Direct Visual Odometry on a GPU

Matias Christensen

Supervisor:
Frank Lindseth

NTNU
Norwegian University of Science and Technology

Department of Computer Science (IDI)
Norwegian University of Science and Technology
TDT4900 - Computer Science, Master's Thesis, Spring 2019
Delivery: 28. May 2019

Abstract

This thesis covers the implementation of the DVO (Direct Visual Odometry) algorithm on a GPGPU (General-purpose Graphical Processing Unit). DVO is a direct visual odometry algorithm capable of accurately estimating the trajectory of a camera from a RGB-D video feed. We cover the necessary mathematical background thready needed to reason about motion in three-dimensional space, as well as how to propose the trajectory as an optimization problem we can solve in a efficient manner. A overview of GPGPU programming is given, illustrating how this differs from CPU programming. We then show how the DVO algorithm can be implemented in a efficient manner on a GPGPU and how this differs from a CPU implementation. Finally, a evaluation of how altering the parameters of the algorithm effects the efficiency and accuracy of the implementation is given.

Abstrakt

Denne masteroppgaven dekker implementasjonen av DVO (Direct Visual Odometry) algoritmen på en GPGPU (General-purpose Graphical Processing Unit). DVO er en direkte visuell odometri algoritme som er i stand til å estimere banen til et kamera på en nøyaktig måte, gitt en RGB-D video. Vi dekker den nødvendige matematiske teorien som trengs for å kunne drøfte rundt bevegelse i tre-dimensjonalt rom, i tilegg til hvordan vi kan definere bane estimeringen som et optimaliseringsproblem og løse det på en effektiv måte. Vi gir så et overblikk over GPGPU programmering som illustrerer hvordan det skiller seg fra CPU programmering. Vi viser så hvordan DVO algoritmen kan implementerer på en effektiv måte på en GPGPU og hvordan den skiller seg fra en CPU implementasjon. Til slutt evaluerer vi hvordan en endring i algoritmens parametere påvirker effektiviteten og nøyaktigheten til implementasjonen.

Table of Contents

List of Figures

Acronyms

CPU Central Processing Unit. 39–41, 56

DSP Digital Signal Processor. 39

DVO Direct Visual Odometry. 51, 52

FPGA Field-programmable Gate Array. 39

GPGPU General-purpose Graphical Processing Unit. 1, 39–42, 51

GPU Graphical Processing Unit. 1, 4, 39, 41–43, 45, 51, 56

IMU Inertial measurement unit. 54

RGB Red Green Blue. 20, 56

SIMD Single Instruction, Multiple Data. 40

SLAM Simultaneous Localization and Mapping. 2, 4, 11, 21, 26, 35

VO Visual Odometry. 2, 4, 11, 21, 26, 35

Symbols

$\mathbf{x}$, $\mathbf{u}$	=	Vectors.
$\tilde{\mathbf{x}}$, $\tilde{\mathbf{u}}$	=	Homogeneous vectors.
$\mathbf{A}$, $\mathbf{K}$	=	Matrices.
$\mathfrak{so}(3)$, $\mathfrak{se}(3)$	=	Lie Algebras.
$\mathbf{SO}(3)$, $\mathbf{SE}(3)$	=	Lie Groups.
$\hat{\boldsymbol{\omega}}$	=	Skew-symmetric matrices.
Ω	=	Set of all point in an image.

Chapter 1

Introduction

We will, in this chapter, introduce this thesis and try to give context for the work that has been done. Section 1.1 overviews the motivation for why the subject matter of this thesis was chosen to be what it is and what the goals where. In section 1.2 we give a short account of some of the historical developments that have been made in the area of visual odometry and SLAM, this is to provide some context for the current state of the field and give some indication for where it is heading. We then provide an overview of how the different visual odometry methods differ in section 1.3. This section attempts to show how the method discussed in this thesis fits into the varied landscape of previous and current visual odometry methods.

1.1 Motivation & Goals

Visual odometry and SLAM methods have in recent years showed them self to be increasingly relevant as their uses increase in emerging areas such as autonomous vehicles, augmented reality, and virtual reality. We have also seen a considerable increase in the performance of Graphical Processing Units (GPUs) and the ability to do General-purpose Graphical Processing Unit (GPGPU) programming. Finally, we have also seen the introduction of small, embeddable GPUs, such as NVIDIA Jetson or NVIDIA Drive, making it possible to run GPGPU programs on mobile robotic platforms such as cars and drones. These factors combine to make the use of GPGPU programming in visual odometry and SLAM, a great opportunity to develop robotic systems with novel capabilities.

We are then left with the question of how we best can utilize such GPU devices for visual odometry, and how can we implement the algorithms using GPGPU programming. For this thesis, we have chosen to focus on the DVO (Direct Visual Odometry) algorithm, first published in [15]. It is a modern direct visual odometry method that has shown good performance and is also simple enough that the GPGPU implementation from scratch is achievable within the scope of this thesis.

We have chosen to formalize the focus of the thesis into three research questions that we aim to answer through the following chapters:

1. How can we best implement DVO on a GPGPU?
2. How does the implementation on a GPGPU differ from the already published CPU implementation?
3. How does a change in the parameters of the GPGPU implementation change the performance and accuracy of the algorithm?

1.2 A Concise History of Visual Odometry & SLAM

The history of computing three-dimensional structure from two-dimensional images long precedes the introduction of computers and digital cameras. Early work such as [17] describes a method of manually calculating three-dimensional structure and its distance from the observer by manually selecting corresponding image points.

The introduction of computers and digital photography allowed the field that we now consider computer vision to begin. Methods such as Harris corner detection[10] and later FAST[35] was introduced to detect corners in the image computationally. Such methods were used to create feature trackers such as the KLT-tracker[34] that tracked identified feature points over multiple frames.

With the introduction of robust feature descriptors such as SIFT[18], SURF[2] and ORB[27] correspondences could now match between images. This method made it possible to use bundle adjustment methods to calculate camera transformations and scene structure.

