometric considerations [5]. The orthogonal group O_n and the special orthogonal group SO_n are matrix groups, studied in linear algebra. Here, they are investigated to define ways of denoting the orientation and therefore the pose of an object, especially in $\mathbb{R}^3$.

In the following, the definitions and properties of O_n and SO_n are based on [32, Chapter 12] and [26, Chapter 1]. In order to formally define this so-called *pose* of an object, some mathematical preparation is required. The *general linear group* $\mathrm{GL}_n(\mathbb{R})$, or from now on just GL_n, is the group of all regular matrices in $\mathbb{R}^{n\times n}$ with the usual matrix multiplication as its group operation. Thus, these matrices represent all bijective linear maps from $\mathbb{R}^n$ to $\mathbb{R}^n$. Starting from GL_n, we can define other matrix groups.

Definition 2.1.1. The *orthogonal group* O_n is the set of all matrices A in $\mathbb{R}^{n\times n}$ that fulfill

$$AA^T = A^T A = I.$$

We have to check, whether the name *group* is justified for O_n. In the following theorem, some basic properties of O_n are investigated.

Theorem 2.1.2 (Properties of O_n)**.** *Let n be a positive natural number. Then the following properties of the orthogonal group O_n hold.*

(i) The orthogonal group O_n is a subgroup of GL_n.

(ii) The column vectors of any matrix $A \in \mathrm{O}_n$ are pairwise orthogonal with respect to the Euclidean inner product $\langle\cdot,\cdot\rangle$ on $\mathbb{R}^n$.

(iii) The column vectors of any matrix $A \in \mathrm{O}_n$ have (Euclidean) norm 1.

(iv) The column vectors of any matrix $A \in \mathrm{O}_n$ form an orthonormal basis of $\mathbb{R}^n$.

(v) For all $A \in \mathrm{O}_n$ the determinant $\det A$ is either $+1$ or -1.

(vi) Any $A \in \mathrm{O}_n$ preserves the inner product on $\mathbb{R}^n$, i.e. $\langle x, y\rangle = \langle Ax, Ay\rangle$.

(vii) If λ is an eigenvalue of an $A \in \mathrm{O}_n$, then $|\lambda| = 1$ holds.

Proof. (i) First, since for all $A \in \mathrm{O}_n$ the property $AA^T = A^TA = I$ holds, the transposed matrix A^T is the inverse of A. Therefore, A is regular and $\mathrm{O}_n \subseteq \mathrm{GL}_n$.

For the identity matrix $I \in \mathbb{R}^{n\times n}$, we see that $I = I^T$ and $II = I$. Thus, the multiplicative identity I is in O_n. Since $A^T = A^{-1}$ holds for all $A \in \mathrm{O}_n$, each element of O_n has its inverse element in O_n. Furthermore, for any $A, B \in \mathrm{O}_n$ the computation

$$(AB)(AB)^T = ABB^TA^T \overset{B\in\mathrm{O}_n}{=} AIA^T \overset{A\in\mathrm{O}_n}{=} I$$

shows the closure of O_n. Also, matrix multiplication is associative.

Thus, O_n is a group and because of $\mathrm{O}_n \subseteq \mathrm{GL}_n$ a subgroup of the general linear group.

(ii) Let A_i denote the i-th column vector of a matrix A and therefore also the i-th row vector of the matrix A^T. The equation $I = A^TA$ for an $A \in \mathrm{O}_n$ translates to

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} = I = A^TA = \begin{pmatrix} \langle A_1, A_1\rangle & \langle A_1, A_2\rangle & \cdots & \langle A_1, A_n\rangle \\ \langle A_2, A_1\rangle & \langle A_2, A_2\rangle & \cdots & \langle A_2, A_n\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle A_n, A_1\rangle & \langle A_n, A_2\rangle & \cdots & \langle A_n, A_n\rangle \end{pmatrix}. \tag{2.1}$$

Thus, the product $\langle A_i, A_j\rangle$ for $i \neq j$ equals 0, so the column vectors of A are pairwise orthogonal.

(iii) From (2.1) directly follows $\langle A_i, A_i\rangle$ for any column A_i of an orthogonal matrix A. Thus, the Euclidean norm of all columns of A is equal to 1.

(iv) From (i) we know that any $A \in \mathrm{O}_n$ is a regular matrix. Hence, the n columns of an orthogonal matrix are linearly independent and therefore, they form a basis of $\mathbb{R}^n$. With (ii) and (iii), it follows that the columns of any $A \in \mathrm{O}_n$ form an orthonormal basis of $\mathbb{R}^n$.

(v) Since the determinant of a matrix is compatible with matrix multiplication and transposition, the equation $I = AA^T$ leads to

$$1 = \det(I) = \det(AA^T) = \det(A)\det(A^T) = \det(A)^2$$

and thus $\det(A) = \pm 1$ for $A \in \mathrm{O}_n$.

(vi) The Euclidean inner product $\langle x, y\rangle$ can be viewed as x^Ty with x and y being column vectors in $\mathbb{R}^n$. Then we have $\langle Ax, Ay\rangle = (Ax)^T(Ay) = x^TA^TAy = x^Ty = \langle x, y\rangle$.

(vii) The eigenvalues of $A \in \mathrm{O}_n$ are the solutions λ of the equation $Av = \lambda v$. First, we take a look at the norm of the left side of the equation. In the following computation, we use the property $\langle x, y\rangle = \bar{x}^T y$ for the inner product in $\mathbb{C}^n$ (because eigenvectors of the real matrix A can be in $\mathbb{C} \setminus \mathbb{R}$) and $A^T = A^{-1}$ and get

$$\begin{aligned} \|Av\|^2 &= \langle Av, Av\rangle \\ &= \overline{(Av)}^T (Av) \\ &= \bar{v}^T A^T A v \\ &= \bar{v}^T v \\ &= \langle v, v\rangle = \|v\|^2. \end{aligned}$$

Thus, $\|v\| = \|Av\| = \|\lambda v\| = |\lambda|\|v\|$ implies $|\lambda| = 1$. So we can conclude that all eigenvalues of an orthogonal matrix have an absolute value of 1.

□

Remark 2.1.3. Property (vi) of the orthogonal group in Theorem 2.1.2 is widely used for an alternative, more general way of defining O_n on any vector space V with an inner product $\langle\cdot,\cdot\rangle$ on V: The orthogonal group O_n is the set of all automorphisms $f\colon V \to V$ that preserve the inner product, i.e. $\langle v, w\rangle = \langle f(v), f(w)\rangle$ [22, §0].

The automorphism $\mathbb{R}^n \to \mathbb{R}^n$ represented by a matrix $A \in \mathrm{O}_n$ is a reflection, a rotation, or a combination of reflection and rotation [26, Chapter 1]. As seen in Theorem 2.1.2, it preserves lengths and angles.

Example 2.1.4. The matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \in \mathrm{O}_2$$

represents a reflection at the line with the equation $y = x$ in $\mathbb{R}^2$. The matrix

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \in \mathrm{O}_2$$

represents the rotation around the origin by an angle of $\frac{\pi}{2}$. So the matrix

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \in \mathrm{O}_2$$

represents the linear transformation that first reflects points at the line $y = x$ and then rotates them by $\frac{\pi}{2}$ around the origin.

Definition 2.1.5. The *special orthogonal group* SO_n is the set of all matrices $A \in \mathrm{O}_n$ with $\det(A) = 1$.

Theorem 2.1.6. *The special orthogonal group* SO_n *is a subgroup of* O_n*.*

Proof. The identity matrix $I \in \mathrm{O}_n$ has determinant 1 and is therefore in SO_n. Since the inclusion $\mathrm{SO}_n \subseteq \mathrm{O}_n$ holds per definition, it holds that $A^T = A^{-1}$ for $A \in \mathrm{SO}_n$, and since $\det(A^T) = \det(A) = 1$, the matrix A has its inverse in SO_n. To show the closure of SO_n, we use the properties of determinants on $A, B \in \mathrm{SO}_n$ to get

$$\det(AB) = \det(A)\det(B) = 1.$$

Thus, SO_n is a group and because of $\mathrm{SO}_n \subseteq \mathrm{O}_n$ a subgroup of the orthogonal group. $\square$

Lemma 2.1.7. *For an odd $n > 1$, all $A \in \mathrm{SO}_n$ have 1 as an eigenvalue.*

Proof. The eigenvalues of a matrix $A \in \mathrm{SO}_n$ are the zeros of the characteristic polynomial $\chi_A(\lambda) = \det(A - \lambda I)$. It holds with $A^T = A^{-1}$, $\det(A) = 1$, and $\det(B) = \det(B^T)$ for all matrices $B \in \mathbb{R}^{n\times n}$

$$\begin{aligned}
\det(A - I) &= \det(A - AA^{-1}) \\
&= \det(A(I - A^{-1})) \\
&= \det(A)\det(I - A^T) \\
&= (-1)^n \det(A^T - I) \\
&= (-1)^n \det((A^T - I)^T) \\
&= (-1)^n \det(A - I).
\end{aligned}$$

For an odd n, the equation reads as $\det(A - I) = -\det(A - I)$, so $\det(A - I) = 0$ holds. Therefore, $\lambda = 1$ is an eigenvalue of A. $\square$

For an even n, all eigenvalues have an absolute value of 1 (see Theorem 2.1.2(vii)). However, a polynomial with an even degree does not even have to have real roots as, for example, the characteristic polynomial of

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \in \mathrm{SO}_2$$

shows.

The special orthogonal group SO_n is also referred to as the rotation group. Geometrically, the matrices in SO_2 represent planar rotations around the origin, and the matrices in SO_3 represent rotations in $\mathbb{R}^3$ around an axis through the origin [26, Chapter 1].

Example 2.1.8 (SO_2). To get a matrix $A \in \mathrm{SO}_2$, the two column vectors $(a_{11}, a_{21})^T$ and $(a_{12}, a_{22})^T$ of A have to be orthogonal, so

$$\left\langle \begin{pmatrix} a_{11} \\ a_{21} \end{pmatrix}, \begin{pmatrix} a_{12} \\ a_{22} \end{pmatrix} \right\rangle = a_{11}a_{12} + a_{21}a_{22} \stackrel{!}{=} 0.$$

Since the column vectors have norm 1, at least one of a_{11} and a_{21} has to be nonzero. Without loss of generality, let $a_{21} \neq 0$. The equation above translates to

$$a_{22} = -\frac{a_{11}}{a_{21}} a_{12}.$$

Therefore, the vector $(a_{12}, a_{22})^T$ is uniquely defined by $(a_{11}, a_{21})^T$ up to a scalar factor. Since both vectors have the same norm, we get

$$\begin{pmatrix} a_{12} \\ a_{22} \end{pmatrix} = \pm \begin{pmatrix} -a_{21} \\ a_{11} \end{pmatrix}.$$

To get unit length vectors, $a_{11}^2 + a_{21}^2 = 1$ has to hold. So, $a_{11} \in [-1, 1]$ follows. Let α be in $\{\arccos a_{11}, -\arccos a_{11}\}$ to get $a_{11} = \cos\alpha$. With the well-known trigonometric property $\cos^2\alpha + \sin^2\alpha = 1$ on the unit circle, we get $a_{21} = \sin\alpha$ if the sign of α was chosen accordingly. Furthermore, to get a positive determinant under these preconditions, we have to set $a_{22} = a_{11}$, because only then

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{21}a_{12} = \cos^2\alpha - \sin\alpha(-\sin\alpha) = \cos^2\alpha + \sin^2\alpha = 1$$

holds.

We conclude that all matrices $A \in \mathrm{SO}_2$ are of the form

$$A = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

for an $\alpha \in [-\pi, \pi]$, which is the range of $\pm\arccos$ on the real interval $[-1, 1]$. Allowing only angles $\alpha \in (-\pi, \pi]$ gives a one-to-one correspondence between SO_2 and the respective α.

As seen in Example 2.1.8, any element of SO_2 can be uniquely determined by one real number in $(-\pi, \pi]$, i.e. the angle of the corresponding planar rotation around the origin. We say that SO_2 has one *degree of freedom*.

Lemma 2.1.9. *Applying two rotations from SO_2 by angles α and β is the same as applying one rotation from SO_2 by the angle $\alpha + \beta$.*

Especially, the group SO_2 with the usual matrix multiplication is commutative.

Proof. Let $A, B \in \mathrm{SO}_2$ be two matrices in the special orthogonal group. We have seen that there exist $\alpha, \beta \in (-\pi, \pi]$ such that

$$A = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix}.$$

Multiplying these two matrices yields

$$\begin{aligned} AB &= \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \\ &= \begin{pmatrix} \cos\alpha\cos\beta - \sin\alpha\sin\beta & -\cos\alpha\sin\beta - \sin\alpha\cos\beta \\ \sin\alpha\cos\beta + \cos\alpha\sin\beta & -\sin\alpha\sin\beta + \cos\alpha\cos\beta \end{pmatrix}. \end{aligned}$$

Using sum identities for trigonometric functions, results in

$$AB = \begin{pmatrix} \cos(\alpha+\beta) & -\sin(\alpha+\beta) \\ \sin(\alpha+\beta) & \cos(\alpha+\beta) \end{pmatrix}$$

which is the matrix for a rotation by the angle $\alpha + \beta$.

Analogous computations for the matrix product BA give the same result which shows commutativity. $\square$

The elements of SO_3, the group of rotations in $\mathbb{R}^3$ [26, Chapter 1], do not behave as nicely as those of SO_2. A rotation in $\mathbb{R}^3$ around an axis through the origin can be viewed as a planar rotation in the plane through the origin that is orthogonal to the rotation axis when we use an orthogonal projection to project $\mathbb{R}^3$ onto this plane. Hence, in the simple case of a rotation around a coordinate axis, we can use the representation of elements of SO_2 we have derived above [32, p. 12.4.14]:

rotation around the x-axis	rotation around the y-axis	rotation around the z-axis
$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$	$\begin{pmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{pmatrix}$	$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$

The angle α gives the rotation in the mathematically positive direction when viewed against the direction of the coordinate axis (e.g. rotations around the z-axis are viewed from the top). The coordinate corresponding to the rotation axis is fixed.

In general, multiplication in SO_3 is not commutative, as a simple calculation with two rotation matrices around different axes shows.

To combine rotations with translations, we inspect two more groups.

Definition 2.1.10. The *Euclidean group* E_n is the set of all matrices of the form

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

where R is a matrix in O_n and t is a translation vector in $\mathbb{R}^n$.

This set is also called the *set of rigid transformations* [20].

Note that the elements of E_n are in $\mathbb{R}^{(n+1)\times(n+1)}$. A translation $\mathbb{R}^n \to \mathbb{R}^n$ is not a linear transformation, thus it cannot be represented by an $(n \times n)$-matrix. Working with homogeneous coordinates $(x^T, 1)^T$ of a point $x \in \mathbb{R}^n$ instead, allows us to represent rotation and translation with one projective transformation [46].

Definition 2.1.11. The *special Euclidean group* SE_n is the set of all matrices of the form

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \tag{2.2}$$

where R is a rotation matrix in SO_n and t is a translation vector in $\mathbb{R}^n$.

This set is also called the *set of proper rigid transformations* [46].

Since $\mathrm{SO}_n \subseteq \mathrm{O}_n$, it holds that $\mathrm{SE}_n \subseteq \mathrm{E}_n$. Furthermore, the following theorem holds.

Theorem 2.1.12. *The Euclidean group* E_n *and the special Euclidean group* SE_n *are subgroups of the general linear group* GL_{n+1}.

Proof. The proof for E_n and SE_n are completely analogous, so only the proof for SE_n is given.

First, we show that SE_n is a subset of GL_{n+1}. The determinant of the block diagonal matrix $A \in \mathrm{SE}_n$ of the form (2.2) can be computed as

$$\det \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} = \det(R) \det(1) = 1.$$

So, A is a regular matrix with the shape $(n+1) \times (n+1)$ and is therefore contained in GL_{n+1}.

We need to verify the group axioms for SE_n next. For $R = I_n \in \mathrm{SO}_n$ and t being the zero vector in $\mathbb{R}^n$, the matrix of the form (2.2) is the identity element in GL_{n+1}. Let

$$A_1 = \begin{pmatrix} R_1 & t_1 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad A_2 = \begin{pmatrix} R_2 & t_2 \\ 0 & 1 \end{pmatrix}$$

be elements of SE_n. Then

$$A_1 A_2 = \begin{pmatrix} R_1 R_2 & R_1 t_2 + t_1 \\ 0 & 1 \end{pmatrix}$$

which is in SE_n, since the product $R_1 R_2$ of two rotation matrices is again a rotation matrix and $R_1 t_2 + t_1 \in \mathbb{R}^n$. If we want A_2 to be the inverse matrix of A_1, then $R_1 R_2 = I$ and $R_1 t_2 + t_1 = 0$ must both be true. So $R_2 = R_1^{-1}$, which exists since rotation matrices are regular and their inverse is again a rotation matrix, and $t_2 = -R_1^{-1} t_1$. To check if this is the inverse of A_1 indeed, we compute

$$A_2 A_1 = \begin{pmatrix} R_1^{-1} & -R_1^{-1} t_1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} R_1 & t_1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R_1^{-1} R_1 & R_1^{-1} t_1 - R_1^{-1} t_1 \\ 0 & 1 \end{pmatrix} = I_{n+1}.$$

As a result, SE_n is a group and altogether a subgroup of GL_{n+1}. □

We can uniquely identify the elements of SE_n with pairs in $\mathrm{SO}_n \times \mathbb{R}^n$ if needed:

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \quad \longleftrightarrow \quad (R, t). \tag{2.3}$$

On the one hand, a matrix $A \in \mathrm{SE}_n$ is the transformation matrix of a linear mapping from $\mathbb{R}^{n+1}$ to $\mathbb{R}^{n+1}$. On the other hand, the pair $T = (R, t) \in \mathrm{SE}_n$ can be interpreted as the function

$$T \colon \begin{cases} \mathbb{R}^n \to \mathbb{R}^n \\ x \mapsto Rx + t \end{cases}. \tag{2.4}$$

For an element of SE_3, one can choose a rotation matrix $R \in \mathrm{SO}_3$ which has three degrees of freedom, and a translation vector $t \in \mathbb{R}^3$ where three values can be chosen independently. In total, SE_3 has six degrees of freedom.

To illustrate SO_3, we look at Euler's Rotation Theorem or as it is called in German *Satz vom Fußball* (theorem of the soccer ball) [18, Section 6.6].

Theorem 2.1.13 (Euler's Rotation Theorem / Satz vom Fußball)**.** *In a soccer match, if just one ball is used and it is placed at the beginning of the match and at the beginning of the second half of the match exactly at the kick-off point, then there are at least two points on the ball that are exactly in the same place both times.*

Proof. During the first half of the match, the ball was subject to rotations and translations. Thus, the transformation of the ball can be represented as a product $T_n T_{n-1} \dots T_1$ of transformations $T_i \in \mathrm{SE}_3$ for $i = 1, \dots, n$, where T_n denotes the last transformation of the ball at the end of the first half back to the kick-off point. According to Theorem 2.1.12, SE_3 is a group and the product is, therefore, some transformation $T = (R, t) \in \mathrm{SE}_3$. Since the ball is at the beginning of the second half at the same position as at the beginning of the first half, the translation part t of T is $(0, 0, 0)^T \in \mathbb{R}^3$. Thus, T represents a rotation $R \in \mathrm{SO}_3$.

For simplicity, we assume the ball to be the unit sphere $S^2 \subseteq \mathbb{R}^3$. To finish the proof, we have to find fixed points of S^2 under the rotation R, i.e. points $v \in \mathbb{R}^3$ with $Rv = v$. Since $n = 3$ is an odd number, Lemma 2.1.7 states that R has an eigenvalue $\lambda = 1$. This implies that there exists an (at least) one-dimensional subspace of $\mathbb{R}^3$ of eigenvectors v that fulfill $Rv = 1v$. A one-dimensional subspace of $\mathbb{R}^3$ is a line through the origin. The intersection of this line with S^2 is two (antipodal) points. These are the fixed points of the rotation R which finishes the proof. $\square$

Remark 2.1.14. The statement of Theorem 2.1.13 can be rephrased as follows: Every matrix in SO_3 represents a rotation in $\mathbb{R}^3$ around exactly one axis through the origin. Euler proved this version of the theorem in his paper [17] because he published the paper in 1775 and the game soccer in its now known form was invented in the 19th century [62].

According to this theorem, a rotation in $\mathbb{R}^3$ can be defined by an axis through the origin and an angle $\alpha \in (-\pi, \pi]$. The axis can be defined by a point in S^2. So we can say that SO_3 has three degrees of freedom [13, Chapter 6].

Now we try to define how to represent the position and orientation of an object, namely the *pose* of this object. Poses are frequently used in robotics and geometry but are rarely ever formally defined. In [13, Appendix B], a pose of a robot in $\mathbb{R}^2$ is defined as an element of SE_2 with the respective rotation matrix in SO_2 and a translation vector in $\mathbb{R}^2$. In [5, Section 2] Brégier et al. choose a more formal way and define poses generally for *rigid objects*. Our definition is based on [5, Section 2]. We will now focus on the typical use cases of poses, that is $\mathbb{R}^2$ or $\mathbb{R}^3$. So from now on in this section, $n \in \{2, 3\}$ if not stated otherwise.

Definition 2.1.15. Given a rigid object in $\mathbb{R}^n$, a *pose* of this object is a distinguishable, static state of this object.

The *pose space* $\mathscr{P}$ of this object is the set of all possible poses P of this object.

This definition of poses seems unnecessarily abstract for the naive intuition of the position and orientation of an object. This information could be described sufficiently by an element of SE_2 or SE_3. Definition 2.1.15 has an advantage over defining the pose of an object as an element of SE_n: *Distinguishable* in our definition means that an object in a pose P_1 can somehow be differentiated from the same object in a different pose P_2, which is a reasonable condition. Viewing the pose of an object with proper symmetry (e.g. the unit circle $S^1 \subseteq \mathbb{R}^2$, a cube in $\mathbb{R}^3$, et cetera) as an element of SE_n would violate the condition of distinguishability since one pose can be represented by more than one element of SE_n. Nevertheless, working with an element of SE_n seems to be easier than working with Definition 2.1.15. So we will identify the pose $P \in \mathscr{P}$ with an equivalence class of elements of SE_n as in [5, Section 2.2]. For objects without proper symmetry these equivalence classes contain for all poses exactly one element of SE_n each. If an object is symmetric, the equivalence classes for the object's poses contain more than one element of SE_n, depending on the symmetry class of the object. For deeper insights into symmetry classes of objects, we refer to the work of Schiller [52] that is related to this thesis.

We start with a reference pose $P_0 \in \mathscr{P}$ for a rigid object in $\mathbb{R}^n$ that can be chosen arbitrarily. Given a new valid pose $P_1 \in \mathscr{P}$ of this object, there exists a proper rigid transformation $T \in \mathrm{SE}_n$ as stated in (2.4) transforming the object from the pose P_0 to the pose P_1 in a way that every point x of the object in the reference pose is transformed to a point $T(x) = Rx + t$ of the object in the new pose P_1. We will also denote this as $T(P_0) = P_1$.

Definition 2.1.16. Given a rigid object in $\mathbb{R}^n$ for $n \in \{2, 3\}$, a reference pose P_0 in the object's pose space $\mathscr{P}$, and two transformations $T, T' \in \mathrm{SE}_n$, we say T and T' generate the same pose if $T(P_0) = T'(P_0) \in \mathscr{P}$. We write

$$T \sim_{P_0} T' \;\Leftrightarrow\; T(P_0) = T'(P_0).$$

The relation $\sim_{P_0}$ is an equivalence relation since reflexivity, symmetry, and transitivity are directly derived from "=". Now we can properly identify a pose in the pose space with a set of proper rigid transformations.

Definition 2.1.17. Given a rigid object in $\mathbb{R}^n$ for $n \in \{2, 3\}$ and a reference pose P_0 in the object's pose space $\mathscr{P}$, we define the class of proper rigid transformations representing a pose $P_1 \in \mathscr{P}$ as

$$\mathcal{T}_{P_0}^{P_1} := \{T \in \mathrm{SE}_n \mid T(P_0) = P_1\}.$$

The index P_0 can be omitted if it is clear or irrelevant which (fixed) pose is currently considered as the reference pose.

While a pose $P \in \mathscr{P}$ can refer to many elements of the special Euclidean group SE_n, any element of SE_n belongs to exactly one pose $P \in \mathscr{P}$ and thus defines a pose of an object uniquely.

After defining poses and pose spaces formally, we return to the actual usage of poses. The translation part of a pose P is relatively easy to handle, since for $(R, t) \in \mathcal{T}^P$ the translation vector t is in $\mathbb{R}^n$, a well-known vector space with the Euclidean inner product. The rotation R – until now viewed as an element of SO_n – is more challenging to manage. The mathematical structure of SO_n will be investigated in detail in Section 2.1.3. Dellaert and Kaess propose in [13, Appendix B] the most common types of representations of rotations in $\mathbb{R}^2$ and $\mathbb{R}^3$.

As stated above, an element of SO_2 can be uniquely determined by a real number in the interval $(-\pi, \pi]$. Moreover, every real number can be interpreted as the angle of the rotation around the origin. By allowing every $\alpha \in \mathbb{R}$, we lose the uniqueness of identifying a number with a rotation, since rotating by the angle α is the same as rotating by the angle $\alpha + 2\pi$. There is a homomorphism between the groups $\mathbb{R}$ and SO_2 given by the function that maps an $\alpha \in \mathbb{R}$ to the rotation in SO_2 by the angle α.

Another useful way to represent rotations in $\mathbb{R}^2$ are complex numbers. In the usual way, we identify $\mathbb{C}$ with $\mathbb{R}^2$ via

$$z = \operatorname{Re} z + i \operatorname{Im} z \in \mathbb{C} \longleftrightarrow \begin{pmatrix} \operatorname{Re} z \\ \operatorname{Im} z \end{pmatrix} \in \mathbb{R}^2.$$

Translating in $\mathbb{C}$ can be done by adding a number $z' \in \mathbb{C}$. Rotating by an angle α can be achieved by multiplying with the complex number $\cos\alpha + i \sin\alpha$ which has length 1. So the group of rotations is the set of complex numbers with length 1 together with multiplication in $\mathbb{C}$. This gives the one-to-one identification between rotations represented by the unit circle in $\mathbb{C}$ and rotations represented by SO_2:

$$\cos\alpha + i \sin\alpha \longleftrightarrow \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Rotations in $\mathbb{R}^3$ have various representations, too. The first way of representing a rotation is to describe it with an element of SO_3 as depicted above. According to Theorem 2.1.13, each element of SO_3 is a rotation around an axis through the origin. Therefore, we can represent a rotation as a pair $(a, \alpha) \in S^2 \times \mathbb{R}$ of an axis a, given by a direction on the unit sphere S^2, and an angle α. While easy to imagine, the description of a rotation in this way is not unique. For instance, the pairs $\left((\frac{\sqrt{2}}{2}, \frac{1}{2}, -\frac{1}{2})^T, \frac{\pi}{3}\right)$ and $\left((-\frac{\sqrt{2}}{2}, -\frac{1}{2}, \frac{1}{2})^T, -\frac{\pi}{3}\right)$ describe the same rotation. Moreover, we know from Theorem 2.1.13 that some rotations around different axes with different angles yield again a rotation around an axis through the origin, but it is not as obvious as with matrices in SO_3 how to combine the rotations.

The equivalent in $\mathbb{R}^3$ for complex numbers for 2D rotations are *quaternions*. Therefore, three pairwise different square roots i, j, k of -1 are introduced. For more detailed information about quaternions, see, for example, [23].

Lastly, another intuitive way of representing rotations that outlines the three degrees of freedom of a 3D rotation is *Euler angles*. Often referred to as *roll* φ, *pitch* ϑ, and *yaw* ψ, they compose a rotation as three consecutive rotations around different axes. There exist different conventions on the order of axes around which rotations occur. Sometimes the first rotation is around the x-axis, the second around the y-axis, and the third around the z-axis. (Proper) Euler angles are given for rotations around the z-, then the (rotated) x-, and then again the (rotated) z-axis. Also configurations like x, then y, then x or y, then z, then y are possible [24, Section 4.4].

From now on, we view the orientation of an object in $\mathbb{R}^3$ as a rotation matrix $R \in \mathrm{SO}_3$ if not stated otherwise.

2.1.2. Manifolds and Tangent Spaces

Pose estimation is the task of finding the pose that best fits some objective function with respect to certain preconditions and constraints [27]. As we know from above, the pose of an object represents its position and orientation. Just searching for the position that optimizes some (differentiable) function can be done with well-known methods such as gradient descent (see Section 2.3.1 for a revision of gradient descent). In general, we want to search in the neighborhood of a possible solution for a solution that is better with respect to the objective function. On the one hand, getting a position in the neighborhood of the position of an object in $\mathbb{R}^3$ can be done by adding some small vector $v \in \mathbb{R}^3$. Searching for an orientation represented as a rotation matrix $R \in \mathrm{SO}_3$ that is somehow close to another orientation, on the other hand, cannot be done by adding an arbitrary, "small" matrix $V \in \mathbb{R}^{3\times 3}$ to the matrix R. We know from Theorem 2.1.6 that SO_n with matrix multiplication is a group. But in general, SO_n is not closed with respect to matrix addition, as for instance the properties for matrices in O_n stated in Theorem 2.1.2 are not closed with respect to matrix or vector addition, so $R+V$ will not be in SO_3, typically. Thus, we have to take a closer look at the geometrical structure of SO_3 [13, Chapter 6]. The special orthogonal group SO_n is a manifold as we will see in 2.1.3. Here, we present the basics of manifolds and tangent spaces from differential geometry.

After introducing some elementary definitions, we will discuss the concept of smooth manifolds. First, we revise a few topological concepts.

Definition 2.1.18. (i) A *Hausdorff space* $(X, \mathcal{T})$ is a topological space that fulfills the T_2 separation axiom. Thus, for all points $x, y \in X$ with $x \neq y$ exist open neighborhoods U_x of x and U_y of y with $U_x \cap U_y = \emptyset$ [47, Chapter 2 §17].

(ii) Let X and Y be topological spaces. A *homeomorphism* $f\colon X \to Y$ is a continuous, bijective function where the inverse function f^{-1} is continuous as well [47, Chapter 2 §18].

(iii) An n-dimensional *topological manifold* M is a topological space with a countable basis that is Hausdorff and has the property that for every point $x \in M$ there exists an open neighborhood U_x of x and an open set $V \subseteq \mathbb{R}^n$ such that U_x is homeomorphic to V [37, Section 2.2].

To properly define smooth manifolds on the basis of topological manifolds, we give some notions from differential geometry. The definitions are based on [37, Section 2.2].

Definition 2.1.19. Let M be an n-dimensional topological manifold.

(i) Let $U \subseteq M$ be an open subset of M. For a homeomorphism $\varphi\colon U \to V$ into an open subset $V \subseteq \mathbb{R}^n$, the pair (U, φ) is a *chart* of M.

For a point $x \in U$, we call $\varphi(x) \in \mathbb{R}^n$ the *coordinates* of x in (U, φ) [1, Section 3.1.1].

(ii) Let $\mathcal{A} = ((U_i, \varphi_i))_{i\in I}$ be a family of charts of M for some index set I. The family $\mathcal{A}$ is called an *atlas* of M if $\bigcup_{i\in I} U_i \supseteq M$ holds.

(iii) The *transition map* between two charts (U_1, φ_1) and (U_2, φ_2) of M with $U_1 \cap U_2 \neq \emptyset$ is the function

$$\varphi_2 \circ \varphi_1^{-1}\colon \varphi_1(U_1 \cap U_2) \longrightarrow \varphi_2(U_1 \cap U_2).$$

(iv) Two charts (U_1, φ_1) and (U_2, φ_2) are called C^k*-compatible* if their transition map is a C^k-diffeomorphism, i.e. $\varphi_2 \circ \varphi_1^{-1}$ is bijective, k times continuously differentiable and its inverse function $\left(\varphi_2 \circ \varphi_1^{-1}\right)^{-1}$ is in C^k as well.

If the transition map $\varphi_2 \circ \varphi_1^{-1}$ is a C^∞-diffeomorphism, we say the charts (U_1, φ_1) and (U_2, φ_2) are C^∞*-compatible* or just *compatible*.

(v) A C^k-*atlas* $\mathcal{A} = ((U_i, \varphi_i))_{i \in I}$ of M is an atlas where the charts (U_i, φ_i) and (U_j, φ_j) are C^k-compatible for all $i, j \in I$ with $i \neq j$.

If the charts of $\mathcal{A}$ are pairwise compatible, we call $\mathcal{A}$ a C^∞-*atlas* or *smooth atlas*.

(vi) A C^k- or C^∞-atlas $\mathcal{A}$ of a topological manifold M is *maximal* if there is no chart (U, φ) of M that is C^k- or C^∞-compatible with all charts of $\mathcal{A}$ and not already contained in $\mathcal{A}$.

These definitions lead to smooth manifolds.

Definition 2.1.20. A C^k- or C^∞-*manifold* is a topological manifold provided with a maximal C^k- or C^∞-atlas.

We call a C^∞-manifold also a *smooth manifold*.

Example 2.1.21. Let's take a look at the set $\mathbb{R}^{m \times n}$ of all $(m \times n)$-matrices with real entries for $m, n \in \mathbb{Z}^+$ as in [1, Section 3.1.5]. Let $\varphi\colon \mathbb{R}^{m \times n} \to \mathbb{R}^{mn}$ be the function defined by

$$\varphi \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} a_{11} \\ \vdots \\ a_{m1} \\ a_{12} \\ \vdots \\ a_{mn} \end{pmatrix}.$$

So $\varphi(A)$ is obtained by stacking the column vectors of the matrix A on one another. This is a linear function between the two vector spaces $\mathbb{R}^{m \times n}$ and $\mathbb{R}^{mn}$. In the sense of Definition 2.1.18(ii), φ is a homeomorphism since it is continuous, bijective and the inverse φ^{-1} is bijective as well. The matrix space $\mathbb{R}^{m \times n}$ is an mn-dimensional topological manifold, because the topology, the Hausdorff property, and the second-countability are simply transferred via φ^{-1} from $\mathbb{R}^{mn}$, and any open neighborhood U_A of a point $A \in \mathbb{R}^{m \times n}$ is homeomorphic to the open set $\varphi(U_A)$ of $\mathbb{R}^{mn}$. Furthermore, $(\mathbb{R}^{m \times n}, \varphi)$ is a chart of $\mathbb{R}^{m \times n}$, and since it covers the whole space, we already have an atlas $\mathcal{A}$. We can add all charts (U, ψ) to the atlas $\mathcal{A}$ that are compatible with the chart $(\mathbb{R}^{m \times n}, \varphi)$. This gives us a maximal smooth atlas. Thus, $\mathbb{R}^{m \times n}$ with this structure is a (smooth) manifold.

Given two manifolds M_1 and M_2 of dimensions d_1 and d_2, the product space $M_1 \times M_2$ can be equipped with the product topology. For charts (U_1, φ_1) and (U_2, φ_2) of M_1 and M_2, the function

$$\psi\colon U_1 \times U_2 \to \mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \colon (x_1, x_2) \mapsto (\varphi_1(x_1), \varphi_2(x_2))$$

is a chart of $M_1 \times M_2$ since all properties directly transfer from φ_1 and φ_2. Thus, two atlases $\mathcal{A}_1$ and $\mathcal{A}_2$ of M_1 and M_2 create an atlas $\mathcal{A}$ of $M_1 \times M_2$. So, the product $M_1 \times M_2$ is a manifold [1, Section 3.1.6].

In Section 2.3.2, we will make use of the *tangent space* T_xM of a point x in a manifold M to optimize a function on the manifold M. There are various equivalent ways to define tangent vectors and spaces [40, Section 3.6], here we use smooth curves on the manifold as in [37, Section 2.6]. A smooth curve is a smooth function $\gamma\colon I \to M$ for an interval $I \subseteq \mathbb{R}$ (see [37, Chapter 2] for more about smooth functions on manifolds).

Definition 2.1.22. Let M be an n-dimensional smooth manifold, $x \in M$ a point on M, and $\mathcal{C}_x^M$ the set of all smooth curves $\gamma\colon I \to M$ on M with $0 \in I$ and $\gamma(0) = x$.

(i) Let $\gamma_1, \gamma_2 \in \mathcal{C}_x^M$ be two curves through $x \in M$. They are called *tangent at* x if there exists a chart (U, φ) of M with $x \in U$ and $(\varphi \circ \gamma_1)'(0) = (\varphi \circ \gamma_2)'(0)$. The function $\varphi \circ \gamma_i$ is a mapping $I \subseteq \mathbb{R} \to \mathbb{R}^n$, thus $(\varphi \circ \gamma_i)'(0)$ is the ordinary derivative. This definition does not depend on the choice of the chart (U, φ) (this can be verified with the chain rule of differentiation for another chart $(\tilde{U}, \tilde{\varphi})$ as in [37, Section 2.6]). Therefore, being tangent at a point x defines an equivalence relation on $\mathcal{C}_x^M$.

(ii) A *tangent vector* to M at x is an equivalence class of the relation defined in (i).

(iii) The set of all tangent vectors to M at x is called the *tangent space* T_xM to M at x.

(iv) The *tangent bundle*
$$TM := \{(x, \xi) \mid x \in M,\ \xi \in T_xM\}$$
of M is the disjoint union of all tangent spaces to M.

To establish a vector space structure on a tangent space T_xM, we define a function $\vartheta_\varphi^{x,M}$ between the tangent space and $\mathbb{R}^n$ similar to [37, Section 2.6.1].

Lemma 2.1.23. *Let M be an n-dimensional smooth manifold, x a point on M, and (U, φ) a chart of M with $x \in U$. The function $\vartheta_\varphi^{x,M}$ defined as*
$$\vartheta_\varphi^{x,M}\colon T_xM \to \mathbb{R}^n\colon \xi = [\gamma] \mapsto (\varphi \circ \gamma)'(0)$$
is a bijection.

Proof. First, we have to show that $\vartheta_\varphi^{x,M}$ is well-defined. According to Definition 2.1.22(i) of the equivalence relation on the set $\mathcal{C}_x^M$ of smooth curves, a different representative $\tilde{\gamma} \in \xi$ gives the same value $(\varphi \circ \tilde{\gamma})'(0)$ as $(\varphi \circ \gamma)'(0)$. Thus, $\vartheta_\varphi^{x,M}(\xi)$ is independent of the choice of a curve γ of ξ. Furthermore, $\varphi \circ \gamma$ has the codomain $\mathbb{R}^n$. Therefore, $\vartheta_\varphi^{x,M}$ is well-defined.

For two tangent vectors $\xi_1, \xi_2 \in T_xM$ with representatives $\gamma_i \in \xi_i$ for $i \in \{1, 2\}$ and $\xi_1 \neq \xi_2$, it holds that $(\varphi \circ \gamma_1)'(0) \neq (\varphi \circ \gamma_2)'(0)$ since γ_1 and γ_2 would have been in the same equivalence class otherwise. Thus, the function $\vartheta_\varphi^{x,M}$ is injective.

Let $p \in \mathbb{R}^n$ be an arbitrary element of the codomain of $\vartheta_\varphi^{x,M}$. We have to find a curve $\gamma\colon I \to M$ such that $(\varphi \circ \gamma)'(0) = p$ and $\gamma(0) = x$. Therefore, a reasonable ansatz is given by $\varphi \circ \gamma(t) = tp + \varphi(x)$. Indeed, the equivalence class ξ of the curve
$$\gamma(t) := \varphi^{-1}(tp + \varphi(x))$$
fulfills $\vartheta_\varphi^{x,M}(\xi) = p$. So, $\vartheta_\varphi^{x,M}$ is surjective and in total bijective. $\square$

This function allows us to equip T_xM with a vector space structure over the scalar field $\mathbb{R}$. Multiplication of a tangent vector $\xi \in T_xM$ with a scalar $s \in \mathbb{R}$ is defined by taking the image of ξ in $\mathbb{R}^n$ under $\vartheta_\varphi^{x,M}$, multiplying the result with s and bring it back to T_xM with the inverse of the bijection $\vartheta_\varphi^{x,M}$, i.e.
$$s\xi := (\vartheta_\varphi^{x,M})^{-1}\left(s\vartheta_\varphi^{x,M}(\xi)\right). \tag{2.5}$$

The addition for $\xi, \eta \in T_x M$ is defined similarly over the addition in $\mathbb{R}^n$ via

$$\xi + \eta := (\vartheta_\varphi^{x,M})^{-1} \left(\vartheta_\varphi^{x,M}(\xi) + \vartheta_\varphi^{x,M}(\eta)\right).$$

Remark 2.1.24. These two operations are well-defined, i.e. independent of the chart (U, φ). To see this, let's take a different chart (V, ψ) of M. According to [37, Section 2.6.1] the function $\vartheta_\psi^{x,M} \circ (\vartheta_\varphi^{x,M})^{-1}$ is linear (see also [37, Section 1.3]). So we can do the following equivalence transformations omitting the indices x and M

$$\begin{aligned}
\vartheta_\varphi^{-1}\left(\vartheta_\varphi(\xi) + \vartheta_\varphi(\eta)\right) &= \vartheta_\psi^{-1}\left(\vartheta_\psi(\xi) + \vartheta_\psi(\eta)\right) \\
\vartheta_\psi \circ \vartheta_\varphi^{-1}\left(\vartheta_\varphi(\xi) + \vartheta_\varphi(\eta)\right) &= \vartheta_\psi(\xi) + \vartheta_\psi(\eta) \\
\vartheta_\psi \circ \vartheta_\varphi^{-1}\left(\vartheta_\varphi(\xi)\right) + \vartheta_\psi \circ \vartheta_\varphi^{-1}\left(\vartheta_\varphi(\eta)\right) &= \vartheta_\psi(\xi) + \vartheta_\psi(\eta) \\
\vartheta_\psi(\xi) + \vartheta_\psi(\eta) &= \vartheta_\psi(\xi) + \vartheta_\psi(\eta)
\end{aligned}$$

that prove that this addition is independent of the chart. Analogously, scalar multiplication as defined above is well-defined.

Furthermore, the vector space axioms transfer directly from $\mathbb{R}^n$ [37, Section 2.6.1].

2.1.3. The Special Orthogonal Group SO_n

After the preparations of Section 2.1.2, we can investigate the geometric structure of the special orthogonal group SO_n. We will see that O_n is a submanifold of $\mathbb{R}^{n\times n}$ and conclude that SO_n is a smooth manifold. Furthermore, we investigate the tangent space of SO_n.

We start with a lemma about the representation of elements of SO_n with orthogonal matrices. The lemma is based on [21, Theorem 12.10] and parts of [21, Theorem 18.1], and will be used later.