Early visual odometer methods like EKF-SLAM[20] used Kalman filters to represent and update the pose through filtering. Later methods such as PTAM[16] showed it was possible to create real-time visual odometry systems based on bundle adjustment.

An early example of dense methods operating directly on the image intensities was shown in [19], which estimated the dense depth in the image. This method was improved further in methods like DTAM[22], which could generate a detailed geometry of the scene using dense methods.

In the last few years, we have seen an increasing number of methods which entirely use a dense formulation such as LSD-SLAM[6] and DSO[5], and others which use a partial direct formulation such as SVO[9].

1.3 Taxonomy of Visual Odometry Methods

In this section, we will give an overview of the three principal axes we can use to classify a Visual Odometry (VO) or Simultaneous Localization and Mapping (SLAM) algorithms. While these distinctions are not binary and solutions falling in the middle of any of the three classifications exists, it is still a useful distinction that allows us to classify and compare different methods.

1.3.1 Filtering vs. Smoothing

The difference between filtering- and smoothing-methods lie mainly in how the methods represent the current state of the model.

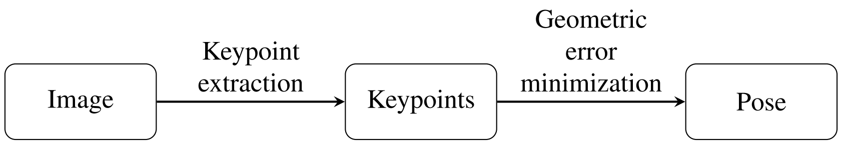
Filtering methods use a joint probability distribution over all states in the system. When the system gets a new measurement, it is used to update the probability distribution and reduce the uncertainty of the system. New parameters are usually added with large uncertainties, and old values are marginalized away when no longer needed.

Smoothing methods, also called optimization methods, for visual odometry, represent the state in the form of a non-linear optimization function that is continuously optimized in the background. They will in most cases, aggressively marginalize variables outside a small window to keep the optimization computationally tractable.

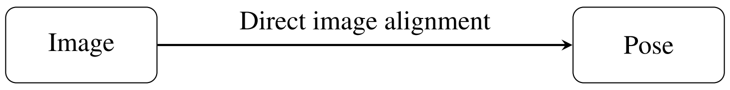
In filtering methods, usually, only the current pose of the camera is represented as the old pose is marginalized as soon as a new pose is obtained. The estimation of a new pose usually involves a linearization of a non-linear function to propagate the probability distribution to the new pose. As the old pose is marginalized away, the system loses the ability to re-linearize at a new point if information giving a better estimate of an old pose is obtain. Smoothing methods do not suffer from this problem as long as old pose remains within the optimization window, as the non-linear optimization function is re-linearized for each step of the optimization function.

In [31], the accuracy of filtering and smoothing methods are compared. Here the conclusion is drawn that filtering methods work best for small problems with few parameters, while larger problems with more points work better with smoothing methods.

1.3.2 Direct vs Indirect



(a) Indirect motion estimation



(b) Direct motion estimation

Figure 1.1: Direct vs. indirect motion estimation

The difference between direct and indirect methods is how we represent the image information and how we optimize using them.

With *indirect* methods, we first perform feature extraction on the image to create a number of feature-descriptors and then perform optimization on those descriptors, as illustrated in fig. 1.1a. These feature-descriptors are created from points in the image with a high gradient, like corners, and take the form of a vector that should be distinct from other feature points. The feature extraction can be computationally costly, and the point correspondence will often contain some outliers because of error in the matching. In the optimization stage, the features are matched with features extracted in earlier images to build point correspondences between images. The pose is retrieved by optimizing over the geometric error, which is the Euclidean distance between the observed feature image coordinate and the coordinate of the matched feature in a different image projected into the image. A geometric error function over two images with known point depths is shown in eq. (1.1), here $\boldsymbol{\xi}$ is the pose between the images, $\mathbf{x}_i$ and $\mathbf{y}_i$ are the same point detected in each image, d_i is the depth and ω is the re-projection function projecting a point in one image to the other given a depth and a pose.

$$E(\boldsymbol{\xi}) = \sum_i \|\mathbf{x}_i - \omega\left(\mathbf{y}_i, \boldsymbol{\xi}, d_i\right)\|_2^2 \tag{1.1}$$

Direct methods, on the other hand, use the image directly without any features extraction step, as shown in fig. 1.1b. There may be a point selection process to select an appropriate subset of the image to use, but this subset of pixels will be used directly without any feature extraction. Direct method will usually use a much larger fraction of the image in the optimization and will, therefore, make use of information that might exist in the image that indirect method will discard, such as low gradient areas. When optimizing, we use a photometric error function that compares the intensity values of pixels re-projected from one image to the other. While direct methods avoid the need to compute feature vectors, the optimization usually involves far more points and is, therefore, more computationally expensive and requires some ingenuity to allow it to run in real-time. A photometric error function for two images with known depth is shown in eq. (1.2). Here $\boldsymbol{\xi}$ is the pose between the images, I_1 and I_2 are the image functions, $\mathbf{x}_i$ are the image points, d_i is the point depths and ω is the warping function that projects a point from one image to the other given a pose and a depth value.

$$E(\boldsymbol{\xi}) = \sum_i \left(I_1(\mathbf{x}_i) - I_2(\omega(\mathbf{x}_i, \boldsymbol{\xi}, d_i))\right)^2 \tag{1.2}$$

Some methods like [9] use a combination of both direct and indirect methods and are referred to as *semi-direct* methods.

1.3.3 Dense vs Sparse

The distinction between dense and sparse method is how the images are used in the method.

Dense methods will use the entire image in the optimization and attempt to estimate depth for every pixel in the image, often combined with a smoothness prior to the depth to convexify the optimization problem. Fully dense method, such as [22] are very computationally expensive and requires GPU implementations to run in real-time. The use of all pixels also doesn't add to the accuracy of the camera trajectory and is usually done in the context of 3D reconstructions rather than VO or SLAM.