Lemma 2.1.25. *Any matrix $R \in \mathrm{SO}_n$ can be represented in the form $R = PBP^T$ with an orthogonal matrix $P \in \mathrm{O}_n$ and a block diagonal matrix*

$$B = \mathrm{diag}(R_1(\alpha_1), R_2(\alpha_2), \ldots, R_m(\alpha_m), 1, \ldots, 1) \tag{2.6}$$

where $R_i(\alpha_i) \in \mathrm{SO}_2$ for $i = 1, \ldots, m$ denotes a rotation matrix

$$R_i(\alpha_i) = \begin{pmatrix} \cos\alpha_i & -\sin\alpha_i \\ \sin\alpha_i & \cos\alpha_i \end{pmatrix} \quad \text{with } 0 < \alpha_i \leq \pi.$$

Proof. Let $R \in \mathrm{SO}_n$ be a rotation matrix. The matrix R is orthogonal, so [21, Theorem 12.10] states that there exist an orthogonal matrix $P \in \mathrm{O}_n$ and a block diagonal matrix $B = \mathrm{diag}(B_1, B_2, \ldots, B_r)$ where the blocks B_j are either 1, -1, or of the form

$$B_j = \begin{pmatrix} \cos\theta_j & -\sin\theta_j \\ \sin\theta_j & \cos\theta_j \end{pmatrix} \quad \text{with } 0 < \theta_j < \pi, \tag{2.7}$$

with $R = PBP^T$ for $j = 1, \ldots, r$. Since $R \in \mathrm{SO}_n$, the determinant

$$\det(R) = \det(PBP^T) = \det(P)\det(B)\det(P^T) = \det(P)^2\det(B) \overset{P\in\mathrm{O}_n}{=} \det(B)$$

has to be $+1$. The determinant of the block diagonal matrix B is given by the product $\det(B_1)\det(B_2)\cdots\det(B_r)$. Blocks of the form (2.7) are in SO_2 and therefore have determinant 1. Thus, there is an even number of blocks of B that are -1. Hence we can assume, that B has the form $\mathrm{diag}(\tilde{B}_1, \tilde{B}_2, \ldots, \tilde{B}_{\tilde{r}})$ with blocks $\tilde{B}_j$ that are either 1 or of the form

$$\tilde{B}_j = \begin{pmatrix} \cos\tilde{\theta}_j & -\sin\tilde{\theta}_j \\ \sin\tilde{\theta}_j & \cos\tilde{\theta}_j \end{pmatrix} \quad \text{with } 0 < \tilde{\theta}_j \leq \pi, \tag{2.8}$$

for $j = 1, \ldots, \tilde{r}$, where two -1 entries create such a (2×2)-block, since $\cos\pi = -1$ and $\sin\pi = 0$. The blocks of B can be reordered by switching rows and columns with some orthogonal matrix S similar to

$$\underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}}_{=S} \begin{pmatrix} 1 & 0 & 0 \\ 0 & a & b \\ 0 & c & d \end{pmatrix} \underbrace{\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}}_{=S^T} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

at the correct positions. After that, B has the form (2.6), and the lemma is proven. □

In the following, we want to show that O_n and consequently SO_n are smooth manifolds. Therefore we give a version of a statement of differential geometry known as *rank theorem*, *regular level set theorem*, or as a theorem about implicitly defined manifolds that can be found in various books on manifolds or calculus like [59, Theorem 9.9], [40, Chapter 5], and [1, Proposition 3.3.3]. We refer to these sources for a proof of that statement.

Lemma 2.1.26. *Let M and N be two smooth manifolds of dimensions m and n with $m \geq n$. Furthermore, let $F\colon M \to N$ be a smooth function and $y \in N$ be a point on the manifold N. The point y is called a* regular value *of F if F has full rank at every point $x \in F^{-1}(y)$, i.e. if $DF(x)[\cdot]$ is surjective at every $x \in F^{-1}(y)$. If $y \in N$ is a regular value of F, the pre-image $F^{-1}(y)$ is a submanifold of M of dimension $m - n$.*

Now we consider the orthogonal group $\mathrm{O}_n \subseteq \mathbb{R}^{n\times n}$. Example 2.1.21 examined that $\mathbb{R}^{n\times n}$ is an n^2-dimensional smooth manifold. Analogously, one can prove that any finite-dimensional vector space V over $\mathbb{R}$ can be equipped with an atlas such that V is a smooth manifold. In particular, the set $\mathrm{Sym}_n \subseteq \mathbb{R}^{n\times n}$ of symmetric $(n\times n)$-matrices, i.e. $A = A^T$, is a smooth manifold. The following considerations are based on [1, Section 3.3.2].

Theorem 2.1.27. *The orthogonal group O_n is a smooth manifold of dimension $\frac{n(n-1)}{2}$.*

Proof. We will show that O_n is an embedded submanifold of $\mathbb{R}^{n\times n}$. Consider the function

$$F\colon \mathbb{R}^{n\times n} \to \mathrm{Sym}_n\colon A \mapsto A^T A - I_n.$$

Since $(A^TA)^T = A^TA$ for all $A \in \mathbb{R}^{n\times n}$, the function F is well-defined. According to Definition 2.1.1, it holds that $\mathrm{O}_n = F^{-1}(\{0_n\})$, where 0_n denotes the zero matrix in $\mathbb{R}^{n\times n}$.

Consider the differential $DF(A)[B]$ of F at A in the direction of B. With the Leibniz rule for differentiation, it holds (see [1, Appendix A.5] for details about matrix differentiation)

$$DF(A)[B] = A^TB + B^TA.$$

The mapping $DF(A)[\cdot]$ is surjective for every $A \in \mathrm{O}_n$ if for every $C \in \mathrm{Sym}_n$ there exists a matrix $B \in \mathbb{R}^{n\times n}$ with $DF(A)[B] = C$. For $A \in F^{-1}(\{0_n\}) = \mathrm{O}_n$ and $C \in \mathrm{Sym}_n$, let $B = \frac{1}{2}AC$, resulting in

$$DF(A)[\frac{1}{2}AC] = A^T\frac{1}{2}AC + (\frac{1}{2}AC)^T A = \frac{1}{2}(A^TAC + C^TA^TA) = C$$

with $A^TA = I_n$, since $A \in \mathrm{O}_n$, and $C = C^T$, since $C \in \mathrm{Sym}_n$. Thus, 0_n is a regular value of F and therefore O_n a submanifold of $\mathbb{R}^{n\times n}$ with Lemma 2.1.26. The vector space Sym_n is $\frac{n(n+1)}{2}$-dimensional since for a symmetric matrix S, every element of the diagonal of S and every element above this diagonal can be chosen independently. Hence, the dimension of O_n is given by $n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$. □

It follows that SO_n is a manifold as well if we can show that it is a connected component of O_n. A connected component of a topological space X is a subset $C \subseteq X$ that is connected and there exists no larger connected set $D \supsetneq C$ in X [47, Chapter 3 §25].

Theorem 2.1.28. *The special orthogonal group* SO_n *is a smooth manifold of dimension* $\frac{n(n-1)}{2}$.

Proof. We show that SO_n is a connected component of O_n as in [61]. Let A be any matrix in SO_n. The goal is to find an arc in SO_n that starts from I_n and ends in A. With Lemma 2.1.25, the matrix A can be represented in the form $A = PBP^T$ with an orthogonal matrix $P \in \mathrm{O}_n$ and a block diagonal matrix

$$B = \mathrm{diag}(R_1(\alpha_1), R_2(\alpha_2), \dots, R_m(\alpha_m), 1, \dots, 1),$$

for rotations $\mathbb{R}_i(\alpha_i) \in \mathrm{SO}_2$ by the angle $\alpha_i \in (0, \pi]$. For a $\lambda \in [0, 1]$, let $B(\lambda)$ define the block diagonal matrix

$$B(\lambda) = \mathrm{diag}(R_1(\lambda\alpha_1), R_2(\lambda\alpha_2), \dots, R_m(\lambda\alpha_m), 1, \dots, 1).$$

and $A(\lambda) = PB(\lambda)P^T$. Clearly, $A(\lambda) \in \mathrm{SO}_n$ for all $\lambda \in [0, 1]$. Furthermore, this arc starts at $A(0) = PI_nP^T = I_n$ and ends at $B(1) = A$. So the special orthogonal group SO_n is connected. Furthermore, since the function $\det\colon \mathrm{O}_n \to \{-1, 1\}$ is continuous, there cannot be a continuous path in O_n from a matrix $C \in \mathrm{O}_n$ with $\det C = 1$ to a matrix $D \in \mathrm{O}_n$ with $\det D = -1$. Hence, SO_n is a connected component of O_n and therefore open.

Thus, SO_n is an $\frac{n(n-1)}{2}$-dimensional topological manifold, since every point $x \in \mathrm{SO}_n$ has an open neighborhood $U_x \subseteq \mathrm{O}_n$ in O_n and therefore an open neighborhood $U_x \cap \mathrm{SO}_n$ in SO_n that is homeomorphic to some subset of $\mathbb{R}^{\frac{n(n-1)}{2}}$. In the same way, charts and atlases transfer from the smooth manifold O_n to SO_n. Consequently, SO_n is a smooth manifold of dimension $\frac{n(n-1)}{2}$. □

We want to investigate the tangent space to the manifold SO_n as in [21, Section 14.7]. At the identity I_n, the tangent vectors are given by the curves $\gamma\colon I \to \mathrm{SO}_n$ with $\gamma(0) = I_n$, w.l.o.g. let $I = (-1, 1)$. Since $\gamma(t)$ denotes a matrix in SO_n, we know $\gamma(t)\gamma(t)^T = I_n$. We can differentiate γ with respect to t resulting in

$$\gamma'(t)\gamma(t)^T + \gamma(t)\gamma'(t)^T = 0_n$$

with the product rule. Since $\gamma(0) = I_n$, this reduces to

$$\gamma'(0) + \gamma'(0)^T = 0_n.$$

Thus, $\gamma'(0)$ is a skew-symmetric matrix. With Skew_n we denote the set of all skew-symmetric matrices in $\mathbb{R}^{n\times n}$. This is a $\frac{n(n-1)}{2}$-dimensional vector space since every element above the diagonal of a matrix can be chosen arbitrarily. We have seen above, that $T_{I_n}\mathrm{SO}_n$ is a $\frac{n(n-1)}{2}$-dimensional vector space over $\mathbb{R}$ as well. Hence, the spaces Skew_n and $T_{I_n}\mathrm{SO}_n$ are equal (more precisely, they can be identified).

At some arbitrary point $B \in \mathrm{SO}_n$, let's consider some curve $\gamma_B\colon (-1,1) \to \mathrm{SO}_n$ with $\gamma_B(0) = B$. Then the curve $\tilde{\gamma}_B(t) := B^T\gamma_B(t)$ passes through I_n at 0. So as seen above, we can write

$$\tilde{\gamma}_B'(0) = B^T\gamma_B'(0) \in T_{I_n}\mathrm{SO}_n = \mathrm{Skew}_n$$

and therefore

$$T_B\mathrm{SO}_n = \{BS \mid S \in \mathrm{Skew}_n\}. \tag{2.9}$$

We have seen above that for two manifolds M_1 and M_2 the product $M_1\times M_2$ is a manifold. This directly implies the following theorem.

Theorem 2.1.29. *The special Euclidean group SE_n is a smooth manifold of dimension $\frac{n(n+1)}{2}$.*

Proof. With (2.3) the special Euclidean group can be identified with $\mathrm{SO}_n \times \mathbb{R}^n$. According to Theorem 2.1.28, SE_n is the product of two manifolds of dimensions $\frac{n(n-1)}{2}$ and n. Thus, SE_n is a smooth manifold of dimension $\frac{n(n+1)}{2}$. □

In the analysis of the pose estimation algorithm, we want to measure the quality of the estimation as the *distance* of the estimated poses to their ground truth. We will simplify the regarded objects to points in $\mathbb{R}^3$ with an orientation in SO_3. Thus, the representation of the poses as elements of SE_3 suffices since a point with an orientation has no symmetry. We can define a simple metric on SO_n and SE_n according to [34, Section 3.5].

Lemma 2.1.30. *The function*

$$d_{\mathrm{SO}_n}\colon \mathrm{SO}_n \times \mathrm{SO}_n \to \mathbb{R}_0^+\colon (R_1,R_2) \mapsto \|I_3 - R_1R_2^T\|_F$$

defines a metric on SO_n, where $\|A\|_F$ denotes the Frobenius norm for quadratic matrices.

Proof. The Frobenius norm $\|A\|_F$ of a matrix $A \in \mathbb{R}^{n\times n}$ is defined as

$$\|A\|_F := \sqrt{\sum_{i,j=1}^{n} |a_{ij}|^2} = \sqrt{\mathrm{tr}\,(AA^T)}$$

and according to [58, Section I.3] a norm. Therefore, $d_{\mathrm{SO}_n}(R_1,R_2) = \|I_n - R_1R_2^T\|_F$ is positive or zero. In particular, $\|I_n - R_1R_2^T\|_F = 0$ if and only if $I_n - R_1R_2^T = 0_n$. This is equivalent to $I_n = R_1R_2^T$, and, since R_2 is in O_n, this is equivalent to $R_1 = R_2$. We conclude that $d_{\mathrm{SO}_n}(R_1,R_2) = 0$ if and only if $R_1 = R_2$.

The computation

$$\|I_n - R_1R_2^T\|_F = \|(I_n - R_1R_2^T)^T\|_F = \|I_n - R_2R_1^T\|_F$$

implies that $d_{\mathrm{SO}_n}(R_1, R_2) = d_{\mathrm{SO}_n}(R_2, R_1)$.

To proof the triangle inequality $d_{\mathrm{SO}_n}(R_1, R_2) \leq d_{\mathrm{SO}_n}(R_1, R_3) + d_{\mathrm{SO}_n}(R_3, R_2)$ for any matrix $R_3 \in \mathrm{SO}_n$, we start by proving $\|I_n - RS^T\|_F = \|S - R\|_F$ for all $R, S \in \mathrm{SO}_n$. With $\|A\|_F^2 = \operatorname{tr}\left(AA^T\right)$ and $RR^T = SS^T = I_n$, we compute

$$\begin{aligned}
\|S - R\|_F^2 &= \operatorname{tr}\left((S - R)(S - R)^T\right) \\
&= \operatorname{tr}\left(SS^T - RS^T - SR^T + RR^T\right) \\
&= \operatorname{tr}\left(I_nI_n^T - (RS^T)I_n^T - I_n(RS^T)^T + I_nI_n^T\right) \\
&= \operatorname{tr}\left((I_n - RS^T)(I_n - RS^T)^T\right) \\
&= \|I_n - RS^T\|_F^2.
\end{aligned}$$

Thus, we conclude

$$\begin{aligned}
d_{\mathrm{SO}_n}(R_1, R_2) &= \|I_n - R_1R_2^T\|_F \\
&= \|R_2 - R_1\|_F \\
&= \|R_2 - R_3 + R_3 - R_1\|_F \\
&\leq \|R_2 - R_3\|_F + \|R_3 - R_1\|_F \\
&= \|I_n - R_3R_2^T\|_F + \|I_n - R_1R_3^T\|_F \\
&= d_{\mathrm{SO}_n}(R_1, R_3) + d_{\mathrm{SO}_n}(R_3, R_2).
\end{aligned}$$

So the triangle inequality holds for d_{SO_n} and therefore d_{SO_n} is a metric. □

Theorem 2.1.31. *The function*

$$d_{\mathrm{SE}_n} \colon \mathrm{SE}_n \times \mathrm{SE}_n \to \mathbb{R}_0^+ \colon (T_1, T_2) = ((R_1, t_1), (R_2, t_2)) \mapsto d_{\mathrm{SO}_n}(R_1, R_2) + d_2(t_1, t_2),$$

where d_2 is the Euclidean metric on $\mathbb{R}^n$, is a metric on SE_n.

Proof. With Lemma 2.1.30, d_{SO_n} is a metric on SO_n. Therefore, positivity, symmetry, and the triangle inequality transfer from the metrics d_{SO_n} and d_2 directly to d_{SE_n}. □

In the following, we will omit the indices SO_n and SE_n if it is clear which metric is used.

2.1.4. The Exponential Map for Quadratic Matrices

As discussed at the beginning of Section 2.1.2, it is not possible to make “small steps” towards an optimal solution on the SO_3 manifold by adding an arbitrary small matrix $V \in \mathbb{R}^{3\times3}$ to a rotation matrix $R \in \mathrm{SO}_3$. But if V has a certain structure, it can easily be brought back onto SO_3 using the exponential map for quadratic matrices [13, Section 6.1].

This exponential map is defined analogously to the power series of the exponential function on $\mathbb{C}$. Here the definition as well as the proof of the well-definedness and basic calculation rules are based on [21, Section 18.1]

Definition 2.1.32. The exponential e^A (sometimes denoted as $\exp A$) of a quadratic matrix $A \in \mathbb{C}^{n\times n}$ is defined as

$$e^A = \sum_{k\geq 0} \frac{A^k}{k!} \tag{2.10}$$

with the matrix powers $A^k = A^{k-1}A$ and $A^0 := I$ for all A.

To prove that e^A is well-defined, we first need a lemma about the powers of matrices.

Lemma 2.1.33. *Let $A \in \mathbb{C}^{n\times n}$ be a real quadratic matrix and let $a_{\max}$ be the maximum value of the absolute values $|a_{ij}|$ of all entries of A. Then for a positive integer $k \in \mathbb{Z}^+$, the absolute values of all entries $a_{ij}^{(k)}$ of the matrix A^k are bounded by $(na_{\max})^k$.*

Proof. Let $a_{\max} := \max_{1\leq i,j\leq n} |a_{ij}|$ be defined as stated above. We prove this lemma by induction on the exponent k of A. For $k = 1$, this statement is trivially true since $a_{ij}^{(1)} = a_{ij}$.

Assuming $|a_{ij}^{(k)}| \leq (na_{\max})^k$ is true for $k \in \mathbb{Z}^+$, consider A^{k+1}. Using $A^{k+1} = A^k A$, we can conclude for an entry $a_{ij}^{(k+1)}$ of A^{k+1}

$$\begin{aligned}
|a_{ij}^{(k+1)}| &= \left| \sum_{1\leq m\leq n} a_{im}^{(k)} a_{mj} \right| \\
&\overset{(1)}{\leq} \sum_{1\leq m\leq n} |a_{im}^{(k)}||a_{mj}| \\
&\overset{(2)}{\leq} \sum_{1\leq m\leq n} (na_{\max})^k a_{\max} \\
&= n^k \sum_{1\leq m\leq n} a_{\max}^{k+1} \\
&= n^k n a_{\max}^{k+1} = (na_{\max})^{k+1}.
\end{aligned}$$

For the inequality in (1), we use the triangle inequality and for the inequality in (2), we use the induction hypothesis.

Thus, $|a_{ij}^{(k+1)}| \leq (na_{\max})^{k+1}$ holds for all $k \in \mathbb{Z}^+$. ☐

Lemma 2.1.34. *The exponential map e^A of a matrix $A \in \mathbb{C}^{n\times n}$ as presented in Definition 2.1.32 is well-defined, i.e. the power series (2.10) converges absolutely for all $A \in \mathbb{C}^{n\times n}$.*

Proof. We say the matrix power series (2.10) converges absolutely if each entry of the matrix sequence

$$\sum_{k=0}^{N} \left| \frac{A^k}{k!} \right|$$

converges for $N \to \infty$. With the notation and result of Lemma 2.1.33 we can write for the series of one entry

$$\sum_{k\geq 0} \frac{|a_{ij}^{(k)}|}{k!} \leq \sum_{k\geq 0} \frac{(na_{\max})^k}{k!} = e^{na_{\max}}$$

and thus the series (2.10) converges absolutely due to the direct comparison test. □

The exponential map has some interesting properties [21, Section 18.1].

Lemma 2.1.35. *Let $A \in \mathbb{R}^{n\times n}$ and $B, C \in \mathbb{C}^{n\times n}$ be quadratic matrices.*

(i) If B and C commute, i.e. $BC = CB$, then

$$e^B e^C = e^{B+C}.$$

(ii) Let P be a regular matrix in $\mathrm{GL}_n(\mathbb{R})$ or in $\mathrm{GL}_n(\mathbb{C})$ and let D be a matrix in $\mathbb{R}^{n\times n}$ or $\mathbb{C}^{n\times n}$. Then, the equation

$$e^{PDP^{-1}} = Pe^D P^{-1}$$

holds.

(iii) For the determinant of e^A holds

$$\det(e^A) = e^{\operatorname{tr} A},$$

where $\operatorname{tr} A$ denotes the trace $a_{11} + a_{22} + \cdots + a_{nn}$ of A.

(iv) The exponential of A is regular, i.e. $e^A \in \mathrm{GL}_n$.

(v) The inverse of e^A is given by

$$(e^A)^{-1} = e^{-A}.$$

Proof. (i) To prove this property, consider the power series representations of e^B and e^C and their Cauchy product. In the following computation, we use the binomial formula for the equality in (1) which only holds since $BC = CB$, so

$$\begin{aligned}
e^B e^C &= \left(\sum_{i\geq 0} \frac{B^i}{i!}\right)\left(\sum_{j\geq 0} \frac{C^j}{j!}\right) \\
&= \sum_{k\geq 0} \sum_{l=0}^{k} \frac{B^l}{l!} \frac{C^{k-l}}{(k-l)!} \\
&= \sum_{k\geq 0} \frac{1}{k!} \sum_{l=0}^{k} \binom{k}{l} B^l C^{k-l} \\
&\overset{(1)}{=} \sum_{k\geq 0} \frac{1}{k!} (B+C)^k = e^{B+C}.
\end{aligned}$$

This proves the statement.

(ii) Since $PP^{-1} = I$ and $(PD^{k-1}P^{-1})(PDP^{-1}) = PD^kP^{-1}$, induction shows that

$$(PDP^{-1})^k = PD^kP^{-1}$$

for every $k \geq 0$. Thus, we see

$$e^{PDP^{-1}} = \sum_{k\geq 0} \frac{(PDP^{-1})^k}{k!} = \sum_{k\geq 0} \frac{PD^kP^{-1}}{k!} = Pe^DP^{-1}.$$

So $e^{PDP^{-1}} = Pe^DP^{-1}$ is shown.

(iii) The characteristic polynomial χ_A of the matrix A decomposes into linear factors over $\mathbb{C}$ due to the fundamental theorem of algebra. So according to [32, Satz 8.7.10] the matrix A interpreted as an element of $\mathbb{C}^{n\times n}$ is similar to a matrix J in Jordan normal form, i.e. there exists a matrix $P \in \mathrm{GL}_n(\mathbb{C})$ with $A = PJP^{-1}$.

Part (ii) implies that

$$e^A = e^{PJP^{-1}} = Pe^JP^{-1}.$$

The Jordan matrix J is an upper triangular matrix with the (complex) eigenvalues λ_i for $i = 1, \ldots, n$ of A in its diagonal (according to their algebraic multiplicity). Again, a simple induction shows that for any upper triangular matrix $T = (t_{ij})$ the matrix e^T is also an upper triangular matrix, with the diagonal entries $e^{t_{ii}}$. So the main diagonal of e^J consists of the exponentials e^{λ_i} of the eigenvalues λ_i of A.

The determinant of a triangular matrix is the product of its diagonal entries. To sum up, the determinant of e^A computes as

$$\det(e^A) = \det(Pe^JP^{-1}) = \det(P)\det(P)^{-1}\det(e^J) = e^{\lambda_1}e^{\lambda_2}\cdots e^{\lambda_n} = e^{\lambda_1+\lambda_2+\cdots+\lambda_n},$$

where the multiplicativity of the determinant and $\det(P^{-1}) = \det(P)^{-1}$ was used.

The trace $\operatorname{tr} A$ of the matrix A is defined as the sum $a_{11}+a_{22}+\cdots+a_{nn}$ of the diagonal entries of A. Furthermore, the trace $\operatorname{tr} A$ is equal to the sum of the eigenvalues of A [33, Section 1.2]. This is no contradiction to complex eigenvalues: Since χ_A is a polynomial with real coefficients, for every zero $u = a + ib$ of χ_A that lies in $\mathbb{C} \setminus \mathbb{R}$, the complex conjugate $\bar{u} = a - ib$ is a zero of χ_A, too. The sum $u + \bar{u} = 2a$ is in $\mathbb{R}$, therefore the sum of all eigenvalues of A is in $\mathbb{R}$.

In total, we have

$$\det(e^A) = e^{\lambda_1+\lambda_2+\cdots+\lambda_n} = e^{\operatorname{tr} A},$$

which completes the proof.

(iv) According to (iii) the determinant of e^A is equal to $e^{\operatorname{tr} A}$ which is positive. Therefore, e^A is regular for any $A \in \mathbb{R}^{n\times n}$.

(v) As we learned in (iv), the matrix e^A is regular, which means that it has an inverse. The matrices A and $-A$ commute, since $A(-A) = -A^2 = (-A)A$. Thus, with (i) follows

$$e^Ae^{-A} = e^{A-A} = e^{0_n} = I_n.$$

A similar computation delivers $e^{-A}e^{A} = I_n$ which shows that e^{-A} is the inverse matrix of e^A.

□

A matrix $A \in \mathbb{R}^{n\times n}$ is *skew-symmetric* if $A = -A^T$. We can derive an explicit formula for e^A if A is a skew-symmetric matrix in $\mathbb{R}^{2\times 2}$.

Theorem 2.1.36. *Let $A \in \mathbb{R}^{2\times 2}$ be a skew-symmetric matrix of the form*

$$A = \begin{pmatrix} 0 & -\theta \\ \theta & 0 \end{pmatrix}.$$

Then the exponential of A is given by

$$e^A = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

Thus, e^A is in SO_2 [21, Section 18.1].

Proof. First, we take a look at the skew-symmetric matrix

$$M := \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

By a simple induction with the induction start

$$M^2 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \quad M^3 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad M^4 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

and $A = \theta M$, we see for any integer $k \geq 0$

$$A^{4k+1} = \theta^{4k+1}\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \theta^{4k+1}M, \quad A^{4k+2} = \theta^{4k+2}\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} = -\theta^{4k+2}I_2,$$

$$A^{4k+3} = \theta^{4k+3}\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} = -\theta^{4k+3}M, \quad A^{4k+4} = \theta^{4k+4}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \theta^{4k+4}I_2.$$

In Lemma 2.1.34 we proved that the power series of e^A converges absolutely. Thus, we can rearrange the terms. So, rearranging and using the series expansions of $\sin\theta$ and $\cos\theta$ gives

$$\begin{aligned}
e^A &= \sum_{k\geq 0}\frac{A^k}{k!} = I_3 + \frac{A}{1!} + \frac{A^2}{2!} + \frac{A^3}{3!} + \frac{A^4}{4!} + \frac{A^5}{5!} + \dots \\
&= \left(I_3 + \frac{1}{2!}A^2 + \frac{1}{4!}A^4 + \dots\right) + \left(A + \frac{1}{3!}A^3 + \frac{1}{5!}A^5 + \dots\right) \\
&= \left(\theta^0 I_2 + \frac{1}{2!}(-\theta^2 I_2) + \frac{1}{4!}\theta^4 I_2 + \dots\right) + \left(\theta M + \frac{1}{3!}(-\theta^3 M) + \frac{1}{5!}\theta^5 M + \dots\right) \\
&= \left(\sum_{k\geq 0}(-1)^k\frac{\theta^{2k}}{(2k)!}\right) I_2 + \left(\sum_{k\geq 0}(-1)^k\frac{\theta^{2k+1}}{(2k+1)!}\right) M \\
&= \cos\theta I_2 + \sin\theta M = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.
\end{aligned}$$

This computation finishes the proof. □

This proves that the exponential e^A of a skew-symmetric matrix $A \in \mathbb{R}^{2\times 2}$ is a rotation matrix in SO_2. This even holds for skew-symmetric matrices in $\mathbb{R}^{n\times n}$ for arbitrary $n \geq 2$. Gallier gives in [21, Theorem 18.1] the following statement.

Theorem 2.1.37. *Let* $\mathrm{Skew}_n \subseteq \mathbb{R}^{n\times n}$ *be the set of all real skew-symmetric* $(n\times n)$*-matrices.*

(i) For all $A \in \mathrm{Skew}_n$*, the exponential* e^A *is an element of* SO_n*.*

(ii) The exponential map

$$e^{\cdot} : \mathrm{Skew}_n \to \mathrm{SO}_n \tag{2.11}$$

with the domain Skew_n *and the codomain* SO_n *is surjective.*

Proof. (i) Let $A \in \mathrm{Skew}_n$ be a skew-symmetric matrix. First, we show that $A \in \mathrm{O}_n$. Since A is skew-symmetric, we know $A^T = -A$. Considering the power series representation of e^A, we see that $(e^A)^T = e^{A^T}$. With Lemma 2.1.35(v) follows

$$\left(e^A\right)^T e^A = e^{A^T} e^A = e^{-A} e^A = I_n$$

and $e^A (e^A)^T = I_n$ analogously. Thus, the matrix e^A is orthogonal, i.e.

$$\left(e^A\right)^{-1} = \left(e^A\right)^T.$$

Second, we compute the determinant $\det(e^A)$. The property $A^T = -A$ implies that the diagonal entries of A are all equal to zero. Thus, $\operatorname{tr} A = 0$. With Lemma 2.1.35(iii) follows

$$\det(e^A) = e^{\operatorname{tr} A} = e^0 = 1.$$

To sum up, from $e^A \in \mathrm{O}_n$ and $\det(e^A) = 1$ follows that $e^A \in \mathrm{SO}_n$.

(ii) Part (i) guarantees that the function (2.11) is well-defined. Let $R \in \mathrm{SO}_n$ be a rotation matrix. With Lemma 2.1.25, the matrix R can be represented in the form $R = PBP^T$ with an orthogonal matrix $P \in \mathrm{O}_n$ and a block diagonal matrix

$$B = \operatorname{diag}(B_1, B_2, \ldots, B_r, 1, \ldots, 1).$$

The blocks $B_j \in \mathrm{SO}_2$ are of the form

$$B_j = \begin{pmatrix} \cos\theta_j & -\sin\theta_j \\ \sin\theta_j & \cos\theta_j \end{pmatrix} \quad \text{with } \ 0 < \theta_j \leq \pi.$$

We have to find a skew-symmetric matrix A with $e^A = R$. Let's define a block diagonal matrix $C = \operatorname{diag}(C_1, C_2, \ldots, C_r, 0, \ldots, 0)$ as follows. For a block B_j let C_j be the (2×2)-matrix

$$C_j := \begin{pmatrix} 0 & -\theta_j \\ \theta_j & 0 \end{pmatrix}. \tag{2.12}$$

The exponential e^C of the block diagonal matrix C can be computed by taking the exponential of the blocks separately as

$$e^C = \operatorname{diag}\left(e^{C_1}, e^{C_2}, \ldots, e^{C_r}, e^0, \ldots, e^0\right).$$

The exponential e^{C_j} for $j = 1, \dots, r$ equals B_j due to Theorem 2.1.36. Since $e^0 = 1$, we have $e^C = B$.

Now, let A be the matrix PCP^T with the orthogonal matrix P from above. The matrix C consists of skew-symmetric blocks on its diagonal. So C is skew-symmetric, implying $C + C^T = 0_n$. We can compute

$$A + A^T = PCP^T + (PCP^T)^T = PCP^T + PC^TP^T = P(C + C^T)P^T = 0_n,$$

which shows that A is skew-symmetric. The exponential of A can be rewritten as

$$e^A = e^{PCP^T} \overset{2.1.35(ii)}{=} Pe^CP^T = PBP^T = R$$

with the considerations above and Lemma 2.1.35(ii).

So, for an arbitrary rotation matrix $R \in \mathrm{SO}_n$, we found a skew-symmetric matrix $A \in \mathrm{Skew}_n$ such that $e^A = R$. Thus, the exponential map is surjective with domain Skew_n and codomain SO_n.

□

For $n = 3$, *Rodrigues' formula* gives an explicit representation of e^A for a skew-symmetric matrix A, allowing an efficient computation of e^A.

Theorem 2.1.38 (Rodrigues' Formula)**.** *Let $A \in \mathbb{R}^{3\times3}$ be a skew-symmetric matrix of the form*

$$A = \begin{pmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{pmatrix}$$

and $\theta := \sqrt{a^2 + b^2 + c^2}$. Then, for $A \neq 0_3$, Rodrigues' formula

$$e^A = I_3 + \frac{\sin\theta}{\theta}A + \frac{1 - \cos\theta}{\theta^2}A^2$$

holds [21, Lemma 18.6].

Proof. For a matrix A as given above, consider the matrix

$$\tilde{A} := \begin{pmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{pmatrix}.$$

The computation

$$A\tilde{A} = \begin{pmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{pmatrix} \begin{pmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{pmatrix} = \begin{pmatrix} -cab + bac & -cb^2 + b^2c & -cbc + bc^2 \\ ca^2 - a^2c & cab - abc & cac - ac^2 \\ -ba^2 + a^2b & -bab + ab^2 & -bac + abc \end{pmatrix} = 0_3$$

and an analogous computation for $\tilde{A}A$ show that $A\tilde{A} = \tilde{A}A = 0_3$. Furthermore, we can represent A^2 by $\tilde{A}$ and $\theta = \sqrt{a^2+b^2+c^2}$ via

$$\begin{aligned} A^2 &= \begin{pmatrix} -c^2-b^2 & ba & ca \\ ab & -c^2-a^2 & cb \\ ac & bc & -b^2-a^2 \end{pmatrix} \\ &= \begin{pmatrix} -c^2-b^2-a^2+a^2 & ab & ac \\ ab & -c^2-a^2-b^2+b^2 & bc \\ ac & bc & -b^2-a^2-c^2+c^2 \end{pmatrix} = \tilde{A} - \theta^2 I_3. \end{aligned}$$

Multiplying this equation by A gives

$$A^3 = A(\tilde{A} - \theta^2 I_3) = -\theta^2 A$$

since $A\tilde{A} = 0_3$. It follows $A^4 = -\theta^2 A^2$. From these considerations, we can deduce for any positive integer k by induction

$$\begin{aligned} A^{4k+1} &= \theta^{4k} A && \mid \cdot A \\ A^{4k+2} &= \theta^{4k} A^2 && \mid \cdot A \\ A^{4k+3} &= \theta^{4k} A^3 = \theta^{4k}(-\theta^2 A) = -\theta^{4k+2} A && \mid \cdot A \\ A^{4k+4} &= -\theta^{4k+2} A^2 \end{aligned}$$

by assuming $A^{4k} = -\theta^{4k-2}A^2$ and using the identity shown above for A^3.

As in the $\mathbb{R}^{2\times 2}$ case, we use the absolute convergence of e^A to rearrange the terms of its power series and the power series expansions of $\sin\theta$ and $\cos\theta$. So we conclude

$$\begin{aligned} e^A &= \sum_{k\geq 0} \frac{A^k}{k!} = I_3 + \frac{A}{1!} + \frac{A^2}{2!} + \frac{A^3}{3!} + \frac{A^4}{4!} + \frac{A^5}{5!} + \frac{A^6}{6!} + \dots \\ &= I_3 + \left(A + \frac{1}{3!}A^3 + \frac{1}{5!}A^5 + \dots\right) + \left(\frac{1}{2!}A^2 + \frac{1}{4!}A^4 + \frac{1}{6!}A^6 + \dots\right) \\ &= I_3 + \left(\theta^0 A + \frac{1}{3!}(-\theta^2 A) + \frac{1}{5!}\theta^4 A + \dots\right) + \left(\frac{1}{2!}\theta^0 A^2 + \frac{1}{4!}(-\theta^2 A^2) + \frac{1}{6!}\theta^4 A^2 + \dots\right) \\ &= I_3 + \frac{1}{\theta}\left(\theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - + \dots\right) A + \frac{1}{\theta^2}\left(1 - 1 + \frac{1}{2!}\theta^2 - \frac{1}{4!}\theta^4 + \frac{1}{6!}\theta^6 - + \dots\right) A^2 \\ &= I_3 + \frac{1}{\theta}\left(\sum_{k\geq 0}(-1)^k \frac{\theta^{2k+1}}{(2k+1)!}\right) A + \frac{1}{\theta^2}\left(1 - \sum_{k\geq 0}(-1)^k \frac{\theta^{2k}}{(2k)!}\right) A^2 \\ &= I_3 + \frac{\sin\theta}{\theta} A + \frac{1-\cos\theta}{\theta^2} A^2. \end{aligned}$$

This computation finishes the proof. □

2.2. Factor Graphs

The idea of a complex object being composed of a few simpler objects directly translates to a graph-theoretic representation of this object. Each considered part of the composite object as well as each (geometrical) relation between these parts is represented by a vertex in the object's graph model. The edges of the graph connect a vertex representing a part with all vertices representing the relations, that part is involved in. Since no two part-vertices and no two relation-vertices are connected by an edge, we receive a bipartite graph.

Example 2.2.1. Let's consider a simplified model of a house and its corresponding factor graph, constructed as described above. Let the house consist of four walls denoted as w_i for $i = 1, 2, 3, 4$, a roof r, and a door d. A graph displaying this house could look like in Figure 2.1.

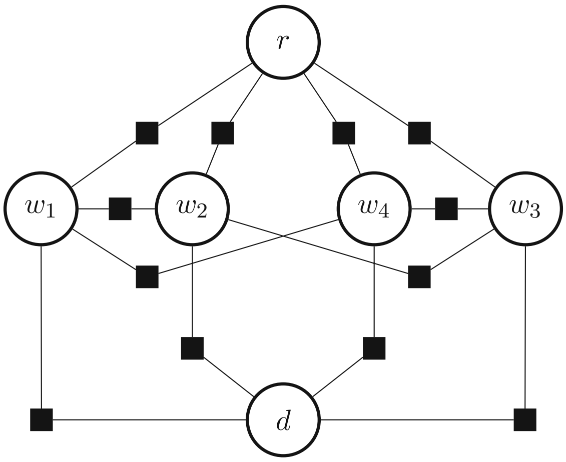


Figure 2.1.: Factor graph for a house viewed as a composite object

The round nodes represent the poses of the different parts of the house. The little black squares connecting two nodes v_1 and v_2 indicate the transformation from the pose of v_1 to the pose of v_2. These relative connections sufficiently describe the house as a composite object. If we want to describe the house in some greater context, i.e. in some world coordinate frame, we need some absolute conditions on a variable as well. In the pose estimation problem of Section 3.2, this is solved with an additional node, connected to only one node of a part of the composite object, called a *prior factor*.

This concept of representing a composite object with a graph is a key feature in our solution method of the pose estimation problem of Section 3.2, where the pose estimation scenario will be modeled with *factor graphs* as they are described in [36].

A factor graph consists of two disjoint sets of vertices, one referred to as *variables* and the other one referred to as *factors*. This leads to the formal definition of factor graphs.

Definition 2.2.2. Let $g \colon D \to R$ be a function with domain $D = A_1 \times A_2 \times \cdots \times A_n$ for some sets A_i with $i = 1, \ldots, n$ and any semiring R as codomain. Suppose that there exist a finite index set J and functions $f_j \colon D_j \to R$ for $j \in J$ with $D_j = A_{j_1} \times \cdots \times A_{j_k}$ and $j_1, \ldots, j_k \in \{1, \ldots, n\}$ pairwise different, such that the function g factorizes as

$$g(x_1, \ldots, x_n) = \prod_{j \in J} f_j(X_j) \tag{2.13}$$

with $X_j = (x_{j_1}, \dots, x_{j_k})$ where $f_j(X_j)$ stands for $f_j(x_{j_1}, \dots, x_{j_k})$. A *factor graph* for this factorization of g is a graph $\mathcal{F} = (V, E)$ with the set of vertices $V = V_v \dot\cup V_f$ and the set of edges E that is based on the factorization (2.13). The function g is called the *global function* of the factor graph $\mathcal{F}$.

The nodes in V_v are called *variable nodes* and represent the variables $x_1, \dots, x_n$ of $g(x_1, \dots, x_n)$, the nodes in V_f are called *factor nodes* and represent the factors f_j for $j \in J$ of the factorization of g. The set E contains no edge between two variable nodes respectively two factor nodes. The factor node for a factor f_j is connected to the variable node x_i by an edge $\{f_j, x_i\} \in E$ if and only if x_i is an argument of f_j.

So, every node in a factor graph is equipped with either a variable or a function. With the set of variable nodes (equipped with variables) V_v and the set of factor nodes (equipped with functions) V_f, we also write $\mathcal{F} = (V_v, V_f, E)$ for the factor graph $\mathcal{F}$.

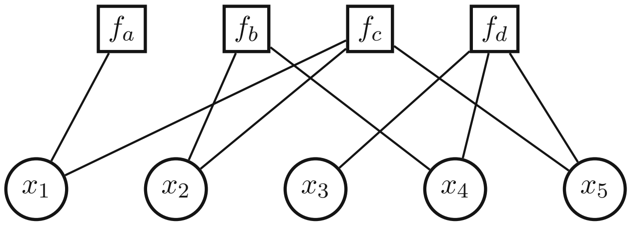


Figure 2.2.: Factor graph for the product $f_a(x_1)f_b(x_2, x_4)f_c(x_1, x_2, x_5)f_d(x_3, x_4, x_5)$.

Example 2.2.3. Let $g\colon \mathbb{Z}_3^5 \to \mathbb{Z}_3$ be a function with the factorization

$$g(x_1, x_2, x_3, x_4, x_5) = \underbrace{x_1^2}_{=:f_a(x_1)} \underbrace{(2x_2 + x_4 + 1)}_{=:f_b(x_2,x_4)} \underbrace{(x_1 + x_2^4 + x_5^3 + 2)}_{=:f_c(x_1,x_2,x_5)} \underbrace{(x_3 + x_4 + x_5)}_{=:f_d(x_3,x_4,x_5)}.$$

With the index set $J = \{a, b, c, d\}$ and the respective factors, we can draw the factor graph in Figure 2.2. If we change the split of the function g into factors to

$$g(x_1, x_2, x_3, x_4, x_5) = \underbrace{x_1^2(2x_2 + x_4 + 1)}_{=:f_e(x_1,x_2,x_4)} \underbrace{(x_1 + x_2^4 + x_5^3 + 2)}_{=:f_c(x_1,x_2,x_5)} \underbrace{(x_3 + x_4 + x_5)}_{=:f_d(x_3,x_4,x_5)},$$

we receive a different factor graph. So, the factor graph depends on the factorization of the function g.

We collect some simple properties of factor graphs.

Lemma 2.2.4. *Considering factor graphs as given in Definition 2.2.2, the following properties hold.*

(i) A factor graph is a bipartite graph.

(ii) For any simple bipartite graph $G = (V_1, V_2, E)$ of vertices $V = V_1 \dot\cup V_2$ and edges $E \subseteq V_1 \times V_2$, we can equip V_1 with variables and V_2 with factors such that G is a factor graph.

(iii) *For a function $g: D \to R$ with a factorization $\prod_{j \in J} f_j(X_j)$ as in (2.13) there exists exactly one factor graph $\mathcal{F}$ representing this factorization.*

Proof. (i) Follows directly from the split of the set of vertices in variable nodes and factor nodes and that no two factor nodes and no two variable nodes are connected in the graph.

(ii) For each node in $V_v := V_1$, we introduce a variable x_i together with a set of values A_i as the domain of this variable and for each node in $V_f := V_2$, we introduce a function f_j with a semiring R as common codomain. Let the function f_j depend on all variables, the respective node is connected to. The product of all these functions gives a function $g(x_1, \dots, x_n)$ with domain $A_1 \times \cdots \times A_n$ and codomain R. Thus, we have created a function g with a factorization according to Definition 2.2.2 that has the bipartite graph G as its factor graph.

(iii) This can be seen by introducing variable nodes for each variable x_i of g and factor nodes for each factor f_j of g and connecting them accordingly.

□

So we have seen, how we can turn functions into factor graphs and factor graphs into functions. This allows us to switch between factor graphs and functions easily.

2.2.1. Applications of Factor Graphs

Factor graphs offer a variety of applications. The most important one in our setting is probabilistic modeling with factor graphs, used for example in certain navigation and location tasks. Moreover, factor graphs are used for instance in robotics [10], coding theory [42], and artificial intelligence [63].

In the pose estimation problem of Section 3.2, we search for an assignment of the different parts of a truck to poses, that fits some prior information about relations between the parts and the observations of the parts best. We can translate this in probability theoretical terms: We search for the state X (a variable assignment) that is most likely under the given preconditions, assumptions, and observations Z, thus, we want to maximize the *posterior density* $p(X|Z)$ [13, Section 1.6]. The following lemma can rephrase this maximization problem.

Lemma 2.2.5. *The maximum a posteriori estimate $X^{\mathrm{MAP}} := \arg\max_X p(X|Z)$ is given by the joint probability function $p(X, Z)$ as*

$$X^{\mathrm{MAP}} = \arg\max_X p(X, Z)$$

Proof. Bayes' law states in this context that

$$p(X|Z) = \frac{p(Z|X)p(X)}{p(Z)}.$$

The preconditions, assumptions, and observations Z are given, hence the term $p(Z)$ is some constant, positive factor, not influencing the maximal argument. Therefore, maximizing the posterior $p(X|Z)$ translates as

$$\arg\max_X p(X|Z) = \arg\max_X \frac{p(Z|X)p(X)}{p(Z)} = \arg\max_X p(Z|X)p(X).$$

The term $p(Z|X)p(X)$ equals the joint probability $p(X, Z)$ according to the definition of conditional probability [6, Section 1.3]. Thus, the maximum a posteriori estimate X^{MAP} can be computed by maximizing the joint probability $p(X, Z)$. □

Remark 2.2.6. The term $p(Z|X)$ is also called likelihood (function) in statistics and can be denoted with $L(X|Z)$, indicating, that this is seen as a function of X and not as a function of Z [6, Section 6.3.1].

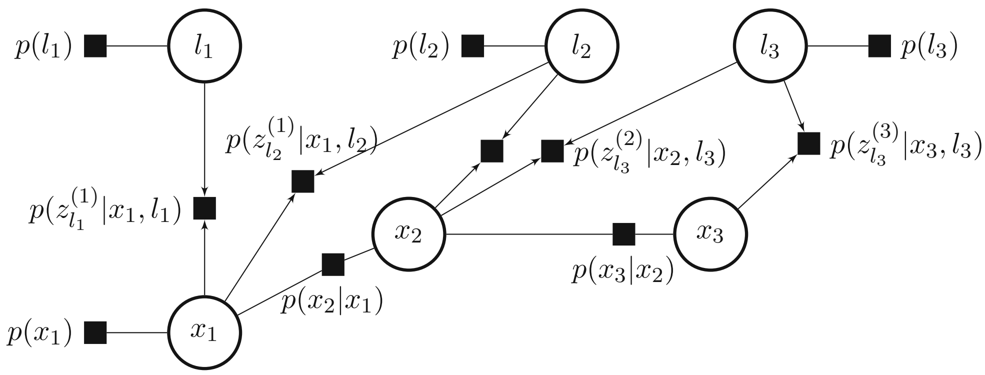


Figure 2.3.: Factor graph modeling a sensor observing landmarks over time

Example 2.2.7. Figure 2.3 shows a simple example of a factor graph $\mathcal{F}$, similar to [13, Chapter 1], modeling the situation of a sensor x moving past some landmarks l_1, l_2, l_3 of unknown poses, e.g. the three wheels on one side of a truck. The set of variable nodes V of the factor graph $\mathcal{F} = (V, F, E)$ is given by $V = \{x_1, x_2, x_3, l_1, l_2, l_3\}$ and the set F of the factor nodes is visualized by the black squares. The variables x_i for $i = 1, 2, 3$ denote the sensor positions at three consecutive time steps. If the sensor detects at time step i the landmark l_j, the observation $z_{l_j}^{(i)}$ is generated. These observations are seen as fixed given values, so there are no variable nodes for them. A factor graph is undirected, the arrows in the factor graph indicate that the poses of the landmarks and the sensor influence the observations, resulting in the probability densities $p(z_{l_j}^{(i)}|x_i, l_j)$.

With the definition of conditional probability [6, Section 1.3], the joint probability $p(x_1, x_2, x_3)$ is given by

$$p(x_1, x_2, x_3) = p(x_3|x_2, x_1)p(x_2|x_1)p(x_1) = p(x_3|x_2)p(x_2|x_1)p(x_1)$$

where the last equality follows if we assume that the *Markov property* holds. The Markov property states in this case that the pose of the sensor at time step 3 is just dependent on the pose of the sensor at time step 2 [6, Section 5.8.5]. Similarly, the joint probability $p(X, Z)$ of all sensor poses, landmarks, and observations is given by

$$p(X, Z) = p(x_3|x_2)p(x_2|x_1)p(x_1)p(l_1)p(l_2)p(l_3) \prod_{i,j} p(z_{l_j}^{(i)}|x_i, l_j),$$

which is exactly the global function $f(X)$ defined as the product of all factors of the factor graph $\mathcal{F}$.

A common application of factor graphs among many other applications in robotics is *Simultaneous Localization and Mapping* (SLAM) [13, Chapter 2]. It is similar to our pose estimation approach and the scene displayed in Figure 2.3 is an instance of a SLAM problem. A robot tries to orient itself in an unknown environment. Equipped with some sensors, the robot moves around and detects objects (here called *landmarks*) and their approximate relative pose to the robot. These measurements are represented by factor nodes between the variables x_t of the robot's pose at a time step t and variables l_i introduced for each landmark. Furthermore, odometry measurements – information about the movement (e.g. velocity and acceleration) of the robot [56, Section 5.4] – are taken into consideration. They translate to factors between the variables (x_1, x_2, and x_3 in the example above) denoting the position of the robot at certain time steps. Prior factors on landmarks or the robot's starting pose enable absolute location in the environment.

Factor graphs can be used in coding theory to model and decode certain codes [42]. The indicator function $I_C \colon A^n \to \{0, 1\}$ for a code C over the alphabet A, that maps a word $c \in A^n$ to 1 if c is a code word and to 0 otherwise, is interpreted as the global function of a factor graph. This factor graph allows efficient decoding, for instance with the sum-product algorithm.

In artificial intelligence, factor graphs can be used to model neural networks. Zhang et al. describe in [63] factor graph neural networks to model dependencies between different variables.

2.3. Optimization

Optimization is the task of minimizing or maximizing a function f under certain constraints [25]. Finding a minimum of the function $g(x) = x^2 - 3x + 2$ can be done by exploiting the properties of continuously differentiable functions. In many real-life scenarios, there is no explicit representation of a function that can easily be differentiated. Hence, we need optimization methods to get as close as possible to a local or global optimum.

In Section 2.3.1, an optimization technique is introduced. Levenberg-Marquardt optimization is used by GTSAM [11], the Python and C++ package employed in the solution of the pose estimation problem, to optimize factor graphs.

Many optimization problems are set in a vector space, e.g. in $\mathbb{R}^n$, where simple and intuitive optimization techniques are applicable. However, there are problems where a function f needs to be optimized on a (nonlinear) manifold M. There, we have to consider the issue of moving on the manifold efficiently, which can be solved by retractions as described in Section 2.3.2 and applied to SO_2 and SO_3 in Section 2.3.3. Section 2.3.4 deals with optimization in SE_3.

Furthermore, factor graphs can be optimized as well if they are equipped with some probability structure. Section 2.3.5 deals with the optimization of factor graphs.

2.3.1. Levenberg-Marquardt Optimization

In Section 2.1, we have established the necessary geometric background for optimizing the pose of an object. Now, we will briefly discuss the actual scheme for nonlinear optimization used in this project. In particular, in Section 3.2 we describe the part of the project that deals with pose estimation with factor graphs using the Python and C++ library GT-SAM [11] which provides implementations of various optimization algorithms. We use the Levenberg-Marquardt algorithm [45] that can be seen as a combination of gradient descent and Gauss-Newton optimization [13, Section 2.5.3]. Marquardt describes this algorithm and the theoretical background in [45].

Gradient descent, Gauss-Newton, and Levenberg-Marquardt optimization are iterative algorithms to optimize (w.l.o.g. minimize) a function g, starting from an initial estimate $x^{(0)}$ and updating an estimate $x^{(t)}$ by the rule

$$x^{(t+1)} = x^{(t)} + \alpha\delta^{(t)} \tag{2.14}$$

with $\alpha \in \mathbb{R}$ and some update step $\delta^{(t)}$ depending on the method. The scaling factor α is chosen concerning the specific use case, ensuring that the convergence speed is fast enough on the one hand and that the updates are safe, i.e. the steps are not too big to risk divergence, on the other hand. This process continues until the solutions $x^{(t)}$ converge, i.e. $\delta^{(t)}$ is smaller than some threshold. The following considerations are based on [13, Section 2.5] where these three methods are discussed.

Gradient descent is a simple nonlinear optimization technique to minimize a differentiable function g by taking steps in the direction of the steepest descent, given by the negative gradient $-\operatorname{grad} g(x^{(t)})$ of the current guess $x^{(t)}$. Thus, the update rule for gradient descent is given by

$$x^{(t+1)} = x^{(t)} + \alpha\delta^{(t)}_{\mathrm{GD}} = x^{(t)} - \alpha \operatorname{grad} g(x^{(t)}).$$

Gradient descent has a slow convergence speed close to the minimum.

The Gauss-Newton method is a technique to minimize a sum of squared continuously differentiable functions. This least squares problem is given by

$$\min_{x\in\mathbb{R}^n} \sum_{i=1}^{m} (g_i(x))^2 \tag{2.15}$$

with the function $g = (g_1, \dots, g_m)$ and $g_i \colon \mathbb{R}^n \to \mathbb{R}$ for each $i = 1, \dots, m$. The idea is to approximate g by a Taylor series of first order [45], thus

$$g(x) \approx g(x_0) + \sum_{j=1}^{n} \operatorname{grad}_j g(x_0)(x - x_0)_j = g(x_0) + J(x_0)(x - x_0) \tag{2.16}$$

where J denotes the well-known Jacobian matrix

$$J(x) := \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(x) & \cdots & \frac{\partial g_1}{\partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial x_1}(x) & \cdots & \frac{\partial g_m}{\partial x_n}(x) \end{pmatrix}.$$

Minimizing as in (2.15) with this linearization of g instead of g itself can be done by taking the gradient of the squared norm of the right-hand side of (2.16) and setting it to zero. This results in the equation

$$\operatorname{grad} \|g(x_0) + J(x_0)(x - x_0)\|^2 = 0 \quad \Rightarrow \quad J^T(x_0)\left(J(x_0)(x - x_0) + g(x_0)\right) = 0.$$

Here, we take the linearization point $x^{(t)}$ and evaluate the Taylor series expansion at the point $x^{(t+1)} = x^{(t)} + \delta_{\text{GN}}^{(t)}$ for some unknown $\delta_{\text{GN}}^{(t)}$. Therefore, the update step $\delta_{\text{GN}}^{(t)}$ is implicitly defined by the equation

$$J^T(x^{(t)})J(x^{(t)})\delta_{\text{GN}}^{(t)} = -J^T(x^{(t)})g(x^{(t)}). \tag{2.17}$$

Thus, the Gauss-Newton update rule is

$$x^{(t+1)} = x^{(t)} - \alpha \underbrace{\left(J^T(x^{(t)})J(x^{(t)})\right)^{-1} J^T(x^{(t)})g(x^{(t)})}_{=\delta_{\text{GN}}^{(t)}},$$

where α is again some scaling factor [45]. This method can have poor convergence behavior if g is not nearly quadratic [13, Section 2.5.2].

In the Levenberg-Marquardt optimization, the Gauss-Newton update is modified by introducing a real factor $\lambda \geq 0$ in (2.17) to get the equation

$$\left(J^T(x^{(t)})J(x^{(t)}) + \lambda \operatorname{diag}\left(J^T(x^{(t)})J(x^{(t)})\right)\right)\delta_{\text{LM}}^{(t)} = -J^T(x^{(t)})g(x^{(t)}). \tag{2.18}$$

where $\operatorname{diag}(A)$ denotes the diagonal matrix $\operatorname{diag}(a_{11}, a_{22}, \ldots, a_{ll})$ consisting of the entries of the diagonal of a quadratic matrix $A \in \mathbb{R}^{l \times l}$. This results in larger steps towards the direction of the steepest descent if the gradient is small. If the gradient is big, the steps are smaller to reduce the risk of divergence. Another modification can be made by rejecting steps that lead to an increase in the value that should be minimized. If a step is rejected, the value of λ is increased (e.g. in [13, Algorithm 2.1] λ is multiplied by 10) and the last step is retaken with the new λ. If a step is accepted, λ is diminished again (e.g. λ is divided by 10). Marquardt suggests in [45] that this algorithm combines the advantage of gradient descent that it converges from rather far away, and the advantage of the Gauss-Newton method that it converges rapidly when we are already close to a solution.

2.3.2. Optimization on Manifolds

Optimization methods like gradient descent, Gauss-Newton, and Levenberg-Marquardt rely on the update rule (2.14) to gradually improve some estimate $x^{(t)}$. This works well in vector spaces since $\delta^{(t)}$ is rather easy to define and compute. On a manifold M, it is not that easy to take a step and still stay on the manifold, thus, resulting in a new valid estimate $x^{(t+1)} \in M$. Absil, Mahony, and Sepulchre describe in [1, Section 4.1] a method to take a step in a vector space and consequently bring the resulting point back onto the manifold. A function that achieves this mapping from a vector space back to the manifold is called a *retraction*.

In a topological sense, a retraction r is a continuous function from a topological space X to a subspace Y of X with $r(y) = y$ for all $y \in Y$ [31, Chapter 0]. Clearly, r is idempotent, i.e. $r \circ r = r$, and therefore the topological analog to a projection in linear algebra. Here, we need retractions on manifolds. The idea stays the same: A retraction brings a point from the tangent space onto the manifold. The formal definition of retractions on manifolds given in [1, Definition 4.1.1] is as follows.

Definition 2.3.1. Let M be a manifold. A *retraction* on M is a smooth function

$$\mathcal{R} \colon TM \to M$$

such that the following properties hold for the restriction $\mathcal{R}_x := \mathcal{R}|_{T_xM}$ for every $x \in M$ (i.e. in $\mathcal{R}_x$, we drop the first part of a pair $(x, \xi) \in TM$).

(i) Let 0_x denote the zero vector of the vector space T_xM, then $\mathcal{R}_x(0_x) = x$.

(ii) The differential $D\mathcal{R}_x(0_x)[\cdot]$ is equal to the identity id_{T_xM} on the tangent space T_xM.

These conditions ensure that the gradient at x is preserved under $\mathcal{R}_x$ [1, Section 4.1]. This can be visualized as in Figure 2.4.

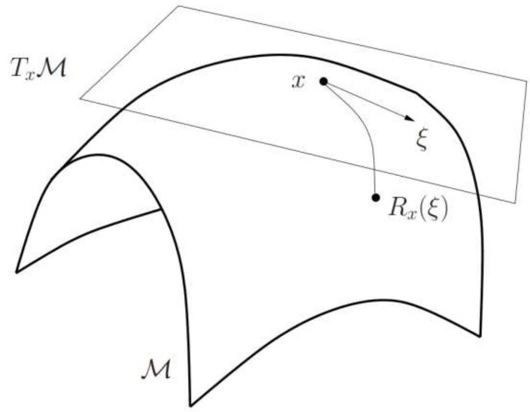


Figure 2.4.: Visualization of a retraction taken from [1, Figure 4.1].

A deeper examination of differential geometry and the theory behind Definition 2.3.1 would be beyond the scope of this work. We will just make a brief remark and provide references to the corresponding sources.

Remark 2.3.2. The zero element 0_x of T_xM in condition (i) of Definition 2.3.1 is according to (2.5) the pre-image of $0_n \in \mathbb{R}^n$ under some $\vartheta_\varphi^{x;M}$ of Lemma 2.1.23. Thus, the tangent vector 0_x belongs to some (and therefore all) curve(s) γ with $(\varphi \circ \gamma)'(0) = 0_n$.

Part (ii) of Definition 2.3.1 is often referred to as the *local rigidity condition* [1, Section 4.1]. The differential $D\mathcal{R}_x(0_x)[\cdot]$ is a function that takes a tangent vector η of the domain of $\mathcal{R}_x$ at 0_x and maps it to the tangent vector $D\mathcal{R}_x(0_x)[\eta]$ in the tangent space of the codomain of $\mathcal{R}_x$. So to be precise, $D\mathcal{R}_x(0_x)[\cdot]$ is a function $T_{0_x}(T_xM) \to T_xM$ and is only equal to the identity on T_xM if we identify $T_{0_x}(T_xM)$ and T_xM as described in [1, Section 3.5.2]. See [1, Section 3.5.6] for a more detailed description of this differential.

Using a retraction $\mathcal{R}$ in an optimization problem on a manifold M offers two advantages [1, Section 4.1]. First, $\mathcal{R}$ brings elements of tangent spaces T_xM back onto the manifold M. So starting from an $x^{(t)} \in M$ we can easily take steps in the tangent space $T_{x^{(t)}}M$ since it is a vector space and, furthermore, closely related to $\mathbb{R}^n$ due to the map defined in Lemma 2.1.23. The retraction $\mathcal{R}$ maps this new point in $T_{x^{(t)}}M$ onto the manifold M and therefore delivers a new estimate $x^{(t+1)}$.

Second, in an optimization problem, there is usually a cost function $c\colon M \to \mathbb{R}$ on the manifold M that should be minimized. But when the steps are taken in the tangent space, the cost function needs to be lifted to T_xM as well. The function

$$c^{\mathcal{R}} := c \circ \mathcal{R}\colon \quad TM \;\to\; \mathbb{R}$$

lifts the cost function to the tangent bundle. For a point $x \in M$, we denote the restriction of $c^{\mathcal{R}}$ to the tangent space T_xM by

$$c_x^{\mathcal{R}} := c^{\mathcal{R}}|_{T_xM} = c \circ \mathcal{R}_x\colon \quad T_xM \;\to\; \mathbb{R}$$

which is a function from a vector space to $\mathbb{R}$. Due to the chain rule [1, Section 1.3] and condition (ii) of Definition 2.3.1, the differential of $c_x^{\mathcal{R}}$ computes as $Dc_x^{\mathcal{R}}(0_x) = Dc(x)$.

Remark 2.3.3. Absil, Mahony, and Sepulchre state in [1, (4.4)] that even

$$\operatorname{grad} c_x^{\mathcal{R}}(0_x) = \operatorname{grad} c(x)$$

holds if the manifold M is endowed with a *Riemannian metric* (which is the case for SO_n). This would again go beyond the scope of this work. See [1, Section 3.6] for more information about Riemannian metrics.

Example 2.3.4. There are several retractions for the special orthogonal group SO_n. We will focus on a retraction that uses the exponential map according to [2, Example 2]. For a rotation $R \in \mathrm{SO}_n$ a retraction is given by

$$\mathcal{R}_R\colon T_R\mathrm{SO}_n \to \mathrm{SO}_n\colon \eta \mapsto Re^{R^{-1}\eta}. \tag{2.19}$$

The tangent space $T_R\mathrm{SO}_n$ is given by (2.9), so η is of the form $R\hat{\xi}$ for some $\hat{\xi} \in \mathrm{Skew}_n$ (the hat-operator is used for consistent notation with the next Chapter, see Definition 2.3.5). Hence, $\mathcal{R}_R(\eta)$ reduces to

$$\mathcal{R}_R(\eta) = Re^{R^{-1}\eta} = Re^{R^{-1}R\hat{\xi}} = Re^{\hat{\xi}}.$$

The exponential $e^{\hat{\xi}}$ of the skew-symmetric matrix $\hat{\xi}$ is according to Theorem 2.1.37 in SO_n. Thus, the matrix product $Re^{\hat{\xi}}$ is in SO_n and the function $\mathcal{R}_R$ well-defined.

To get a retraction in the sense of Definition 2.3.1, let $\mathcal{R}$ be the function that maps a pair $(R, \eta) \in T\mathrm{SO}_n$ to $\mathcal{R}_R(\eta)$. Condition (i) holds due to the computation

$$\mathcal{R}_R(0_R) = Re^{R^{-1}0_R} = Re^{R^{-1}R0_n} = Re^{0_n} = RI_n = R$$

with the zero element 0_R of the tangent space $T_R\mathrm{SO}_n$ and the zero element 0_n of the vector space Skew_n. For the second part of the definition, we have to compute the differential $D\mathcal{R}(0_R)[\cdot]$. Najfeld and Havel give in [48, Section 1.2] the general definition

$$De^{tA}[V] = \lim_{h\to 0} \frac{1}{h}\left(e^{t(A+hV)} - e^{tA}\right)$$

and an explicit way to calculate the directional derivative of the matrix exponential e^{tA} in the direction V by

$$De^{tA}[V] = \int_0^t e^{(t-\tau)A} V e^{\tau A} d\tau.$$

Here, with $\eta = R\hat{\xi}_\eta \in T_R\mathrm{SO}_n$ and $\zeta = R\hat{\xi}_\zeta \in T_R\mathrm{SO}_n$, this translates to

$$\begin{aligned}
D\mathcal{R}_R(0_R)[\zeta] &= \lim_{h\to 0} \frac{1}{h}\left(Re^{R^{-1}(\eta+h\zeta)} - Re^{R^{-1}\eta}\right)\Big|_{\eta=0_R} \\
&= \lim_{h\to 0} \frac{1}{h}\left(Re^{R^{-1}(R\hat{\xi}_\eta+hR\hat{\xi}_\zeta)} - Re^{R^{-1}R\hat{\xi}_\eta}\right)\Big|_{R\hat{\xi}_\eta=0_n} \\
&= R\lim_{h\to 0} \frac{1}{h}\left(e^{\hat{\xi}_\eta+h\hat{\xi}_\zeta} - e^{\hat{\xi}_\eta}\right)\Big|_{\hat{\xi}_\eta=0_n} \\
&= R\, De^{0_n}[\hat{\xi}_\zeta] \\
&= R\int_0^1 e^{(1-\tau)0_n}\hat{\xi}_\zeta e^{\tau 0_n} d\tau \\
&= R\int_0^1 I_n\hat{\xi}_\zeta I_n d\tau = R\hat{\xi}_\zeta = \zeta.
\end{aligned}$$

This shows that $D\mathcal{R}_R(0_R)[\cdot]$ is the identity on $T_R\mathrm{SO}_n$. Thus, $\mathcal{R}$ is a retraction.

2.3.3. Optimization on SO_2 and SO_3

After the preparations made in Section 2.1, we can formalize *small steps* on a rotation manifold. Instead of adding an arbitrary, small matrix, we take small rotations in some natural representation, translate that into a matrix, use the exponential map for quadratic matrices as a retraction to get a rotation matrix, and then compose it with the base rotation.

In SO_2, this process is quite straightforward as described in [13, Section 6.1.5]. The special orthogonal group SO_2 is according to Theorem 2.1.28 a ($\frac{2(2-1)}{2} = 1$)-dimensional manifold. The tangent space $T_B\mathrm{SO}_2$ at some point $B \in \mathrm{SO}_2$, given by (2.9), is a 1-dimensional vector space. A planar rotation around the origin can be uniquely defined by a single number $\xi \in \mathbb{R}$. In [13, Section 6.1.3], Dellaert and Kaess refer to this number and its SO_3-equivalent (see below), that represents a step in an incremental rotation, in this context as *local coordinates* or a *local parametrization*. Here, we follow their proposed way of optimizing on the rotation manifolds SO_2 and SO_3.

The special orthogonal group SO_3 is a ($\frac{3(3-1)}{2} = 3$)-dimensional manifold with the tangent space $T_B\mathrm{SO}_3 = \{BS \mid S \in \mathrm{Skew}_3\}$. As seen in Section 2.1.1, a rotation in $\mathbb{R}^3$ can be represented by an axis $a \in S^2$ and an angle $\alpha \in \mathbb{R}$. Since $\|a\| = 1$, the rotation is also uniquely defined by the local coordinates $\xi := \alpha a \in \mathbb{R}^3$.

To use the retraction $\mathcal{R}$ given in example 2.3.4 for SO_2 and SO_3, we need tangent vectors of these groups, i.e. skew-symmetric matrices, instead of the local coordinates stated above. In [43, Section 3], this lifting from local coordinates in $\mathbb{R}^3$ to skew-symmetric matrices is described by the matrix $\hat{\xi} \in \mathrm{Skew}_3$ that is the cross-product matrix of ξ, i.e. $\hat{\xi}c = \xi \times c$ for any $c \in \mathbb{R}^3$. The $\mathbb{R}^2$ equivalent is the function that maps a vector $c = (c_1, c_2)^T$ to its orthogonal vector $(-c_2, c_1)^T$. The matrix of this linear mapping is again skew-symmetric. The hat-operator defines this mapping from local coordinates to cross-product matrices.

Definition 2.3.5. The hat-operator for planar rotations is the function defined as

$$\hat{\cdot}\colon\ \mathbb{R} \to \mathbb{R}^{2\times 2}\colon\ \ \xi \mapsto \hat{\xi} := \begin{pmatrix} 0 & -\xi \\ \xi & 0 \end{pmatrix}.$$

The hat-operator for rotations in $\mathbb{R}^3$ is the function defined as

$$\hat{\cdot}\colon\ \mathbb{R}^3 \to \mathbb{R}^{3\times 3}\colon\ \ \xi = \begin{pmatrix} \xi_x \\ \xi_y \\ \xi_z \end{pmatrix} \mapsto \hat{\xi} := \begin{pmatrix} 0 & -\xi_z & \xi_y \\ \xi_z & 0 & -\xi_x \\ -\xi_y & \xi_x & 0 \end{pmatrix}.$$

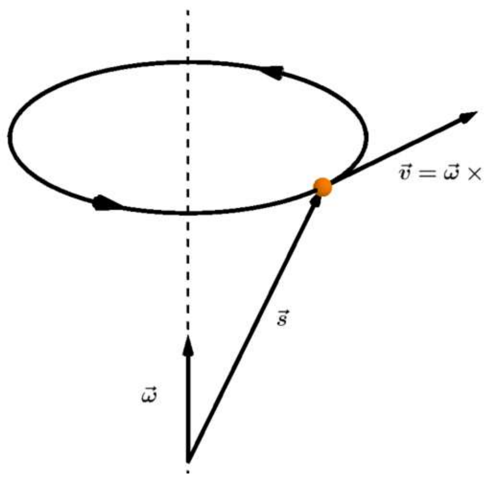


Figure 2.5.: Visualization of $\vec{\omega}$ and $\vec{v}$ as in [55, Figure 7-4].

We provide a physical interpretation of why it is reasonable to use the hat-operator here. The velocity vector $\vec{v}$ of a point that rotates around an axis is given by the cross product $\vec{v} = \vec{\omega} \times \vec{s}$ of the angular velocity $\vec{\omega}$ and the position $\vec{s}$ of the object [38, Chapter VI §31]. This is visualized in Figure 2.5. The angular velocity $\vec{\omega}$ refers to the local coordinates $\xi \in \mathbb{R}^3$ defined by the product αa of the angle α and the axis a. The tangent vectors in the tangent space $T_B\mathrm{SO}_3$ can be seen as the speed vectors of any curves at the point B [40, Section 3.5]. Thus, the velocity vector $\vec{v}$ refers to the tangent vectors in $T_B\mathrm{SO}_3$ given by the product of B with $\hat{\xi}$.

So, when optimizing in SO_2, the hat-operator transforms the rotation angle ξ into a skew-symmetric (2×2) matrix $\hat{\xi}$. Starting at a rotation R_θ, defined by the angle θ, the retraction can be used to update this rotation by a local parametrization ξ as

$$\begin{aligned}\mathcal{R}_{R_\theta}(R_\theta\hat{\xi}) &= R_\theta e^{R_\theta^{-1}R_\theta\hat{\xi}} = R_\theta e^{\hat{\xi}}\\ &\overset{2.1.36}{=} \begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}\cos\xi & -\sin\xi\\ \sin\xi & \cos\xi\end{pmatrix}\\ &\overset{2.1.9}{=} \begin{pmatrix}\cos(\theta+\xi) & -\sin(\theta+\xi)\\ \sin(\theta+\xi) & \cos(\theta+\xi)\end{pmatrix}.\end{aligned}$$

We introduce a notation for updating a base rotation R_0 by local coordinates ξ.

Definition 2.3.6. Given a rotation R_0 in SO_2 or SO_3 and local coordinates ξ in $\mathbb{R}$ respectively $\mathbb{R}^3$ as described above, we can define the *local update* $\oplus$ of the rotation R_0 by the local coordinates ξ as

$$R_0 \oplus \xi := \mathcal{R}_{R_0}(R_0\hat{\xi}).$$

Note, that the matrix $e^{\hat{\xi}}$ for $\xi = \alpha a$, that appears in a local update in SO_3, has the explicit representation

$$e^{\hat{\xi}} = I_3 + \frac{\sin\alpha}{\alpha}\hat{\xi} + \frac{1-\cos\alpha}{\alpha^2}\hat{\xi}^2$$

according to Rodrigues' formula 2.1.38 and since

$$\|\xi\| = \sqrt{(\alpha a_x)^2 + (\alpha a_y)^2 + (\alpha a_z)^2} = |\alpha|\|a\| = |\alpha|$$

holds. Hence, this retraction can be computed efficiently in SO_2 and SO_3.

Example 2.3.7. Let us examine how a simplified version of the pose estimation problem of Section 3.2 would translate to an optimization problem on SO_3. For now, we just want to optimize for the orientation, so assume there is a camera on a pole at a fixed, known position on one side of a truck. The camera can only rotate itself but cannot change its position. It can detect the three wheels and the loading edge on the side of the truck (the view of the camera might look like in Figure 3.12). Each detection generates a vector $z = (z^{(e)}, z^{(w_1)}, z^{(w_2)}, z^{(w_3)})^T \in \mathbb{R}^{12}$ where $z^{(e)}, z^{(w_1)}, z^{(w_2)}, z^{(w_3)} \in \mathbb{R}^3$ denote the measurements of the loading edge and the three wheels as the direction vectors from the camera to the respective parts of the truck (simplified as points) in the current camera frame (i.e. the x-axis points in the direction the camera is looking at, the z-axis points upwards, and the y-axis points to the left, such that all axes together create a right-handed coordinate system).

Assume that a rough model of the relations between the different parts of the truck is known and that we have access to an estimation function

$$h\colon \mathrm{SO}_3 \to \mathbb{R}^{12}.$$

For a given rotation $R \in \mathrm{SO}_3$, this function h estimates the corresponding measurement $z_R \in \mathbb{R}^{12}$ based on the assumed model of the truck. This function is not surjective, in particular, it does not have an inverse function. Furthermore, h might be way too complicated to invert it even if the codomain were restricted to $h(\mathrm{SO}_3)$ and h were injective. On top of that, the measurements obtained by the camera are not exact but noisy and the model of the truck is not exact as well. Thus, a measurement obtained by the camera does not have to be an element of $h(\mathrm{SO}_3)$. For these various reasons, getting the rotation of the camera from a given measurement $z \in \mathbb{R}^{12}$ is a challenging task that can be tackled by optimizing

$$R^* = \underset{R \in \mathrm{SO}_3}{\arg\min} \|h(R) - z\|^2.$$

In the following, we will examine, how $\|h(R) - z\|^2$ can be optimized similar to [13, Section 6.1.3] with Levenberg-Marquardt optimization discussed in Section 2.3.1.

Given an estimation function $h\colon \mathrm{SO}_3 \to \mathbb{R}^n$, the goal is to find

$$R^* = \underset{R \in \mathrm{SO}_3}{\arg\min} \|h(R) - z\|^2.$$

Methods like gradient descent and Levenberg-Marquardt start at some initial estimate $R^{(0)}$ and iteratively take steps $\delta^{(t)}$ towards a new estimate $R^{(t+1)}$ for $t \geq 0$ by minimizing

$$\delta^{(t)} = \underset{\xi \in \mathbb{R}^3}{\arg\min} \|h(R^{(t)} \oplus \xi) - z\|^2$$

to get $R^{(t+1)} = R^{(t)} \oplus \delta^{(t)}$. Let's define $g_R(\xi) := h(R \oplus \xi)$. An approximation of the Jacobian G_R of g_R can be computed through, for instance, numerical differentiation or automated differentiation. As in (2.16), we can approximate $g_R(\xi)$ by

$$g_R(\xi) \approx g_R(0) + G_R(0)(\xi - 0) = h(R) + G_R\xi.$$

Therefore, we can define the update step $\delta^{(t)}_{\mathrm{LM}}$ as in (2.18).

2.3.4. Optimization in SE_3

Until now, we just considered optimization of rotations. In the pose estimation problem of Section 3.2, we search for optimal poses defined by elements of SE_3 rather than just optimal rotations. This is just a simple generalization from SO_3 to SE_3, analogously to [13, Sections 6.2.2 and 6.2.3].

Local coordinates ξ in SO_3 were given by the product of an axis $a \in S^2$ and an angle $\alpha \in \mathbb{R}$. For local coordinates in SE_3, we simply expand the local coordinates of SO_3 by a vector $v \in \mathbb{R}^3$ modeling translations in $\mathbb{R}^3$. Thus, we receive 6-dimensional local coordinates ξ of the 6-dimensional manifold SE_3 (see Theorem 2.1.29).

The tangent space $T_{(R,t)}\mathrm{SE}_3$ of the product manifold $\mathrm{SE}_3 = \mathrm{SO}_3 \times \mathbb{R}^3$ is according to [59, solution to Problem 8.7] isomorphic to the product of the tangent spaces $T_R\mathrm{SO}_3 \times T_t\mathbb{R}^3$. Thus, we view tangent vectors to SE_3 as a pair $(\eta, s) \in T_R\mathrm{SO}_3 \times T_t\mathbb{R}^3$ or a matrix

$$\begin{pmatrix} \eta & s \\ 0 & 0 \end{pmatrix} \in \mathbb{R}^{4\times 4}$$

if necessary, with $\eta \in R\mathrm{Skew}_3$ according to (2.9) and $s \in \mathbb{R}^3$.

Therefore, we can define a retraction for the special Euclidean group SE_3 as in Example 2.3.4. For $(R, t) \in \mathrm{SE}_3$ and $(\eta, s) = (R\hat{\xi}_{\mathrm{SO}_3}, s) \in T_R\mathrm{SO}_3 \times \mathbb{R}^3$, we define the retraction as

$$\mathcal{R}_{(R,t)} \colon T_{(R,t)}\mathrm{SE}_3 \to \mathrm{SE}_3 \colon (\eta, s) \mapsto \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} e^{R^{-1}\eta} & R^{-1}s \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} Re^{R^{-1}\eta} & s+t \\ 0 & 1 \end{pmatrix}. \tag{2.20}$$

We see that $\mathcal{R}_{(R,t)}(\eta, s) = (\mathcal{R}_R^{\mathrm{SO}_3}(\eta), s+t) \in \mathrm{SE}_3$ with the retraction $\mathcal{R}_R^{\mathrm{SO}_3}$ of Example 2.3.4. At the zero element $0_{(R,t)}$ of $T_{(R,t)}\mathrm{SE}_3$, we compute

$$\mathcal{R}_{(R,t)}(0_{(R,t)}) = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} e^{R^{-1}0_R} & R^{-1}0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} I_4 = (R, t).$$

With the computation of the differential $D\mathcal{R}_R^{\mathrm{SO}_3}(0_R)[\zeta]$ in Example 2.3.4 we conclude for some tangent vector $(\zeta, u) = (R\hat{\xi}_\zeta, u) \in T_{(R,t)}\mathrm{SE}_3$

$$\begin{aligned} D\mathcal{R}_{(R,t)}(0_{(R,t)})[(\zeta, u)] &= \lim_{h\to 0} \frac{1}{h} \left(\mathcal{R}_{(R,t)}(\eta + h\zeta, s + hu) - \mathcal{R}_{(R,t)}(\eta, s)\right)\Big|_{(\eta,s)=(0_R,0_3)} \\ &= \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{\xi}_\zeta & \lim_{h\to 0} \frac{1}{h} R^{-1}hu \\ 0 & \lim_{h\to 0} \frac{1}{h} \end{pmatrix} \\ &= \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{\xi}_\zeta & R^{-1}u \\ 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} R\hat{\xi}_\zeta & RR^{-1}u + 0t \\ 0 & 0 \end{pmatrix} \\ &= (\zeta, u). \end{aligned}$$

This proves that the function defined in (2.20) is indeed a retraction for SE_3 in the sense of Definition 2.3.1. We can use this retraction to define local updates in SE_3.

Definition 2.3.8. For $T_0 = (R_0, t_0) \in \mathrm{SE}_3$ and local coordinates

$$\xi = \begin{pmatrix} \omega \\ v \end{pmatrix} \in \mathbb{R}^6$$

with $\omega = \alpha a \in \mathbb{R}^3$ and $v \in \mathbb{R}^3$ as described above, we define the *local update* $\oplus$ of T_0 by the local coordinates ξ as

$$T_0 \oplus \xi := \mathcal{R}_{T_0}(R_0\hat{\omega}, v)$$

Now we can proceed as in Section 2.3.3. Let $h \colon \mathrm{SE}_3 \to \mathbb{R}^n$ be an estimation function. To find

$$T^* = \operatorname*{arg\,min}_{T\in\mathrm{SE}_3} \|h(T) - z\|^2 \tag{2.21}$$

for some measurement z, we compute the step

$$\delta^{(t)} = \operatorname*{arg\,min}_{\xi\in\mathbb{R}^6} \|h(T^{(t)} \oplus \xi) - z\|^2$$

to update the current estimate $T^{(t)}$ as $T^{(t+1)} = T^{(t)} \oplus \delta^{(t)}$.

2.3.5. Optimizing a Factor Graph

Now that we know how to optimize for poses in SE_3, we examine how to optimize a factor graph. In the pose estimation problem described in Section 3.2, we will combine these optimization approaches by optimizing for poses of different objects in SE_3 that are linked by a factor graph, simultaneously. This brief introduction to factor graph optimization is based on [13, Sections 1.6, 1.7, and 2.2].

Let $\mathcal{F} = (V, F, E)$ be a factor graph with variables V, factors F and edges $E \subseteq V \times F$. Let furthermore $f(X) = \prod_j f_j(X_j)$ be the global function of the factor graph, defined by the product of the factors $f_j \in F$. In general, we can pose the question, which variable assignment $X^{\max}$ maximizes the global function f, i.e.

$$X^{\max} = \arg\max_X f(X) = \arg\max_X \prod_j f_j(X_j).$$

Depending on the structure of $\mathcal{F}$ and the factors f_j, we can make certain reductions to the problem.

Let the factors of $\mathcal{F}$ denote probability densities as in Example 2.2.7. Thus, the global function $f(X)$ of this factor graph is some joint probability density $p(X, Z)$ for unknown states X and given observations and assumptions Z. Lemma 2.2.5 states that maximizing the joint probability density $p(X, Z)$ gives the same argument X as maximizing the posterior density $p(X|Z)$. The maximum a posteriori estimate X^{MAP}, in turn, is what we are looking for if we search for the state X that is most likely under certain preconditions, assumptions, and observations Z.

Let's assume that the factors $f_j(X_j)$ are probability densities of some multivariate Gaussian distribution. As a reminder, the multivariate Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ is given by the density

$$f(x) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)},$$

where $x, \mu \in \mathbb{R}^n$ and the covariance matrix $\Sigma \in \mathbb{R}^{n \times n}$ [30, Chapter I, Section 5]. In particular, the factors are proportional

$$f_j(X_j) \propto e^{-\frac{1}{2}\left(h(X_j)-z_j)^T \Sigma_j^{-1}(h(X_j)-z_j)\right)} = e^{-\frac{1}{2}\|h(X_j)-z_j)\|^2_{\Sigma_j}} \tag{2.22}$$

with the notation $(h(X_j) - z_j)^T \Sigma_j^{-1}(h(X_j) - z_j) = \|h(X_j) - z_j)\|^2_{\Sigma_j}$ from [13, Section 2.2]. The measurements z_j and the estimation function h are used as in Section 2.3.3. Thus, the error $h(X_j) - z_j$ of the estimation function h regarding the measurement z_j is normally distributed around the mean 0 with the covariance matrix Σ_j.

With the considerations above, we can compute

$$\begin{aligned}
X^{\text{MAP}} &\overset{2.2.5}{=} \arg\max_X p(X, Z) \\
&= \arg\max_X \prod_j f_j(X_j) \\
&\overset{(1)}{=} \arg\max_X \prod_j e^{-\frac{1}{2}\|h_j(X_j)-z_j\|^2_{\Sigma_j}} \\
&\overset{(2)}{=} \arg\max_X \log\left(\prod_j e^{-\frac{1}{2}\|h_j(X_j)-z_j\|^2_{\Sigma_j}}\right) \\
&= \arg\max_X -\frac{1}{2}\sum_j \|h_j(X_j) - z_j\|^2_{\Sigma_j} \\
&\overset{(3)}{=} \arg\min_X \sum_j \|h_j(X_j) - z_j\|^2_{\Sigma_j}.
\end{aligned}$$

The equality in (1) holds since the maximal argument does not change if the objective function is multiplied by a constant positive factor. For the equality in (2), we used that the natural logarithm is a strictly increasing function and, thus, preserves the maximal argument. To obtain the equality in (3), we used that the arg max of some function g multiplied with a negative constant factor changes to the arg min of g.

For X_j in some manifold, the resulting optimization problem

$$X^{\text{MAP}} = \arg\min_X \sum_j \|h_j(X_j) - z_j\|^2_{\Sigma_j} \tag{2.23}$$

can be solved with the methods discussed in Sections 2.3.1 and 2.3.4 similar to the optimization problem (2.21).

Remark 2.3.9. The later used library GTSAM [11] uses this optimization scheme for optimization on manifolds with factor graphs as documented in [12].

2.4. Geometric Algorithms and Data Structures

The loading edge detection problem, described in Section 3.1, raised various geometric issues. The data generated by the sensors was given as point clouds. Section 2.4.1 discusses point cloud manipulation techniques and a data structure that allows to efficiently store, process, and find points in the point cloud.

In Section 2.4.2, we describe and analyze the classic geometric algorithm RANSAC in detail. This algorithm is used several times in different variants in the implementation of the loading edge detection problem.

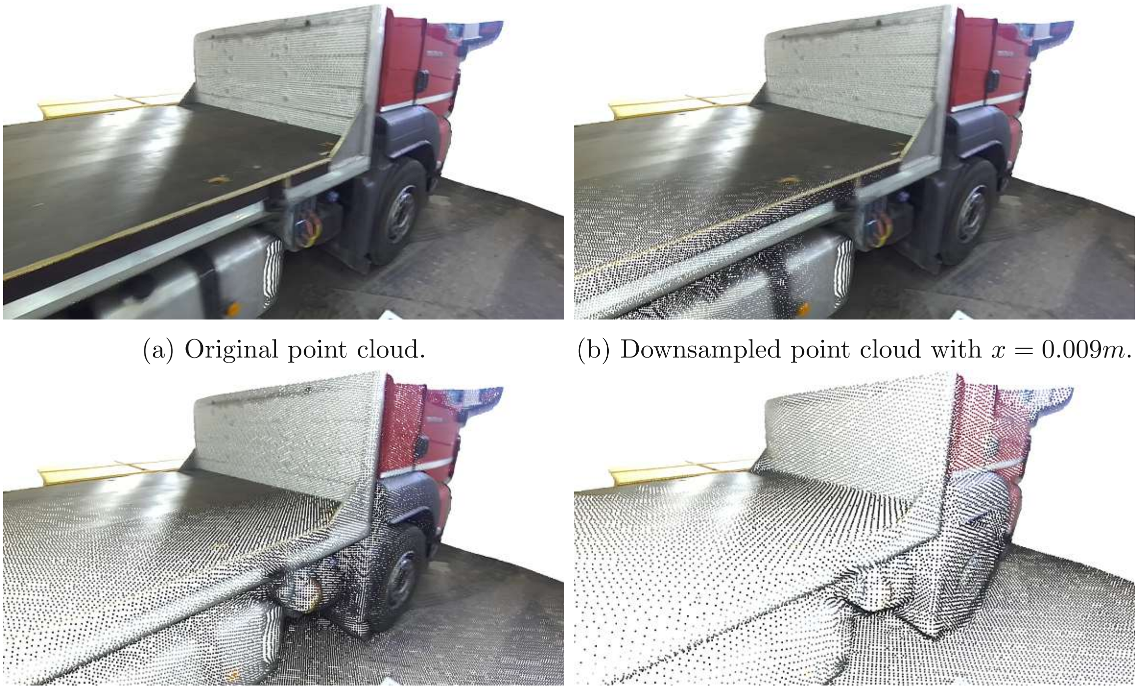
2.4.1. Point Cloud Processing

The most common ways to represent a geometric object in computer vision, computer graphics, or computer-aided geometric design are *polygon meshes* and *point clouds* [41].

For more information on polygon meshes and data structures to represent them, see for instance [57]. In this project, we work with point clouds in the loading edge detection problem and mainly with poses of objects represented by one element of SE_3 in the pose estimation problem.

Here, we focus on point clouds and their processing and manipulation. According to [41], a point cloud is a set of points (in our case in $\mathbb{R}^3$) that represents the surface of one or more objects. They can be generated using for example LiDAR sensors or ZED Cameras (see Chapter 3). Sometimes, models of objects generate these data because they are easier to obtain than real data.

In the preprocessing of point clouds, some kind of *downsampling* is used to reduce the number of points and, consequently, increase the speed of computations on this point cloud. There are several types of downsampling. Some libraries like the open-source Python and C++ package Open3D provide a variety of point cloud manipulating functions, including downsampling functions as described in [64].



(a) Original point cloud.
(b) Downsampled point cloud with $x = 0.009m$.
(c) Downsampled point cloud with $x = 0.018m$.
(d) Downsampled point cloud with $x = 0.036m$.