In *sparse* method, we instead use a subset of independent points in the image without any smoothness prior. This change gives us far fewer point to optimize over and lowers the computational overhead considerably.

There exist also methods classified as *semi-dense*, such as [6]. Here connected patches of high gradient areas in the image are used and optimized with a smoothness prior on the depth.

Chapter 2

Theory

2.1 Three-dimensional Geometry

In this theses, we will concern our self with the estimation of motion within a 3D environment. Thus we require mathematical tools to describe the three-dimensional world. This section describes how we will represent position, translation, rotation, and motion mathematically, and how we will work with them.

2.1.1 Position & Translation

We describe a point in three-dimensional space with a vector $\mathbf{x}$, which consist of three scalar values which indicate it's position along the cardinal directions.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

We can translate the point $\mathbf{x}$ by adding a translation $\mathbf{v}$ to it.

$$\mathbf{x} + \mathbf{v} = \begin{bmatrix} x_1 + v_1 \\ x_2 + v_2 \\ x_3 + v_3 \end{bmatrix}$$

While a translation and a point may appear identical, it is useful to separate the two as distinct types of objects by letting them represent different things. We let a point be a position within a coordinate system, while a translation is a linear movement that can be applied to any point.

2.1.2 Rotation

There are multiple ways of representing rotations three-dimensional space. We will here give an outline of the most common methods and justify our representation of choice.

Euler-angles

The simplest way to describe three-dimensional rotation is to use Euler-angles. Here we describe the final rotation by successive rotation around the principal axis, as shown in fig. 2.1. There are multiple valid sequences of axes to rotate around when using which introduces ambiguity. Euler-angles also suffer from a phenomenon known as "gimbal-lock," where they lose a degree of freedom. This problem makes Euler-angles unsuitable for the optimization methods we will employ in the latter part of this thesis.

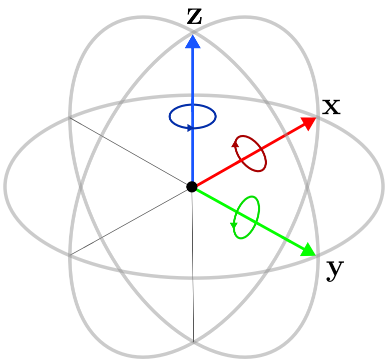


Figure 2.1: Euler angles

Quaternions

Quaternions represent rotation as a point on a four-dimensional sphere. Unlike Euler-angles, quaternions are a *singularity-free* representation of rotation. They are parameterized by four real values and are given by $\mathbf{q} = (x, y, z, w)^T \in \mathbb{S}^3$. Quaternions are however constrained, as a valid quaternion must be of unit length, that is $\sqrt{x^2 + y^2 + z^2 + w^2} = 1$. This constraint makes optimization with quaternions difficult, as only a small subset of possible iteration step will move a valid quaternion to another valid quaternion. We could mitigate this problem by heavily penalizing a quaternion with a unit length different than 1 in the optimization cost function. It is also possible to convert any invalid quaternion to the closest possible valid quaternion after each step, but this can significantly slow down the optimization. Neither of these solutions is ideal, which is why we will avoid the use of quaternions in our optimization.

Lie Representation

In this thesis, we will instead employ Lie Groups and Lie Algebra for representing rotations, which we will introduce in section 2.1.5. We shall see that this method avoids any of the problems of Euler-angles or quaternions, and are therefore well suited for our purposes. We shall also see that the representation can be extended to represent any Euclidean transformation, that is a combination of a rotation and a translation.

2.1.3 Homogeneous Coordinates

The position vectors introduced in section 2.1.1 may be extended to what is called homogeneous coordinates. We will later show in section 2.1.5 why this is useful.

Given a vector $\mathbf{u}$ we can extend it to a homogeneous vector $\tilde{\mathbf{u}}$ by appending a one as the fourth element, as shown in eq. (2.1).

$$\mathbf{u} = \begin{bmatrix} x \\ y \\ x \end{bmatrix} \rightarrow \tilde{\mathbf{u}} = \begin{bmatrix} x \\ y \\ x \\ 1 \end{bmatrix} \tag{2.1}$$

The vector may be scaled by a scalar w as in eq. (2.2). We consider two homogeneous vectors that differ only by a scaling factor to be representing the same vector.

$$\tilde{w} \cdot \tilde{\mathbf{u}} = \tilde{w} \begin{bmatrix} x \\ y \\ x \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{x} \\ \tilde{w} \end{bmatrix} \tag{2.2}$$

To recover an inhomogeneous vector from a homogeneous one, we can divide the vector by the fourth element to get the correct scale. And then remove the fourth element, as shown in eq. (2.3).

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{x} \\ \tilde{w} \end{bmatrix} \rightarrow \frac{1}{\tilde{w}} \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{x} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} x \\ y \\ x \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ x \end{bmatrix} \tag{2.3}$$

2.1.4 Skew-symmetric matrices

$$\begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \tag{2.4}$$

A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is called *skew-symmetric* if $\mathbf{A}^T = -\mathbf{A}$. A skew-symmetric matrix $\hat{u}$ of size 3×3 must be on the form given in eq. (2.4). This can trivially be seen, as any diagonal elements must be equal to 0, to be equal to it's own negation, that is $\forall i, j | u_{ij} = -u_{ji} \implies u_{ii} = 0$. Any off-diagonal element must be equal to the negation of its corresponding transpose element, that is $\forall i, j | u_{ij} = -u_{ji}$. We therefore get eq. (2.4) as the only 3×3 solution.

$$\boldsymbol{\omega} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \quad \hat{\boldsymbol{\omega}} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \tag{2.5}$$

As the skew-symmetric is parameterized by three real values, we can define a **hat-operator** $\hat{(\cdot)}$ which converts any three-element vector $\boldsymbol{\omega} = (\omega_1, \omega_2, \omega_3)^T$ to it's corresponding skew-symmetric matrix as given in eq. (2.5).

$$\left(\begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}\right)^\vee = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{2.6}$$

Similarly, we can define a inverse **vee-operator** $(\cdot)^\vee$ which converts any skew-symmetric 3×3 matrix to it's vector representation as show in eq. (2.6). We therefore have that $(\hat{\boldsymbol{\omega}})^\vee = \boldsymbol{\omega}$.