Figure 2.6.: Point cloud of a truck downsampled with voxel downsampling for different voxel sizes x.

The first downsampling method discussed here is *voxel downsampling*. *Voxel* is short for *volumetric pixel* and can be seen as a pixel in three dimensions [8]. When a point cloud is downsampled by voxel downsampling, the considered space is subdivided into a grid of voxels of some fixed size x (i.e. x gives the side length of the cubes representing the voxels), and the points of the point cloud are assigned to the voxel they are located in. A voxel V containing the points $p_1, p_2, \ldots, p_m$ then generates one point in the new point cloud by averaging its assigned points as

$$\frac{1}{m}\sum_{i=1}^{m} p_i.$$

The resulting point cloud consists of more uniformly distributed looking points that mitigate real data noise to some extent. Regions with a high point density are thinned out compared to regions with a low point density. This new point cloud is highly dependent on the parameter x of the size of a voxel. If x is too small, almost no downsampling happens because there are no voxels with a high number of points in them. If x is too large, the point cloud can lose some of its characteristic features as the extreme example of just one big voxel shows. In Figure 2.6, a point cloud of a truck before and after downsampling is shown.

Another way to downsample a point cloud is to randomly select every n-th point from the original point cloud to create the downsampled point cloud. By using this method, dense regions of a point cloud remain dense compared to sparse regions. It depends on the actual use case whether this method is preferred over voxel downsampling.

Real data usually comes with unwanted noise. Sometimes the depth of a point computed from camera images is too far away from the real depth due to some errors. Especially points in the background of a scene are often poorly matched. To remove these outliers, one can search the neighborhood of each point. If the number of points in the ball with radius r around a point p is below a certain threshold N, the point is considered an outlier and is removed from the point cloud. The parameters r and N have to be chosen according to the point cloud, depending on how the point cloud was generated or already downsampled.

If we know which region of a point cloud contains the desired information, we can cut the point cloud accordingly, for instance, remove all points that lie below a plane or outside of a ball with a certain radius and center.

In practical applications, a combination of these methods is chosen to obtain a point cloud in the desired format, ensuring that algorithms deliver correct results and work efficiently. In Section 3.1.1, we explain the downsampling techniques employed in the loading edge detection algorithm.

Some downsampling techniques as well as our implementation of the loading edge detector need an efficient data structure for organizing point clouds. For example, if we want to find all points in a point cloud $\mathcal{C}$ that lie within a certain distance from a reference point p naively, we just iterate over all points in $\mathcal{C}$, compute the distance, and check if this distance is below a certain threshold. Computing the neighbors of all n points in $\mathcal{C}$ takes in total $\Theta(n^2)$ time, which is pretty bad. Thus, we use and briefly describe kd-trees, introduced by Bentley in [4].

This is a data structure for k-dimensional data. Here, we deal with 3-dimensional points $p = (p_1, p_2, p_3)^T \in \mathbb{R}^3$, but this concept generalizes to k dimensions easily. First, we split the set of points according to their first coordinate: We take a plane parallel to the second and third axis such that there is approximately an equal number of points on both sides of the plane. Then, we divide the set of points $\mathcal{C}$ into two sets $\mathcal{C}_1, \mathcal{C}_2$ for the two sides of the plane. The set $\mathcal{C}_1$ is forwarded to the left child of the root and the set $\mathcal{C}_2$ to the right child. We split $\mathcal{C}_1$ according to the second coordinate of the points (we divide the space with a plane parallel to the first and third axis) into the sets $\mathcal{C}_{11}$ and $\mathcal{C}_{12}$. Then, we split $\mathcal{C}_{11}$ according to their third coordinate (we divide the space with a plane parallel to the first and second axis) into the sets $\mathcal{C}_{111}$ and $\mathcal{C}_{112}$. After that, we start again by dividing the points according to their first coordinate. We continue for each set $\mathcal{C}_{n_1 n_2 \ldots n_l}$ with this procedure until each point can be identified uniquely. This can be seen as a higher dimensional generalization of binary search trees. Querying a *kd*-tree with an axis-aligned search region takes $O(n^{1-\frac{1}{k}} + m)$ time, where n is the number of points, k the dimension of the data, and m the number of points in the output of the range query.

A *kd*-tree can be used to estimate the normal vectors of points in a point cloud. Assuming that the points in the point cloud represent the surface of an object, it makes sense to equip the points with the normal vector of the represented surface at the respective positions. The Open3D function `estimate_normals` realizing these considerations, uses a *kd*-tree to find all points that are close to the considered point p [64]. Let N_p be the set of neighbors of p found with the *kd*-tree. To compute the normal of that point, we first compute the covariance matrix as

$$S = \frac{1}{n} \sum_{x \in N_p} (x - \mu)(x - \mu)^T,$$

where n denotes the number of points and $\mu = \frac{1}{n} \sum_{x \in N_p} x$ the center of the points in N_p. The two eigenvectors v_1 and v_2 to the two largest eigenvalues of S define the two principal components of N_p [15, Section 10.13.1]. Thus, the normal vector n_p of the point p is estimated as the cross-product $v_1 \times v_2$ of the two principal directions.

2.4.2. RANSAC

The fitting of lines, planes, circles, parabolas, or many other simple geometric objects is an important issue, that appears in many real-life applications. It is a geometric optimization problem: Fitting a parabola given by $f(x) = ax^2 + bx + c$ in a set of data points $\{(x_i, y_i)_{i=1}^N\} \subseteq \mathbb{R}^2$ using least-squares regression is the task of minimizing the sum of squared errors

$$\sum_{i=1}^{N} (y_i - f(x_i))^2$$

with respect to the parameters $a, b, c \in \mathbb{R}$ [49].

Fitting a plane given of the equation $E\colon ax + by + cz + d = 0$ in a set of data points $\{(x_i, y_i, z_i)_{i=1}^N\} \subseteq \mathbb{R}^3$ using orthogonal regression is the task of minimizing the sum of squared orthogonal distances

$$\sum_{i=1}^{N} d(E, p_i)^2$$

with respect to the parameters $a, b, c, d \in \mathbb{R}$, where $d(E, p_i)$ denotes the orthogonal distance of the point $p_i = (x_i, y_i, z_i)$ to the plane E [29, Section 4.7.1].

These approaches as well as many other geometric estimation approaches consider outlier points to some degree. In Section 3.1 we want to fit lines in a 3D point cloud of a scene captured with sensors and processed with some functions. The line should estimate the loading edge of a truck. In this scenario, the two approaches mentioned above could not deliver a reasonable result since they try to minimize the error to points that have nothing to do with the loading edge. Here we use the *RANSAC* algorithm proposed by Fischler and Bolles in [19].

RANSAC stands for *random sample consensus* and is a model-fitting method that is robust with respect to outliers. Contrary to the regression techniques described above, outliers that do not resemble the optimal solution are eliminated instead of trying to adapt the solution to them. We start with a geometric model that can be defined by at least m points. Then, m points are randomly sampled from the data set. For the instance of the model defined by these points, the total number of data points that lie close enough to the model is counted. The algorithm repeats this process and keeps the best solution with respect to the number of *votes*. Algorithm 1 shows a simple pseudo-code of this approach.

Algorithm 1 Random Sample Consensus (RANSAC)

Input: point cloud $\mathcal{C}$, model tolerance ε, maximum number of iterations N
Output: best model parameters M

$M_{\text{best}} \leftarrow$ None
$n_{\text{best}} \leftarrow 0$
for $i \leftarrow 1$ **to** N **do**
  randomly sample m points from $\mathcal{C}$
  $M \leftarrow$ parameters for the model defined by the sampled points
  $n \leftarrow$ number of points in $\mathcal{C}$ with a distance $< \varepsilon$ to the model with parameters M
  if $n > n_{\text{best}}$ **then**
    $n_{\text{best}} \leftarrow n$
    $M_{\text{best}} \leftarrow M$
  end if
end for
return M_{best}

Examples of simple use-cases of RANSAC are line fitting, where a line is defined by two points, plane fitting, where a plane is defined by three non-collinear points, and circle fitting, where a circle is defined by three non-collinear points. Some models underlie restrictions regarding the defining points. There is the case of, for instance, a plane that is not sufficiently defined by three collinear points. This can be fixed by sampling more points until there are three non-collinear points given. A circle on the other hand cannot be defined by adding more sample points if the initial three points are collinear (if no circle of infinite radius, i.e. a line, is allowed). This situation could be handled by keeping two points and resampling the third one until they are not collinear. These model-specific issues can be addressed when the parameters for the model are computed.

We will choose the model tolerance ε as seen in Algorithm 1 according to the scene we are working on. In particular, for the loading edge detection, ε is chosen with respect to the accuracy of the generated point cloud. If the loading edge is nearly a straight line, ε can be very small. Otherwise, ε has to be chosen bigger, such that all points on the loading edge are considered for a good RANSAC approximation. In Section 3.1 we chose a tolerance of 0.08 meters.

The maximum number of iterations N is important for the running time of the algorithm.

For finding a line in a point cloud of 1000 points, there are already $\binom{1000}{2} = 499500$ possible point pairs. This number grows like $\Theta(N^2)$. So, N has to be chosen small enough for the algorithm to run efficiently but big enough that a good approximation can be found.

Fischler and Bolles, who first described the RANSAC algorithm, propose in [19, Section II.B.] a way of estimating the maximum number of iterations N required to get a good solution with a certain probability. We adapt this approach here. Let us assume that we have a set $\mathcal{C}$ of c points and the true optimal solution $S \subseteq \mathcal{C}$ contains $n \leq c$ points. Furthermore, let $m \leq n$ be the number of model parameters of the model we try to fit into the set $\mathcal{C}$. We will call the n points in S *inlier points* or just *inliers*.

Let K_m be the random variable giving the number of iterations of the `for`-loop of RANSAC until we find a set of model parameters M defining the solution S. Let us assume that this is achieved if and only if the m chosen points are in the solution set, i.e. $M \subseteq S$. For a small tolerance ε, this gives a good approximation. Then the expected value $\mathbb{E}(K_m)$ of the number of trials is given by

$$\mathbb{E}(K_m) = \sum_{k \geq 1} \mathbb{P}(K_m = k)k,$$

where $\mathbb{P}(K_m = k)$ denotes the probability that the correct solution is obtained in the k-th trial for the first time. Let

$$q_m := \frac{\binom{n}{m}}{\binom{c}{m}} = \frac{n!}{m!(n-m)!}\frac{m!(c-m)!}{c!} = \frac{n!(c-m)!}{c!(n-m)!} \tag{2.24}$$

be the probability that m randomly chosen points in $\mathcal{C}$ lie in the set of S. Then the probability $\mathbb{P}(K_m = k)$ is obtained by $k-1$ unsuccessful attempts followed by a successful trial as

$$\mathbb{P}(K_m = k) = (1 - q_m)^{k-1} q_m.$$

Now we can compute the expected value as

$$\begin{aligned}
\mathbb{E}(K_m) &= \sum_{k \geq 1} (1 - q_m)^{k-1} q_m k \\
&\overset{(1)}{=} q_m \sum_{k \geq 1} k \bar{q}_m^{k-1} \\
&\overset{(2)}{=} q_m \left(\sum_{k \geq 0} \bar{q}_m^k \right)' \\
&\overset{(3)}{=} q_m \left(\frac{1}{1 - \bar{q}_m} \right)' \\
&= q_m(-1)\frac{-1}{(1 - \bar{q}_m)^2} = \frac{q_m}{q_m^2} = \frac{1}{q_m}.
\end{aligned}$$

In the above computation, the equality in (1) is obtained by defining $\bar{q}_m := 1 - q_m$. The equalities in (2) and (3) follow from the rules of differentiating formal power series and the formal power/Laurent series identity $\sum_{i \geq 0} X^i = \frac{1}{1-X}$ [51, Section 8.4].

The variance $\mathbb{V}(K_m)$ of the number of iterations till success is then given by

$$\begin{aligned}
\mathbb{V}(K_m) &= \mathbb{E}(K_m^2) - \mathbb{E}(K_m)^2 \\
&= \sum_{k\geq 1} \mathbb{P}(K_m = k)k^2 - \left(\sum_{k\geq 1} \mathbb{P}(K_m = k)k\right)^2 \\
&= \sum_{k\geq 1} (1-q_m)^{k-1} q_m k^2 - \left(\frac{1}{q_m}\right)^2 \\
&= q_m \sum_{k\geq 1} k(k-1+1)\bar{q}_m^{k-1} - \frac{1}{q_m^2} \\
&\overset{(4)}{=} q_m \bar{q}_m \sum_{k\geq 2} k(k-1)\bar{q}_m^{k-2} + q_m \sum_{k\geq 1} k\bar{q}_m^{k-1} - \frac{1}{q_m^2} \\
&\overset{(5)}{=} q_m \bar{q}_m \left(\sum_{k\geq 0} \bar{q}_m^k\right)'' + q_m \left(\sum_{k\geq 0} \bar{q}_m^k\right)' - \frac{1}{q_m^2} \\
&\overset{(6)}{=} q_m \bar{q}_m (-1)\frac{-2}{(1-\bar{q}_m)^3} + \frac{1}{q_m} - \frac{1}{q_m^2} \\
&= \frac{2-2q_m}{q_m^2} - \frac{1-q_m}{q_m^2} = \frac{1-q_m}{q_m^2}.
\end{aligned}$$

In (4), the index can be shifted to $k \geq 2$ since for $k = 1$ the summand $k(k-1)\bar{q}_m^{k-2}$ is 0. The equalities in (5) and (6) work as in (2) and (3) above.

This gives just a rough idea of the magnitude of a reasonable number of iterations N. We can also choose N such that all chosen points lie in the solution set S in at least one of N iterations with a certain probability p as in [29, Section 4.7.1]. This is the same as failing to choose all points in S in all N iterations with a probability of $1-p$. Thus, with the notation above, the equation

$$1 - p = (1-q_m)^N$$

defines N for given p and q_m. Taking the log of this equation and dividing by $\log(1-q_m)$ gives the equivalent equation (under the assumption $p, q_m \notin \{0,1\}$)

$$N = \frac{\log(1-p)}{\log(1-q_m)}. \tag{2.25}$$

Lastly, we can simplify the definition of q_m as seen in (2.24). Usually, we do not know the exact numbers c of total points and n of inlier points. Instead, we can estimate the percentage of inlier points compared to the total number of points in $\mathcal{C}$. Furthermore, m is typically small (e.g. 2 for lines, 3 for planes) compared to the total number of points and inliers. Thus, we can assume that the probability of choosing an inlier remains constant for all chosen points. Let r be this probability (approximately $\frac{n}{c}$ with the notation above), then q_m simplifies to

$$q_m = r^m.$$

Example 2.4.1. Let $\mathcal{C} \subseteq \mathbb{R}^2$ be a point cloud as in Figure 2.7. We do not want to find a regression line but rather the longest line of points in the point cloud. Thus, RANSAC is an appropriate approach. About $\frac{1}{3}$ of the points lie on the longest line. The expected number of trials to success is then given by

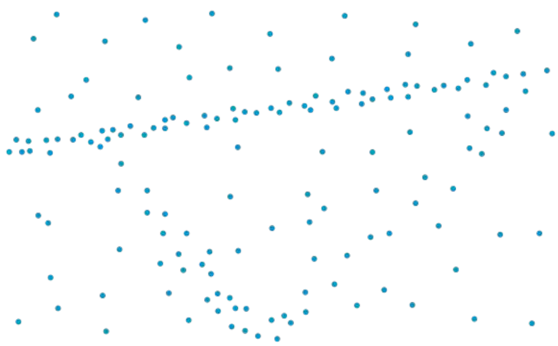
$$\mathbb{E}(K_2) = \frac{1}{\left(\frac{1}{3}\right)^2} = 9$$

and the variance and standard deviation by

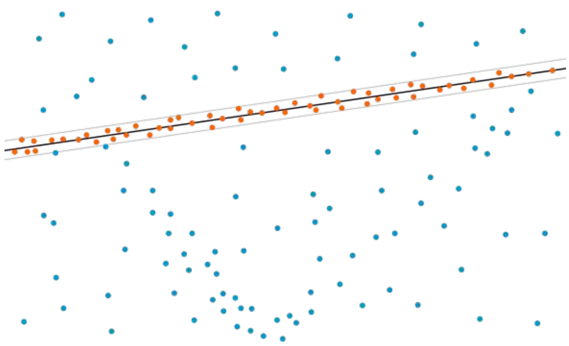
$$\mathbb{V}(K_2) = \frac{1 - \frac{1}{9}}{\left(\frac{1}{9}\right)^2} = 72 \quad \text{and} \quad \sqrt{\mathbb{V}(K_2)} = 6\sqrt{2} \approx 8.49.$$

To get approximately this line with a probability of at least 99%, we perform RANSAC with a maximum number of $N = 40$ iterations according to

$$\frac{\log(1 - 0.99)}{\log\left(1 - \frac{1}{9}\right)} = \frac{\log(0.01)}{\log\left(\frac{8}{9}\right)} \approx 39.10.$$



(a) Point cloud in $\mathbb{R}^2$.



(b) Line containing the most points.

Figure 2.7.: RANSAC line detection example.

3. Composite Object Detection in a Loading Scenario of a Truck

This thesis is part of a bigger project at the Austrian Institute of Technology (AIT), dealing with the autonomous loading of trucks. Here, we address two specific problems of a loading scenario.

The first issue is to detect the loading edge of a truck in a point cloud that is generated by LiDAR sensors or depth cameras. Our solution method includes several geometric considerations on these points and is presented and analyzed in Section 3.1. The essential parts of the code of our implementation can be found in Appendix A.

In the second and central part of this chapter, we view the whole truck as an object that is composed of simpler objects, namely wheels, lights, and the loading platform (respectively the two loading edges on both sides). In a loading scenario, a forklift equipped with sensors should be able to estimate its own pose relative to the truck, given only a rough approximation of the configuration of the truck, i.e. the relative poses of the different parts of the truck to one another. This pose estimation approach is examined in Section 3.2. In Appendix B, we present the crucial parts of our implementation for solving this pose estimation problem.

The algorithms and methods used in this chapter are based on the mathematical foundation examined in Chapter 2. In Section 3.1, we use point cloud manipulation techniques and the RANSAC algorithm of Section 2.4. The pose estimation problem of Section 3.2 is based on factor graphs (see Section 2.2) and optimization on manifolds (see Section 2.3).

All data used in this work come from real recordings of a truck. We used a *MAN TGS 26.440* truck [44], a *PALFINGER BM 214 truck-mounted forklift* (also called *Crayler*) [50], and a *ZED 2i Camera* [54], all provided by Palfinger and the AIT. Figure 3.1 displays the used equipment. The truck was used in both the loading edge detection of Section 3.1 and the pose estimation problem of Section 3.2 as the composite object to be detected. The ZED Camera was mounted on the top of the forklift and the forklift drove around the truck, observing it with the camera. Above the ZED Camera, we installed a LiDAR sensor (short for *Light Detection and Ranging*), which generates a point cloud of its surroundings with laser scanning. The data obtained from the ZED Camera and the LiDAR sensor could be combined to get better data. Here, we only used the ZED Camera for recording.

The ZED Camera is equipped with two cameras, recording at the same time. Thus, it captures stereo images enabling depth perception. The depth data is computed by bundle adjustment. This technique compares matching points in the left and right image to compute the corresponding 3D point by optimizing a nonlinear least squares problem [7]. The internal coordinate system of the ZED Camera depends on the application and the software used to generate the data. In Remark 3.1.1, we describe the two variants used in this work. We refer to [53] for further information on the coordinate frames of this camera.

(a) Truck and autonomous forklift.

(b) ZED Camera of Stereolabs [54].

Figure 3.1.: Truck, autonomous forklift, and ZED Camera used in our experiments.

3.1. Loading Edge Detection

When a truck is being loaded from the side, the essential part of the truck to detect is the loading edge, i.e. the side boundary of the loading platform. In Figure 3.2 one can see a point cloud of a truck recorded with a ZED Camera, where the loading edge is detected with our proposed algorithm and marked in red.

The presented method of loading edge detection relies on the geometric properties of points on the loading edge. Additionally, it uses initial estimates of the height of the loading edge above the ground as well as the height and angle of the camera. These parameters are usually easy to get before using the algorithm with some knowledge about the setting of the camera attached to the autonomous forklift and the truck whose loading edge has to be detected. The more accurate these estimates are, the faster the algorithm gets, because the part of the point cloud where the loading edge could be located can be restricted accordingly.

In light of the pose estimation task described in Section 3.2, we are not only interested in the pose of the loading edge for the autonomous forklift to know where to place the load. The beginning of the left loading edge will mark the origin of the world coordinate frame in which the sensor should locate itself. The direction of the x-axis of the world frame will be defined by the left loading edge and the z-axis will point upwards, orthogonal to the loading platform. The y-axis will be chosen such that the world frame forms a right-handed coordinate system. In Figure 3.12 the coordinate system of the world frame is marked with three arrows defining the axes.

Thus, this algorithm also tries to find the edge between the loading platform and the rear wall to get the beginning of the loading edge and to define the plane incident to the loading platform and therefore the direction of the z-axis. In Appendix A the important parts of the code of our approach to the loading edge detection problem can be found.

In Section 3.1.1 we describe how we solved and implemented the loading edge detection problem and briefly discuss different approaches for edge detection algorithms. Then, we analyze our code and the parameters of our code in Section 3.1.2.

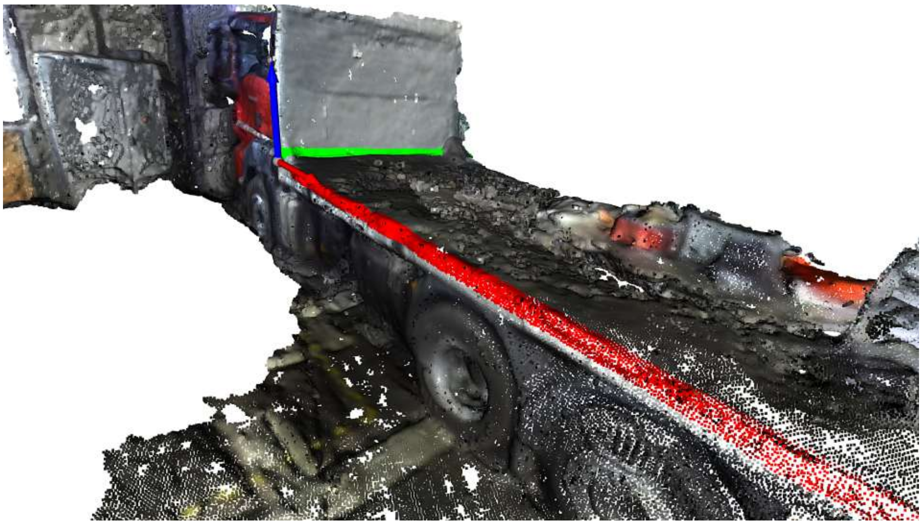


Figure 3.2.: Point cloud of a truck with the detected loading edge marked in red.

3.1.1. Description of the Algorithm

The presented algorithm deals a lot with point cloud processing and manipulation. The Python and C++ library *Open3D* [64] is specialized in 3D Data Processing and was used in the version 0.17.0. It is compatible with the Python package *NumPy* [28] which speeds up computation and makes arrays of any shape easy to handle. In addition to these two packages, the python *time* package was used for measuring the performance of the algorithm in terms of running time and comparing different parameter combinations.

We chose an object-oriented approach. For each point cloud, we want to find the loading edge in, an instance of the class `LoadingEdgeDetection` is created and initialized with this Open3D point cloud, an estimated height and angle of the sensor(s) (ZED Camera and/or LiDAR sensor), and an estimated height of the loading platform. Furthermore, some other parameters can be set optionally when initializing a class instance, otherwise, their default values are used. They will be described below.

When an instance of the `LoadingEdgeDetection` class is initialized, the up-vector u^* (z-axis of the world frame) of the scene is estimated in the sensor coordinate frame from the given depth angle of the camera. This is equivalent to finding a rotation in SO_3 that transforms the z-axis of the sensor frame to the z-axis of the world frame. Here, the sensor frame is defined as a right-handed coordinate system with the x-axis pointing in the direction the sensor is looking at, the y-axis pointing to the left, and the z-axis pointing upwards. Since u^* is only a rough estimation of the real up-vector of the scene, we assume that the camera is just tilted to the front. Therefore, we only need a rotation around the y-axis to rotate the z-axis of the sensor coordinate frame upwards. Thus, for a given depth angle α of the camera, the up-vector is given by

$$u^* = \begin{pmatrix} \cos(-\alpha) & 0 & -\sin(-\alpha) \\ 0 & 1 & 0 \\ \sin(-\alpha) & 0 & \cos(-\alpha) \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -\sin(-\alpha) \\ 0 \\ \cos(-\alpha) \end{pmatrix}.$$

Remark 3.1.1. The choice of the sensor frame is dependent on the hardware and software used to obtain the data. There is no general convention for sensor coordinate frames. Here, we obtained the data with a ZED Camera using the camera frame standard of the robot operating system (ROS). In Section 3.2, we use a sensor frame where the sensor looks in negative z-direction, the y-axis points upwards, and the x-axis to the right since the data used in that algorithm are obtained by Blender which is based on the OpenGL standard camera definition, where this is the standard sensor frame [53].

Working with real data presents some difficulties. First, we have only limited control over the amount of data generated by the sensors. Here, with over 2.7 million points generated by the ZED Camera, computation would take way too long for practical purposes. Thus, we need to preprocess the point cloud with techniques described in Section 2.4.1. Figure 3.3 shows the downsampling process used here.

We start by performing voxel downsampling using a voxel size of 0.015 meters reducing the point cloud to roughly 700,000 points. In the next step, outliers are removed, such that around 600,000 points remain. For a point cloud generated by a ZED Camera, a point p is considered an outlier if there are less than 30 points within a radius of 0.05 meters around p. Lastly, we use the estimated height of the loading platform and the estimated direction of u^* to get an approximate pose of the plane L that is incident to the loading platform. Then we cut the point cloud around this plane and remove every point that is too far away from L. About 200,000 points remain. The parameter `search_width` defines the width of the remaining strip of points. Besides lowering the number of points and thus lowering the computation time, the last step has another effect: We cut away a lot of points of the scene that could be detected as edge points by the algorithm, for instance, the edge between the floor and a wall is an edge that could be declared as loading edge falsely. Thus, `search_width` should especially be chosen small enough to cut the floor away. We refer to the point cloud obtained after these downsampling steps as $\mathcal{C}$.

Now the preprocessing of the point cloud is finished and the search for the loading edge can start. As mentioned above, the goal is to find the beginning of the loading edge, i.e. the loading edge e_l and the edge e_w between the loading platform and the rear wall of the truck, if possible. In Figure 3.2 these two edges are marked in green and red. The algorithm does not work with semantic information about e_l, in particular, only the geometric properties of the points are used to find the loading edge. To be precise, the algorithm first looks for the longest edge e_1 in the point cloud $\mathcal{C}$ that is approximately orthogonal to the estimated up-vector u^* with the function `find_longest_edge()`. In the second step, the function `find_orthogonal_line()` searches for an edge e_2 that is approximately orthogonal to e_1 and u^*. Depending on the camera's pose and the resulting perspective, either e_l or e_w is found first.

Let us assume that every edge detection and every estimation of angles and heights works perfectly (or sufficiently well). Then the algorithm delivers the correct result or approximation if the longest visible edge e_1 in the point cloud is in $\{e_l, e_w\}$ and the longest visible edge orthogonal to e_1 and u^* is also in $\{e_l, e_w\}$.

To find the edge e_1, we first compute possible edge points in $\mathcal{C}$. The method that works best among all methods considered here in this context uses the estimated normals of the points in the point cloud. The normals of points in a point cloud can be estimated as described in Section 2.4.1. Figure 3.4 shows the edge points detected by this method. Alternative approaches to finding edge points are discussed at the end of this section.

To get edge points, we compute the *kd*-tree of the points in $\mathcal{C}$ (see Section 2.4.1 for more about *kd*-trees) and estimate the normals of the points. To further accelerate the computation, the point cloud $\mathcal{C}$ is then downsampled one more time. Now, we use the uniform downsampling method of Open3D with a downsampling factor of 40, which randomly selects every 40-th point, to get the point cloud $\mathcal{C}'$. The roughly 200,000 points in $\mathcal{C}$ reduce to about 5,000 points in $\mathcal{C}'$. Now we search for edge points in the point cloud $\mathcal{C}'$.

(a) Original point cloud.

(b) Voxel downsampling of the point cloud (a).

(c) Outliers of the point cloud (b) removed.

(d) Point cloud cut to the interesting region.

Figure 3.3.: Downsampling process of a point cloud of a truck: First, we apply voxel downsampling, then we remove outlier points, and then we restrict the point cloud to a small area around the loading platform.

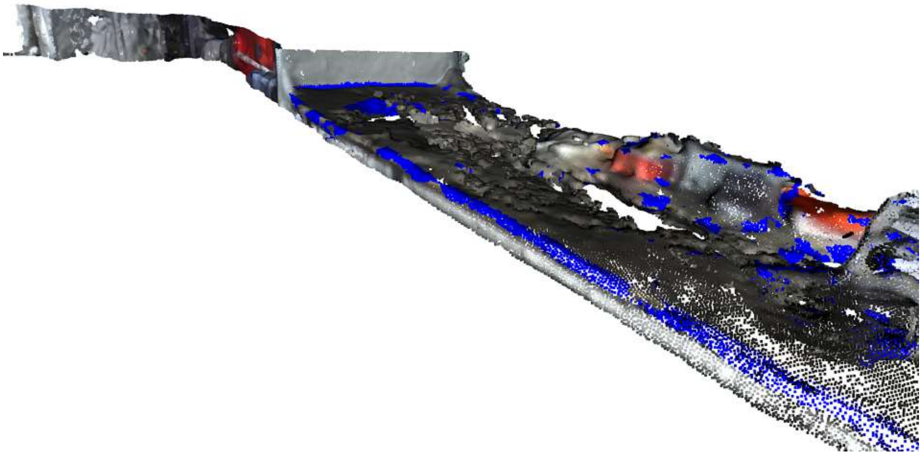


Figure 3.4.: The blue points are the edge points that are detected by using the estimated point normals. For illustration purposes, the edge point detection was carried out on $\mathcal{C}$ instead of $\mathcal{C}'$.

For each point p in $\mathcal{C}'$, let $N_p \subseteq \mathcal{C}$ be the set of neighbors of p, i.e. all points in $\mathcal{C}$ that lie within a radius of 0.1 meters around p. This set can be efficiently obtained by using the kd-tree computed for $\mathcal{C}$. Let $N_p^{\text{up}} \subseteq N_p$ be the set of all points whose normals are (approximately) parallel to the up-vector u^* and let $N_p^{\text{rest}} := N_p \setminus N_p^{\text{up}}$ be the set of all other points. In this approach, we call a point p an edge point as declared in the following definition.

Definition 3.1.2 (Edge Points using Estimated Normals)**.** Concerning the conditions and definitions stated above, we call a point p an *edge point* if all of the following properties hold.

(i) The normals of the points in N_p^{up} have (approximately) the same direction v_p^{up}.

(ii) The normals of the points in N_p^{rest} have (approximately) the same direction v_p^{rest}.

(iii) The vectors v_p^{up} and v_p^{rest} are (approximately) orthogonal.

(iv) The sets N_p^{up} and N_p^{rest} are (approximately) the same size.

All these conditions are just *approximately* because real data as well as the estimated normals are not perfect. This is a reasonable way to define edge points because under these conditions, a point p lies in the intersection of two planes (all points in a plane have the same normal vector). Part (iii) of the definition ensures that the edge found is created by two nearly orthogonal planes, just like the loading edge. This definition of edge points is similar to the edge detection approach described in [60, Section 4.2].

Figure 3.5 visualizes this edge point definition. The arrows indicate the estimated normals for points close to the considered (orange) point p. If the neighbors of p lie in a plane, all estimated normals are approximately parallel. If the points in N_p lie on a curved surface, the estimated normals point in various directions. In the last case, p is indeed an edge point. The estimated normals cluster in two approximately orthogonal groups as described above.

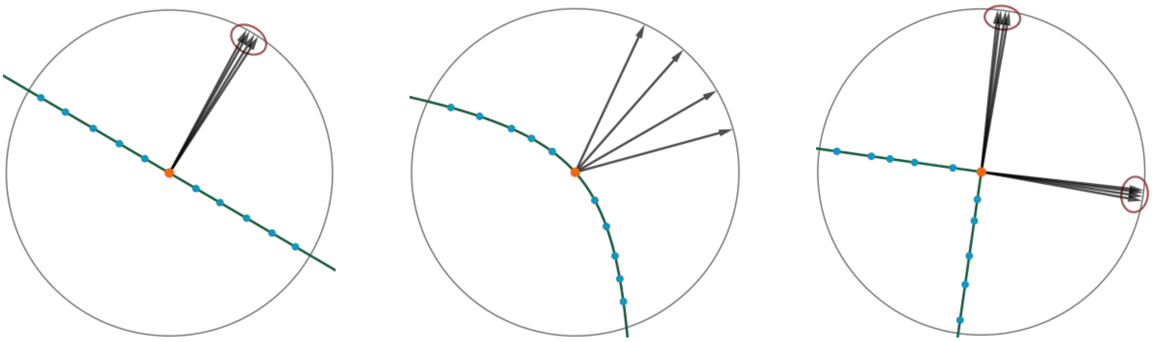


Figure 3.5.: Normals for points on planes, curved surfaces, and edges as [60, Figure 3].

We can define the vectors v_p^{up} and v_p^{rest} as

$$v_p^{\mathrm{up}} = \frac{1}{|N_p^{\mathrm{up}}|} \sum_{q \in N_p^{\mathrm{up}}} n_q \quad \text{and} \quad v_p^{\mathrm{rest}} = \frac{1}{|N_p^{\mathrm{rest}}|} \sum_{q \in N_p^{\mathrm{rest}}} n_q,$$

where n_q is the estimated normal vector of a point q. The normals n_q in N_p^{up} have approximately the same direction if the length of v_p^{up} is close to 1. So, parts (i) and (ii) are checked by computing v_p^{up} and v_p^{rest} and their lengths. Allowing shorter vectors leads to more accepted edge points. So if (i) and (ii) hold, the points of N_p^{up} and the points of N_p^{rest} lie in a plane each.

Condition (iii) of Definition 3.1.2 is only checked if both vectors v_p^{up} and v_p^{rest} are not the zero-vector. Thus, we can compute the inner product

$$\left\langle \frac{1}{\|v_p^{\mathrm{up}}\|} v_p^{\mathrm{up}}, \frac{1}{\|v_p^{\mathrm{rest}}\|} v_p^{\mathrm{rest}} \right\rangle. \tag{3.1}$$

If this inner product is 0, these vectors are orthogonal. Allowing inner products close to 0 leads to more accepted edge points. This relaxation makes sense since the inner product (3.1) relates to the angle α between the corresponding vectors by the formula [39, Section 6.1 (2)]

$$\cos \alpha = \frac{\langle v_p^{\mathrm{up}}, v_p^{\mathrm{rest}} \rangle}{\|v_p^{\mathrm{up}}\| \|v_p^{\mathrm{rest}}\|} = \left\langle \frac{1}{\|v_p^{\mathrm{up}}\|} v_p^{\mathrm{up}}, \frac{1}{\|v_p^{\mathrm{rest}}\|} v_p^{\mathrm{rest}} \right\rangle.$$

If this condition holds, the edge is created by two (nearly) orthogonal planes.

Lastly, part (iv) of Definition 3.1.2 can easily be checked by comparing the number of points in both sets. Allowing small differences in the number of contained points leads to more accepted edge points. This condition ensures that the points accepted as edge points lie sufficiently close to the edge. In an extreme case, the set N_p^{rest} could, for example, contain only one point with all other points belonging to N_p^{up} and thus, under condition (i), lying in a horizontal plane, rather far away from the edge.

The partition into the sets N_p^{up} and N_p^{rest} as well as the checks of the conditions above happen in the functions `find_edge_points_normals()` and `cluster_normals()`.

Now, we have a set $\mathcal{C}_{\text{edge}} \subseteq \mathcal{C}'$ of edge points. The edge e_1 is defined as the line passing through the most points in $\mathcal{C}_{\text{edge}}$, with a small tolerance allowing points to lie close to the line. This is done by a RANSAC approach as described in Section 2.4.2. In Figure 3.4 about $\frac{1}{3}$ of all edge points lie on the loading edge. Thus, with Formula (2.25), at least 40 iterations of RANSAC are required to find this edge with a probability of 99%.

To find e_2, the function `find_orthogonal_line()` searches in $\mathcal{C}_{\text{edge}}$ for the longest edge (approximately) orthogonal to e_1, using a RANSAC approach. In particular, with RANSAC the line with the most votes in $\mathcal{C}_{\text{edge}}$ is found. If this line is approximately orthogonal to e_1 and u^*, it is accepted as e_2. Otherwise, the points on this line are deleted from $\mathcal{C}_{\text{edge}}$ and the process is started again. The parameter `max_lines` defines how often this search is restarted. By doing this, we want to avoid accepting an edge orthogonal to e_1 that consists of too few points, because the edge between the loading platform and the rear wall of the truck does not have to be visible in every analyzed scene. So if no second edge e_2 is detected within a few iterations, the algorithm just finds one edge.

If a second edge was found, we want to declare a point as the origin of the world frame. Using perfect data, this would be the intersection point of e_1 and e_2. Generally, these two lines do not intersect in practice. Let E_i for $i = 1, 2$ be the plane determined by e_i and the estimated up-vector u^*. Let S_1 be the intersection point $e_1 \cap E_2$ and S_2 be the intersection point $e_2 \cap E_1$. The origin is then defined as the midpoint $\frac{1}{2}(S_1 + S_2)$ between the two intersection points.

In the next step, the directions of the x-axis and the y-axis are defined. The coordinate frame should look as in Figure 3.12, so the loading edge and the edge between the loading platform and the rear wall lie in positive x- respectively y-direction of the origin. Again with a RANSAC-like approach, we look for the endpoints of the lines e_1 and e_2 in $\mathcal{C}_{\text{edge}}$: Two edge points are sampled and the pair of points with the greatest distance between them is accepted as the pair of endpoints of the line. This does not have to be the very best solution possible, some good approximation is sufficient. The vector from the origin to the endpoint that is further away defines the direction of the corresponding axis.

As stated above, it is not clear if e_l or e_w is the line e_1 that is found first by the algorithm. To differentiate e_l from e_w, we look at a difference between these two edges, that can easily be detected in a point cloud: Directly above the loading edge e_l are no points, while directly above e_w there are still many points of the rear wall. So both lines are lifted by a few centimeters in the direction of u^* and the number of points on the lines before and after the lifting are compared. The line with the smaller quotient of the number of points on the line after and before lifting is declared as the loading edge e_l.

Let v_l and v_w be the direction vectors of e_l and e_w. If the truck is seen from the left side, the direction vectors v_x^{left}, v_y^{left}, and v_z^{left} are defined as

$$v_x^{\text{left}} := v_l, \qquad v_z^{\text{left}} := v_x^{\text{left}} \times v_w, \qquad v_y^{\text{left}} := v_z^{\text{left}} \times v_x^{\text{left}}.$$

The order of the factors of the cross-product is important and can be checked with the right-hand rule. If the truck is seen from the right side, this process does not define the origin of the world frame. But concerning the factor graph approach for the pose estimation problem in Section 3.2, we define the pose of the right loading edge by the coordinate frame where e_l defines the y-axis instead of the x-axis as above. So, let

$$v_y^{\text{right}} := v_l, \qquad v_z^{\text{right}} := v_w \times v_y^{\text{right}}, \qquad v_x^{\text{right}} := v_y^{\text{right}} \times v_z^{\text{right}}$$

be the direction vectors of the axes if the right loading edge was detected.

If only the loading edge e_l was detected, the origin is set to one endpoint randomly. The z-axis is defined by the up-vector and the last axis is defined by the cross product of v_l and u^*. Whether the last axis is taken as $v_l \times u^*$ or as $u^* \times v_l$ is determined by moving the line of the loading edge a bit to the side to detect on which side of the loading edge the loading platform is located.

This is our approach to solving the loading edge detection problem. A different approach to defining the edge points was attempted in this project: Instead of estimating the normals of all points in the neighborhood N_p of a potential edge point p, we compute the centroid $c_p = \frac{1}{n} \sum_{p' \in N_p} p'$ of the neighbors of p and measure the distance $d(p, c_p)$. This edge detection method is also described in [3]. If this distance is larger than a certain threshold, the point p is declared as an edge point. Figure 3.6 shows the edge points detected in that way. We can see that a lot more points on edge-like structures in the point cloud (especially on things lying on the loading platform) are declared as edge points. Furthermore, points on borders of the point cloud fulfill this property too. Methods like this centroid approach lack the possibility of considering previous knowledge about the edge direction, such as a normal vector to that direction. A possible improvement would be to consider the vector $c_p - p$ instead of just the distance $d(p, c_p) = \|c_p - p\|_2$. Du summarizes and compares different edge detection techniques in his work [14], including the normal vector approach and the centroid approach.

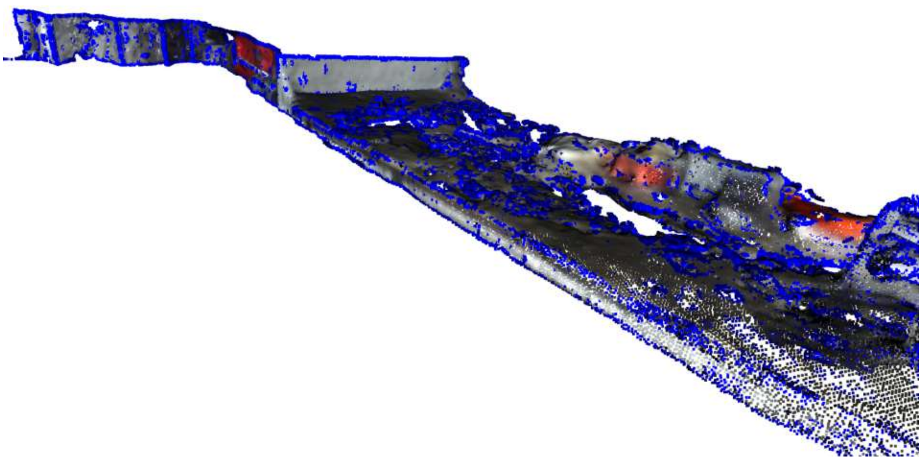


Figure 3.6.: The blue points are the edge points that are detected by using the centroids of the neighbors.

3.1.2. Parametrization and Analysis of the Algorithm

The quality of the result of the loading edge detection algorithm depends on the quality of the data. For a perfectly generated point cloud and exact estimates of the height and angle of the camera, this algorithm delivers the correct result efficiently. LiDAR sensors generate point clouds of good quality. Therefore, a LiDAR sensor would work well together with the loading edge detector. Since the necessary hardware was not yet installed properly, we only have limited access to testing data. We will analyze the influence of different parameters on the algorithm's running time, using a point cloud generated with data from a ZED Camera. The ground truth of the detected loading edge can be seen in Figure 3.7.

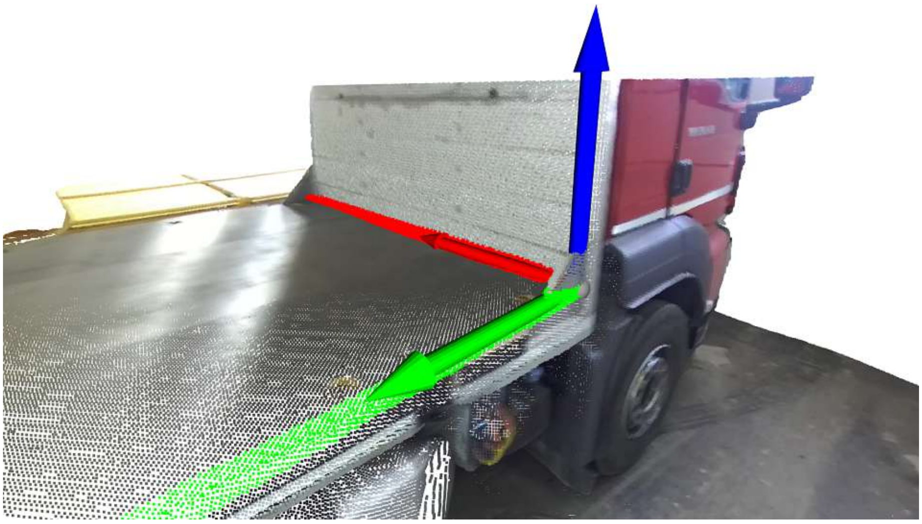


Figure 3.7.: Point cloud of the truck, where the right loading edge (green) is detected.

First, we analyze the used *voxel size*, i.e. the size of the voxel grid, that we use to downsample the point cloud in the preprocessing step. Figure 3.8 displays the running time of the algorithm, the error to the optimal solution, and the success rate for different voxel sizes, where each parameter set was tested 100 times. The lines indicate the mean values of the 100 trials and the shaded regions around the lines indicate the empirical standard deviation. As the voxel size increases, fewer points remain in the point cloud, in which we want to find the loading edge. The running time of operations like finding the neighbors of all points in a *kd*-tree and computing the edge points depends on the number of points in the point cloud. Additionally, the voxel downsampling is more computationally expensive if the voxel size is smaller. Therefore, a larger voxel size results in a shorter running time of the algorithm (as portrayed in Figure 3.8) and shorter preprocessing time. When point clouds are downsampled, it results in a loss of information. The error in Figure 3.8 is measured as

$$\|o^{\text{est}} - o^{\text{true}}\|_2 + \|v_y^{\text{est}} - v_y^{\text{true}}\|_2 + \|v_x^{\text{est}} - v_x^{\text{true}}\|_2$$

with the estimated origin $o^{\text{est}} \in \mathbb{R}^3$, the estimated y- and x-axis v_y^{est} and v_x^{est} in S^2, and their ground truth counterparts o^{true}, v_y^{true}, and v_x^{true}. We see, that the error slightly increases for increased voxel size. The error is only measured if the algorithm finds two orthogonal edges. The success rate indicates how often two orthogonal edges could be found. For a voxel size of about 0.020 and more, the algorithm does not find two edges in all trials anymore. Similar results are found for other point clouds of the truck. To minimize the running time of the algorithm while maintaining robust results, we choose a voxel size of 0.018 for the other testing trials. The optimal values for this and the other parameters depend on the way, the investigated point cloud is recorded and generated. LiDAR point clouds, for example, probably need a different voxel size.

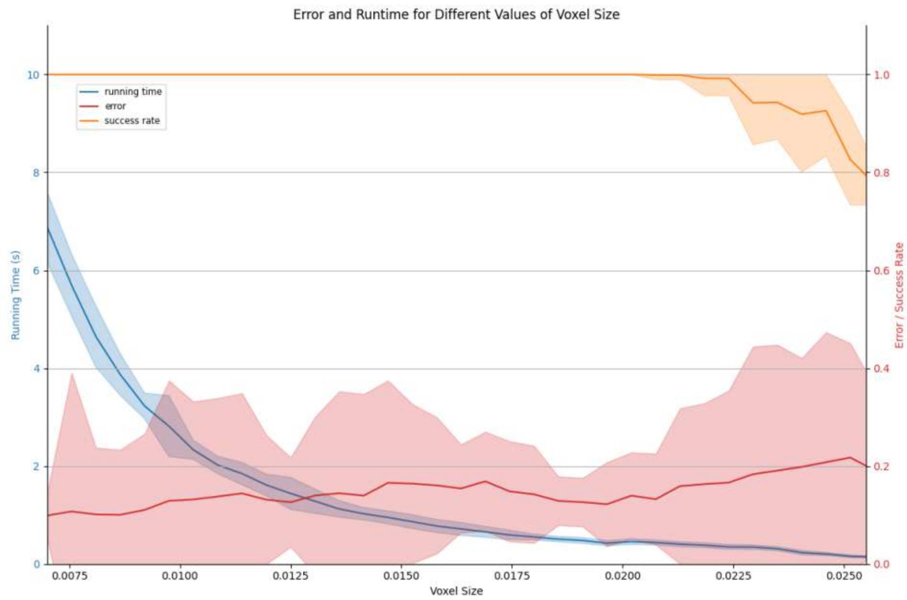


Figure 3.8.: Running time, error, and success rate for different voxel sizes for 100 test trials.

Figure 3.9 shows the importance of restricting the edge search to a certain area in the point cloud. The *search width* is the width of the strip around the estimated plane the loading platform lies in, where we search for the loading edge. This plane is computed with the estimated height and angle of the camera and is therefore not exact. Hence, if the search width is too small, regions containing important information may be removed from the point cloud, leading to a low success rate and high errors. If the search width is too high, the investigated point cloud will contain too many potential edge points leading to unpredictable behaviour in detecting two orthogonal edges.

The uniform downsampling step before computing the edge points has a great influence on the running time of the algorithm. Figure 3.10 shows how the running time changes with different uniform downsampling factors. Around a downsampling factor of 50, the first signs of unwanted behavior of the algorithm appear. For point clouds with more noise, this can happen more intensively. Throughout the few available testing point clouds, a downsampling factor of 40 has proven to be effective.

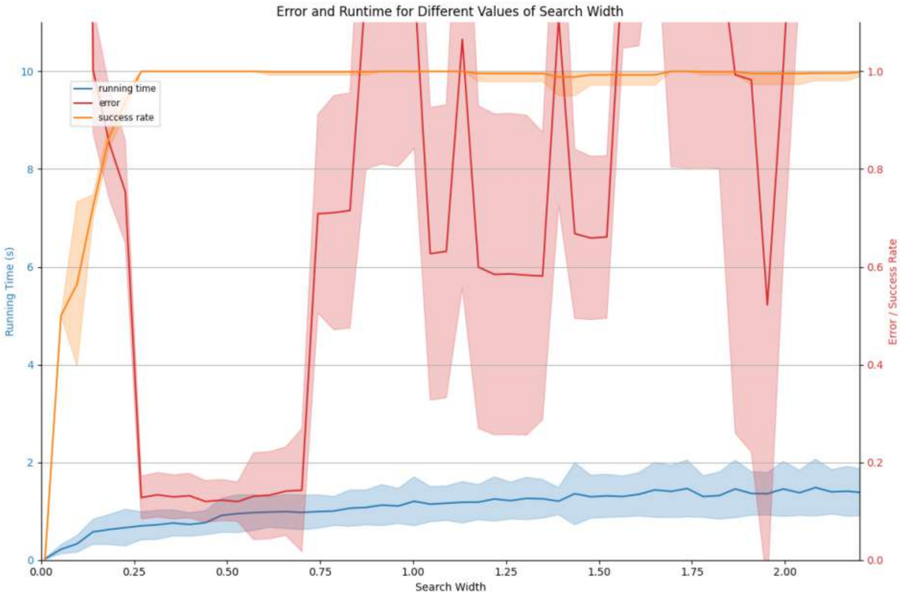


Figure 3.9.: Running time, error, and success rate for different search widths for 100 test trials.

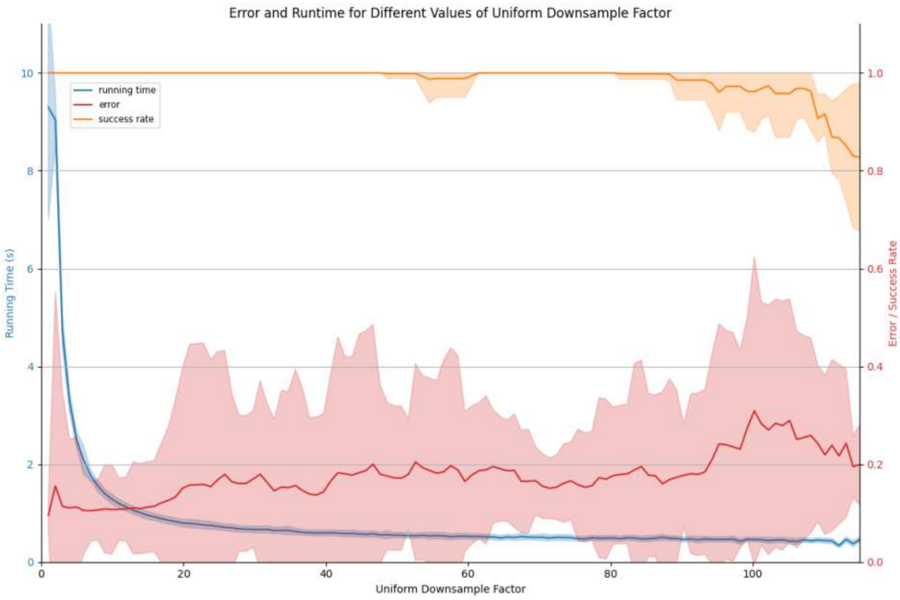
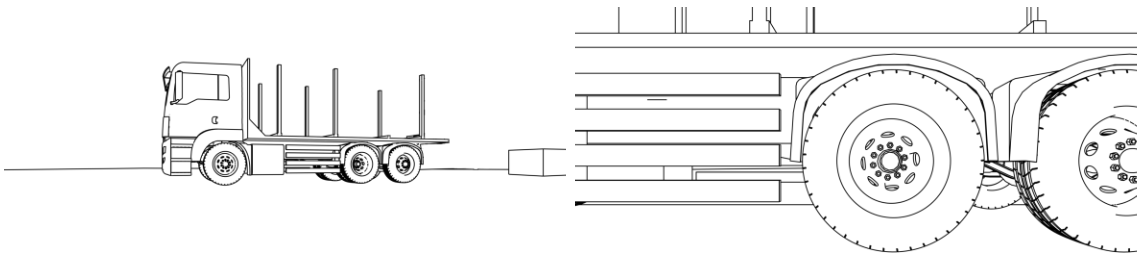


Figure 3.10.: Running time, error, and success rate for different uniform downsampling factors for 100 test trials.

3.2. Part-Based Pose Estimation Using Factor Graphs

The second and central problem of this work deals with the location of a sensor throughout the entire process of automated loading of a truck. The truck is viewed as an object composed of some simpler objects like the wheels of the truck. Initially, the truck's configuration is only roughly known. Throughout the loading process and with every analyzed detection, this model of the truck is updated and improved such that the sensor pose can be estimated even if only a small part of the truck is visible to the sensors. Figure 3.11 displays this principle of collecting information from far away such that robust pose estimation is possible when the sensor is close to the truck.



(a) Truck viewed from far away. (b) Close up of the truck.

Figure 3.11.: Depending on the position of the forklift, the sensors observe different sections of the truck.

To estimate the truck configuration and the sensor pose simultaneously, this structure of the truck as a composite object is modeled as a factor graph as described in Section 2.2. The parts of the truck as well as the sensor in different time steps are represented by variables. Approximate relations between the various parts translate to factors between the respective variables and the observations of the parts of the truck in the sensor frame are expressed as factors between the sensor and the corresponding parts. By optimizing the factor graph as seen in Section 2.3.5, the model is updated and the pose of the sensor in the world frame is estimated. The origin of the world coordinate frame is defined by the beginning of the left loading edge and the axes by the loading edge, the edge between the loading platform and the rear wall, and the vertical direction. Figure 3.12 shows this world frame. Section 3.2.1 describes this translation to a factor graph in more detail.

This process is realized in an algorithm and tested on data obtained from videos of the truck filmed by sensors attached to a forklift. The tool for detecting the parts of the truck correctly is still in development, so the parts of the truck were manually annotated. We use the library GTSAM [11] for factor graph modeling as it provides good ways of defining and using factor graphs, particularly, it is equipped with efficient algorithms for optimizing factor graphs. Section 3.2.1 also gives a brief introduction to the functionality and usage of GTSAM and Section 3.2.2 describes the algorithm of our solution of the part-based pose estimation problem. Finally, in Section 3.2.3, we analyze the algorithm and some parameters. In Appendix B the important parts of the code of our solution can be found.

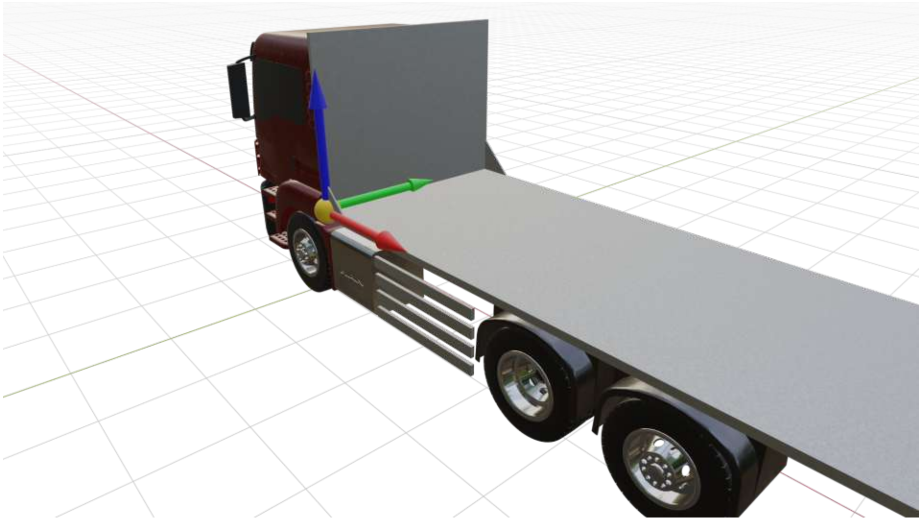


Figure 3.12.: World coordinate frame for an instance of a truck.

3.2.1. Composite Object as a Factor Graph and GTSAM

Factor graphs as described in Section 2.2 are an elegant way of representing a composite object. As they are used in the pose estimation problem, they combine the geometrical structure of the considered object with probability theoretical relations. The Python and C++ library GTSAM [11] (Georgia Tech Smoothing and Mapping) provides data structures and functions for representing and optimizing factor graphs in a variety of applications.

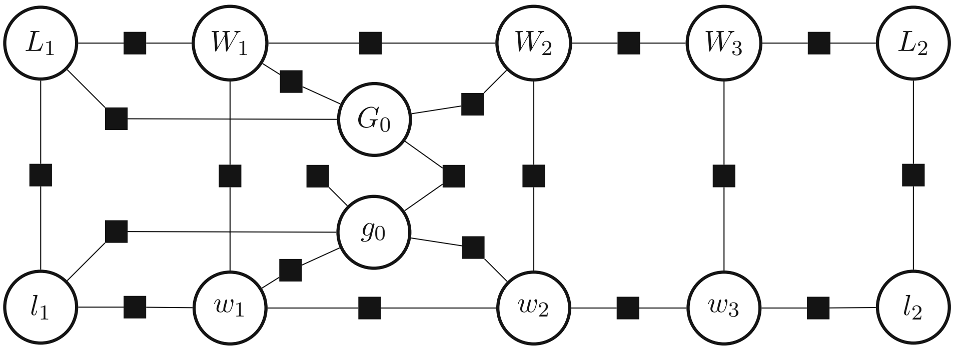


Figure 3.13.: Factor graph representing the truck as a composition of wheels, lights, and loading edges.

We view the truck that can be seen in Figure 3.11 as a composition of its six wheels, two front lights, two rear lights, and the loading edges on both sides. In the code and the following, we refer to the three wheels on the left side of the truck as w_i, for $i \in \{1, 2, 3\}$, starting from the front wheel w_1 back to the last wheel w_3. On the right side, the wheels are denoted with W_1, W_2, and W_3. The front lights are named l_1 and L_1 and the rear lights l_2 and L_2, where the lowercase letters denote the lights on the left side of the truck, and the uppercase letters their counterparts on the right side. The (beginning of the) left and right loading edges are symbolized as g_0 and G_0. The beginning of the left loading edge denotes the origin of the world/global frame. These names can be realized in GTSAM with *symbols* consisting of one letter and one number each.

In our implemented solution of the pose estimation problem, we create a factor graph at the beginning, representing the geometrical relations between the parts described above. Figure 3.13 visualizes this factor graph. The round vertices depict the variables of the factor graph that represent the parts of the truck. The factor nodes are symbolized by the black squares that lie between pairs of variables. We do not connect each possible pair of two variables, but just those that are somehow close to each other and/or related. For example, the left front wheel w_1 is connected to its counterpart W_1 on the other side of the truck, to the left front light l_1, to the left loading edge g_0 and to the wheel w_2.

We realize the factors in GTSAM as `BetweenFactorPose3`-factors that can be defined by the transformation $T \in \mathrm{SE}_3$ that relates the poses P_{p_1} and P_{p_2} of the parts p_1 and p_2 of the truck as $T(P_{p_1}) = P_{p_2}$. When viewing the wheels of the truck as right circular cylinders, the class of proper rigid transformations $\mathcal{T}_{P_w}^{P_{w'}}$ for two wheels w and w' of Definition 2.1.17 contains more than one element of SE_3. Here, we ignore this problem of symmetry that would result in transformations $T, T' \in \mathrm{SE}_3$ with $T \neq T'$ but $T \sim_{P_1} T'$ according to Definition 2.1.16, i.e. T and T' both could define the factor possibly causing problems when optimizing the factor graph since GTSAM has no functionality for handling symmetries properly. This problem is considered in a different part of the project at AIT that is not regarded in this work. Here, we assume that each object is given by one point in $\mathbb{R}^3$ (the centers of the base circles of the wheels viewed as right circular cylinders, the centers of the lights, and the beginnings of the loading edges), defining the position of this object, and a coordinate frame, defining the orientation. Moreover, the notation $T(P_{p_1}) = P_{p_2}$ would not be well-defined otherwise, since P_{p_1} and P_{p_2} denote elements of the pose spaces of objects p_1 and p_2 that can only be compared by a transformation $T \in \mathrm{SE}_3$ if p_1 and p_2 are the same geometric object.

Until now, we just use relative poses between different parts. Thus, the truck could be anywhere in the world frame. But since we define the origin of the world frame at the beginning of the left loading edge, the pose of the whole truck in the world frame cannot be arbitrary. To solve this, we add a factor that is just connected to g_0. In GTSAM this is called `PriorFactorPose3` and we initialize this factor with $(R, t) \in \mathrm{SE}_3$ with $t = (0,0,0)^T \in \mathbb{R}^3$ and $R = I_3 \in \mathrm{SO}_3$.

Later, the sensors start moving around the scene and observing some parts of the truck. For each time step t, a new variable s_t is introduced, representing the pose of the sensor at that time step. The observations $o_p^{(t)}$ in the sensor coordinate frame at time step t define the factor nodes between the sensor s_t and the respective part p of the truck. These factors are also realized with `BetweenFactorPose3` in GTSAM. After three time steps, the factor graph could look like in Figure 3.14.

Roughly speaking, we start from a model of the truck that defines the factor nodes in the factor graph $\mathcal{F}_0$ in Figure 3.13. In each time step t, a new sensor node s_t is introduced with the observations as factors. The new factor graph $\mathcal{F}_t$ is then optimized as illustrated in Section 2.3.5. The factors f_j of $\mathcal{F}_t$ are defined as probability densities as in (2.22). For more detailed information on how GTSAM defines the factors, see the documentation [12].

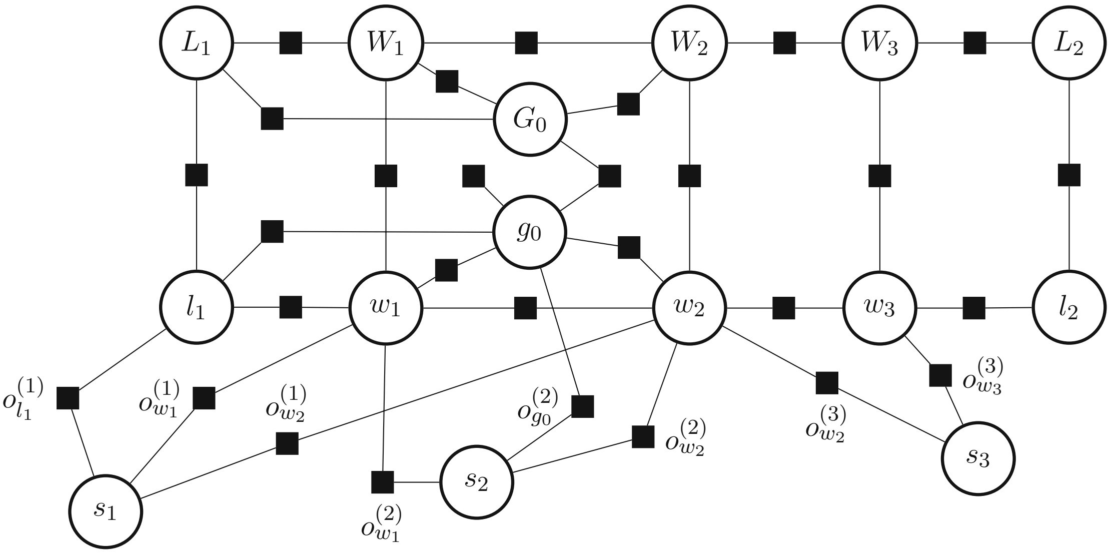


Figure 3.14.: Factor graph $\mathcal{F}_3$ for the parts of the truck after 3 time steps.

Subsequently, the model is updated according to the outcome of the optimization, i.e. the poses of the parts of the truck in the world frame are being updated as described in Section 3.2.2 in more detail. The model of the truck includes on one hand for each part the estimated pose in the world frame represented as an element of SE_3. On the other hand, some dependencies between the different parts describe the configuration of the truck in more detail. For instance, the wheels w_1 and W_1 have the same x-coordinate, the three wheels on the left, respectively right side of the truck have (approximately) the same y-coordinate, and all six wheels have the same height (z-coordinate). These special properties of the truck have to be taken into account when updating the model. Furthermore, some parts can have certain degrees of freedom regarding their pose: The steering angle of the front wheels is not fixed, they can rotate around a vertical axis. Furthermore, the height of the loading edge above the ground is dependent on the weight of the load that is placed on the loading platform (and typically changes throughout the loading process). These features describe the truck (for our purposes) sufficiently well.

3.2.2. Description of the Algorithm

Our solution to the pose estimation problem works with one class, keeping track of the currently assumed model of the truck and the factor graph that is enlarged with every time step. The code could be adapted to fit various other composite objects. For using this code on other composite objects, the initial model, defined via the symbols of the different parts, approximate poses of the parts in some world frame, and other object-specific settings, have to be adapted.

The program starts with initializing the factor graph of the truck as well as some other attributes. In each time step, we follow the same three phases. In the first phase, we generate the data. The data consists of the poses of the observed parts. The loading edge can be detected by the loading edge detector described in Section 3.1. The detection of the other parts is not part of this work. The parts are now manually annotated and will be automatically detected by a machine-learning approach once the whole project is finished. The generated data is handed over to the pose estimator.

The second phase is the pose estimation itself. In particular, the pose of the sensor in the world frame is estimated from the observations of the current time step, previous observations, and the estimated model of the truck. Thus, the sensor tries to locate itself relative to the truck. This is done with the factor graph approach described in Section 3.2.1. The function `estimate_sensor_pose()` starts by defining initial guesses for the poses of all variables in the factor graph to get a reasonable starting point for the optimization. Using good starting points increases the convergence speed and results in a higher chance of reaching a global optimum. The poses of the variables of the different parts are initialized by their estimated pose in the world frame, given by the current model of the truck. The sensor symbols from previous time steps remain for a certain amount of time steps in the factor graph. Now, old sensor symbols are either deleted from the factor graph to keep the factor graph efficient, or their initial guess is set to their estimated pose from the last time step. The factor graph is expanded by the sensor variable s_t of the current time step. The initial guess for the pose of s_t is either set randomly, if we are in the first time step, or the pose is initialized by the estimated sensor pose of the last time step. Considering prior knowledge about the starting point of the sensors or odometry information throughout the loading process would give better initial guesses. Subsequently, we optimize the factor graph with the GTSAM version of the Levenberg-Marquardt algorithm as explained in Section 2.3.5.

The final phase of one step of the pose estimation algorithm consists of the model update. The method `update_truck_configuration()` first updates the poses of all parts and all variables $s_{t'}$ for $t' < t_0$ with the current time step t_0 that are not deleted from the graph. For a part p with the old pose $T_p^{\text{old}} = (R_p^{\text{old}}, t_p^{\text{old}}) \in \mathrm{SE}_3$, the translation t_p^{old} and the rotation R_p^{old} are updated separately. Let t_p^{old} be the old position in the model, t_p^{est} the position that was estimated in phase 2 of this time step, and t_p^{new} the position that will be the new position of p in the model after this time step. Then we define

$$t_p^{\text{new}} := w t_p^{\text{est}} + (1-w) t_p^{\text{old}} \tag{3.2}$$

for some weight $w \in [0,1]$. We take the weighted average of the old and the current estimate of t_p to consider the estimated or assumed model of the truck as well as the new estimation. The estimation is based on observations that are noisy due to the restricted capabilities of the sensors and errors in the detection and preprocessing algorithms. Thus, the estimated position t_p^{est} of p is in general not equal to the true position t_p and is possibly even further away from t_p than t_p^{old}. On the other hand, the model of the truck, and therefore t_p^{old}, was initialized with a very rough approximation of the true, unknown configuration of the truck, and then updated with noisy data. So the two positions are averaged by (3.2) to hopefully compensate for each other's errors to a certain degree.

The weighting factor w is reduced over time. In the beginning, the model of the truck is assumed to be quite imprecise, so new estimations are weighted more. Throughout the loading process, the truck is observed a lot of times, so the model of the truck gets better over time. Therefore, the current model of the truck gets more weight compared to new estimations. The weight gets reduced by the function

$$w(n) = \lambda_w^n (w_0 - c) + c, \tag{3.3}$$

where $w(n)$ is the value of the weight after n time steps. The value $w_0 \in (0, 1]$ is the starting value of the function, λ_w is the reducing factor in $(0, 1)$, and $c \in [0, w_0)$ the lower bound of w. According to the computation

$$\begin{aligned} w(n) &= \lambda_w^n (w_0 - c) + c \\ \lambda_w w(n) &= \lambda_w^{n+1} (w_0 - c) + \lambda_w c \\ \lambda_w^{n+1} (w_0 - c) &= \lambda_w w(n) - \lambda_w c \end{aligned}$$

the weight $w(n + 1)$ can be computed from $w(n)$ as

$$w(n + 1) = \lambda_w w(n) + (1 - \lambda_w)c.$$

The influence of this factor λ_w is displayed in Figure 3.18b.

Updating the rotation R_p^{old} is more complicated.

Remark 3.2.1. In [5, Section 7], Brégier et al. describe a method to average two rotations in SO_3. First, we take the naive weighted average

$$Q := wR_p^{\text{est}} + (1 - w)R_p^{\text{old}}.$$

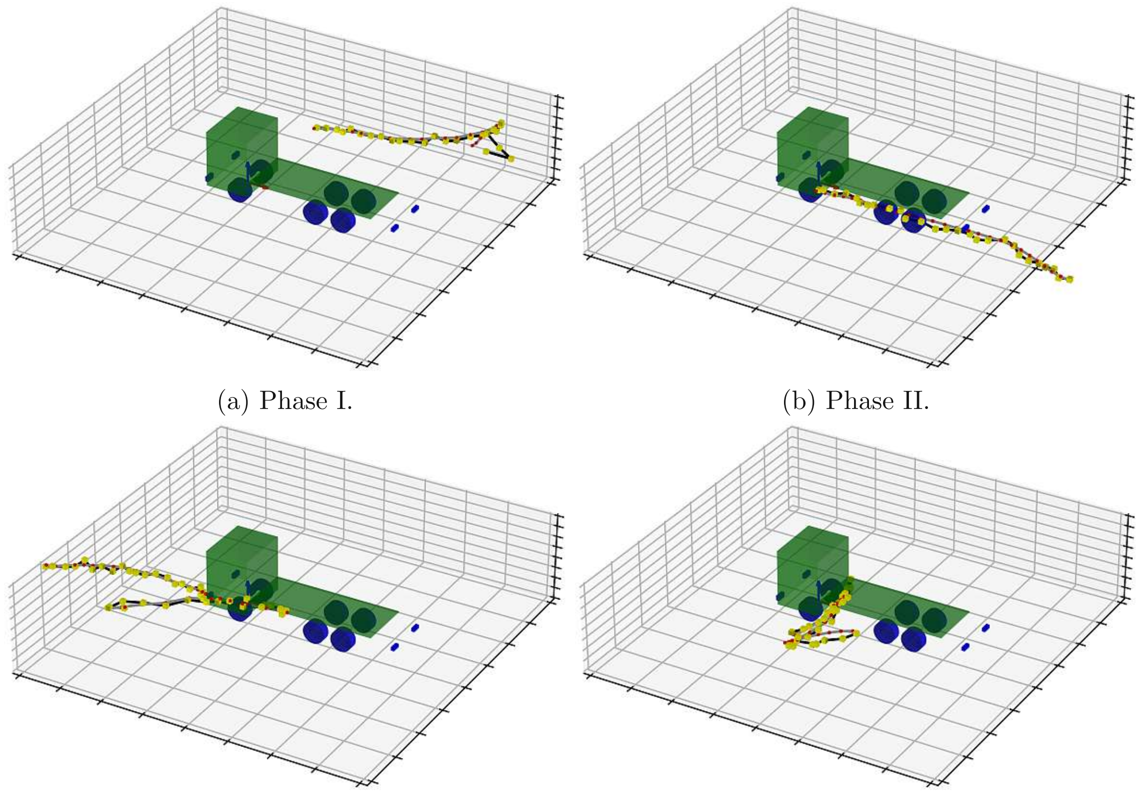
Then, we compute a singular value decomposition $Q = UDV^T$ with $U, V \in \mathrm{O}_3$ and the diagonal matrix $D = \operatorname{diag}(\alpha_1, \alpha_2, \alpha_3)$ with $\alpha_1 \geq \alpha_2 \geq \alpha_3 \geq 0$. The averaged rotation is then given by $R_p^{\text{new}} = USV^T$ with $S = \operatorname{diag}(1, 1, \operatorname{sgn}(\det(U) \det(V)))$. The matrix R_p^{new} is indeed in SO_3.

The new poses $T_p^{\text{new}} = (R_p^{\text{new}}, t_p^{\text{new}})$ for all parts p are saved in the model. Special relations between certain parts, as outlined in Section 3.2.1, are considered in two ways. First, GTSAM provides options to equip factors with noise models (e.g. in Section 2.3.5, we modeled the noise of some estimation function h to measurements or assumed relations as normally distributed around 0 with the covariance matrix Σ). The noise for respective coordinates of the construction factors between dependent parts of the truck, e.g. the noise for the x-coordinate of the factor between w_1 and W_1, is set to a small value by adapting the covariance matrix Σ of the noise accordingly. Therefore, when optimizing the factor graph, these relations are valued more than other factors like noisy observations since the probability of lying far from the measurement z is low according to the probability density functions of these construction factors defined via Σ. Nevertheless, this does not give perfect alignments of dependent parts, so they have to be aligned explicitly every few time steps. The poses of all parts of one dependency group are averaged as described above, regarding their dependent coordinate to obtain a valid model of the truck.

Finally, the model is updated and we are ready for the next time step $t + 1$, where we start by updating the factors in the factor graph $\mathcal{F}_t$ according to the new model of the truck to get the factor graph $\mathcal{F}_{t+1}$. With $\mathcal{F}_{t+1}$, we continue as described above.

3.2.3. Parametrization and Analysis of the Algorithm

The proposed algorithm for the pose estimation of a composite object works in real-time, due to efficient optimization algorithms for factor graphs provided by GTSAM. It is designed to continually receive new observations from the sensors to improve the estimated model of the truck and estimate the pose of the sensors relative to the truck to enable autonomous loading. A limiting factor for realizing this approach is the correct detection and classification of the different parts of the truck. The respective machine-learning tool, that is planned to be employed in this project, is still in development. Since time is not a crucial aspect of this approach, we want to analyze the influence of different parameters on the outcome of the model.



(a) Phase I. (b) Phase II.

(c) Phase III. (d) Phase IV.

Figure 3.15.: Estimated sensor trajectories in four different phases: The yellow cubes indicate the estimated sensor positions and the small red cubes indicate the true sensor positions.

As stated above, there are still missing parts in the whole project of automated loading of a truck. Therefore, it was not possible to generate the data under conditions that replicate those present in a real-world deployment of the pose estimator. With a ZED Camera attached to a forklift, we recorded four sequences of a truck while driving around the truck. The blender model of the truck was then fitted with GeoTracker for Blender from KeenTools [35] into the scene and tracked throughout the recordings. We define a time step as one second. For each time step, the poses of the different parts of the truck that were visible for the sensor (i.e. within the field of view and not hidden by the rest of the truck) in the respective sensor coordinate frames were extracted and used as the testing data. Figure 3.15 shows the estimated trajectories of the four different loading scenarios. In Phases II and IÍI, the truck was approached once from the back and once from the front. In Phase I, the truck was viewed from the other side, and Phase IV simulated a loading process.

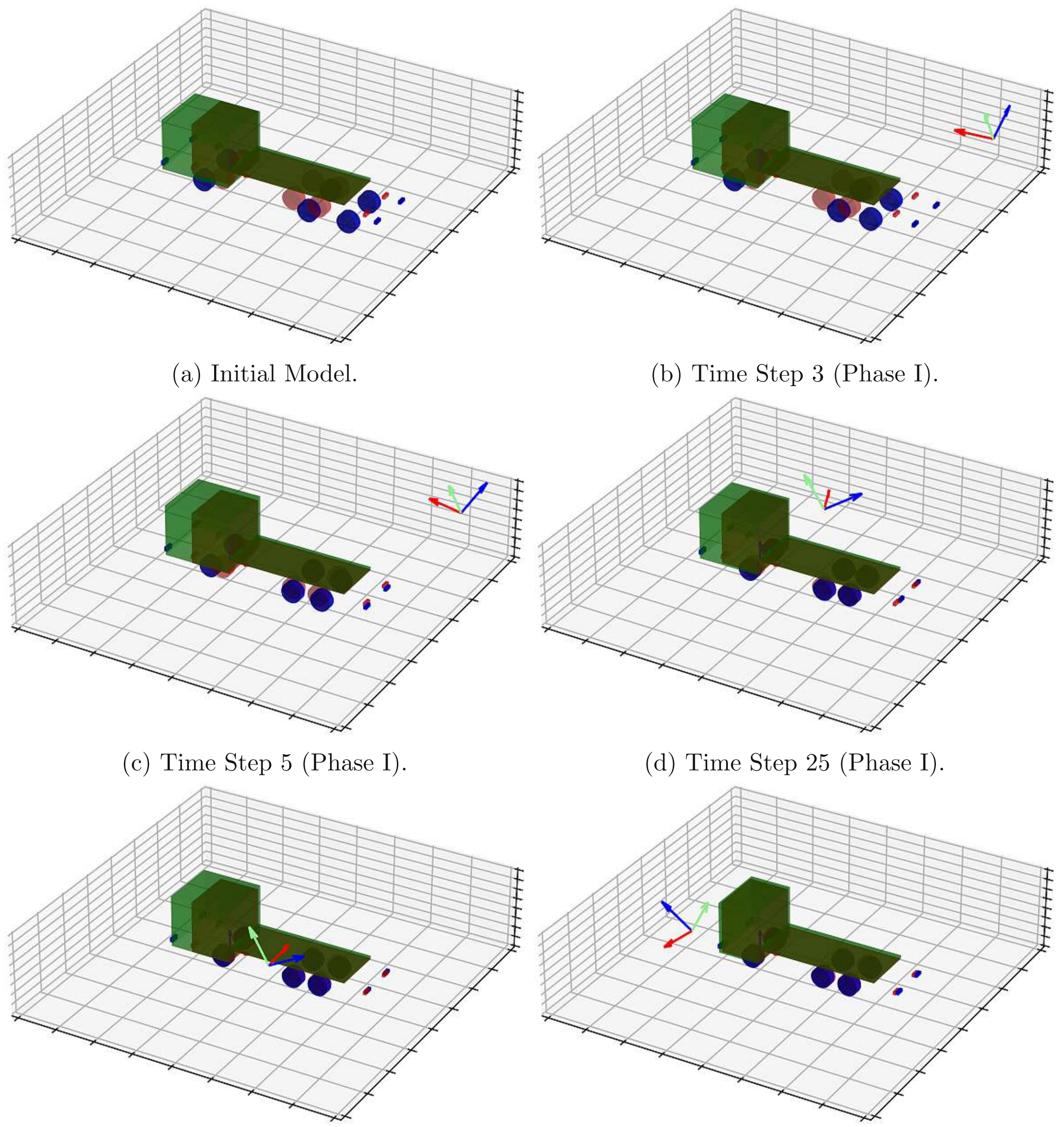
But first, we want to see how our pose estimation approach updates the model of the truck. Figure 3.16 shows some time steps in the pose estimation process. We start with a very rough approximation of the model as displayed in Figure 3.16a. The reddish truck pictures the assumed ground truth of the model, the blue cylinders portray the estimated (or initial) poses of the wheels, the blue cuboids portray the estimated (or initial) poses of the front and the rear lights, and the green cuboids picture the estimated (or initial) poses of the loading platform and the driver's cabin (for illustration purposes). The three arrows at the beginning of the left loading edge visualize the origin of the world coordinate frame. The other three arrows show the estimated pose of the sensor at that time step. As usual, the red arrow denotes the x-axis, the green arrow denotes the y-axis, and the blue arrow denotes the z-axis. Since the poses of the parts were annotated with Blender [9], the camera looks in the negative z-direction. At time step 5 in 3.16c, the rear wheels and lights are already approximately at the correct positions. Their positions become more accurate when the sensor detects the rear lights again in Phase II as seen in 3.16e. In Phase III, the front lights are observed for the first time, thus, Figure 3.16f already provides a good model.

We analyze several different parameters and their influence on the model and the estimation of the sensor pose. To measure the performance of the algorithm with a certain parameter set, we use the metric d on SE_3 defined in Theorem 2.1.31. We define the *cumulated model error* $\mathrm{err}^{(t)}$ in time step t as

$$\mathrm{err}^{(t)} := \sum_{p \in P} d(T_p^{(t)}, T_p^{\mathrm{true}}),$$

where P denotes the set of all parts of the truck, $T_p^{(t)} \in \mathrm{SE}_3$ defines the pose of part $p \in P$ at time step t, and T_p^{true} defines the pose of the part p in the assumed ground truth. We take a look at the initial weights for updating the translation and rotation of a part, at the construction and observation noise, and at the factors for reducing the construction noise and the weights. For further descriptions of these parameters, see Section 3.2.2.

For each parameter λ at a time, we analyze the cumulated model error $\mathrm{err}^{(t)}$ for the test data described above by setting λ to different values, assuming the rough initial model as in the first picture of Figure 3.16, and passing the data to the pose estimator multiple times, each time with some additional random noise on the initial model and the observations. For the initial model, we add random numbers of the normal distribution $\mathcal{N}(0, 0.5^2)$ to certain distances of parts of the model (e.g. the distance between wheels w_1 and w_2). To add noise to the rotations, we take a vector $\xi \in \mathbb{R}^3$ with entries sampled from $\mathcal{N}(0, 0.02^2)$ and update the rotation R with the local update $\mathcal{R} \oplus \xi$ of Definition 2.3.6. The standard deviation for the rotation is rather small since the parts of trucks have fixed rotations to



(a) Initial Model.

(b) Time Step 3 (Phase I).

(c) Time Step 5 (Phase I).

(d) Time Step 25 (Phase I).

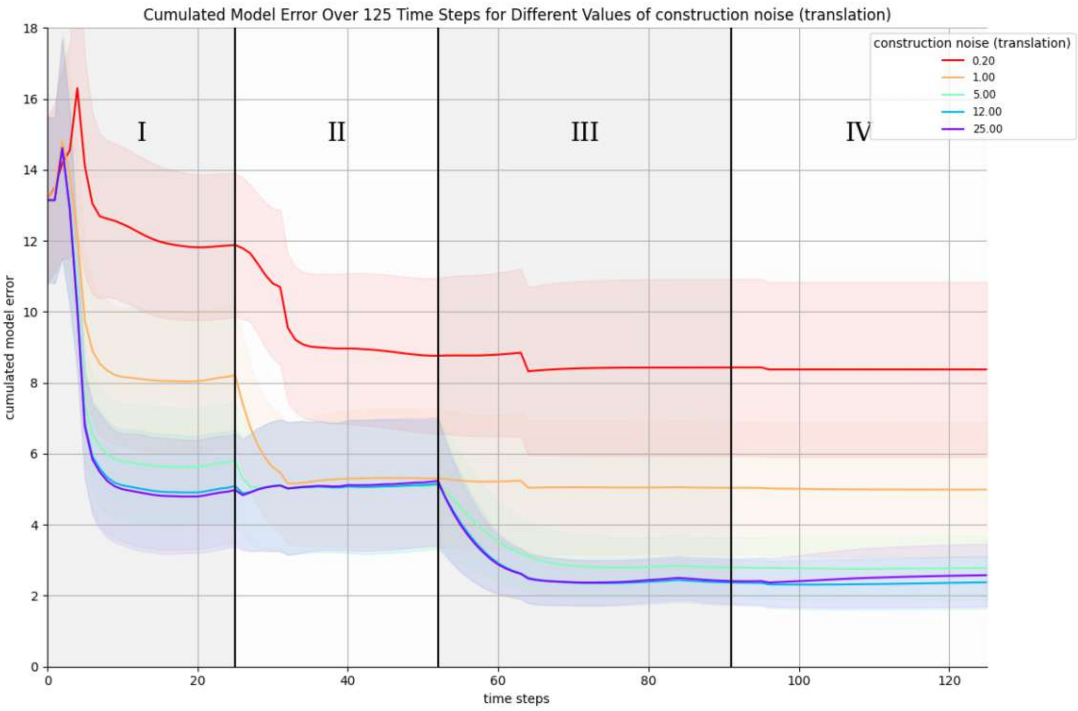
(e) Time Step 47 (Phase II).

(f) Time Step 67 (Phase III).

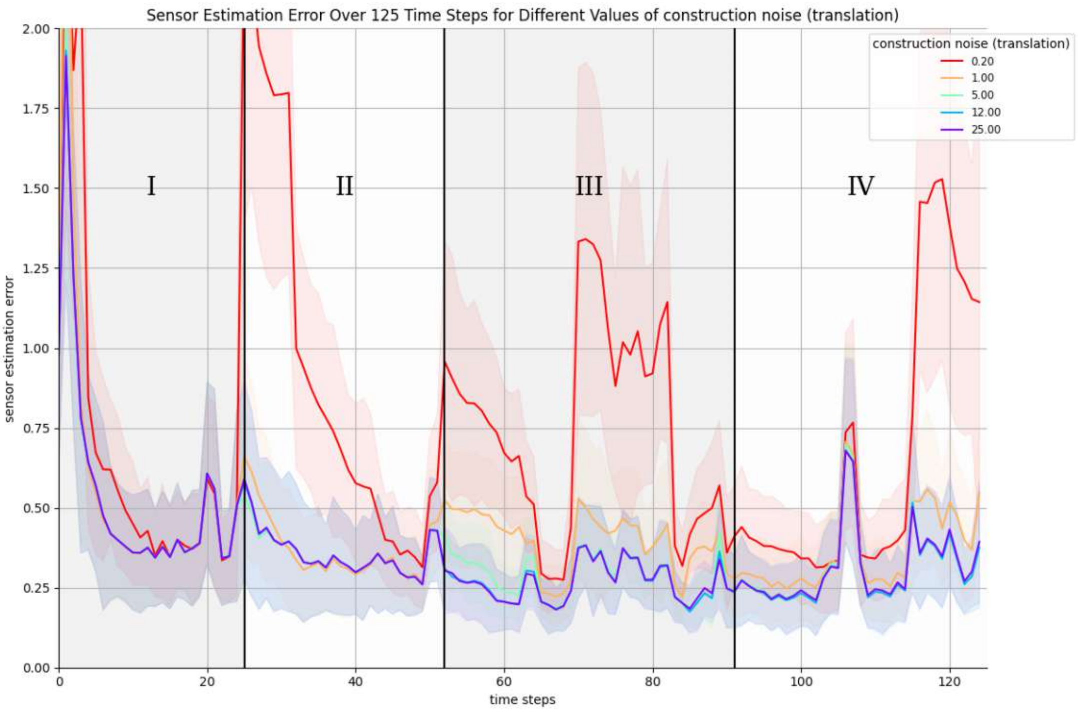
Figure 3.16.: Estimated model (green and blue) and ground truth (reddish) of the truck for a pose estimation run of Phases I to IV.

each other, regardless of the specific truck configuration, e.g. the two loading edges are always parallel and the lights are always oriented directly to the front and the back. The observations are treated similarly but with a standard deviation of 0.1 on the translation and 0.05 on the rotation.