The skew-symmetric matrix corresponding to a given vector has the property that multiplication of the skew-symmetric matrix with another vector is equivalent to the cross-product of the two vectors, as provided in eq. (2.7).

$$u \times v = \hat{u}v \tag{2.7}$$

2.1.5 Lie Algebra & Lie Groups

We will in this section, describe the use of Lie Groups and Lie Algebra for representing rotations and Euclidean transformations of rigid-bodies. We shall see that this form of representation will allow us to avoid the problems outlined in section 2.1.2, as well as providing methods for composing, inverting, and differentiating transformations.

Definitions

The non-rigorous definition of a *Lie Group* is a *group* that is also a *differentiable manifold*. A group is set, $\mathbf{G}$, combined with a binary operator, $\circ$, which together follow the four group axioms:

- **Closure:** For all $a, b \in \mathbf{G}$, we have that $a \circ b \in \mathbf{G}$.
- **Associativity:** For all $a, b, c \in \mathbf{G}$, we have that $(a \circ b) \circ c = a \circ (b \circ c)$.
- **Identity element:** There exist an unique element e such that for all $a \in \mathbf{G}$, we have that $a \circ e = e \circ a = a$.
- **Inverse element:** For all $a \in \mathbf{G}$ there exist and inverse element a^{-1}, such that $a \circ a^{-1} = a^{-1} \circ a = e$.

A differentiable manifold is a manifold where each point on the manifold is locally similar to Euclidean space, as illustrated in fig. 2.2. This definition means that for any given point on the manifold and a maximum non-zero error, we can define a sufficiently small neighborhood, with non-zero area, around the point where the error between the manifold and the Euclidean space is less than the given error.

We can, therefore, see a Lie Group as a smooth, connected manifold where we continuously move from any point on the manifold to any other, along its surface. We may also employ the methods calculus on the manifold to reason about its curvature.

A *Lie Algebra* is a vector space $\mathfrak{g}$ together with binary operator $\mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}; (x, y) \mapsto [x, y]$ called the Lie bracket, which satisfy the following axioms:

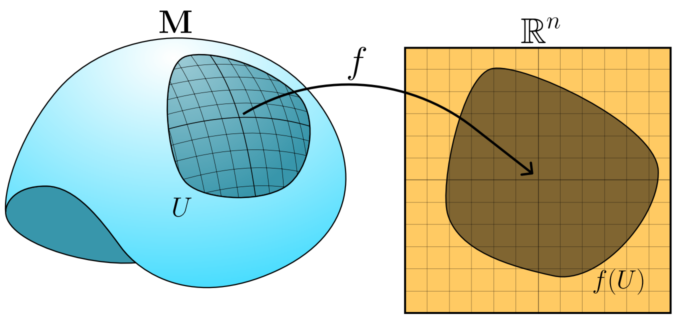


Figure 2.2: The function f maps the region U of the differential manifold M to the region $f(U)$ of the euclidean space $\mathbb{R}^n$.

- **Bilinearity:** For all $a, b \in \mathbb{R}$ and $x, y, z \in \mathfrak{g}$ we have that $[ax + by, z] = a[x, z] + b[y, z]$ and $[z, ax + by] = a[z, x] + b[z, y]$.
- **Alternativity:** For all $x \in \mathfrak{g}$ we have that $[x, x] = 0$.
- **The Jacobi identity:** For $x, y, z \in \mathfrak{g}$ we have that $[x, [y, z]] + [z, [x, y]] + [y, [z, x]] = 0$.

There exist a close connection between any Lie group $\mathbf{G}$ and its corresponding Lie algebra $\mathfrak{g}$. A Lie algebra is the tangent space of the identity element of the Lie group. We illustrate an example of this in fig. 2.3, with the Lie group $\mathbf{SO}(3)$ and Lie algebra $\mathfrak{so}(3)$, which we will introduce in detail in the following section. There also a function $\exp(\boldsymbol{\omega})$ which maps any element $\boldsymbol{\omega}$ of the Lie algebra to its corresponding element in the Lie group, as well as a $\log(\mathbf{R})$ function, which performs the inverse.

There exists many corresponding pairs of Lie groups and Lie algebras. We will, in this thesis, concern our self with those that are relevant in the context VO and SLAM methods. Those are the Lie groups $\mathbf{SO}(3)$ and $\mathbf{SE}(3)$ and their corresponding Lie algebras $\mathfrak{so}(3)$ and $\mathfrak{se}(3)$. The pair $\mathbf{SIM}(3)$ and $\mathfrak{sim}(3)$ are also used in some methods, such as [6], but will not be covered here for the sake of brevity.