We start with the different noise models that are used here. In Section 2.3.5 it is stated that the probability density of a factor in a factor graph can be interpreted as a noise model.

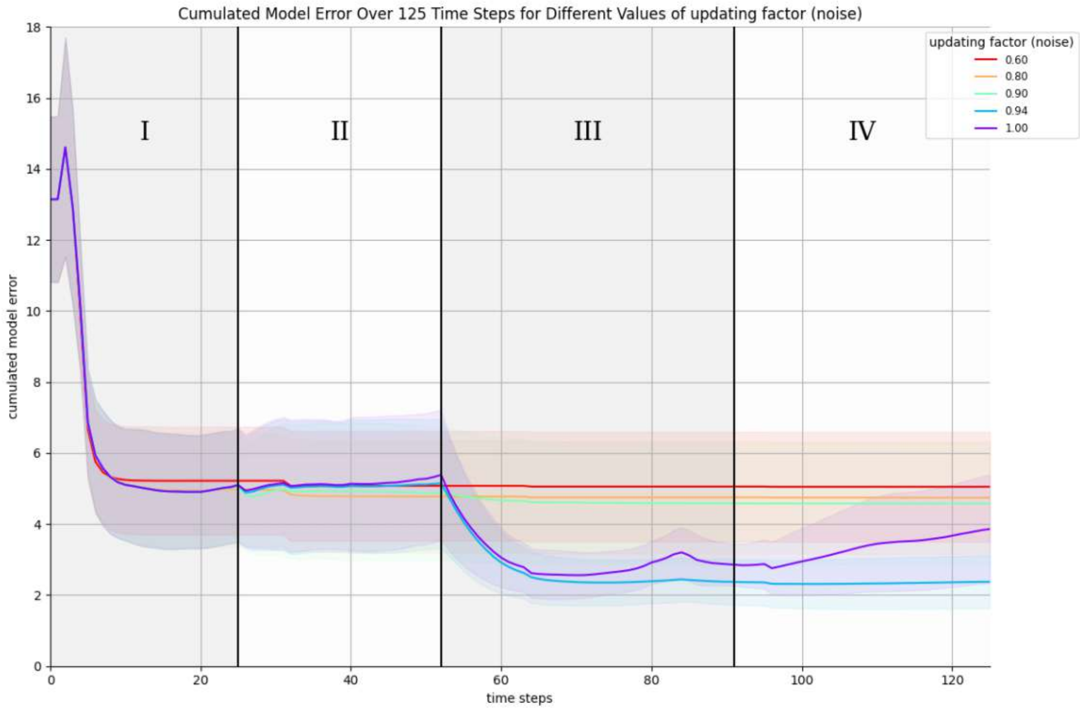


(a) Model error.

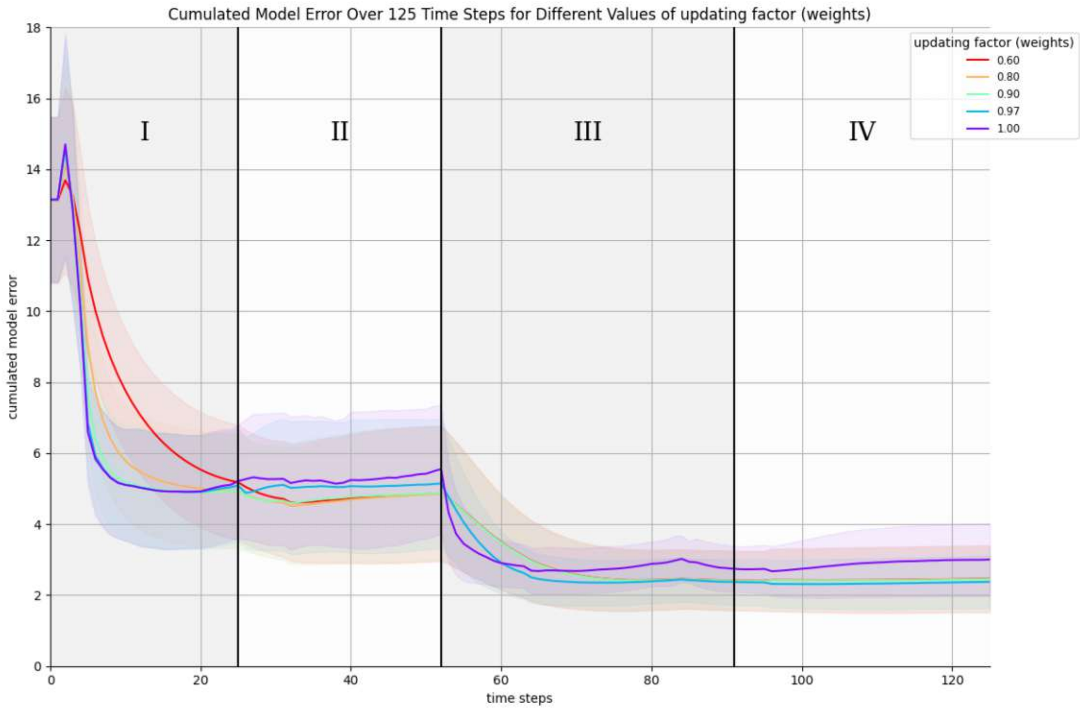


(b) Sensor estimation error.

Figure 3.17.: Model and sensor estimation error depending on the noise for the translation of the construction factors.



(a) Model error depending on the updating factor λ_c for the noise of the construction factors.



(b) Model error depending on the updating factor λ_w for the weight w of model updates.

Figure 3.18.: Model error for different values of the updating factors λ_c and λ_w.

In GTSAM, each factor is initialized with a noise model. In the created factor graph (see Figure 3.14), we distinguish among the *construction factors* between two different parts of the truck, that are defined by the currently estimated model, and the *observation factors* $o_p^{(t)}$ defined by the observations of different parts.

The construction factors express in the beginning a rough model with additional noise. Thus, we need higher values for the construction noise than the standard deviation of the additional noise. Figure 3.17a displays the model error for a few different values of the construction noise. The Regions I, II, III, and IV indicate the four different sequences of the sensor movement around the truck as seen in Figure 3.15. First, the sensor approaches the truck from the back right, then from the back left, then from the front left, and in Phase IV, the forklift simulates the loading of a palette. This is a reasonable real-life scenario since the forklift moves around near the truck before it starts loading stuff. Thus, the model must be quite accurate in Phase IV, because when the forklift loads palettes onto the truck, the sensors are too close to detect many parts.

For each parameter value, the computation was simulated 50 times. The lines display the arithmetic mean of the model errors for a certain parameter value. The shaded areas around the lines indicate the corresponding empirical standard deviations. We see that if the construction noise is too small, the factor graph relies too much on wrong assumptions and fails to optimize the model properly. If the construction noise is too high towards the end, the model error rises again since the already well-adapted model is loaded with too much uncertainty and new noisy observations make the model worse.

Figure 3.17b was created for the same situation but displays the error of the sensor estimation measured as $d(T_{s_t}^{\text{est}}, T_{s_t}^{\text{true}})$ with the metric d on SE_3 of Theorem 2.1.31, the estimated sensor pose $T_{s_t}^{\text{est}}$ at time step t, and the corresponding ground truth $T_{s_t}^{\text{true}}$. This gives similar results on good values for the construction noise. In Region IV, around the timesteps 105 to 110, we see a short rise in the estimation error. At that time, the forklift placed the load on the truck and detected almost no parts of the truck.

These considerations indicate that the noise for the construction factors should change over time. In the beginning, a high construction noise ensures a flexible model. Towards the end, we already have a good approximation of the model, so a low construction noise ensures that we consider this model enough. Therefore, we introduce an updating parameter λ_c for the construction noise similar to the updating parameter λ_w in (3.3) and analyze λ_c in Figure 3.18a. If this factor is too low, the construction noise gets too small too fast, and the model does not update properly anymore. If λ_c is too high or even set to 1 on the other hand, the effect of a rising model error towards the end, as described above, occurs. For the updating factor λ_w of the weights for updating the model, we observe similar results as displayed in Figure 3.18b.

We set the standard deviation of the artificial noise on the translation part of the observations to 0.1. Considering the noise occurring in the data generation process, this noise is probably a bit higher than 0.1. Indeed, as seen in Figure 3.19, for a value of 0.17, the model gets updated optimal. If the observation noise is too low, the noisy observations are considered too much. If the observation noise is too high, at some point, the model cannot be improved anymore.

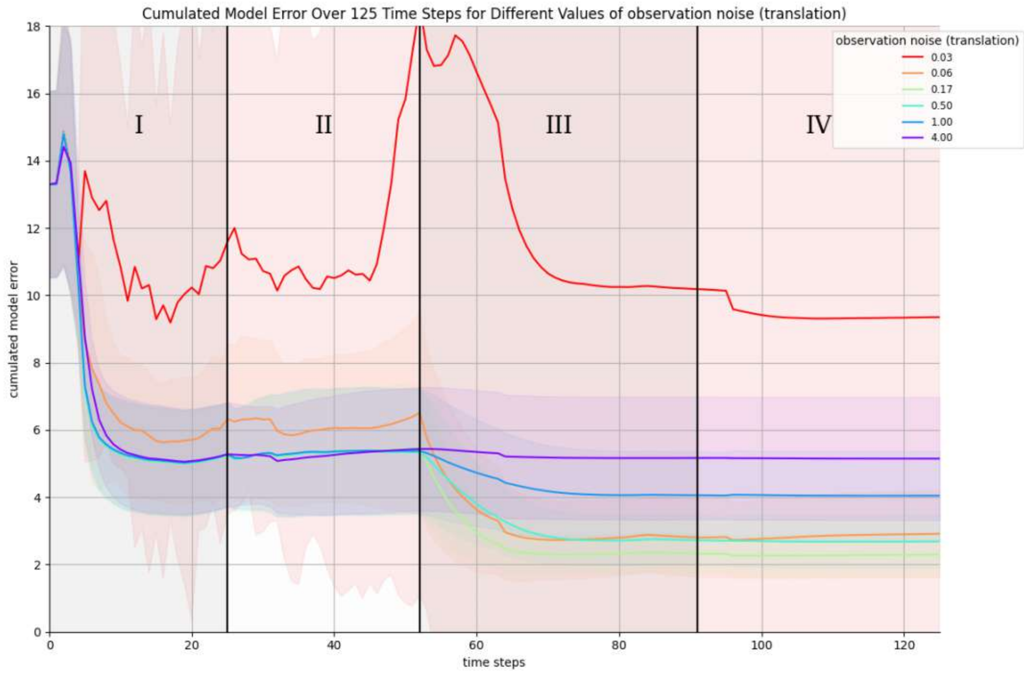


Figure 3.19.: Model error depending on the noise for the translation of the observation factors.

3.3. Possible Improvements and Further Work

There are several directions, that go beyond the scope of this master's thesis, to extend the described workflow (especially the pose estimator). The greatest part is the integration of the loading edge detector and the pose estimator in the bigger project at AIT. Once the relevant tools, such as the machine-learning tool for detecting and classifying various parts, are completed, the code and parameters must be adjusted to meet the requirements of the new data and any downstream algorithms that rely on the pose estimator.

Furthermore, the pose estimator could be adapted to better fit real-life situations. For instance, we could introduce and test objects with certain degrees of freedom. The orientation of the front wheels depends on the current steering angle and is therefore flexible throughout the algorithm. Additionally, the height of the loading platform depends on the weight of the load placed on it. Degrees of freedom could be realized by special factors or noise models that express these specific properties. Likewise, the algorithm could be adapted to accept incomplete poses, e.g. the loading edge is correctly detected, but the beginning of the loading edge cannot be determined. Similar to poses with degrees of freedom, this could be handled by adapting the observation noise in the corresponding directions. Considering the short rise of the estimation error of the sensor pose when the load is being placed on the truck (as seen between time steps 105 and 110 in Figure 3.17b) one might pose the question of how to handle the situation if only little or no parts of the truck can be detected. Here, we cut the parts of the video, where no parts of the truck were observed. In real applications, a system switching between this pose estimator and other methods of navigating around the truck could be employed. For instance, GPS and odometry measurements could be taken into account in the pose estimator as well as in other methods. With all these improvement ideas, one has to be careful to maintain an efficient algorithm.

With tools that generate data for this algorithm automatically, we are faced with the problem of incorrect data. For example, the classification of the wheels could be mixed up, mislabeling the front wheel as w_3 or a wheel on the left side as W_2. Moreover, one has to be prepared for a completely wrong detection, where the data generating tool detects a street lamp as some light of the truck or some circular load as a wheel of the truck. Since the factor graph already carries a probability structure, we could estimate how likely certain detections of different parts of the truck are, and consequently discard wrong detections before estimating a new model. More comprehensive testing of various situations and different trucks is necessary to ensure a responsible and practical implementation of this technology in real life.

Lastly, one could test the concept of detection of composite objects on objects other than the truck. A bicycle is composed of two wheels (one with a certain degree of freedom), a handlebar, a saddle, a frame (consisting of cylinder-like shapes), pedals, lights, a chain, and so on. Depending on the use case and available sensors, one could go into detail or stick to fewer and simpler objects. A robot arm loading different kinds of bicycles in some compartments might need to detect bicycles without knowing their specific configurations.

Thus, there are various related problems and use cases this work could be extended to.

4. Conclusion

In this thesis, we developed and analyzed two steps of an automated truck-loading process and examined the mathematical foundations of the used methods.

The loading edge detection algorithm is based on the geometric properties of points in point clouds. First, the point cloud is downsampled with voxel downsampling and outlier removal. This downsampling process is a tradeoff between the efficiency and robustness of the algorithm. For a small voxel size, the downsampling of the point cloud and all steps of the algorithm that are executed on all points in the corresponding point cloud take a lot of time. If the voxel size is too big, too much information about the truck gets lost and the error of the detected loading edge to the ground truth rises, while the success rate of the algorithm shrinks.

Then, the resulting point cloud is cut to the strip of points containing the loading platform, using the estimated height and angle of the camera. A test on the width of this strip showed the importance of this cutting step, and therefore the importance of accurate estimates of the height and angle of the camera in this approach. The camera sensors produce noisy data, forcing us to relax the conditions in the search for edge points. Subsequently, the risk of declaring points as edge points incorrectly rises. So, if the search width is smaller, more edge-like structures in the point cloud are cut away.

The edge point detection is then executed on every n-th point in the remaining strip. This random downsampling factor n has proven to have a great influence on the running time of the algorithm. In real-life applications, this parameter has to be adapted to the density of the point cloud (depending on the sensors and the previous downsampling steps). Some testing under the actual circumstances is necessary to find the range of this parameter, where the algorithm runs efficiently, but the success rate stays high and the error stays low. This loading edge detection algorithm can be used in the data generation process for the pose estimator.

The pose estimator takes observations of some parts of the truck and tries to reconstruct the truck's configuration from a rough initial model as well as the pose of the sensor relative to the truck. Therefore, we considered the truck as a composite object, composed of its wheels, lights, and loading edges, and represented this structure as a factor graph. A rough initial approximation of the spatial relations of the different parts of the truck to each other translated to the construction factors between the respective variables.

The spatial relations were given as rigid transformations in SE_3. We have seen that SO_3 and SE_3 are smooth manifolds, allowing the use of manifold optimization techniques in this pose estimation problem. The retraction for these manifolds was defined via the exponential map for matrices. The power series of the exponential map has for skew-symmetric matrices in $\mathbb{R}^{3\times 3}$ an explicit representation as Rodrigues' formula. For manifold optimization problems on a higher dimensional manifold SO_n for $n > 3$, either an approximation of the exponential map has to be used or different retractions have to be considered to obtain efficient optimization algorithms.

We visualized the updating process of the model. After a small number of time steps, the estimated poses of the observed parts got close to the ground truth. To model certain dependencies of parts on each other, the probability densities of the respective factors have to be adapted accordingly. Due to this functionality, the estimations of the poses of the wheels of the truck on both sides improved even while the camera stayed on one side of the truck observing only the right wheels.

The uncertainty of the model was implemented via the construction noise. Tests on this parameter showed that the construction noise has to be high enough to reflect the uncertainties of the construction factors. Especially at the beginning of the pose estimation process, the model is only a rough estimation of the real truck configuration. The model of the truck improves with every step, therefore, the construction noise needs to be carefully reduced throughout the loading process. If the construction noise is reduced too quickly or too slowly, we receive high model errors towards the end of the optimization process as the respective test showed.

The observations of the parts of the truck are represented by the observation factors in the factor graph. The corresponding observation noise should reflect the errors of the sensors and the preprocessing algorithms. For the construction and the observation noise, more testing on real data in various scenarios is required. The estimation of the sensor pose depends on the accuracy of the model and the number and quality of observations. We have seen that during the actual loading process, the sensors get so close to the truck that little to no parts can be observed. This leads to high sensor pose estimation errors during this phase. For real-life applications, different methods or additional sensors pointing in different directions have to be considered during the loading phase.

We presented a new application of factor graph modeling. The truck serves as an example of this composite object detection approach. For a full proof of concept, more tests on different composite objects have to be made.

A. Code Loading Edge Detection

```python
import numpy as np
import open3d as o3d
import time


def run_loading_edge_detection(filename, inputs):
    """
    Run the loading edge detection.

    Parameters
    ----------
    filename : str
        Name of the file containing the point cloud.
    inputs : dict
        Dictionary containing some custom parameter values.

    Returns
    -------
    loading_edge : ndarray of shape (2,3)
        Two points that define the loading edge.
    other_edge : ndarray of shape (2,3)
        Two points that define the second edge.
    running_time : float
        Duration of the loading edge detection.
    num_edges_found : int
        0, 1, or 2, dependent on the number of edges found.
    """
    # read in the point cloud
    pcd = o3d.io.read_point_cloud(filename)

    # define an instance of the class with the desired parameters
    led = LoadingEdgeDetection(o3d_cloud=pcd, **inputs)

    # perform voxel downsampling
    voxel_downsampled_pcd = led.voxel_downsample_pcd(led.pcd, voxel_size=0.015)

    # remove the outliers
    final_pcd = led.remove_outliers(voxel_downsampled_pcd, nb_points=30, radius=0.05)

    # find the loading edge
    loading_edge, other_edge, running_time, num_edges_found = led.find_two_edges(final_pcd)

    return loading_edge, other_edge, running_time, num_edges_found


class LoadingEdgeDetection:
    def __init__(
        self,
        o3d_cloud,
        est_angle_camera=32.5,
        est_height_camera=2.05,
        est_height_loading_platform=1.35,
        search_width=0.5,
        tol_orthogonal=0.3,
```

```
        max_lines=5,
        uniform_downsample_factor=40,
    ):
        """
        Constructor.

        Parameters
        ----------
        o3d_cloud : open3d PointCloud
            Point cloud, where the loading edge should be detected.
        est_camera_angle : float, optional
            Estimated camera angle in degree to the horizontal plane. The default is 32.5.
        est_height_camera : float, optional
            Estimated height of the camera above the ground in meters. The default is 2.05.
        est_height_loading_platform : float, optional
            Estimated height of the loading platform above the ground in meters. The default is 1.35.
        search_width : float, optional
            Width of the remaining point cloud, when cutting the pcd above and below
            the estimated plane of the loading platform. The default is 0.5.
        tol_orthogonal : float, optional
            Tolerance when searching for orthogonal lines. Has to be between 0 and 1.
            The default is 0.3.
        max_lines : int, optional
            Maximum number of lines to be found in 'line_ransac'. The default is 5.
        uniform_downsample_factor : int, optional
            Downsample factor when reducing the number of points for the edge point search.
            The default is 40.

        Attributes
        ----------
        pcd : open3d PointCloud
            Original point cloud of the scene.
        points : ndarray of shape (num_points, num_dimensions)
            Points of the original point cloud as numpy array.
        est_height_camera : float
            Estimated height of the camera above the ground in meters. Used to estimate the pose
            of the loading platform.
        est_height_loading_platform : float
            Estimated height of the loading platform above the ground in meters. Used to estimate
            the pose of the loading platform.
        up_vector : ndarray of shape (3,)
            Estimated up-vector of the scenery. Computed with the estimated angle of the camera.
        point_loading_platform_below_camera : ndarray of shape (3,)
            Point in the estimated height of the loading platform, vertically below the camera.
            Computed by the estimated up-vector and height of the camera and the loading platform.
        search_width : float
            Width of the remaining point cloud, when cutting the pcd above and below the estimated
            plane of the loading platform. Smaller search width speeds up the computation and
            eliminates other edge points. Can be decreased, if angle and height estimates are good.
        tol_orthogonal : float in [0, 1]
            Tolerance when searching for orthogonal lines. It holds:
                v is orthogonal to w  <=>  v @ w = 0.
            Normalized vectors v and w are approximately orthogonal, if v @ w < tol_orthogonal.
        max_lines : int
            Maximum number of lines to be found in 'line_ransac' before the algorithm stops.
            Larger max_lines is more likely to find a second edge,
            but also more likely to declare a line as an edge, that is not an edge.
        origin : ndarray of shape (3,)
            Origin of the world coordinate frame, defined in the method find_origin(). The origin of
            the world frame is defined as the beginning of the left loading edge. If the truck is seen
            from the right side, 'origin' states the beginning of the right loading edge.
        coordinate_frame : dict
            Contains the directions of the coordinate axes of the world frame, defined in the method
```

```
            rotate_coordinate_frame(). The keys are 'x', 'y', and 'z'.
        uniform_downsample_factor : int
            The edge point detection is only performed on some points, to save computation time.
            For a downsampling factor of n, every n-th point is investigated.
        """
        # define the point cloud
        self.pcd = o3d_cloud

        # define the estimated height of the camera and the loading platform
        self.est_height_camera = est_height_camera
        self.est_height_loading_platform = est_height_loading_platform

        # compute the estimated up-vector of the scenery from the estimated camera angle
        # and the point below the camera, in the estimated height of the loading platform
        self.up_vector = self.get_up_vector(est_angle_camera)
        self.point_loading_platform_below_camera = self.get_point_loading_platform_below_camera()

        # define additional parameters
        self.search_width = search_width
        self.tol_orthogonal = tol_orthogonal
        self.max_lines = max_lines
        self.coordinate_frame = {}
        self.uniform_downsample_factor = uniform_downsample_factor

    #############################################################################################
    # Point Cloud Processing
    #############################################################################################

    def cut_pcd_with_plane(self, pcd, weights, dist_to_original_plane=0.5, inside=False):
        """
        Cut a point cloud with a plane.
        Use this function, to speed up computation by restricting the search space.

        Parameters
        ----------
        pcd : open3d PointCloud
            Point cloud to be cut.
        weights : list of float
            Weights a, b, c, d of the plane: a*x + b*y + c*z + d = 0.
        dist_to_original_plane : float, optional
            Distance of the planar cut to the original plane. The default is 0.5.
        inside : bool, optional
            If True, the points inside the cut region are returned.
            If False, the points outside the cut region are returned.
            The default is False.

        Returns
        -------
        filtered_pcd : open3d PointCloud
            Cut point cloud.
        """
        # get the indices of the points inside of the cut region
        filtered_indices = self.cut_pcd_with_plane_indices(
            self.pcd_to_np(pcd), weights, dist_to_original_plane, inside
        )

        # select the points with the respective indices
        filtered_pcd = pcd.select_by_index(filtered_indices)

        return filtered_pcd

    def cut_pcd_with_plane_indices(self, pcd_np, weights, dist_to_original_plane, inside):
        """
```

```
        Computes the indices of the points inside or outside the cut region for cut_pcd_with_plane().

        Parameters
        ----------
        pcd_np : ndarray of shape (num_points, num_dimensions)
            Point cloud as numpy array.
        weights : list of float
            Weights a,b,c,d of the plane: a*x + b*y + c*z + d = 0.
        dist_to_original_plane : float
            Distance of the planar cut to the original plane.
        inside : bool
            If True, the points inside the cut region are returned.
            If False, the points outside the cut region are returned.

        Returns
        -------
        list of int
            Indices of the points inside or outside the cut region.
        """
        # transpose the array to ease computation (pcd_np[0] is a vector of all first coordinates)
        pcd_np = pcd_np.T

        # compute the values of the plane equation for all points as a numpy array
        values = weights[0] * pcd_np[0] + weights[1] * pcd_np[1] + weights[2] * pcd_np[2] + weights[3]

        # cut in both directions
        above = values > np.abs(dist_to_original_plane)
        below = values < -np.abs(dist_to_original_plane)

        # return the indices of the points inside or outside the cut region
        if not inside:
            return [i for i in range(len(above)) if above[i] or below[i]]
        else:
            return [i for i in range(len(above)) if not above[i] and not below[i]]

    """
    Additional functions, not shown here:
        def divide_inlier_outlier(self, pcd, inlier_ind, color)
        def voxel_downsample_pcd(self, pcd, voxel_size)
        def remove_outliers(self, pcd, nb_points, radius)
        def pcd_to_np(self, pcd)
        def draw_point_cloud(self, list_of_pcds, show_normals, estimate_normals)
    """

    ##############################################################################################
    # Truck Specific Functions
    ##############################################################################################

    def get_up_vector(self, camera_angle):
        """
        Compute the estimated up-vector (up-direction of the scenery in the world frame) from the
        estimated camera angle as np.array([np.sin(-camera_angle), 0, np.cos(-camera_angle)]).

        Parameters
        ----------
        camera_angle : float
            Estimated camera angle in degrees to the horizontal plane.

        Returns
        -------
        up_vector : ndarray of shape (3,)
            Approximate up-vector.
        """
```

```python
        # z-axis goes up (x to the front, y to the side)
        up_vector_camera = np.array([0, 0, 1])

        # angles to rotate around (rotate around the y-axis) in radians
        angles = np.deg2rad(np.array([0, -camera_angle, 0]))

        # rotation matrix
        rot_matrix = o3d.geometry.get_rotation_matrix_from_axis_angle(angles)

        return rot_matrix @ up_vector_camera

    def get_point_loading_platform_below_camera(self):
        """
        Computes the point below (in the direction of the estimated up-vector) the camera
        that lies in the estimated height of the loading platform.

        Returns
        -------
        point : ndarray of shape (3,)
            Estimated position of the point in the height of the loading platform, below the camera.
            Given in sensor frame coordinates.
        """
        # the camera defines the origin of the sensor frame
        pos_camera = np.array([0, 0, 0])

        # height difference between camera and loading platform
        height_diff_camera_loading_platform = (
            self.est_height_camera - self.est_height_loading_platform
        )

        return pos_camera - height_diff_camera_loading_platform * self.up_vector

    def cut_pcd_around_loading_platform(self, pcd):
        """
        Cuts a point cloud above and below the estimated loading platform. The normal vector of the
        plane, the height of the plane, and the width of the cutted point cloud are set as attributes.

        Parameters
        ----------
        pcd : open3d PointCloud
            Input point cloud.

        Returns
        -------
        open3d PointCloud
            Output point cloud.

        """
        # equation of the plane: nX = nP -> nX - nP = 0
        # the up-vector defines the normal vector n of the plane
        weights = list(self.up_vector)

        # append -nP to the weights
        weights.append(-self.up_vector @ self.point_loading_platform_below_camera)

        # cut the point cloud above and below the plane according to the search width
        return self.cut_pcd_with_plane(pcd, weights, self.search_width / 2, True)

    def find_origin(self, P_line1, P_line2):
        """
        Find the origin of the coordinate frame as the intersection point of two given lines.
        These lines do not intersect exactly, so we intersect one of the lines with the plane that is
        defined by the other line and the vertical direction.
```

```
        Then we take the average of the two intersection points.

        Parameters
        ----------
        P_line1 : ndarray of shape (2,3)
            Two points that define the first line.
        P_line2 : ndarray of shape (2,3)
            Two points that define the second line.
        set_origin : bool, optional
            If True, the origin of the coordinate frame is set as an attribute of the class.
            The default is False.

        Returns
        -------
        origin : ndarray of shape (3,)
            Found origin of the coordinate frame.
        """
        # get the intersection points
        intersection_point1 = self.intersect_line_plane(
            P_line1, np.array([P_line2[0], P_line2[1], P_line2[0] + self.up_vector])
        )
        intersection_point2 = self.intersect_line_plane(
            P_line2, np.array([P_line1[0], P_line1[1], P_line1[0] + self.up_vector])
        )

        # compute the origin and set it as a class attribute
        origin = (intersection_point1 + intersection_point2) / 2
        self.origin = origin

        return origin

    def find_direction(self, edge_pcd, origin, P):
        """
        This function is used to find the true direction of an axis (deciding between vec and -vec)
        to find the correct coordinate frame. The correct direction is the vector from the origin
        to the endpoint of P that is further away from the origin. This is done by finding the
        endpoints of the line and comparing the distances of these endpoints to the origin.

        Parameters
        ----------
        edge_pcd : open3d PointCloud
            Point cloud that contains the points on the edge.
            This point cloud should already be the largest cluster of a found edge.
        origin : ndarray of shape (3,)
            Origin of the coordinate frame.
        P : ndarray of shape (2,3)
            Two points that define the line.
            The line defined by P does not have to be parallel to the line defined by the endpoints.
            The two endpoints might just give a rough approximation of the direction of the line.

        Returns
        -------
        P : ndarray of shape (2,3)
            Two points that define the line.
            They are the same points as the input P, but the order of the points might be switched.
        """
        # direction vector of the line
        vec = P[1] - P[0]

        # get the approximate endpoints of the line
        endpoint1, endpoint2 = self.find_endpoints(edge_pcd, max_iter=100)

        # compare the distances of the endpoints to the origin
```

```
        dist_origin_endpoint1 = np.linalg.norm(endpoint1 - origin)
        dist_origin_endpoint2 = np.linalg.norm(endpoint2 - origin)
        if dist_origin_endpoint1 > dist_origin_endpoint2:
            # if endpoint1 is further away from the origin than endpoint2
            # -> the correct approximate direction is the vector from the origin to endpoint1
            correct_direction = endpoint1 - origin
        else:
            # otherwise the correct approximate direction is the vector from the origin to endpoint2
            correct_direction = endpoint2 - origin

        # compare the correct direction to the original direction
        P_start = origin
        correct_direction /= np.linalg.norm(correct_direction)
        if correct_direction @ vec > 0:
            # if their inner product is positive, the direction is correct
            P_end = P[1]

        else:
            # otherwise the direction is wrong and has to be switched
            P_end = P[0]

        return np.array([P_start, P_end])

    def rotate_coordinate_frame(self, vec_loading_edge, vec_other_edge):
        """
        Rotate the right-handed coordinate frame such that the loading edge is the x-axis (left
        loading edge) or the y-axis (right loading edge). The vector of the loading edge defines its
        axis exactly, the z-axis is then the cross-product of vec_loading_edge and vec_other_edge,
        and the last axis is the cross-product of the z-axis and the axis of the loading edge.
        We assume, that the coordinate frame is already centered at the correct origin.
        Since we transform an orthonormal basis to another orthonormal basis,
        the rotation matrix is given by the new axes as columns.

        Parameters
        ----------
        vec_loading_edge : ndarray of shape (3,)
            The direction vector of the detected loading edge.
        vec_other_edge : ndarray of shape (3,)
            The direction vector of the other edge.

        Returns
        -------
        rot_matrix : ndarray of shape (3,3)
            Rotation matrix to the new coordinate frame.
        """
        # check if the loading edge becomes the x- or the y-axis
        if self.up_vector @ np.cross(vec_loading_edge, vec_other_edge) > 0:
            # the loading edge is the x-axis
            new_x_axis = vec_loading_edge
            new_z_axis = np.cross(new_x_axis, vec_other_edge)
            new_y_axis = np.cross(new_z_axis, new_x_axis)

        else:
            # the loading edge is the y-axis
            new_y_axis = vec_loading_edge
            new_z_axis = np.cross(vec_other_edge, new_y_axis)
            new_x_axis = np.cross(new_y_axis, new_z_axis)

        # normalize the new axes to get the rotation matrix
        new_x_axis /= np.linalg.norm(new_x_axis)
        new_y_axis /= np.linalg.norm(new_y_axis)
        new_z_axis /= np.linalg.norm(new_z_axis)
        rot_matrix = np.array([new_x_axis, new_y_axis, new_z_axis]).T
```

```
        # set the axes and the rotation matrix as attributes of the class
        self.coordinate_frame["x"] = new_x_axis
        self.coordinate_frame["y"] = new_y_axis
        self.coordinate_frame["z"] = new_z_axis
        self.rot_matrix_to_coordinate_frame = rot_matrix

        return rot_matrix

    ##############################################################################################
    # Edge Detection
    ##############################################################################################

    def find_two_edges(self, pcd):
        """
        Find two orthogonal edges in a point cloud. The first edge is found by 'find_longest_edge'.
        Then, a second edge is found that is horizontal and orthogonal to the first edge. If two edges
        are found, the function 'differentiate_edges' is used to distinguish the loading edge from
        some other edge. If necessary, the two edges are switched. The origin of the coordinate frame
        is set as the intersection point of the two edges. The orientation of the coordinate frame is
        set such that the loading platform is located in the first quadrant of the xy-plane.
        The two edges are then drawn together with the rest of the point cloud.

        Parameters
        ----------
        pcd : open3d PointCloud
            Point cloud, where the edges should be detected.

        Returns
        -------
        edge1_P : ndarray of shape (2,3)
            Two points that define the loading edge.
        edge2_P : ndarray of shape (2,3)
            Two points that define the second edge.
        time : float
            Duration of the loading edge detection.
        num_edges_found : int
            0, 1, or 2, dependent on the number of edges found.
        """
        # start the timer
        start = time.time()

        # find the longest edge in the point cloud
        edge1_P, edge1_pcd, edge1_ind, edge_points_pcd, found_longest_edge = self.find_longest_edge(
            pcd
        )

        # if there are enough edge points to find an edge, we look for a second edge
        if found_longest_edge:
            # find a second edge that is orthogonal to the first edge
            edge2_P, second_line_found = self.find_orthogonal_line(
                edge_points_pcd, P_original=edge1_P, max_lines=self.max_lines
            )

            # if a second edge is found, the coordinate frame can be set
            if second_line_found:
                num_edges_found = 2
                origin = self.find_origin(edge1_P, edge2_P)

                edge2_ind = self.get_points_on_line(pcd, edge2_P)
                edge2_pcd, _ = self.divide_inlier_outlier(pcd, edge2_ind, color=[0, 1, 0])

                edge1_P = self.find_direction(edge1_pcd, origin, edge1_P)
```

```
            edge2_P = self.find_direction(edge2_pcd, origin, edge2_P)

            # define the direction of the up-vector (currently it is up or down)
            # the origin lies below the camera, so the angle between the up-vector
            # and the vector world_origin -> sensor_origin should be smaller than 90°
            # if the angle is larger than 90°, the up-vector is pointing downwards
            if self.up_vector @ (np.array([0, 0, 0]) - origin) < 0:
                self.up_vector *= -1

            # check if edge1 or edge2 is the loading edge
            _, _, correct_order = self.differentiate_edges(pcd, edge1_P, edge2_P)

            # switch edge1 and edge2 if necessary
            if not correct_order:
                edge1_P, edge2_P = edge2_P, edge1_P
                edge1_pcd, edge2_pcd = edge2_pcd, edge1_pcd
                edge1_ind, edge2_ind = edge2_ind, edge1_ind

            # get the rotation matrix to rotate the coordinate frame
            rot_matrix, axis_loading_edge = self.rotate_coordinate_frame(
                edge1_P[1] - edge1_P[0], edge2_P[1] - edge2_P[0]
            )

            end = time.time()

            # here would be the place to visualize the results

            return edge1_P, edge2_P, end - start, num_edges_found

        else:
            num_edges_found = 1
            # if only one edge is found,
            # the second horizontal direction is determined by the first edge and the up-vector
            edge1_P = self.find_endpoints(edge1_pcd)

            # the origin is one endpoint of the first edge
            origin = edge1_P[0]

            # get the direction of the second axis
            potential_second_axis = np.cross(edge1_P[1] - edge1_P[0], self.up_vector)

            # the loading platform locates on the side of the first edge, where more points are
            lifting_factor = 0.3
            lift1 = lifting_factor * potential_second_axis
            edge1_P_lifted1 = edge1_P + lift1
            lift2 = lifting_factor * (-potential_second_axis)
            edge1_P_lifted2 = edge1_P + lift2

            num_votes_lifted_1 = len(
                self.get_points_on_line(
                    pcd, edge1_P_lifted1, tol=0.1, cluster=False, between_points=True
                )
            )
            num_votes_lifted_2 = len(
                self.get_points_on_line(
                    pcd, edge1_P_lifted2, tol=0.1, cluster=False, between_points=True
                )
            )

            if num_votes_lifted_1 < num_votes_lifted_2:
                second_axis = -potential_second_axis
            else:
                second_axis = potential_second_axis
```

```
            second_axis /= np.linalg.norm(second_axis)
            # get the rotation matrix to rotate the coordinate frame
            rot_matrix = self.rotate_coordinate_frame(edge1_P[1] - edge1_P[0], second_axis)
            end = time.time()
            # here would be the place to visualize the results
            edge2_P = np.array([origin, origin + second_axis])
            return (edge1_P, edge2_P, end - start, num_edges_found)
    def find_longest_edge(self, pcd):
        """
        Find the longest edge in a point cloud. First we find the edge points and then a line that
        contains many of these edge points.
        Parameters
        ----------
        pcd : open3d PointCloud
            Point cloud, where the edge should be detected.
        Returns
        -------
        edge_P : ndarray of shape (2,3)
            The endpoints of the dominant edge in the point cloud.
        edge_pcd : open3d PointCloud
            Point cloud that contains the points on the dominant edge.
        edge_ind : list of int
            Indices of the points on the dominant edge.
        edge_points_pcd : open3d PointCloud
            Point cloud that contains the edge points.
        edge_found : bool
            Indicates if the algorithm found an edge.
        """
        edge_found = True
        # cut the point cloud above and below the estimated plane of the loading platform
        pcd_cut = self.cut_pcd_around_loading_platform(pcd)
        # find the edge points with the normals of the points
        edge_points_ind = self.find_edge_points_normals(pcd_cut)
        # check if enough edge points were found
        if len(edge_points_ind) < 3:
            edge_found = False
            return None, None, None, None, edge_found
        # divide the cut point cloud into edge points and non-edge points
        edge_points_pcd, _ = self.divide_inlier_outlier(pcd_cut, edge_points_ind)
        # find a line that contains many of these edge points
        P, ransac_completed = self.line_ransac(edge_points_pcd)
        if ransac_completed:
            # find the points of the original point cloud on this line
            edge_ind = self.get_points_on_line(pcd, P, between_points=False)
            # divide the original point cloud into edge points and non-edge points
            edge_pcd, _ = self.divide_inlier_outlier(pcd, edge_ind, color=[1, 0, 0])
```

```
                edge_P = self.find_endpoints(edge_pcd)

                return edge_P, edge_pcd, edge_ind, edge_points_pcd, edge_found

            else:
                edge_found = False
                return None, None, None, None, edge_found

        def find_edge_points_normals(self, pcd, radius=0.1):
            """
            Find edge points of a point cloud by computing the normals of the neighbors of each point.
            If the normals can be clustered into two groups reasonably, such that the mean of one group
            is nearly parallel to the vertical direction, and the mean of the other group is nearly
            orthogonal to the vertical direction, the point is considered an edge point.
            Uses the function 'cluster_normals' to cluster the normals.

            Parameters
            ----------
            pcd : open3d PointCloud
                Point cloud, where the edge points should be found.
            radius : float, optional
                Radius around a point, where the kd-tree looks for neighbors. The default is 0.1.

            Returns
            -------
            edge_points : list of int
                Indices of the edge points.
            """
            # compute the kd-tree of the point cloud to enable fast neighbor search
            kdtree = o3d.geometry.KDTreeFlann(pcd)

            # compute the normals of the point cloud
            pcd.estimate_normals()

            # downsample the point cloud to speed up computation
            downsampled_pcd = pcd.uniform_down_sample(self.uniform_downsample_factor)

            edge_points_ind = []
            for i in range(len(downsampled_pcd.points)):
                # find the neighbors of the point
                _, point_indices, _ = kdtree.search_radius_vector_3d(downsampled_pcd.points[i], radius)

                # get the point cloud of the neighbors
                neighbor_pcd = pcd.select_by_index(point_indices)

                # cluster the normals of the neighbors into two groups
                orthogonal, _ = self.cluster_normals(neighbor_pcd)

                # if the normals could be clustered into two groups that are approximately orthogonal,
                # the point is considered an edge point
                if orthogonal:
                    edge_points_ind.append(self.uniform_downsample_factor * i)

            return edge_points_ind

        def cluster_normals(self, pcd, tol_cluster=0.4):
            """
            Helper function for 'find_edge_points_normals'. Clusters the normals of a point cloud into
            two groups. If the mean of one group is nearly parallel to the vertical direction, and the
            mean of the other group is nearly orthogonal to the vertical direction, and there is
            approximately an equal number of points in both groups, the point is considered an edge point.

            Parameters
```

```
        ----------
        pcd : open3d PointCloud
            Point cloud, where the normals should be clustered. Usually the neighbors of a point.
        tol_cluster : float, optional
            Tolerance of the inner product of the mean of the two clusters and the vertical direction.
            If the inner product is 0, the rest is orthogonal to the up-vector. The default is 0.4.

        Returns
        -------
        orthogonal : bool
            The two clusters are approximately orthogonal to each other.
        center_rest : ndarray of shape (3,)
            Mean of the cluster that is that is not the vertical direction.
            If the two clusters are not approximately orthogonal, None is returned.
        """
        # get the normals of the points in the point cloud
        normals = np.asarray(pcd.normals)

        # tolerance for the length of the difference of the normals and the vertical direction
        tol_parallel = 0.3

        # all normals that are NOT approximately vertical (parallel to up_vector)
        rest = normals[
            np.logical_and(
                np.linalg.norm(normals - self.up_vector, axis=1) > tol_parallel,
                np.linalg.norm(normals + self.up_vector, axis=1) > tol_parallel,
            )
        ]

        # if there are too little normals of one kind, this is not an edge point
        if 4 * len(rest) > len(normals) and 4 * len(rest) < 3 * len(normals):
            # compute the mean and the norm of the rest
            center_rest = np.sum(rest, axis=0) / len(rest)
            norm_rest = np.linalg.norm(center_rest)

            # tolerance for the length of the mean of the rest
            tol_norm = 0.4

            # if the norm is too short, the normals are too far away from each other -> return False
            if norm_rest > tol_norm:
                # normalize the mean of the rest
                center_rest /= norm_rest

                # compute the inner product of the mean of the rest and the vertical direction
                inner_product = center_rest @ self.up_vector

                # if the inner product of the centers is close to zero,
                # the clustered normals are nearly orthogonal
                if np.abs(inner_product) < tol_cluster:
                    return True, center_rest

            return False, center_rest

        return False, None

    def differentiate_edges(self, pcd, edge1_P, edge2_P, lifting_vector=None):
        """
        Differentiate between a loading edge and the edge between loading platform and rear wall.
        Key difference: above the loading edge, there should be no points -> translate the edge a bit
            in the direction of the up_vector and check, which line hits less points -> loading edge

        Parameters
        ----------
```

```
        pcd : open3d PointCloud
            Point cloud, where the edges should be differentiated.
        edge1_P : ndarray of shape (2,3)
            Two points that define the first edge.
        edge2_P : ndarray of shape (2,3)
            Two points that define the second edge.
        lifting_vector : ndarray of shape (3,), optional
            Vector in the direction of which the edges are translated. The default is None.
            If None, the up_vector is used.

        Returns
        -------
        loading_edge : ndarray of shape (2,3)
            Two points that define the loading edge.
        other_edge : ndarray of shape (2,3)
            Two points that define the other edge.
        correct_order : bool
            True, if the loading edge is edge1_P, False if the loading edge is edge2_P.
        """
        # define the lifting vector
        if lifting_vector is None:
            lifting_vector = self.up_vector

        # lift both edges by a certain factor
        lifting_factor = 0.3
        lift = lifting_factor * lifting_vector
        edge1_P_lifted = edge1_P + lift
        edge2_P_lifted = edge2_P + lift

        # compute the number of points for both edges and both lifted edges
        num_votes_lifted_1 = len(
            self.get_points_on_line(pcd, edge1_P_lifted, tol=0.1, cluster=False, between_points=True)
        )
        num_votes_lifted_2 = len(
            self.get_points_on_line(pcd, edge2_P_lifted, tol=0.1, cluster=False, between_points=True)
        )
        num_votes_1 = len(
            self.get_points_on_line(pcd, edge1_P, tol=0.1, cluster=False, between_points=True)
        )
        num_votes_2 = len(
            self.get_points_on_line(pcd, edge2_P, tol=0.1, cluster=False, between_points=True)
        )

        # compute the relative votes of the lifted edges compared to the original edges
        rel_votes1 = num_votes_lifted_1 / num_votes_1
        rel_votes2 = num_votes_lifted_2 / num_votes_2

        min_rel = 0.0001
        if rel_votes1 < min_rel and rel_votes2 < min_rel:
            # if both edges have little to no points above them, we take the longer edge
            if np.linalg.norm(edge1_P[1] - edge1_P[0]) > np.linalg.norm(edge2_P[1] - edge2_P[0]):
                loading_edge = edge1_P
                other_edge = edge2_P
                correct_order = True

            else:
                loading_edge = edge2_P
                other_edge = edge1_P
                correct_order = False

        elif rel_votes1 < rel_votes2:
            # edge1 is the loading edge
            loading_edge = edge1_P
```

```python
            other_edge = edge2_P
            correct_order = True

        else:
            # edge2 is the loading edge
            loading_edge = edge2_P
            other_edge = edge1_P
            correct_order = False

        return loading_edge, other_edge, correct_order

    ###############################################################################################
    # Geometric Functions
    ###############################################################################################

    def line_ransac(self, pcd, tol=0.08, max_iter=100):
        """
        Find a line in a point cloud by RANSAC. Typical usecase: The given point cloud consists of
        previously detected edge points, and we want to find the dominant edge, i.e. the edge
        containing the most points.

        Parameters
        ----------
        pcd : open3d PointCloud
            Point cloud, where the line should be detected.
            Usually this point cloud consists of previously detected edge points.
        tol : float, optional
            Tolerance of the distance between a point and the line. The default is 0.08.
        max_iter : int, optional
            Maximum number of iterations of the RANSAC algorithm. The default is 100.

        Returns
        -------
        best_line : ndarray of shape (2,3)
            Two points that define the line.
        finished_ransac : bool
            States if there are enough points in the point cloud to perform RANSAC.
        """
        if len(pcd.points) > 1:
            # initialize the best line and the number of votes
            best_votes = 0
            best_line = None

            for _ in range(max_iter):
                # sample the points and get the votes for this line
                P1, P2, votes = self.line_ransac_votes(pcd, tol)

                # update the best line and the number of votes if necessary
                if votes > best_votes:
                    best_votes = votes
                    best_line = np.array([P1, P2])

            return best_line, True

        else:
            return None, False

    def line_ransac_votes(self, pcd, tol, P=None):
        """
        Helper function for 'line_ransac'. Counts the number of points that lie within some small
        tolerance around a line. If P is None, two random points are sampled from pcd.

        Parameters
```

```
        ----------
        pcd : open3d PointCloud
            Point cloud, where the line should be detected.
        tol : float
            Tolerance of the distance between a point and the line.
        P : list of ndarray of shape (2,3) or None, optional
            If None, two random points are chosen and the line between them is used.
            If not None, the line between the two points is used. The default is None.

        Returns
        -------
        P1 : ndarray of shape (3,)
            First point of the line.
        P2 : ndarray of shape (3,)
            Second point of the line.
        votes : int
            Number of points that lie within the tolerance around the line.
        """
        # sample points if necessary
        if P is None:
            ind1, ind2 = np.random.randint(len(pcd.points), size=2)
            P1 = pcd.points[ind1]
            P2 = pcd.points[ind2]

        else:
            P1 = P[0]
            P2 = P[1]

        # compute the distance of each point to the line
        vecs_to_points = pcd.points - P1

        # get the vector of the line
        vec_line = P2 - P1

        # if the two points are too close to each other, we return zero votes
        norm_line = np.linalg.norm(vec_line)
        if norm_line < 1e-10:
            return P1, P2, 0

        # normalize the vector of the line
        vec_line /= norm_line

        # project the vectors to the points onto the line
        projected_length = vecs_to_points @ vec_line

        # to avoid numerical errors, we set negative values to zero
        squared_dist = np.square(np.linalg.norm(vecs_to_points, axis=1)) - np.square(projected_length)
        squared_dist[squared_dist < 0] = 0
        dist_to_line = np.sqrt(squared_dist)

        # we count the number of points that lie within some small tolerance around the line
        votes = np.count_nonzero(dist_to_line < tol)

        return P1, P2, votes

    def get_points_on_line(self, pcd, P, tol=0.05, cluster=True, between_points=False):
        """
        Returns the indices of the points of a point cloud that lie on a given line.
        If cluster=True, the indices of the largest cluster of points on the line are returned.
        If between_points=True, only the points between the two given points are returned.

        Parameters
        ----------
```

```
        pcd : open3d PointCloud
            Point cloud, where the points on the line should be found.
        P : ndarray of shape (2,3)
            Two points that define the line.
        tol : float, optional
            Tolerance of the distance between a point and the line. The default is 0.05.
        cluster : bool, optional
            If True, the indices of the largest cluster of points on the line are returned.
            If False, all the indices of the points on the line are returned.
            Only relevant, if between_points=False. The default is True.
        between_points : bool, optional
            If True, only the points between the two given points are returned.
            If False, all the points on the line are returned. The default is False.

        Returns
        -------
        points_on_line_ind : list of int
            Indices of the points on the line.
        """
        # get the two points that define the line
        P1, P2 = P[0], P[1]

        # get the distance of all points to the first point of the line
        vecs_to_points = pcd.points - P1

        # get the vector of the line and its length
        vec_line = P2 - P1
        original_len_line = np.linalg.norm(vec_line)

        # normalize the vector of the line
        vec_line /= original_len_line

        # project the vectors to the points onto the vector of the line
        projected_length = vecs_to_points @ vec_line

        if between_points:
            # consider only points between P1 and P2
            # compute the distance of the points to the line with the Pythagorean theorem
            squared_dist = np.square(np.linalg.norm(vecs_to_points, axis=1)) - np.square(
                projected_length
            )

            # avoid numerical errors and get the distance to the line
            squared_dist[squared_dist < 0] = 0
            dist_to_line = np.sqrt(squared_dist)

            points_on_line_ind = []
            for i in range(len(projected_length)):
                if (
                    projected_length[i] > 0
                    and projected_length[i] < original_len_line
                    and dist_to_line[i] < tol
                ):
                    points_on_line_ind.append(i)

            return points_on_line_ind

        else:
            # consider all points
            # compute the distance of the points to the line with the Pythagorean theorem
            squared_dist = np.square(np.linalg.norm(vecs_to_points, axis=1)) - np.square(
                projected_length
            )
```

```python
            # avoid numerical errors and get the distance to the line
            squared_dist[squared_dist < 0] = 0
            dist_to_line = np.sqrt(squared_dist)

            # get the indices of the points that lie within the tolerance
            points_on_line_ind = np.where(dist_to_line < tol)[0]

            # return the indices of the largest cluster
            if cluster:
                return self.find_largest_cluster(pcd, points_on_line_ind)
            else:
                return points_on_line_ind

    def find_largest_cluster(self, pcd, pcd_ind=None, eps=0.5, min_points=10):
        """
        Find the largest cluster of a point cloud.

        Parameters
        ----------
        pcd : open3d PointCloud
            Point cloud, where the largest cluster should be found.
        pcd_ind : list of int or None, optional
            Indices of the points in pcd that we want to find a cluster in.
            If None, we search for the largest cluster in the whole point cloud, e.g. if we want to
            find the largest cluster of a line in a point cloud, pcd_ind gives the indices of the
            points of the line in pcd. The default is None.
        eps : float, optional
            Epsilon for the DBSCAN algorithm. The default is 0.5.
        min_points : int, optional
            Minimum number of points for a cluster. The default is 10.

        Returns
        -------
        inlier_ind : list of int
            Indices of the points of the largest cluster.
        """
        if pcd_ind is None:
            # search for the largest cluster in the whole point cloud
            pcd_ind = list(range(len(pcd.points)))
        else:
            # or just in the points with the given indices
            pcd = pcd.select_by_index(pcd_ind)

        # cluster the pcd using dbscan
        cluster_labels = np.array(pcd.cluster_dbscan(eps=eps, min_points=min_points))
        # points labeled -1: noise

        # get a list of all cluster labels
        unique_labels = np.unique(cluster_labels)

        # count for each cluster label the amount of points in that label
        num_points_per_cluster = [len(np.where(cluster_labels == i)[0]) for i in unique_labels]

        # If the two largest clusters have the same amount of points, we have to decide somehow
        # between them. Generally, this is bad, because then there is no unique 'largest' cluster
        # just take the first cluster -> [0] at the end
        # (the first [0] gives us the first entry of the tuple (array,) that we get from np.where)
        max_points_per_cluster = np.max(num_points_per_cluster)
        max_label = np.where(num_points_per_cluster == max_points_per_cluster)[0][0]

        # get the label of the largest cluster
        largest_cluster_label = int(unique_labels[max_label])
```

```
        # get all the indices of pcd_ind that correspond to the points of the largest cluster
        # in the larger point cloud that pcd is embedded in
        inlier_ind = [pcd_ind[i] for i in list(np.where(cluster_labels == largest_cluster_label)[0])]

        return inlier_ind

    def find_orthogonal_line(self, edge_pcd, P_original, tol=0.08, max_iter=100, max_lines=5):
        """
        For a given line, find a line that is approximately orthogonal to it within a point cloud.
        The line is found by a version of RANSAC. At most max_lines lines are found.
        The first line that is orthogonal is returned.
        If no orthogonal line is found, the original line is returned, together with False.

        Parameters
        ----------
        edge_pcd : open3d PointCloud
            Point cloud, where the line should be detected.
            Usually this point cloud consists of previously detected edge points.
        P_orthogonal : ndarray of shape (2,3)
            Two points that define the line that is orthogonal to the line we are looking for.
        tol : float, optional
            Tolerance of the distance between a point and the line. The default is 0.08.
        max_iter : int, optional
            Maximum number of iterations of the RANSAC algorithm for one line. The default is 100.
        max_lines : int, optional
            Maximum number of lines that are tried to find. The default is 5.

        Returns
        -------
        best_line : ndarray of shape (2,3)
            Two points that define the line.
        orthogonal : bool
            True, if an orthogonal line was found.
        """
        # initialize the best line and the number of votes
        best_votes = 0
        best_line = P_original

        # define the original line
        line_original = P_original[1] - P_original[0]
        line_original /= np.linalg.norm(line_original)

        for j in range(max_lines):
            for _ in range(max_iter):
                # if the point cloud is too small, we cannot find a line
                if len(edge_pcd.points) >= 2:
                    P1, P2, votes = self.line_ransac_votes(edge_pcd, tol)

                else:
                    return P_original, False

                # update the best line and the number of votes if necessary
                if votes > best_votes:
                    best_votes = votes
                    best_line = np.array([P1, P2])

            # compute the vector of the best line
            vec_line = best_line[1] - best_line[0]
            vec_line /= np.linalg.norm(vec_line)

            # if the line is orthogonal to the other line and to the vertical direction,
            # we have found the correct line
```

```
            if (
                np.abs(vec_line @ line_original) < self.tol_orthogonal
                and np.abs(vec_line @ self.up_vector) < self.tol_orthogonal
            ):
                return best_line, True

            else:
                # delete the points on this line from the edge point cloud and start again
                new_ind = self.get_points_on_line(edge_pcd, best_line, tol=0.05)
                edge_pcd = edge_pcd.select_by_index(new_ind, invert=True)
                best_votes = 0
                best_line = P_original

        # if no orthogonal line is found within the maximum number of iterations,
        # return the original line and False
        return P_original, False

    def find_endpoints(self, edge_pcd, max_iter=100):
        """
        Find the endpoints of a line, i.e. the points on the line that are the furthest away from each
        other. This is done by a variant of RANSAC. Usually a small number of iterations is enough,
        since this does not have to be the best pair of points, a close approximation suffices.
        If necessary, increase the number of iterations given by max_iter.

        Parameters
        ----------
        edge_pcd : open3d PointCloud
            Point cloud that contains the points on the edge.
            This point cloud should already be the largest cluster of a found edge.
        max_iter : int, optional
            Maximum number of iterations. The default is 100.

        Returns
        -------
        best_P1 : ndarray of shape (3,)
            First endpoint.
        best_P2 : ndarray of shape (3,)
            Second endpoint.
        """
        # initialize best distance and best points
        best_dist = 0
        best_P1 = None
        best_P2 = None

        # find the best pair of points within max_iter iterations
        for i in range(max_iter):
            ind1, ind2 = np.random.randint(len(edge_pcd.points), size=2)
            P1 = edge_pcd.points[ind1]
            P2 = edge_pcd.points[ind2]

            new_dist = np.linalg.norm(P2 - P1)
            if new_dist > best_dist:
                best_dist = new_dist
                best_P1 = P1
                best_P2 = P2

        return best_P1, best_P2

    """
    Additional functions, not shown here:
        def intersect_line_plane(self, P_line, P_plane)
        def intersect_two_planes(self, weights1, weights2)
    """
```

```
    ##########################################################################################
    # Alternative Approach
    ##########################################################################################

    def find_edge_points_centroids(self, pcd, radius=0.09, max_nn=40, tol=0.009):
        """
        Find edge points of a point cloud by computing the centroid of the neighbors of each point.
        If the centroid is too far away from the point, the point is considered an edge point.
        Problem of this method: Finds also points at the border of the point cloud.

        Parameters
        ----------
        pcd : open3d PointCloud
            Point cloud, where the edge points should be found.
        radius : float, optional
            Radius around a point, where the kd-tree looks for neighbors. The default is 0.09.
        max_nn : int, optional
            Maximum number of neighbors to be found. The default is 40.
        tol : float, optional
            Tolerance of the distance between the centroid and the point. The default is 0.009.
            If the distance is larger than tol, the point is considered an edge point.

        Returns
        -------
        edge_points_ind : list of int
            Indices of the edge points.
        """
        # compute the kd-tree of the point cloud to enable fast neighbor search
        kdtree = o3d.geometry.KDTreeFlann(pcd)

        # downsample the point cloud to speed up computation
        downsampled_pcd = pcd.uniform_down_sample(self.uniform_downsample_factor)

        edge_points_ind = []
        for i in range(len(downsampled_pcd.points)):
            # find the neighbors of the point
            _, point_indices, _ = kdtree.search_hybrid_vector_3d(
                query=downsampled_pcd.points[i], radius=radius, max_nn=max_nn
            )

            # get the point cloud of the neighbors
            neighbor_pcd = pcd.select_by_index(point_indices)

            # compute the centroid of the neighbors
            sum_of_points = np.sum(np.asarray(neighbor_pcd.points), axis=0)
            centroid = sum_of_points / len(neighbor_pcd.points)

            # compute the distance between the centroid and the point
            dist_to_point = np.linalg.norm(centroid - downsampled_pcd.points[i])

            # if the distance is larger than tol, the point is considered an edge point
            if dist_to_point > tol:
                edge_points_ind.append(self.uniform_downsample_factor * i)

        return edge_points_ind
```