Rotations in 3D: $\mathbf{SO}(3)$ & $\mathfrak{so}(3)$

The elements of the Lie Group $\mathbf{SO}(3)$ are represented by 3-by-3 orthonormal matrices. To rotate a point $\mathbf{x}$ by a rotation given as an matrix $\mathbf{R}$ of $\mathbf{SO}(3)$, we perform matrix-vector multiplication as shown in eq. (2.8).

$$\mathbf{x}' = \mathbf{R}\mathbf{x} \tag{2.8}$$

As the elements of $\mathbf{SO}(3)$ are orthonormal matrices, the inverse is equal to the transpose, that is $\mathbf{R}^{-1} = \mathbf{R}^T$. This rule can trivially be seen from eq. (2.9) where we view $\mathbf{R}$ as a matrix of column vectors. By multiplying with the transpose, we get a matrix of dot products. As each column vector is orthogonal to the others, the dot product of any two different column vectors will be zero. Likewise, as all the column vectors are unit length,

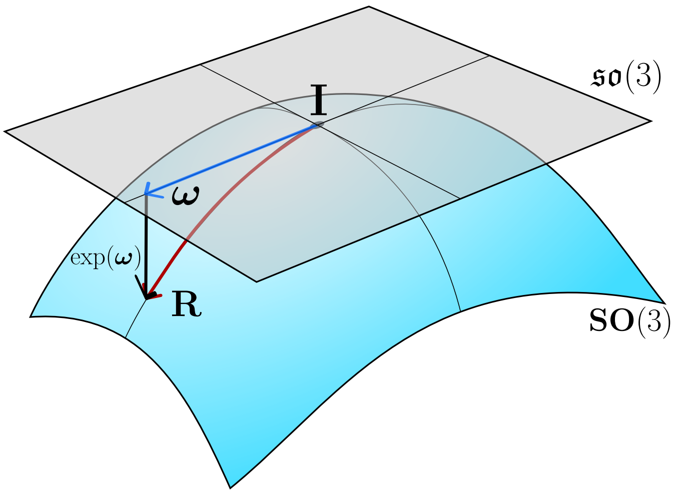


Figure 2.3: $\mathfrak{so}(3)$ exists as a 3-dimensional vector space in $\mathbb{R}^9$ while $\mathbf{SO}(3)$ is a 3-dimensional differential manifold in $\mathbb{R}^9$. $\mathfrak{so}(3)$ lies tangent to $\mathbf{SO}(3)$ at the identity $\mathbf{I}$.

the dot product of a vector with itself will give the value one. The result is the identity matrix, which again implies that the transpose is equal to the inverse.

$$\begin{aligned}
\mathbf{R}\mathbf{R}^T &= \begin{bmatrix} | & | & | \\ r_1 & r_2 & r_3 \\ | & | & | \end{bmatrix} \begin{bmatrix} — & r_1^T & — \\ — & r_2^T & — \\ — & r_3^T & — \end{bmatrix} \\
&= \begin{bmatrix} a_1 \cdot a_1 & a_1 \cdot a_2 & a_1 \cdot a_3 \\ a_2 \cdot a_1 & a_2 \cdot a_2 & a_2 \cdot a_3 \\ a_3 \cdot a_1 & a_3 \cdot a_2 & a_3 \cdot a_3 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{I}
\end{aligned} \tag{2.9}$$

Two rotations compose into a single rotation by performing matrix-matrix multiplication. It is, therefore, as shown in eq. (2.10), equivalent to multiply a vector with a series of rotation matrices in turn, as it is to multiply it once with the resulting composed rotation matrix. Note that this operation is not commutative, and $\mathbf{R}_1 \cdot \mathbf{R}_2 \neq \mathbf{R}_2 \cdot \mathbf{R}_1$ for most rotation matrices.

$$\begin{aligned}
\mathbf{x}' &= \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{x} \\
&= (\mathbf{R}_2 \cdot \mathbf{R}_1) \cdot \mathbf{x} \\
&= \mathbf{R} \cdot \mathbf{x}, \quad \mathbf{R} = \mathbf{R}_2 \cdot \mathbf{R}_1
\end{aligned} \tag{2.10}$$

The elements of $\mathfrak{so}(3)$ are the set of 3-by-3 skew-symmetric matrices. We can represent this set as the linear combination of three matrices. These matrices are called the generators of $\mathfrak{so}(3)$ and are given in eq. (2.11). The generators correspond to the derivatives of rotation around each axis, evaluated at identity.

$$\mathbf{G}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{G}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \quad \mathbf{G}_3 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.11}$$

We can represent any element in $\mathfrak{so}(3)$ by a three-value vector indicating the corresponding linear combination of the generators, as shown in eq. (2.12). We can see that this implements the vee-operator from section 2.1.4.

$$\hat{\boldsymbol{\omega}} = \omega_1 \mathbf{G}_1 + \omega_2 \mathbf{G}_2 + \omega_3 \mathbf{G}_3 \in \mathfrak{so}(3), \quad \forall \boldsymbol{\omega} \in \mathbb{R}^3 \tag{2.12}$$

The exponential map is the mapping from elements in $\mathbf{SO}(3)$, to elements in $\mathfrak{so}(3)$ — this mapping smooth, meaning that a small change in the input causes a small change in the output. The mapping is also differentiable so that we can calculate a Jacobian of the function. And it is invertible.

The exponential map takes the 3-by-3 skew-symmetric matrix and maps it to the corresponding rotation matrix and is defined as in eq. (2.13).

$$\exp(\hat{\boldsymbol{\omega}}) = \mathbf{I} + \hat{\boldsymbol{\omega}} + \frac{1}{2!}\hat{\boldsymbol{\omega}}^2 + \frac{1}{3!}\hat{\boldsymbol{\omega}}^3 + \cdots \tag{2.13}$$

We can rewrite the terms in pairs, as shown in eq. (2.14).

$$\exp(\hat{\boldsymbol{\omega}}) = \mathbf{I} + \sum_{i=0}^{\infty} \left[\frac{\hat{\boldsymbol{\omega}}^{2i+1}}{(2i+1)!} + \frac{\hat{\boldsymbol{\omega}}^{2i+2}}{(2i+2)!} \right] \tag{2.14}$$

Using the property of skew-symmetric matrices, that its cube can be calculated as given in eq. (2.15), we can calculate the identities provided in eq. (2.16).

$$\hat{\boldsymbol{\omega}}^3 = -(\boldsymbol{\omega}^T \boldsymbol{\omega}) \cdot \hat{\boldsymbol{\omega}} \tag{2.15}$$

$$\begin{aligned} \theta^2 &\equiv \boldsymbol{\omega}^T \boldsymbol{\omega} \\ \hat{\boldsymbol{\omega}}^{2i+1} &= (-1)^i \theta^{2i} \hat{\boldsymbol{\omega}} \\ \hat{\boldsymbol{\omega}}^{2i+2} &= (-1)^i \theta^{2i} \hat{\boldsymbol{\omega}}^2 \end{aligned} \tag{2.16}$$

Using these identities we can rewrite eq. (2.14) as shown in eq. (2.17). This final formula is known as the Rodrigues formula and is what we will use to calculate to exponential of $\mathbf{SO}(3)$ elements. This will let us convert any element $\hat{\boldsymbol{\omega}} \in \mathfrak{so}(3)$ to its corresponding element $\mathbf{R} \in \mathbf{SO}(3)$.

$$\begin{aligned}
\exp(\hat{\boldsymbol{\omega}}) &= \mathbf{I} + \left(\sum_{i=0}^{\infty} \frac{(-1)^i \theta^{2i}}{(2i+1)!}\right)\hat{\boldsymbol{\omega}} + \left(\sum_{i=0}^{\infty} \frac{(-1)^i \theta^{2i}}{(2i+2)!}\right)\hat{\boldsymbol{\omega}}^2 \\
&= \mathbf{I} + \left(1 - \frac{\theta^2}{3!} + \frac{\theta^4}{5!} + \cdots\right)\hat{\boldsymbol{\omega}} + \left(\frac{1}{2!} - \frac{\theta^2}{4!} + \frac{\theta^4}{6!} + \cdots\right)\hat{\boldsymbol{\omega}}^2 \\
&= \mathbf{I} + \left(\frac{\sin\theta}{\theta}\right)\hat{\boldsymbol{\omega}} + \left(\frac{1-\cos\theta}{\theta^2}\right)\hat{\boldsymbol{\omega}}^2
\end{aligned} \tag{2.17}$$

To reverse this process, that is convert an element in $\mathbf{R} \in \mathbf{SO}(3)$ to it's respective element in $\hat{\boldsymbol{\omega}} \in \mathfrak{so}(3)$, we can use the equation given in eq. (2.18). To retrieve the vector $\boldsymbol{\omega}$ we can use the vee-operator, $\boldsymbol{\omega} = (\ln(\mathbf{R}))^{\vee}$.

$$\begin{aligned}
\theta &= \arccos\left(\frac{\operatorname{tr}(\mathbf{R} - 1}{2}\right) \\
\ln(\mathbf{R}) &= \frac{\theta}{2\sin\theta}\left(\mathbf{R} - \mathbf{R}^T\right)
\end{aligned} \tag{2.18}$$

When optimizing over elements in $\mathfrak{so}(3)$, we will need to be able to calculate the gradient of expressions such as the one given in eq. (2.19). This equation is a function rotating a point $\mathbf{x}$ using the rotation matrix $\mathbf{R}$, which again is created from the skew-symmetric matrix $\hat{\boldsymbol{\omega}}$ parameterized by the vector $\boldsymbol{\omega}$.

$$\mathbf{y} = f(\mathbf{R}, \mathbf{x}) = \mathbf{R} \cdot \mathbf{x}, \quad \mathbf{R} = \exp(\hat{\boldsymbol{\omega}}) \tag{2.19}$$

The gradient of this function with respect to $\mathbf{x}$ is given in eq. (2.20), and is simply $\mathbf{R}$. This result makes intuitive sense as any small change in the original point will cause the same change in the resulting point with the direction rotated.

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{R} \tag{2.20}$$

We can calculate the gradient of the same function with respect to the vector $\boldsymbol{\omega}$, which is given in eq. (2.21). We use the fact that any infinitesimal change around a specific rotation $\mathbf{R}$, can be modeled as $\mathbf{R} \cdot \exp(\hat{\boldsymbol{\psi}})$ where $\exp(\hat{\boldsymbol{\psi}})$ is an infinitesimal rotation away from identity. The gradient around the identity rotation is given by the generators of $\mathfrak{so}(3)$, as given in eq. (2.11). We, therefore, get that the final gradient is given by the skew-symmetric matrix of $-\mathbf{y}$.

$$\begin{aligned}
\frac{\partial \mathbf{y}}{\partial \boldsymbol{\omega}} &= \frac{\partial}{\partial \boldsymbol{\omega}} \mathbf{R} \cdot \mathbf{x} \\
&= \frac{\partial}{\partial \boldsymbol{\psi}}\bigg|_{\boldsymbol{\psi}=\mathbf{0}} \Big(\exp(\hat{\boldsymbol{\psi}}) \cdot \mathbf{R}\Big) \cdot \mathbf{x} \\
&= \frac{\partial}{\partial \boldsymbol{\psi}}\bigg|_{\boldsymbol{\psi}=\mathbf{0}} \Big(\exp(\hat{\boldsymbol{\psi}})\Big) \cdot (\mathbf{R} \cdot \mathbf{x}) \\
&= \frac{\partial}{\partial \boldsymbol{\psi}}\bigg|_{\boldsymbol{\psi}=\mathbf{0}} \Big(\exp(\hat{\boldsymbol{\psi}})\Big) \cdot \mathbf{y} \\
&= (\mathbf{G}_1\mathbf{y}|\mathbf{G}_2\mathbf{y}|\mathbf{G}_3\mathbf{y}) \\
&= -\hat{\mathbf{y}}
\end{aligned} \tag{2.21}$$

Euclidean transformations in 3D: $\mathbf{SE}(3)$ & $\mathfrak{se}(3)$

The elements $\mathbf{T}$ of the Lie-group $\mathbf{SE}(3)$ are represented by a 4×4 matrix as shown in block form in eq. (2.22). The upper left 3×3 sub-matrix is an element of $\mathbf{SO}(3)$, encoding the rotation of the transformation. The translation is represented by the vector $\mathbf{t}$.