B. Code Pose Estimation

```
import gtsam
import numpy as np
import copy
import pickle


class Truck3D:
    def __init__(
        self,
        observation_noise_translation_default=0.17,
        observation_noise_rotation_default=0.14,
        construction_noise_translation_default=12,
        construction_noise_rotation_default=1.5,
        updating_factor_noise=0.94,
        updating_factor_weights=0.97,
        initial_weights_translation=0.8,
        initial_weights_rotation=0.8,
        align_every_n_steps=32,
        saving_old_factors_until_n_time_steps=50,
        **truck_configuration,
    ):
        """
        Constructor.

        Parameters
        ----------
        observation_noise_translation_default : float, optional
            Standard deviation of the noise for the translation of the observations.
            The default is 0.17.
        observation_noise_rotation_default : float, optional
            Standard deviation of the noise for the rotation of the observations. The default is 0.14.
        construction_noise_translation_default : float, optional
            Default standard deviation of the noise for the translation of the construction factors.
            Used for construction factors of parts that are in no special relation
            (dependency group or degree of freedom) with each other. The default is 12.
        construction_noise_rotation_default : float, optional
            Default standard deviation of the noise for the rotation of the construction factors.
            Used for construction factors of parts that are in no special relation
            (dependency group or degree of freedom) with each other. The default is 1.5.
        updating_factor_noise : float, optional
            Factor to update the construction noise after each time step.
            Update the default construction noise with this factor after each time step.
            The default is 0.94.
        updating_factor_weights : float, optional
            Factor to update the weights of the factors after each time step.
            The default is 0.97.
        initial_weights_translation : float, optional
            Initial weight for updating the translation of parts of the truck after a new observation.
            The default is 0.8.
        initial_weights_rotation : float, optional
            Initial weight for updating the rotation of parts of the truck after a new observation.
            The default is 0.8.
        align_every_n_steps : int, optional
            How often dependent parts of the model are aligned.
```

```
            If 0, dependent parts are never aligned.
            If 1, dependent parts are aligned after each time step.
            The default is 32.
        saving_old_factors_until_n_time_steps : int, optional
            Number of time steps, old observation factors should stay in the graph.
            The default is 50.
        **truck_configuration : dict
            Dictionary with the configuration of the truck. The keys of the dictionary are some
            features of the truck (e.g. width of the loading platform). Not all features
            have to be given (for the features that are not given, default values are used).

        Attributes
        ----------
        graph : gtsam.NonlinearFactorGraph
            Current factor graph.
        updating_factor_noise : float in (0, 1]
            Factor to update the construction noise after each time step.
            The construction noise is updated with the function update_construction_noise().
        updating_factor_weights : float in (0, 1]
            Factor to update the weights for model updates after each time step.
            The weights are updated with the function update_weights().
        lower_bound_construction_noise : float >= 0
            Lower bound for the construction noise in the updating function.
        lower_bound_weights : float in [0, 1]
            Lower bound for the weights in the updating function.
        weight_rotation : float in (lower_bound_weights, 1]
            Current weight for the updates of the rotation.
        weight_translation : float in (lower_bound_weights, 1]
            Current weight for the updates of the translation.
        observation_noise_rotation_default : float >= 0
            Standard deviation of the noise for the rotation of the observations.
        observation_noise_translation_default : float >= 0
            Standard deviation of the noise for the translation of the observations.
        construction_noise_rotation_default : float
            Default standard deviation of the noise for the rotation of the construction factors.
            Used for construction factors of parts that are in no special relation to each other.
        construction_noise_rotation_dependent : float
            Standard deviation of the noise for the rotation of the construction factors between
            dependent parts.
        construction_noise_translation_default : float
            Default standard deviation of the noise for the translation of the construction factors.
            Used for construction factors of parts that are in no special relation to each other.
        construction_noise_translation_dependent : float
            Standard deviation of the noise for the translation of the construction factors between
            dependent parts.
        align_every_n_steps : int >= 0
            How often dependent parts of the model are aligned.
            If 0, dependent parts are never aligned.
            If 1, dependent parts are aligned after each time step.
        factors_to_reuse : list of gtsam.NonlinearFactorGraph
            List of the observation factors that are saved to reuse them after each time step
            in the recreation of the factor graph. Old observations stay the same each time
            the factor graph is recreated and should be kept in the factor graph for some time.
        saving_old_factors_until_n_time_steps : int > 0
            Number of time steps, old observation factors should stay in the graph.
        time_step_factors_to_reuse : list of int
            List of the corresponding time step of the factors to reuse.
            Used to check if observations are too old and therefore deleted from the list.
        symbols_sensor : list of gtsam.Symbol
            List of the symbols for the sensor in each time step.
        sensor_trajectory : list of gtsam.Pose3
            List of the estimated poses of the sensor in the world frame.
        current_num_time_steps : int
```

```
        Number of time steps that have already been processed.
    g0, G0, w1, w2, w3, W1, W2, W3, l1, l2, L1, L2 : gtsam.Symbol
        Symbols for the variables.
        g0 ... origin of the world/global frame
        G0 ... not any origin, but to stay consistent with the naming convention
        w1, w2, w3 ... left wheels (one front wheel (w1) and two rear wheels (w2, w3))
        W1, W2, W3 ... right wheels (one front wheel (W1) and two rear wheels (W2, W3))
        l1, l2 ... left lights (one front light (l1) and one rear light (l2))
        L1, L2 ... right lights (one front light (L1) and one rear light (L2))
    symbols_parts : list of gtsam.Symbol
        List of all symbols for the variables.
    dependency_groups : dict
        Dependency groups contain parts of the truck that have a fixed spatial relation
        to each other, e.g. all wheels have the same z-coordinate in the world frame.
        Therefore, if the z-coordinate of one wheel is changed, the z-coordinates of all other
        wheels have to change as well. Given as a dictionary with the axes as keys,
        where the values are dictionarys with the dependency groups as values.
    groups_with_degrees_of_freedom : dict
        Some parts have certain degrees of freedom, e.g. the front wheels can be rotated
        around the z-axis (but both front wheels are rotated the same amount). Given as
        a dictionary with the axes as keys. The values are again dictionarys containing
        groups of symbols that are in a fixed relation to each other. A group is given as
        a dictionary containing "symbols" (list of symbols) and "std" (additional
        standard deviation of the noise model). If a single element has a degree of freedom,
        it can be added here as well (e.g. a wheel can be rotated around the y-axis).
    symbols_info : dict
        Dictionary with information about each symbol defining the model of the truck implicitly.
        For each symbol, the following information is contained:
            num_observed: integer, indicating how often the symbol has been observed.
            pose_in_world_frame: dictionary with keys "x", "y", "z", "rotation_matrix".
            dependency_groups: dictionary with keys "x", "y", "z", "roll", "pitch", "yaw"
                and the symbols in their respective group as values.
            groups_with_degrees_of_freedom: dictionary with only the necessary keys of "x", "y",
            "z", "roll", "pitch", and "yaw" and the symbols in their respective group as values.
            std_degrees_of_freedom: dictionary with the additional standard deviation for
                the respective groups with degrees of freedom.
    construction_noise_dictionary : dict
        Dictionary with the standard deviations of the noise models for the construction factors.
        This information is saved such that it does not have to be computed in each time step.
    length_loading_platform : float
        Length of the loading platform.
    true_truck_configuration : dict
        Dictionary that contains the ground truth of the truck configuration. Used for testing.
    """
    self.graph = gtsam.NonlinearFactorGraph()

    # define input parameters for noise, weights, and updating factors as attributes
    self.updating_factor_noise = updating_factor_noise
    self.updating_factor_weights = updating_factor_weights
    self.lower_bound_construction_noise = 0.1
    self.lower_bound_weights = 0.1
    self.weight_rotation = initial_weights_rotation
    self.weight_translation = initial_weights_translation
    self.observation_noise_rotation_default = observation_noise_rotation_default
    self.observation_noise_translation_default = observation_noise_translation_default
    self.construction_noise_rotation_default = construction_noise_rotation_default
    self.construction_noise_rotation_dependent = 0
    self.construction_noise_translation_default = construction_noise_translation_default
    self.construction_noise_translation_dependent = 0

    # define how often dependent parts of the model are aligned
    self.align_every_n_steps = align_every_n_steps
```

```
        # define empty lists for recreating the factor graph
        self.factors_to_reuse = []
        self.saving_old_factors_until_n_time_steps = saving_old_factors_until_n_time_steps
        self.time_step_factors_to_reuse = []

        # define empty lists for the symbols and the estimated poses of the sensor
        self.symbols_sensor = []
        self.sensor_trajectory = []

        # define a counter for the number of time steps
        self.current_num_time_steps = 0

        # define all available symbols for the variables
        (
            self.g0,
            self.G0,
            self.w1,
            self.w2,
            self.w3,
            self.W1,
            self.W2,
            self.W3,
            self.l1,
            self.l2,
            self.L1,
            self.L2,
        ) = self.define_symbols()
        self.symbols_parts = [
            self.g0,
            self.G0,
            self.w1,
            self.w2,
            self.w3,
            self.W1,
            self.W2,
            self.W3,
            self.l1,
            self.l2,
            self.L1,
            self.L2,
        ]

        # dictionary with symbols that are spatially dependent on each other in some way (i.e. axis)
        self.dependency_groups = {
            "x": {
                0: [self.g0, self.G0],
                1: [self.w1, self.W1],
                2: [self.w2, self.W2],
                3: [self.w3, self.W3],
                4: [self.l1, self.L1],
            },
            "y": {
                0: [self.w1, self.w3],
                1: [self.W1, self.W3],
            },
            "z": {
                0: [self.w1, self.w2, self.w3, self.W1, self.W2, self.W3],
                1: [self.l1, self.L1],
                2: [self.l2, self.L2],
            },
        }

        # dictionary with symbols that have some degrees of freedom in their relation to other symbols
```

```
        # and the parts that are in a fixed relation to them
        # the additional uncertainty is given as the standard deviation of the noise model ("std")
        self.groups_with_degrees_of_freedom = {
            "yaw": {
                # the yaw of the front wheels might change depending on the steering angle
                0: {
                    "symbols": [self.w1, self.W1],
                    "std": 1.5 * np.pi,
                    "rotate_second_matrix": np.pi,
                },
            },
            "z": {
                # the height of the loading edge might change depending on the load
                0: {"symbols": [self.g0, self.G0], "std": 2},
            },
        }

        # dictionary with information about the variables
        self.symbols_info = {}
        self.build_symbols_info()

        # get the construction noise model depending on the degrees of freedom and dependent parts
        self.construction_noise_dictionary = {}
        self.build_construction_noise_dictionary()

        # define the approximate truck configuration
        self.length_loading_platform = 5.87
        self.define_estimated_truck_configuration(truck_configuration)

        # define the ground truth configuration of the truck
        self.true_truck_configuration = {}
        self.define_ground_truth_truck_configuration()

    ##########################################################################################
    # Definition of the truck
    ##########################################################################################

    def build_symbols_info(self):
        """
        Build the symbols_info dictionary at the initialization of the class.
            It contains for each symbol the number of times this symbol has been observed,
            the poses of the parts in the world frame,
            the dependency groups the symbol is involved in,
            and the groups with degrees of freedom the symbol is involved in
            with the respective standard deviation.
        """
        self.symbols_info = {
            symbol: {
                "num_observed": 0,
                "pose_in_world_frame": {
                    "x": 0,
                    "y": 0,
                    "z": 0,
                    "rotation_matrix": gtsam.Rot3(),
                },
                "dependency_groups": {
                    "x": [],
                    "y": [],
                    "z": [],
                    "roll": [],
                    "pitch": [],
                    "yaw": [],
                },
```

```
                "groups_with_degrees_of_freedom": {},
                "std_degrees_of_freedom": {},
            }
            for symbol in self.symbols_parts
        }

        # adapt the dependency groups and groups with degrees of freedom for each symbol
        for symbol in self.symbols_parts:
            for axis in self.dependency_groups:
                for group in self.dependency_groups[axis]:
                    # if the symbol is in this specific group, we add all other symbols in this group
                    # to the dependency groups of the symbol
                    if symbol in self.dependency_groups[axis][group]:
                        for other_symbol in self.dependency_groups[axis][group]:
                            if other_symbol != symbol:
                                self.symbols_info[symbol]["dependency_groups"][axis].append(
                                    other_symbol
                                )

            for axis in self.groups_with_degrees_of_freedom:
                for group in self.groups_with_degrees_of_freedom[axis]:
                    if symbol in self.groups_with_degrees_of_freedom[axis][group]["symbols"]:
                        # if the symbol is in this specific group, we add all other symbols in this
                        # group to the dependency groups of the symbol, and we add the standard
                        # deviation of the degrees of freedom to the symbols_info dictionary

                        # gather all other symbols in this group
                        other_symbols = []
                        for other_symbol in self.groups_with_degrees_of_freedom[axis][group][
                            "symbols"
                        ]:
                            if other_symbol != symbol:
                                other_symbols.append(other_symbol)

                        self.symbols_info[symbol]["groups_with_degrees_of_freedom"][
                            axis
                        ] = other_symbols
                        self.symbols_info[symbol]["std_degrees_of_freedom"][
                            axis
                        ] = self.groups_with_degrees_of_freedom[axis][group]["std"]

    def define_estimated_truck_configuration(self, truck_configuration={}):
        """
        Define the estimated configuration of the truck by defining the poses of all parts
        in the world frame in the symbols_info dictionary.
        Calls the function define_symbols_info_for_estimated_truck_configuration().

        Parameters
        ----------
        truck_configuration : dict, optional
            Dictionary with the configuration of the truck. The keys of the dictionary are some
            features of the truck (e.g. width of the loading platform). Not all features have to be
            given (for the features that are not given, default values are used). The default is {}.
        """
        # we have to check whether the length of the loading platform is given
        if "length_loading_platform" not in truck_configuration:
            self.length_loading_platform = 5.87
        else:
            self.length_loading_platform = truck_configuration["length_loading_platform"]

        # define the poses of all parts in the world frame in the symbols_info dictionary
        # pass the truck configuration with ''**'' to unpack the dictionary
        self.define_symbols_info_for_estimated_truck_configuration(**truck_configuration)
```

```python
    def define_symbols_info_for_estimated_truck_configuration(
        self,
        rotation_matrix_right_loading_edge=None,
        rotation_matrix_right_wheels=None,
        rotation_matrix_front_lights=None,
        rotation_matrix_rear_lights=None,
        width_loading_platform=2.47,
        height_loading_platform_above_wheels=0.70,
        distance_g0_w1=0.79,
        distance_w1_w2=3.84,
        distance_w2_w3=1.35,
        height_loading_platform_above_front_lights=0.38,
        height_loading_platform_above_rear_lights=0.46,
        distance_lights_in_front_of_loading_platform=2.19,
        distance_lights_behind_loading_platform=0.79,
        distance_front_lights=1.73,
        distance_rear_lights=1.96,
    ):
        """
        Define the poses of all parts in the world frame in the symbols_info dictionary.
        This function is a helper function for define_estimated_truck_configuration().

        Parameters
        ----------
        rotation_matrix_right_loading_edge : ndarray, optional
            Rotation matrix for the right loading edge. The default is None.
        rotation_matrix_right_wheels : ndarray, optional
            Rotation matrix for the right wheels. The default is None.
        rotation_matrix_front_lights : ndarray, optional
            Rotation matrix for the front lights. The default is None.
        rotation_matrix_rear_lights : ndarray, optional
            Rotation matrix for the rear lights. The default is None.
        width_loading_platform : float, optional
            Width of the loading platform. The default is 2.47.
        height_loading_platform_above_wheels : float, optional
            Height of the loading platform above the wheels. The default is 0.70.
        distance_g0_w1 : float, optional
            Distance between the origin of the world frame and the left front wheel.
            The default is 0.79.
        distance_w1_w2 : float, optional
            Distance between the left front wheel and the left rear wheel. The default is 3.84.
        distance_w2_w3 : float, optional
            Distance between the left rear wheel and the right rear wheel. The default is 1.35.
        height_loading_platform_above_front_lights : float, optional
            Height of the loading platform above the front lights. The default is 0.38.
        height_loading_platform_above_rear_lights : float, optional
            Height of the loading platform above the rear lights. The default is 0.46.
        distance_lights_in_front_of_loading_platform : float, optional
            Distance between the front lights and the loading platform. The default is 2.19.
        distance_lights_behind_loading_platform : float, optional
            Distance between the rear lights and the loading platform. The default is 0.79.
        distance_front_lights : float, optional
            Distance between the two front lights. The default is 1.73.
        distance_rear_lights : float, optional
            Distance between the two rear lights. The default is 1.96.
        """
        # if no rotation matrices are given, we use the default matrices
        if rotation_matrix_right_loading_edge is None:
            rotation_matrix_right_loading_edge = self.get_rotation_matrix(-np.pi / 2, axis="z")
        if rotation_matrix_right_wheels is None:
            rotation_matrix_right_wheels = self.get_rotation_matrix(np.pi, axis="z")
        if rotation_matrix_front_lights is None:
```

```
            rotation_matrix_front_lights = self.get_rotation_matrix(np.pi / 2, axis="z")
        if rotation_matrix_rear_lights is None:
            rotation_matrix_rear_lights = self.get_rotation_matrix(-np.pi / 2, axis="z")

        y_middle = width_loading_platform / 2

        # define the poses of all parts in the world frame in the symbols_info dictionary
        self.symbols_info[self.g0]["pose_in_world_frame"] = {
            "x": 0,
            "y": 0,
            "z": 0,
            "rotation_matrix": gtsam.Rot3(),
        }
        self.symbols_info[self.G0]["pose_in_world_frame"] = {
            "x": 0,
            "y": width_loading_platform,
            "z": 0,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_right_loading_edge),
        }
        self.symbols_info[self.w1]["pose_in_world_frame"] = {
            "x": -distance_g0_w1,
            "y": 0,
            "z": -height_loading_platform_above_wheels,
            "rotation_matrix": gtsam.Rot3(),
        }
        self.symbols_info[self.W1]["pose_in_world_frame"] = {
            "x": -distance_g0_w1,
            "y": width_loading_platform,
            "z": -height_loading_platform_above_wheels,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_right_wheels),
        }
        self.symbols_info[self.w2]["pose_in_world_frame"] = {
            "x": -distance_g0_w1 + distance_w1_w2,
            "y": 0,
            "z": -height_loading_platform_above_wheels,
            "rotation_matrix": gtsam.Rot3(),
        }
        self.symbols_info[self.W2]["pose_in_world_frame"] = {
            "x": -distance_g0_w1 + distance_w1_w2,
            "y": width_loading_platform,
            "z": -height_loading_platform_above_wheels,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_right_wheels),
        }
        self.symbols_info[self.w3]["pose_in_world_frame"] = {
            "x": -distance_g0_w1 + distance_w1_w2 + distance_w2_w3,
            "y": 0,
            "z": -height_loading_platform_above_wheels,
            "rotation_matrix": gtsam.Rot3(),
        }
        self.symbols_info[self.W3]["pose_in_world_frame"] = {
            "x": -distance_g0_w1 + distance_w1_w2 + distance_w2_w3,
            "y": width_loading_platform,
            "z": -height_loading_platform_above_wheels,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_right_wheels),
        }
        self.symbols_info[self.l1]["pose_in_world_frame"] = {
            "x": -distance_lights_in_front_of_loading_platform,
            "y": y_middle - distance_front_lights / 2,
            "z": -height_loading_platform_above_front_lights,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_front_lights),
        }
        self.symbols_info[self.L1]["pose_in_world_frame"] = {
            "x": -distance_lights_in_front_of_loading_platform,
```

```python
            "y": y_middle + distance_front_lights / 2,
            "z": -height_loading_platform_above_front_lights,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_front_lights),
        }
        self.symbols_info[self.l2]["pose_in_world_frame"] = {
            "x": self.length_loading_platform + distance_lights_behind_loading_platform,
            "y": y_middle - distance_rear_lights / 2,
            "z": -height_loading_platform_above_rear_lights,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_rear_lights),
        }
        self.symbols_info[self.L2]["pose_in_world_frame"] = {
            "x": self.length_loading_platform + distance_lights_behind_loading_platform,
            "y": y_middle + distance_rear_lights / 2,
            "z": -height_loading_platform_above_rear_lights,
            "rotation_matrix": gtsam.Rot3(rotation_matrix_rear_lights),
        }

    """
    Additional functions, not shown here:
        def define_symbols(self)
        def get_sensor_symbol(self, time_step)
        def define_ground_truth_truck_configuration(self)
        def symbols_info_to_pose_in_world_frame(self, symbol, truck_configuration, true_or_estimated)
    """

    ############################################################################################
    # Update of the truck configuration
    ############################################################################################

    def update_truck_configuration(self, result):
        """
        Update the poses of the parts of the truck in the world frame according to the weights in the
        symbols_info dictionary. The model of the truck is updated after each time step, when new
        observations lead to new optimization results and a new estimation of the truck configuration.
        If we want to align the dependent parts explicitly, we can set align_explicitly to True.

        Parameters
        ----------
        result : gtsam.Values
            Result of the optimization containing the new poses of the parts in the world frame.
        """
        for symbol in self.symbols_info.keys():
            # for all variables (parts of the truck and recent sensors): get the new pose from the
            # optimization result and update the pose in the symbols_info dictionary
            new_pose = result.atPose3(symbol)
            self.update_pose_in_world_frame(symbol, new_pose)

        # update the length of the loading platform, which is dependent on the poses of some parts
        self.update_dependent_parameters()

        # align the dependent parts explicitly if desired
        align_explicitly = False
        if self.align_every_n_steps > 0 and self.current_num_time_steps > 0:
            if self.current_num_time_steps % self.align_every_n_steps == 0:
                align_explicitly = True

        if align_explicitly:
            self.align_dependent_parts()

        # transform the whole model such that the origin of the world frame
        # is at the beginning of the left loading edge
        trafo_g0_to_origin = self.symbols_info_to_pose_in_world_frame(self.g0).between(gtsam.Pose3())
```

```
        for symbol in self.symbols_info.keys():
            new_pose = self.symbols_info_to_pose_in_world_frame(symbol).compose(trafo_g0_to_origin)

            self.symbols_info[symbol]["pose_in_world_frame"] = {
                "x": new_pose.translation()[0],
                "y": new_pose.translation()[1],
                "z": new_pose.translation()[2],
                "rotation_matrix": new_pose.rotation(),
            }

    def update_pose_in_world_frame(self, symbol, new_pose):
        """
        Update the pose of a part in the world frame according to the updating weights.

        Parameters
        ----------
        symbol : gtsam.Symbol
            Symbol of the part.
        new_pose : gtsam.Pose3
            New estimated pose.
        """
        old_pose = self.symbols_info_to_pose_in_world_frame(symbol)

        # update the rotation and translation according to the weights
        updated_rotation = self.update_rotation(old_pose, new_pose, self.weight_rotation)
        updated_translation = self.update_translation(old_pose, new_pose, self.weight_translation)

        # update the pose in the symbols_info dictionary
        self.symbols_info[symbol]["pose_in_world_frame"] = {
            "x": updated_translation[0],
            "y": updated_translation[1],
            "z": updated_translation[2],
            "rotation_matrix": updated_rotation,
        }

    def update_rotation(self, old_pose, new_pose, weight):
        """
        Update the rotation of a pose according to some weight.

        Parameters
        ----------
        old_pose : gtsam.Pose3
            Old pose.
        new_pose : gtsam.Pose3
            New pose.
        weight : float
            Weight for the update.

        Returns
        -------
        gtsam.Rot3
            Updated rotation.
        """
        rotation_matrix_new = new_pose.rotation().matrix()
        rotation_matrix_old = old_pose.rotation().matrix()

        # compute the naive weighted average
        average_rotation_matrix = weight * rotation_matrix_new + (1 - weight) * rotation_matrix_old

        # project this matrix onto SO(3)
        # compute the singular value decomposition of the matrix
        U, _, V_t = np.linalg.svd(average_rotation_matrix, full_matrices=False)
        S = np.diag([1, 1, np.sign(np.linalg.det(U) * np.linalg.det(V_t))])
```

```python
        average_rotation_matrix = U @ S @ V_t
        return gtsam.Rot3(average_rotation_matrix)
    def update_translation(self, old_pose, new_pose, weight):
        """
        Update the translation of a pose according to some weight.
        Parameters
        ----------
        old_pose : gtsam.Pose3
            Old pose.
        new_pose : gtsam.Pose3
            New pose.
        weight : float
            Weight of the new pose.
        Returns
        -------
        gtsam.Point3
            Updated translation.
        """
        return weight * new_pose.translation() + (1 - weight) * old_pose.translation()
    def update_dependent_parameters(self):
        """
        After changing the configuration of the truck by changing the poses of some parts
        in the world frame, we have to update the parameters dependent on these poses,
        i.e. the length of the loading platform. The new length is computed as the weighted average
        of certain distances. Parts that have been observed more often get more weight.
        """
        weight_g0 = max(self.symbols_info[self.g0]["num_observed"], 1)
        weight_G0 = max(self.symbols_info[self.G0]["num_observed"], 1)
        weight_l2 = max(self.symbols_info[self.l2]["num_observed"], 1)
        weight_L2 = max(self.symbols_info[self.L2]["num_observed"], 1)

        x_front_end = (
            weight_g0 * self.symbols_info[self.g0]["pose_in_world_frame"]["x"]
            + weight_G0 * self.symbols_info[self.G0]["pose_in_world_frame"]["x"]
        ) / (weight_g0 + weight_G0)
        x_rear_end = (
            weight_l2 * self.symbols_info[self.l2]["pose_in_world_frame"]["x"]
            + weight_L2 * self.symbols_info[self.L2]["pose_in_world_frame"]["x"]
        ) / (weight_l2 + weight_L2)
        distance_lights_behind_loading_platform = 1.04
        self.length_loading_platform = (
            abs(x_rear_end - x_front_end) - distance_lights_behind_loading_platform
        )
    def align_dependent_parts(self):
        """
        If the position of a part is updated, the positions of the dependent parts have to be updated
        as well, e.g. if the x-coordinate of the wheel w2 is updated, the x-coordinate of the wheel W2
        has to be updated as well. This is done according to the align_every_n_steps parameter.
        """
        # we iterate over all axes ...
        for axis in self.dependency_groups.keys():
            # ... and all groups to calculate the mean of the updated symbols in the current group
            for group_list in self.dependency_groups[axis].values():
                # average for translation axes
                if axis in ["x", "y", "z"]:
                    average_translation = self.get_average_translation(group_list, axis)
```

```python
                    for symbol in group_list:
                        self.symbols_info[symbol]["pose_in_world_frame"][axis] = average_translation

                # average for rotation axes
                else:
                    average_rotation_matrix, _ = self.get_average_rotation(group_list)
                    for symbol in group_list:
                        self.symbols_info[symbol]["pose_in_world_frame"][
                            "rotation_matrix"
                        ] = gtsam.Rot3(average_rotation_matrix)

        # and do the same for the groups with degrees of freedom
        for axis in self.groups_with_degrees_of_freedom.keys():
            for group in self.groups_with_degrees_of_freedom[axis].values():
                group_list = group["symbols"]

                if axis in ["x", "y", "z"]:
                    average_translation = self.get_average_translation(group_list, axis)
                    for symbol in group_list:
                        self.symbols_info[symbol]["pose_in_world_frame"][axis] = average_translation

                else:
                    # we have to check if the rotation should be the same or rotated by a fixed angle
                    average_rotation_matrix1, average_rotation_matrix2 = self.get_average_rotation(
                        group_list, axis, rotate_second_matrix=group["rotate_second_matrix"]
                    )

                    self.symbols_info[group_list[0]]["pose_in_world_frame"][
                        "rotation_matrix"
                    ] = gtsam.Rot3(average_rotation_matrix1)
                    self.symbols_info[group_list[1]]["pose_in_world_frame"][
                        "rotation_matrix"
                    ] = gtsam.Rot3(average_rotation_matrix2)

    def get_average_translation(self, list_of_symbols, axis):
        """
        Get the average translation of a list of symbols in a certain direction.
        The poses are weighted according to how often they have been observed.

        Parameters
        ----------
        list_of_symbols : list of gtsam.Symbol
            The list of symbols, we want to find the average of.
        axis : str
            The axis for which we want to find the average. One of ["x", "y", "z"].

        Returns
        -------
        weighted_pose_average : float
            The average translation in the given direction.
        """
        sum_pose, sum_observed = 0, 0
        for symbol in list_of_symbols:
            obs = max(self.symbols_info[symbol]["num_observed"], 1)
            sum_pose += self.symbols_info[symbol]["pose_in_world_frame"][axis] * obs
            sum_observed += obs

        weighted_pose_average = sum_pose / sum_observed

        return weighted_pose_average

    def get_average_rotation(self, list_of_symbols, axis=None, rotate_second_matrix=False):
        """
```

```
        Get the average rotation of a list of rotation matrices. If the rotations are dependent on
        each other, but rotated by a fixed angle, one matrix is rotated by this angle,
        then they are averaged, and then this matrix is rotated back.

        Parameters
        ----------
        list_of_symbols : list of gtsam.Symbol
            The list of symbols, we want to find the average of.
        axis : str
            The axis around which we want to find the average. One of ["roll", "pitch", "yaw"].
        rotate_second_matrix : bool or angle, optional
            If not False, this is the angle around which the second rotation matrix is rotated.

        Returns
        -------
        average_rotation_matrix : ndarray
            The average rotation around the given axis.
        """
        if rotate_second_matrix is not False:
            rot_matrix1 = self.symbols_info[list_of_symbols[0]]["pose_in_world_frame"][
                "rotation_matrix"
            ].matrix()
            rot_matrix2 = self.symbols_info[list_of_symbols[1]]["pose_in_world_frame"][
                "rotation_matrix"
            ].matrix()

            angle = rotate_second_matrix
            helper_rot_matrix = self.get_rotation_matrix(angle, axis)
            rot_matrix2 = helper_rot_matrix @ rot_matrix2

            # compute the naive average
            average_rotation_matrix = (rot_matrix1 + rot_matrix2) / 2

        else:
            list_of_matrices = []
            for symbol in list_of_symbols:
                list_of_matrices.append(
                    self.symbols_info[symbol]["pose_in_world_frame"]["rotation_matrix"].matrix()
                )

            # compute the naive average
            average_rotation_matrix = np.sum(np.array(list_of_matrices), axis=0) / len(
                list_of_matrices
            )

        # project this matrix onto SO(3) and compute the singular value decomposition
        U, _, V_t = np.linalg.svd(average_rotation_matrix, full_matrices=False)
        S = np.diag([1, 1, np.sign(np.linalg.det(U) * np.linalg.det(V_t))])
        average_rotation_matrix = U @ S @ V_t

        if rotate_second_matrix:
            return average_rotation_matrix, helper_rot_matrix.T @ average_rotation_matrix
        else:
            return average_rotation_matrix, None

    ##############################################################################################
    # Pose estimation
    ##############################################################################################

    def estimate_sensor_pose(self, observations, observations_info=None, random_guess=False):
        """
        Estimate the pose of the sensor in the world frame that fits the observations best.
```

```
        Parameters
        ----------
        observations : dict
            Dictionary with the symbols of the observed parts as keys and the observations as values.
        observations_info : dict, optional
            Dictionary with the symbols of the observed parts as keys and the information about the
            observations as values. Such information is an axis with certain degree of freedom due to
            incomplete observations, e.g. we only observe a part of the loading edge
            but not its beginning. The default is None.
        random_guess : bool, optional
            If True or the sensor trajectory is empty, the initial guess for the optimization
                of the sensor pose is a random pose.
            Else, the initial guess for the optimization is the estimated pose of the sensor
                in the previous time step.
            The default is False.

        Returns
        -------
        best_pose_new_sensor : gtsam.Pose3
            Estimated pose of the sensor in the world frame.
        best_error : float
            Error of the optimization. Note: This is not the error of the estimated sensor pose
            to the ground truth, but the error of the optimization.
        result : gtsam.Values
            Result of the optimization.
        """
        # define initial guess for the optimization
        if random_guess or self.current_num_time_steps == 0:
            initial_guess_sensor_pose = gtsam.Pose3(
                gtsam.Rot3(), gtsam.Point3(np.random.rand(3) * 10)
            )
        else:
            initial_guess_sensor_pose = self.get_current_sensor_pose()

        # insert the observations as factors in the factor graph
        self.observe(observations, observations_info)

        # get the initial values for all parts and previous sensor poses
        initial = self.get_initial_values()

        # estimate the pose of the sensor
        best_pose_new_sensor, best_error, result = self.optimize_sensor_pose(
            initial, initial_guess_sensor_pose
        )

        # add the estimated pose to the sensor trajectory and the symbols_info dictionary
        self.add_pose_to_trajectory(best_pose_new_sensor)

        return best_pose_new_sensor, best_error, result

    def observe(self, observations, observations_info=None):
        """
        First, we recreate the factor graph with the new estimated truck configuration.
        Then, we add a new sensor to the factor graph.
        Finally, add the observations as factors to the factor graph.

        Parameters
        ----------
        observations : dict
            Dictionary with the symbols of the observed parts as keys and the observations as values.
        observations_info : dict, optional
            Dictionary with the symbols of the observed parts as keys and the information about the
            observations as values. Such information is an axis with certain degree of freedom due to
```

```
            incomplete observations, e.g. we only observe a part of the loading edge
            but not its beginning. The default is None.
        """
        # create the factor graph from scratch (this substitutes the old factor graph)
        self.create_factor_graph()

        # add the old observation factors
        indices_to_delete = []
        for i in range(len(self.factors_to_reuse)):
            if (
                self.time_step_factors_to_reuse[i]
                < self.current_num_time_steps - self.saving_old_factors_until_n_time_steps
            ):
                indices_to_delete.append(i)

            # the list is monotonically increasing, so we can break here
            else:
                break

        # delete the old factors from the list of factors to reuse
        for i in reversed(indices_to_delete):
            del self.factors_to_reuse[i]
            del self.time_step_factors_to_reuse[i]

        # add the old factors to the new factor graph
        for factor in self.factors_to_reuse:
            self.graph.add(factor)

        # we also have to remove the old sensor symbol from the dictionary of all variables
        if self.current_num_time_steps - self.saving_old_factors_until_n_time_steps > 0:
            self.symbols_info.pop(
                self.get_sensor_symbol(
                    self.current_num_time_steps - self.saving_old_factors_until_n_time_steps - 1
                )
            )

        # add a new sensor and the observations as factors to the factor graph
        self.add_new_sensor(observations, observations_info)

    def add_new_sensor(self, observations, observations_info):
        """
        Add a new sensor and the observations to the factor graph of the current time step.

        Parameters
        ----------
        observations : dict
            Dictionary with the symbols of the observed parts as keys and the observations as values.
        observations_info : dict, optional
            Dictionary with the symbols of the observed parts as keys and the information about the
            observations as values. Such information is an axis with certain degree of freedom due to
            incomplete observations, e.g. we only observe a part of the loading edge
            but not its beginning. The default is None.
        """
        # get the symbol for the sensor in the next time step and add it to the list of sensor symbols
        next_time_step = self.current_num_time_steps
        symbol_new_sensor = self.get_sensor_symbol(next_time_step)
        self.symbols_sensor.append(symbol_new_sensor)

        # add the observations as factors to the factor graph
        self.observations_to_factors(symbol_new_sensor, observations, observations_info)

    def observations_to_factors(self, sensor_symbol, observations, observations_info):
        """
```

```
        Convert the observations to factors between the sensor and the parts and insert them into the
        factor graph. Used in the function add_new_sensor(). The observations_info dictionary gives
        additional information about the observations, i.e. uncertainty in certain directions,
        e.g. observations_info = {w1: {"x": 2.4, "roll": 0.6*np.pi}}
            ... if the observation of w1 is unsure in x-direction and roll rotation.

        Parameters
        ----------
        sensor_symbol : gtsam.Symbol
            Symbol of the sensor in the current time step.
        observations : dict
            Dictionary with the symbols of the observed parts as keys and the observations as values.
        observations_info : dict
            Dictionary with the symbols of the observed parts as keys and the information about the
            observations as values. Such information is an axis with certain degree of freedom due to
            incomplete observations, e.g. we only observe a part of the loading edge
            but not its beginning. The default is None.
        """
        for symbol in observations:
            # define the observation noise model
            observation_noise_model = {
                "roll": self.observation_noise_rotation_default,
                "pitch": self.observation_noise_rotation_default,
                "yaw": self.observation_noise_rotation_default,
                "x": self.observation_noise_translation_default,
                "y": self.observation_noise_translation_default,
                "z": self.observation_noise_translation_default,
            }

            # adapt the observation noise model according to observations_info
            if observations_info is not None:
                if symbol in observations_info.keys():
                    for axis in observations_info[symbol].keys():
                        observation_noise_model[axis] = observations_info[symbol][axis]

            # define the observation noise model
            observation_noise_model = self.get_noise_model(**observation_noise_model)

            # define the factor and add it to the graph
            factor = gtsam.BetweenFactorPose3(
                sensor_symbol, symbol, observations[symbol], observation_noise_model
            )
            self.graph.add(factor)

            # add the factor to the list of factors that should be reused in the next time step
            self.factors_to_reuse.append(factor)
            self.time_step_factors_to_reuse.append(self.current_num_time_steps)

            # adapt the num_observed attribute of the observed part
            self.symbols_info[symbol]["num_observed"] += 1

        # adapt the weights for updating rotation and translation
        self.update_weights("translation")
        self.update_weights("rotation")

    def optimize_sensor_pose(self, initial, initial_guess_new_sensor):
        """
        Find the pose of the sensor in the world frame that fits the observations and the assumed
        model best. The initial guess is the estimated pose of the sensor in the world frame at the
        beginning of the optimization.

        Parameters
        ----------
```

```
        initial : gtsam.Values
            Initial values of the variables except the new sensor.
        initial_guess_new_sensor : gtsam.Pose3
            Initial guess for the pose of the sensor in the world frame.

        Returns
        -------
        best_pose_new_sensor : gtsam.Pose3
            Pose of the sensor in the world frame that fits the observations best.
        best_error : float
            Optimization error of the best pose of the sensor in the world frame.
        result : gtsam.Values
            Optimized values of all variables.
        """
        # get the current sensor symbol
        symbol_new_sensor = self.get_sensor_symbol(self.current_num_time_steps)

        # insert the initial guess into the current result
        initial.insert(symbol_new_sensor, initial_guess_new_sensor)

        # optimize the graph with the current result
        result = self.optimize_LM(initial)
        error = self.graph.error(result)
        new_pose_new_sensor = result.atPose3(symbol_new_sensor)

        return new_pose_new_sensor, error, result

    def add_pose_to_trajectory(self, pose):
        """
        Add the estimated pose of the sensor in the world frame to the sensor trajectory and to the
        symbols_info dictionary, increase the number of time steps by one, and update the default
        construction noise.

        Parameters
        ----------
        pose : gtsam.Pose3
            Estimated pose of the sensor in the world frame.
        """
        # add the pose to the sensor trajectory
        self.sensor_trajectory.append(pose)

        # add a new entry to the symbols_info dictionary for the sensor in the new time step
        self.symbols_info[self.get_sensor_symbol(self.current_num_time_steps)] = {
            "num_observed": 0,
            "pose_in_world_frame": {
                "x": pose.translation()[0],
                "y": pose.translation()[1],
                "z": pose.translation()[2],
                "rotation_matrix": pose.rotation(),
            },
            "dependency_groups": {},
            "groups_with_degrees_of_freedom": {},
            "std_degrees_of_freedom": {},
        }

        # increase the number of time steps by one
        self.current_num_time_steps += 1

        # adapt construction noise
        self.construction_noise_rotation_default *= self.updating_factor_noise
        self.construction_noise_translation_default *= self.updating_factor_noise

    """
```

```python
    Additional functions, not shown here:
        def get_initial_values(self)
    """

    ###############################################################################################
    # Factor graph
    ###############################################################################################

    def build_construction_noise_dictionary(self):
        """
        Build the construction noise dictionary. This function is called once in the beginning such
        that the dictionary does not have to be rebuilt in each time step. The keys of the dictionary
        are tuples of two symbols, e.g. (self.w1, self.L2). The values are dictionaries containing the
        construction noise for each axis,
        e.g. {"x": 0.1, "y": 0.1, "z": 0.1, "roll": 0.1, "pitch": 0.1, "yaw": 0.1}.
        """
        # define which variables are connected by factors
        self.define_used_factors()

        # the order of the symbols stays the same, so we do not have to check each pair twice
        for tuple_of_symbols in self.construction_noise_dictionary.keys():
            first_symbol = tuple_of_symbols[0]
            second_symbol = tuple_of_symbols[1]

            # initialize the construction noise with "default"
            # this stays the same if the parts are not dependent on each other
            construction_noise = {
                "roll": "default",
                "pitch": "default",
                "yaw": "default",
                "x": "default",
                "y": "default",
                "z": "default",
            }

            # adapt the construction noise according to the dependency groups
            for axis in self.symbols_info[first_symbol]["dependency_groups"].keys():
                if second_symbol in self.symbols_info[first_symbol]["dependency_groups"][axis]:
                    construction_noise[axis] = "dependent"

            # adapt the construction noise according to the degrees of freedom
            for axis in self.symbols_info[first_symbol]["groups_with_degrees_of_freedom"]:
                new_construction_noise = self.get_construction_noise_with_degrees_of_freedom(
                    first_symbol, second_symbol, axis
                )
                construction_noise[axis] = new_construction_noise

            # add the construction noise to the dictionary
            self.construction_noise_dictionary[tuple_of_symbols] = copy.deepcopy(construction_noise)

    def define_used_factors(self):
        """
        Define which parts of the truck should be connected with factors.
        The variables have a fixed order, so each combination of symbols appears exactly once.
        Define the construction noise dictionary.
        """
        # define the keys in the form of a dictionary (if w1 is in factors[g0], [(g0, w1)] is a key)
        factors = {
            self.g0: [self.l1, self.w1, self.w2, self.G0],
            self.G0: [self.L1, self.W1, self.W2],
            self.w1: [self.l1, self.w2, self.W1],
            self.w2: [self.w3, self.W2],
            self.w3: [self.l2, self.W3],
```

```
            self.W1: [self.L1, self.W2],
            self.W2: [self.W3],
            self.W3: [self.L2],
            self.l1: [self.L1],
            self.l2: [self.L2],
            self.L1: [],
            self.L2: [],
        }

        # build the structure of the construction noise dictionary
        for symbol in factors.keys():
            for other_symbol in factors[symbol]:
                self.construction_noise_dictionary[(symbol, other_symbol)] = {}

    def create_factor_graph(self):
        """
        (Re-)Create the factor graph and add the prior and construction factors to the graph.
        """
        # create an empty graph
        self.graph = gtsam.NonlinearFactorGraph()

        # the world origin is exactly at the start of the left loading edge
        # therefore, we set the respective noise to zero
        world_origin_noise = gtsam.noiseModel.Diagonal.Sigmas(np.array([0] * 6))

        # define the prior factor and add it to the graph
        factor_prior = gtsam.PriorFactorPose3(self.g0, gtsam.Pose3(), world_origin_noise)
        self.graph.add(factor_prior)

        # add the construction factors
        self.add_construction_factors()

    def add_construction_factors(self):
        """
        Function to add the construction factors to the graph. Since these factors depend on
        the model of the truck, we have to add them after each time step (after recreating the graph).
        The values of the BetweenFactorPose3 are based on the transformation between the poses of
        two parts in the world frame (as stated in symbols_info). The noise model is based on
        the construction noise dictionary.
        """
        # define the factors between all pairs of parts (order as stated in self.symbols_parts)
        for tuple_of_symbols in self.construction_noise_dictionary.keys():
            first_symbol = tuple_of_symbols[0]
            second_symbol = tuple_of_symbols[1]

            # get the transformation between the two parts
            trafo_first_to_second_symbol = self.symbols_info_to_pose_in_world_frame(
                first_symbol
            ).between(self.symbols_info_to_pose_in_world_frame(second_symbol))

            construction_noise_model = copy.deepcopy(
                self.construction_noise_dictionary[tuple_of_symbols]
            )

            # substitute the entries in the construction noise model by their corresponding values
            for axis in construction_noise_model.keys():
                if construction_noise_model[axis] == "default":
                    if axis in ["x", "y", "z"]:
                        construction_noise_model[axis] = self.construction_noise_translation_default
                    else:
                        construction_noise_model[axis] = self.construction_noise_rotation_default

                elif construction_noise_model[axis] == "dependent":
```

```
                    if axis in ["x", "y", "z"]:
                        construction_noise_model[axis] = self.construction_noise_translation_dependent
                    else:
                        construction_noise_model[axis] = self.construction_noise_rotation_dependent

                # if it is not "default" or "dependent", it gives the additional noise for this factor
                else:
                    if axis in ["x", "y", "z"]:
                        construction_noise_model[axis] += self.construction_noise_translation_default
                    else:
                        construction_noise_model[axis] += self.construction_noise_rotation_default

            # get the noise model for the factor
            construction_noise_model = self.get_noise_model(**construction_noise_model)

            # define the factor and add it to the graph
            factor = gtsam.BetweenFactorPose3(
                first_symbol, second_symbol, trafo_first_to_second_symbol, construction_noise_model
            )
            self.graph.add(factor)

    def update_construction_noise(self):
        """
        Update the default construction noise for rotation and translation according to the number of
        steps. The construction noise follows the function
            c(n) = lambda^n * (c(0) - lower_bound) + lower_bound,
        thus we compute the new construction noise as
            c(n+1) = lambda * c(n) + (1-lambda)*lower_bound.
        """
        self.construction_noise_translation_default = (
            self.updating_factor_noise * self.construction_noise_translation_default
            + (1 - self.updating_factor_noise) * self.lower_bound_construction_noise
        )
        self.construction_noise_rotation_default = (
            self.updating_factor_noise * self.construction_noise_rotation_default
            + (1 - self.updating_factor_noise) * self.lower_bound_construction_noise
        )

    def update_weights(self, update_type):
        """
        Update the weights of a part according to how often it has been observed.
        The weights follow the function
            w(n) = lambda^n * (w(0) - lower_bound) + lower_bound,
        thus we compute the new weight as
            w(n+1) = lambda * w(n) + (1-lambda)*lower_bound.

        Parameters
        ----------
        update_type : str
            Either "translation" or "rotation".
        """
        if update_type == "translation":
            self.weight_translation = (
                self.updating_factor_weights * self.weight_translation
                + (1 - self.updating_factor_weights) * self.lower_bound_weights
            )

        elif update_type == "rotation":
            self.weight_rotation = (
                self.updating_factor_weights * self.weight_rotation
                + (1 - self.updating_factor_weights) * self.lower_bound_weights
            )
```

```python
    def optimize_LM(self, initial):
        """
        Optimize the factor graph using Levenberg-Marquardt optimization of gtsam.

        Parameters
        ----------
        initial : gtsam.Values
            Initial values of all variables.

        Returns
        -------
        result : gtsam.Values
            Optimized values of all variables, including the estimated sensor pose.
        """
        # define the optimizer
        params = gtsam.LevenbergMarquardtParams()
        params.setVerbosityLM("ERROR")
        optimizer = gtsam.LevenbergMarquardtOptimizer(self.graph, initial, params)

        # optimize the graph
        result = optimizer.optimize()

        return result

    ###########################################################################################
    # Helper functions
    ###########################################################################################

    def get_construction_noise_with_degrees_of_freedom(
        self, first_symbol, second_symbol, axis_first_symbol
    ):
        """
        Get the the construction noise for a part in a group with degree of freedom to other parts.

        Parameters
        ----------
        first_symbol : gtsam.Symbol
            Currently investigated symbol.
        second_symbol : gtsam.Symbol
            Symbol of the part we want to investigate the connection to first_symbol with.
        axis_first_symbol : str
            Respective axis for the group with an degree of freedom.

        Returns
        -------
        construction_noise : str or float
            "dependent" if the two symbols are in the same group,
            otherwise the additional standard deviation for the noise is returned.
        """
        if (
            second_symbol
            in self.symbols_info[first_symbol]["groups_with_degrees_of_freedom"][axis_first_symbol]
        ):
            return "dependent"
        else:
            return self.symbols_info[first_symbol]["std_degrees_of_freedom"][axis_first_symbol]

    def get_noise_model(self, roll=0.1, pitch=0.1, yaw=0.1, x=0.3, y=0.3, z=0.3, rot=None, pos=None):
        """
        Get a noise model for the observations.
        The noise model is a diagonal matrix with the given standard deviations as entries.

        Parameters
```

```
        ----------
        roll : float, optional
            Standard deviation of the noise in the rotation around the x-axis. The default is 0.1.
        pitch : float, optional
            Standard deviation of the noise in the rotation around the y-axis. The default is 0.1.
        yaw : float, optional
            Standard deviation of the noise in the rotation around the z-axis. The default is 0.1.
        x : float, optional
            Standard deviation of the noise in the x-coordinate. The default is 0.3.
        y : float, optional
            Standard deviation of the noise in the y-coordinate. The default is 0.3.
        z : float, optional
            Standard deviation of the noise in the z-coordinate. The default is 0.3.
        rot : float, optional
            Standard deviation of the noise in the rotation around all axes.
            If rot is given, roll, pitch and yaw are set to rot. The default is None.
        pos : float, optional
            Standard deviation of the noise in the translation in all directions.
            If pos is given, x, y and z are set to pos. The default is None.

        Returns
        -------
        noise_model : gtsam.noiseModel
            Noise model for the observations.
        """
        if rot is not None:
            roll, pitch, yaw = rot, rot, rot
        if pos is not None:
            x, y, z = pos, pos, pos

        return gtsam.noiseModel.Diagonal.Sigmas(np.array([roll, pitch, yaw, x, y, z]))

    def get_rotation_matrix(self, angle, axis="x"):
        """
        Get the rotation matrix for a rotation around the x, y, or z axis.

        Parameters
        ----------
        angle : float
            Angle of the rotation in radians.
        axis : str, optional
            Axis of the rotation ("x", "y", or "z"). The default is "x".

        Returns
        -------
        rotation_matrix : np.array
            Rotation matrix.
        """
        if axis == "x" or axis == "roll":
            return np.array(
                [
                    [1, 0, 0],
                    [0, np.cos(angle), -np.sin(angle)],
                    [0, np.sin(angle), np.cos(angle)],
                ]
            )
        elif axis == "y" or axis == "pitch":
            return np.array(
                [
                    [np.cos(angle), 0, np.sin(angle)],
                    [0, 1, 0],
                    [-np.sin(angle), 0, np.cos(angle)],
                ]
```

```python
            )
        elif axis == "z" or axis == "yaw":
            return np.array(
                [
                    [np.cos(angle), -np.sin(angle), 0],
                    [np.sin(angle), np.cos(angle), 0],
                    [0, 0, 1],
                ]
            )
        else:  # incorrect input
            print("Incorrect input for axis, please use x, y, or z!")
            return None

    def get_joint_rotation_matrix(self, angles, axes="xyz"):
        """
        Get the rotation matrix for a rotation around the x, y, and/or z axis, angle in radians.

        Parameters
        ----------
        angles : list of float
            Angles of the rotation in radians around the axes specified in "axes".
        axes : str, optional
            Axes of the rotation (sequence of "x", "y", and "z"). The default is "xyz".

        Returns
        -------
        rotation_matrix : np.array
            Rotation matrix.
        """
        rotation_matrix = np.eye(3)
        for i in range(len(angles)):
            rotation_matrix = np.dot(self.get_rotation_matrix(angles[i], axes[i]), rotation_matrix)

        return rotation_matrix

    """
    Additional functions, not shown here:
        def get_dominant_part(self, list_of_parts)
        def get_width_loading_platform(self)
        def get_current_sensor_pose(self)
    """

    ###########################################################################################
    # Visualization
    ###########################################################################################
    """
    Visualization functions, not shown here:
        def plot_sensor_trajectory(...)
        def plot_truck(...)
        def create_cuboid(...)
        def create_cylinder(...)
        def create_coordinate_frame(...)
    """


################################################################################################
# Exemplary Test
################################################################################################


def test_model_error_construction_noise(
    data_file, obs_noise_sd, constr_noise_sd, parameter_values, repeat_test
):
```

```
    """
    Test how fast and well the model of the truck is updated in dependence of the construction noise.

    Parameters
    ----------
    data_file : pickle file
        Pickle file containing the observed data in form of a dictionary.
    obs_noise_sd : float
        Standard deviation of the additional artificial noise for the observations.
    constr_noise_sd : float
        Standard deviation of the additional artificial noise for the construction of the truck.
    parameter_values : list of float
        Different values of the construction noise used for testing.
    repeat_test : int
        States how often the test should be repeated.

    Returns
    -------
    means : ndarray of shape (len(parameter_values), num_timesteps)
        Means of the model error for the different parameter values of all test trials.
    std_devs : ndarray of shape (len(parameter_values), num_timesteps)
        Standard deviations of the model error for the different parameter values of all test trials.
    """
    # set a seed to get reproducible results
    np.random.seed(11)

    # get the observations from the data file
    observation_list = data_to_observations(data_file)

    # basic parameter settings
    observation_noise_translation_default, observation_noise_rotation_default = 0.17, 0.14
    construction_noise_rotation_default = 1.5
    updating_factor_noise, updating_factor_weights = 0.94, 0.97
    weight_translation, weight_rotation = 0.80, 0.80
    align_every_n_steps = 32

    # create a basic instance of the class Truck3D with the parameters defined above
    Truck_basic = Truck3D(
        observation_noise_translation_default=observation_noise_translation_default,
        observation_noise_rotation_default=observation_noise_rotation_default,
        construction_noise_rotation_default=construction_noise_rotation_default,
        updating_factor_noise=updating_factor_noise,
        updating_factor_weights=updating_factor_weights,
        initial_weights_translation=weight_translation,
        initial_weights_rotation=weight_rotation,
        align_every_n_steps=align_every_n_steps,
    )

    # list that will contain the model errors for all test trials
    results = []

    for _ in range(repeat_test):
        # copy the instance of the Truck3D class
        Truck_imprecise_configuration = copy.deepcopy(Truck_basic)

        # define an imprecise truck configuration with additional random noise
        constr_noise = np.random.normal(0, constr_noise_sd, 11)
        Truck_imprecise_configuration = get_imprecise_truck_configuration(
            Truck_imprecise_configuration, constr_noise
        )

        # list that will contain the model errors for one test trial
        model_errors = []
```

```python
        # add random noise to the observations
        noisy_observations = add_noise_to_data(observation_list, obs_noise_sd)

        # run the sensor pose estimations for all parameter values
        for j in range(len(parameter_values)):
            # copy the imprecise Truck3D object
            Truck = copy.deepcopy(Truck_imprecise_configuration)

            # set the value of the construction noise
            parameter = parameter_values[j]
            Truck.construction_noise_translation_default = parameter

            # run the sensor pose estimation on these observations
            model_error, _ = simulate_sensor_trajectory_model_error(Truck, noisy_observations)
            model_errors.append(model_error)

        results.append(model_errors)

    means = np.mean(results, axis=0)
    std_devs = np.std(results, axis=0)

    return means, std_devs


def data_to_observations(data_file):
    """
    Get the observations saved in a data file.

    Parameters
    ----------
    data_file : pickle file
        Opened pickle file containing the data.

    Returns
    -------
    all_observations : list of dict
        List containing the observations for each time step.
    """
    data_dict = pickle.load(data_file)

    all_observations = []
    for frame in data_dict.values():
        current_observation = {}

        for symbol in frame.keys():
            pos, rot = frame[symbol]["pos"], frame[symbol]["rot"]
            current_observation[symbol] = gtsam.Pose3(gtsam.Rot3(rot), pos)

        all_observations.append(current_observation)

    return all_observations


def add_noise_to_data(data, noise_sd):
    """
    Add random additional noise to data.

    Parameters
    ----------
    data : list of dict
        Data of gtsam poses, where we want to add noise.
    noise_sd : float
```

```python
        Standard deviation of the normally distributed noise.

    Returns
    -------
    noisy_data : list of dict
        Data of gtsam poses with random noise.
    """
    noisy_data = copy.deepcopy(data)
    for frame in noisy_data:
        for symbol_string in frame.keys():
            # get random numbers for the noise of position and orientation
            noise_pos = np.random.normal(0, noise_sd, 3)
            noise_rot = np.random.normal(0, min(noise_sd, 0.05), 3)

            # create a rotation matrix from the local coordinates noise_rot of SO(3)
            noise_rot_matrix = local_update_SO3(np.eye(3), noise_rot)

            # transform the noise to a gtsam pose
            noise_pose = gtsam.Pose3(gtsam.Rot3(noise_rot_matrix), noise_pos)

            # add the noise to the data
            frame[symbol_string] = frame[symbol_string].compose(noise_pose)

    return noisy_data


def local_update_SO3(rotation_matrix, vector):
    """
    Local update on SO(3) using the hat-operator and the matrix exponential as a retraction.

    Parameters
    ----------
    rotation_matrix : ndarray of shape (3,3)
        Original rotation matrix R0.
    vector : ndarray of shape (3,)
        Vector xi of the local coordinates on SO(3).

    Returns
    -------
    ndarray of shape (3,3)
        Local update of the original rotation matrix R0.
    """
    return rotation_matrix @ matrix_exp(vector)


def matrix_exp(vector):
    """
    Matrix exponential for a skew-symmetric matrix.
    Uses the hat-operator on SO(3) and computes the resulting matrix with Rodrigues' forumla.

    Parameters
    ----------
    vector : ndarray of shape (3,)
        Vector xi of the local coordinates on SO(3).

    Returns
    -------
    exp(hat(vector)) : ndarray of shape (3,3)
        The matrix exponential of hat(vector).
    """
    theta = np.linalg.norm(vector)
    if theta == 0:
        return np.eye(3)
```

```
    else:
        return (
            np.eye(3)
            + np.sin(theta) / theta * hat_operator(vector)
            + (1 - np.cos(theta)) / theta**2 * hat_operator(vector) @ hat_operator(vector)
        )


def hat_operator(vector):
    """
    Implementation of the hat-operator for SO(3).

    Parameters
    ----------
    vector : ndarray of shape (3,)
        Vector xi of the local coordinates on SO(3).

    Returns
    -------
    hat(vector) : ndarray of shape (3,3)
        Skew-symmetric matrix, output of the hat-operator.
    """
    return np.array(
        [
            [0, -vector[2], vector[1]],
            [vector[2], 0, -vector[0]],
            [-vector[1], vector[0], 0],
        ]
    )


def simulate_sensor_trajectory_model_error(Truck, observations):
    """
    Simulate a run of the sensor pose estimation for given observations.

    Parameters
    ----------
    Truck : Truck3D
        Instance of the Truck3D class with the respective parameters.
    observations : list of dict
        List of the noisy observations.

    Returns
    -------
    model_error : list of float
        List of the model errors for each time step.
    """
    # list containing the model errors for each time step
    model_error = []

    # get the initial model error
    model_error.append(evaluate_model_error(Truck))

    for i in range(len(observations)):
        # estimate the sensor pose
        _, _, result = Truck.estimate_sensor_pose(observations[i])

        # update the truck configuration according to the results of the previous estimation
        Truck.update_truck_configuration(result)

        # evaluate the current error of the estimated model
        model_error.append(evaluate_model_error(Truck))
```

```
    return model_error


def evaluate_model_error(Truck):
    """
    Evaluate the model error of the truck. The model error is defined as the distance between
    the poses of the parts in the model and the poses of the parts in the ground truth.
    The distance in the position is computed as the norm of the difference between the positions.
    The distance in the rotation R1, R2 is computed as the Frobenius norm of I-R1R2^T.

    Parameters
    ----------
    Truck : Truck3D
        Truck, where we look for the error between the assumed model and the ground truth.

    Returns
    -------
    cumulated_distance : float
        Model error cumulated for all parts of the truck.
    """
    cumulated_distance = 0
    for symbol in Truck.symbols_info:
        if symbol not in Truck.symbols_sensor:
            # compute the distance between the model and the ground truth
            pose_difference = Truck.symbols_info_to_pose_in_world_frame(
                symbol, true_or_estimated="estimated"
            ).between(Truck.symbols_info_to_pose_in_world_frame(symbol, true_or_estimated="true"))
            cumulated_distance += pose_norm(pose_difference)

    return cumulated_distance


def pose_norm(pose):
    """
    Norm of the pose used to measure the model error. The norm of the position is computed as the
    Euclidean norm. The norm of the rotation R is computed as the Frobenius norm of I-R.

    Parameters
    ----------
    pose : gtsam.Pose3
        Pose to compute the norm.

    Returns
    -------
    norm : float
        Norm of the given pose.
    """
    pose_norm_position = np.linalg.norm(pose.translation())

    R = pose.rotation().matrix()
    I = np.identity(3)
    pose_norm_rotation = np.linalg.norm(I - R)

    return pose_norm_position + pose_norm_rotation


"""
Additional functions, not shown here:
    def get_imprecise_truck_configuration(Truck, noise)
    visualization functions
"""
```