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}, \quad \mathbf{T} \in \mathbf{SE}(3),\ \mathbf{R} \in \mathbf{SO}(3),\ \mathbf{t} \in \mathbb{R}^3 \tag{2.22}$$

A homogeneous vector $\tilde{\mathbf{x}}$ may be transformed using a simple matrix-vector multiplication as shown in eq. (2.23). The vector does not necessarily have to be normalized as in the example. We will get the same inhomogenous vector regardless of whether we perform the normalization before or after the transformation.

$$\mathbf{T} \cdot \tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.23}$$

Two transformations, $\mathbf{T}_1$ and $\mathbf{T}_2$, may be multiplied as shown on eq. (2.24) to create a new transformation combining the transformation of the two. Note that this is like regular matrix-matrix multiplication not commutative and $\mathbf{T}_1 \cdot \mathbf{T}_2 \neq \mathbf{T}_2 \cdot \mathbf{T}_1$ for most transformations.

$$\begin{aligned}
\mathbf{T}_1 \cdot \mathbf{T}_2 &= \begin{bmatrix} \mathbf{R}_1 & \mathbf{t}_1 \\ \mathbf{0} & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_2 & \mathbf{t}_2 \\ \mathbf{0} & 1 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{R}_1\mathbf{R}_2 & \mathbf{R}_1\mathbf{t}_2 + \mathbf{t}_1 \\ \mathbf{0} & 1 \end{bmatrix}
\end{aligned} \tag{2.24}$$

We can also calculate the inverse transformation $\mathbf{T}^{-1}$, such that $\mathbf{T}^{-1} \cdot \mathbf{T} = \mathbf{I}$, as shown in eq. (2.25). Here we use that $\mathbf{R}$ is orthogonal, and therefore $\mathbf{R}^{-1} = \mathbf{R}^T$ to simplify the calculation.

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{2.25}$$

As with $\mathfrak{so}(3)$, the elements of $\mathfrak{se}(3)$ are the linear combinations of the generators of its lie group, $\mathbf{SE}(3)$. Again, the generators are corresponding to the differential of the rotation and translation. We, therefore, get the six generators, as shown in eq. (2.26).

$$\begin{aligned}
\mathbf{G}_1 &= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad
\mathbf{G}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
\mathbf{G}_3 &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad
\mathbf{G}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
\mathbf{G}_5 &= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad
\mathbf{G}_6 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}
\end{aligned} \tag{2.26}$$

We can therefore parameterize any element of $\mathfrak{se}(3)$ by a vector $\boldsymbol{\xi} = (\boldsymbol{\nu}|\boldsymbol{\omega})^T \in \mathbb{R}^6$. Where $\boldsymbol{\nu}$ are the translational parameters, and $\boldsymbol{\omega}$ are the rotational parameters.

$$\hat{\boldsymbol{\xi}} = \xi_1\mathbf{G}_1 + \xi_2\mathbf{G}_2 + \xi_3\mathbf{G}_3 + \xi_4\mathbf{G}_4 + \xi_5\mathbf{G}_5 + \xi_6\mathbf{G}_6 \in \mathfrak{se}(3), \quad \forall \boldsymbol{\xi} \in \mathbb{R}^6 \tag{2.27}$$

We can define a exponential function that maps from $\hat{\boldsymbol{\xi}} \in \mathfrak{se}(3)$ to its corresponding $\mathbf{T} \in \mathbf{SE}(3)$, as is shown in eq. (2.28). We can see that we end up with a block matrix consisting of $\exp(\hat{\boldsymbol{\omega}})$, which we showed how to calculate in section 2.1.5, and $\mathbf{V}\boldsymbol{\nu}$. Where $\mathbf{V}$ is defined as a series.

$$\begin{aligned}
\exp(\hat{\boldsymbol{\xi}}) &= \exp\left(\begin{bmatrix} \hat{\boldsymbol{\omega}} & \boldsymbol{\nu} \\ \mathbf{0} & 0 \end{bmatrix}\right) \\
&= \mathbf{I} + \begin{bmatrix} \hat{\boldsymbol{\omega}} & \boldsymbol{\nu} \\ \mathbf{0} & 0 \end{bmatrix} + \frac{1}{2!}\begin{bmatrix} \hat{\boldsymbol{\omega}}^2 & \hat{\boldsymbol{\omega}}\boldsymbol{\nu} \\ \mathbf{0} & 0 \end{bmatrix} + \frac{1}{3!}\begin{bmatrix} \hat{\boldsymbol{\omega}}^3 & \hat{\boldsymbol{\omega}}^2\boldsymbol{\nu} \\ \mathbf{0} & 0 \end{bmatrix} + \cdots \\
&= \begin{bmatrix} \exp(\hat{\boldsymbol{\omega}}) & \mathbf{V}\boldsymbol{\nu} \\ \mathbf{0} & 1 \end{bmatrix}, \quad \mathbf{V} = \mathbf{I} + \frac{1}{2!}\hat{\boldsymbol{\omega}} + \frac{1}{3!}\hat{\boldsymbol{\omega}}^2 + \cdots
\end{aligned} \tag{2.28}$$

We can again use the identities defined in eq. (2.15), to rewrite the series as shown in eq. (2.29). These identities allow us to rewrite the series to a closed form formula for $\mathbf{V}$.

$$\begin{aligned}
\mathbf{V} &= \mathbf{I} + \sum_{i=0}^{\infty} \left[\frac{\hat{\boldsymbol{\omega}}^{2i+1}}{(2i+2)!} + \frac{\hat{\boldsymbol{\omega}}^{2i+2}}{(2i+3)!} \right] \\
&= \mathbf{I} + \left(\sum_{i=0}^{\infty} \frac{(-1)^i \theta^2 i}{(2i+2)!} \right) \hat{\boldsymbol{\omega}} + \left(\sum_{i=0}^{\infty} \frac{(-1)^i \theta^2 i}{(2i+3)!} \right) \hat{\boldsymbol{\omega}}^2 \\
&= \mathbf{I} + \left(\frac{1}{2!} - \frac{\theta^2}{4!} + \frac{\theta^4}{6!} + \cdots \right) \hat{\boldsymbol{\omega}} + \left(\frac{1}{3!} - \frac{\theta^2}{5!} + \frac{\theta^4}{7!} + \cdots \right) \hat{\boldsymbol{\omega}}^2 \\
&= \mathbf{I} + \left(\frac{1-\cos\theta}{\theta^2} \right) \hat{\boldsymbol{\omega}} + \left(\frac{\theta - \sin\theta}{\theta^3} \right) \hat{\boldsymbol{\omega}}^2
\end{aligned} \tag{2.29}$$