Bibliography

[1] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds*. 1st ed. Princeton: Princeton University Press, 2008. ISBN: 978-0-691-13298-3.

[2] R. Adler et al. "Newton's Method on Riemannian Manifolds and a Geometric Model for the Human Spine". In: *IMA Journal of Numerical Analysis* vol. 22 (2002), pp. 359–390. DOI: 10.1093/imanum/22.3.359.

[3] S. Ahmed et al. "Edge and Corner Detection for Unorganized 3D Point Clouds with Application to Robotic Welding". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 7350–7355. DOI: 10.1109/IROS.2018.8593910.

[4] J. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* vol. 18, no. 9 (1975), pp. 509–517. DOI: 10.1145/361002.361007.

[5] R. Brégier et al. "Defining the Pose of Any 3D Rigid Object and an Associated Distance". In: *Int J Comput Vis* vol. 126 (2018), pp. 571–596. DOI: 10.1007/s11263-017-1052-4.

[6] G. Casella and R. Berger. *Statistical Inference*. 2nd ed. Belmont CA: Duxbury, 2002. ISBN: 978-8-131-50394-2.

[7] Y. Chen, Y. Chen, and G. Wang. "Bundle Adjustment Revisited". In: *ArXiv* (2019). DOI: 10.48550/arXiv.1912.03858.

[8] S. Chmielewski and P. Tompalski. "Estimating Outdoor Advertising Media Visibility with Voxel-Based Approach". In: *Applied Geography* vol. 87 (2017), pp. 1–13. DOI: 10.1016/j.apgeog.2017.07.007.

[9] Blender Online Community. *Blender - a 3D Modelling and Rendering Package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: http://www.blender.org.

[10] F. Dellaert. "Factor Graphs: Exploiting Structure in Robotics". In: *Annual Review of Control, Robotics, and Autonomous Systems* vol. 4 (2021), pp. 141–166. DOI: 10.1146/annurev-control-061520-010504.

[11] F. Dellaert and GTSAM Contributors. *borglab/gtsam*. Version 4.2a8. May 2022. DOI: 10.5281/zenodo.5794541. URL: https://github.com/borglab/gtsam.

[12] F. Dellaert and GTSAM Contributors. *GTSAM: Georgia Tech Smoothing and Mapping library*. https://gtsam.org/doxygen. Accessed: 2023-11-11.

[13] F. Dellaert and M. Kaess. *Factor Graphs for Robot Perception*. Foundations and Trends in Robotics, Vol. 6, 2017. URL: http://www.cs.cmu.edu/~kaess/pub/Dellaert17fnt.pdf.

[14] L. Du. *Edge Detection in 3D Point Clouds for Industrial Applications*. 2020.

[15] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. 2nd ed. New York: Wiley, 2001. ISBN: 978-0-471-05669-0.

[16] J. Elseberg, D. Borrmann, and A. Nüchter. "Efficient Processing of Large 3D Point Clouds". In: *2011 XXIII International Symposium on Information, Communication and Automation Technologies*. 2011, pp. 1–7. DOI: 10.1109/ICAT.2011.6102102.

[17] L. Euler. "Formulae Generales pro Translatione Quacunque Corporum Rigidorum". In: *Novi Commentarii Academiae Scientiarum Petropolitanae* vol. 20 (1775), pp. 189–207.

[18] G. Fischer and B. Springborn. *Lineare Algebra: Eine Einführung für Studienanfänger*. 19th ed. Munich: Springer Spektrum, 2020. ISBN: 978-3-662-61644-4.

[19] M. Fischler and R. Bolles. "Randum Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Communications of the ACM* vol. 24 (1981), pp. 381–395. DOI: 10.1145/358669.358692.

[20] A. Galarza and J. Seade. *Introduction to Classical Geometries*. 1st ed. Basel: Birkhäuser, 2007. ISBN: 978-3-764-37517-1.

[21] J. Gallier. *Geometric Methods and Applications: For Computer Science and Engineering*. 2nd ed. Philadelphia: Springer, 2011. ISBN: 978-1-441-99960-3.

[22] D. Ginzburg, I. Piatetski-Shapiro, and S. Rallis. "*L* Functions for the Orthogonal Group". In: *Memoirs of the American Mathematical Society* vol. 128, no. 611 (1997).

[23] R. Goldman. "Understanding Quaternions". In: *Graphical Models* vol. 73 (2010), pp. 21–49. DOI: 10.1016/j.gmod.2010.10.004.

[24] H. Goldstein, C. Poole, and J. Safko. *Classical Mechanics*. 3rd ed. USA: Pearson, 2001. ISBN: 978-0-201-65702-9.

[25] B. Guenin, J. Könemann, and L. Tuncel. *A Gentle Introduction to Optimization*. 1st ed. Waterloo: Cambridge University Press, 2014. ISBN: 978-1-107-05344-1.

[26] B. Hall. *Lie Groups, Lie Algebras, and Representations: An Elementary Introduction*. 2nd ed. Springer, 2015. ISBN: 978-3-319-37433-8.

[27] R. Haralick et al. "Pose Estimation from Corresponding Point Data". In: *IEEE Transactions on Systems, Man, and Cybernetics* vol. 19, no. 6 (1989), pp. 1426–1446. DOI: 10.1109/21.44063.

[28] C. Harris et al. "Array Programming with NumPy". In: *Nature* vol. 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[29] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. New York: Cambridge University Press, 2004. ISBN: 978-0-521-54051-3.

[30] J. Hartung and B. Elpelt. *Multivariate Statistik: Lehr- und Handbuch der Angewandten Statistik*. 7th ed. München/Wien: R Oldenbourg Verlag, 2007. ISBN: 978-3-486-58234-5.

[31] A. Hatcher. *Algebraic Topology*. 1st ed. Cambridge University Press, 2002. ISBN: 978-0-521-79540-1.

[32] H. Havlicek. *Lineare Algebra für Technische Mathematiker*. 3rd ed. Berlin: Heldermann Verlag, 2012. ISBN: 978-3-885-38116-7.

[33] R. Horn and C. Johnson. *Matrix Analysis*. 2nd ed. New York: Cambridge University Press, 2013. ISBN: 978-0-521-54823-6.

[34] D. Huynh. "Metrics for 3D Rotations: Comparison and Analysis". In: *J Math Imaging Vis* vol. 35 (2009), pp. 155–164. DOI: 10.1007/s10851-009-0161-2.

[35] KeenTools. *GeoTracker*. 2023. URL: https://keentools.io/products/geotracker-for-blender.

[36] F. Kschischang, B. Frey, and H. Loeliger. "Factor Graphs and the Sum-Product Algorithm". In: *IEEE Transactions on Information Theory* vol. 47, no. 2 (2001), pp. 498–519. DOI: 10.1109/18.910572.

[37] J. Lafontaine. *An Introduction to Differential Manifolds*. 2nd ed. Montpellier: Springer, 2015. ISBN: 978-3-319-20734-6.

[38] L. Landau and E. Lifshitz. *Mechanics*. 3rd ed. Moscow: Elsevier, 1976. ISBN: 978-0-750-62896-9.

[39] D. Lay, S. Lay, and J. McDonald. *Linear Algebra and its Applications*. 5th ed. Washington: Pearson, 2016. ISBN: 978-0-321-98238-4.

[40] J. Lee. *Introduction to Smooth Manifolds*. 2nd ed. Seattle: Springer, 2013. ISBN: 978-1-441-99981-8.

[41] L. Linsen. *Point Cloud Representation*. Technical Report. Faculty of Computer Science, University of Karlsruhe, 2001. 18 pp.

[42] H. Loeliger. "An Introduction to Factor Graphs". In: *IEEE Signal Processing Magazine* vol. 21 (2004), pp. 28–41. DOI: 10.1109/MSP.2004.1267047.

[43] J. Mäkinen. "Rotation Manifold SO(3) and its Tangential Vectors". In: *Computational Mechanics* vol. 42 (2008), pp. 907–919. DOI: 10.1007/s00466-008-0293-z.

[44] MAN. *Der MAN TGS: Am Liebsten am Limit*. Accessed: 2023-11-11. URL: https://www.man.eu/de/de/lkw/alle-modelle/der-man-tgs/uebersicht/uebersicht-tgs.html.

[45] D. Marquardt. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters". In: *Journal of the Society for Industrial and Applied Mathematics* vol. 11, no. 2 (1963), pp. 431–441. DOI: 10.1137/0111030.

[46] J.M. McCarthy. *Introduction to Theoretical Kinematics*. 1st ed. Irvine: MIT Press, 1990. ISBN: 978-0-262-13252-7.

[47] J. Munkres. *Topology*. 2nd ed. Edinburgh: Pearson, 2014. ISBN: 978-1-292-02362-5.

[48] I. Najfeld and T. Havel. "Derivatives of the Matrix Exponential and Their Commputation". In: *Advances in Applied Mathematics* vol. 16 (1995), pp. 321–375. DOI: 10.1006/aama.1995.1017.

[49] E. Ostertagova. "Modelling Using Polynomial Regression". In: *Procedia Engineering* vol. 48 (2012), pp. 500–506. DOI: 10.1016/j.proeng.2012.09.545.

[50] Palfinger. *BM 214*. Accessed: 2023-11-11. URL: https://www.palfinger.com/de/produkte/mitnahmestapler/modelle/bm-214_p_562.

[51] K. Rosen. *Discrete Mathematics and its Applications*. 7th ed. New York: McGraw-Hill, 2012. ISBN: 978-0-073-38309-5.

[52] P. Schiller. "Robust Pose Estimation of 3D Objects with Symmetries in Point Clouds". Personal Communication. 2024.

[53] Stereolabs. *Coordinate Frames: Selecting a Coordinate Frame*. Accessed: 2023-11-11. URL: https://www.stereolabs.com/docs/positional-tracking/coordinate-frames/.

[54] Stereolabs. *ZED 2i*. Accessed: 2023-11-11. URL: https://www.stereolabs.com/zed-2i/.

[55] K. Symon. *Mechanics*. 2nd ed. Massachusetts: Addison-Wesley Publishing Company, Inc., 1960. ISBN: 978-0-014-04666-9.

[56] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Cambridge: The MIT Press, 2005. ISBN: 978-0-262-20162-9.

[57] R. Tobler and S. Maierhofer. "A Mesh Data Structure for Rendering and Subdivision". In: *The 14-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2006*. WSCG '2006. Pilsen, 2006, pp. 157–162. ISBN: 978-8-086-94305-3.

[58] L. Trefethen and D. Bau. *Numerical Linear Algebra*. Philadelphia: Society for Industrial and Applied Mathematics, 1997. ISBN: 978-0-898-71487-6.

[59] L. Tu. *An Introduction to Manifolds*. 2nd ed. Medford: Springer, 2011. ISBN: 978-1-441-97399-3.

[60] C. Weber, S. Hahmann, and H. Hagen. "Methods for Feature Detection in Point Clouds". In: *Visualization of Large and Unstructured Data Sets - Applications in Geospatial Planning, Modeling and Engineering (IRTG 1131 Workshop)*. Vol. 19. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany, 2011, pp. 90–99. ISBN: 978-3-939-89729-3. DOI: 10.4230/OASIcs.VLUDS.2010.90.

[61] Y. Wong and Y. Au-Yeung. "An Elementary and Simple Proof of the Connectedness of the Classical Groups". In: *The American Mathematical Monthly* vol. 74, no. 8 (1967), pp. 964–966. DOI: 10.2307/2315278.

[62] K. Zeyringer. *Fußball: Eine Kulturgeschichte*. Frankfurt am Main: Fischer Taschenbuch, 2016. ISBN: 978-3-596-03587-8.

[63] Z. Zhang, F. Wu, and W. Lee. "Factor Graph Neural Network". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS'20. New York, 2020, pp. 8577–8587. ISBN: 978-1-713-82954-6.

[64] Q. Zhou, J. Park, and V. Koltun. "Open3D: A Modern Library for 3D Data Processing". In: *arXiv* (2018). DOI: 10.48550/arXiv.1801.09847.