Combining all of this, we get the definition of the exponential function for $\mathfrak{se}(3)$ as shown in eq. (2.30).

$$\begin{aligned}
\boldsymbol{\xi} &= (\boldsymbol{\nu} | \boldsymbol{\omega}) \in \mathbb{R}^6 \\
\theta &= \sqrt{\boldsymbol{\omega}^T \boldsymbol{\omega}} \\
\mathbf{A} &= \frac{\sin\theta}{\theta} \\
\mathbf{B} &= \frac{1-\cos\theta}{\theta^2} \\
\mathbf{C} &= \frac{1-\mathbf{A}}{\theta^2} \\
\mathbf{R} &= \mathbf{I} + \mathbf{A}\hat{\boldsymbol{\omega}} + \mathbf{B}\hat{\boldsymbol{\omega}}^2 \\
\mathbf{V} &= \mathbf{I} + \mathbf{B}\hat{\boldsymbol{\omega}} + \mathbf{C}\hat{\boldsymbol{\omega}}^2 \\
\exp(\hat{\boldsymbol{\xi}}) &= \begin{bmatrix} \mathbf{R} & \mathbf{V}\boldsymbol{\nu} \\ \mathbf{0} & 0 \end{bmatrix}
\end{aligned} \tag{2.30}$$

The logarithm function that maps from $\mathbf{T} \in \mathbf{SE}(3)$ to $\boldsymbol{\xi} = (\boldsymbol{\omega} | \boldsymbol{\nu}) \in \mathfrak{se}(3)$, can be defined by calculating $\boldsymbol{\omega}$ as shown in eq. (2.18). Then we can calculate $\boldsymbol{\nu}$ as $\boldsymbol{\nu} = \mathbf{V}^{-1} \cdot \mathbf{t}$, where $\mathbf{V}^{-1}$ is defined as shown in eq. (2.31).

$$\mathbf{V}^{-1} = \mathbf{I} - \frac{1}{2}\hat{\boldsymbol{\omega}} + \frac{1}{\theta^2}\left(1 - \frac{\mathbf{A}}{2\mathbf{B}}\right)\hat{\boldsymbol{\omega}}^2 \tag{2.31}$$

We can define a function f, shown in eq. (2.32), which takes a point $\mathbf{x}$ and transforms it according to a transformation $\mathbf{T} \in \mathbf{SE}(3)$.

$$\mathbf{y} = f(\mathbf{T}, \mathbf{x}) = (\mathbf{R}|\mathbf{t}) \cdot (\mathbf{x}|1)^T = \mathbf{R} \cdot \mathbf{x} + \mathbf{t}, \quad \mathbf{T} = \exp(\hat{\boldsymbol{\xi}}) \tag{2.32}$$

The gradient of this function with respect to change in $\mathbf{x}$ is shown in eq. (2.33). As a constant translation does not affect the gradient; this is the same as eq. (2.20).

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{R} \tag{2.33}$$

We can calculate the gradient with respect to $\boldsymbol{\xi}$. Using the same procedure as in eq. (2.21), we arrive at the expression given in eq. (2.34).

$$\begin{aligned} \frac{\partial \mathbf{y}}{\partial \boldsymbol{\xi}} &= (\mathbf{G}_1\mathbf{y}| \cdots |\mathbf{G}_6\mathbf{y}) \\ &= (\mathbf{I}| -\hat{\mathbf{y}}) \end{aligned} \tag{2.34}$$

How changes in the Lie algebra $\boldsymbol{\xi}$ affects the Lie group $\mathbf{T}$ is illustrated in figs. 2.4a to 2.4f. The figures show the effect of small perturbations on each of the elements of $\boldsymbol{\xi}$ as its resulting $\mathbf{SE}(3)$ transforms the points of a rectangular prism.

2.2 The Mathematics of Cameras

In Visual Odometry, we are trying to estimate the ego-motion of cameras based solely on the images we record with the cameras. We, therefore, require a proper mathematical model of how images are formed in a camera and how they relate the motion of the camera and the geometry of the real world. In this section, we will describe this relationship and introduce the mathematical tools we will use to model it.

In section 2.2.1, we will describe the mathematical model of an image and the technical details of digital images. section 2.2.2 describes the model of image projection, which is the relationship of three-dimensional objects and their projection onto a two-dimensional image. section 2.2.4 is about the distortion effects introduced by real-world camera lenses. section 2.2.5 describes the mathematical model needed to adequately describe color altering effects such as vignetting, exposure-time, and non-linear response functions of the camera. In section 2.2.6, we describe the use of stereo-cameras with a known transformation between them to estimate depth.

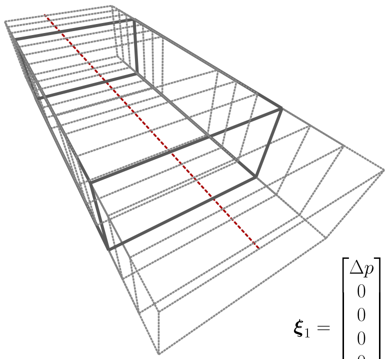
2.2.1 Images

While we in a real-life implementation must perform quantization of images to store and work with them digitally, it is often useful to model them mathematically as continuous objects. The continuous representation allows us to more easily reason about images, without complicating the argument with implementation details. In this section, we will first introduce the continuous mathematical model of an image and then discuss the implementation details of digital quantization as well as the effects of rolling and global shutters.

Mathematical Model of an Image

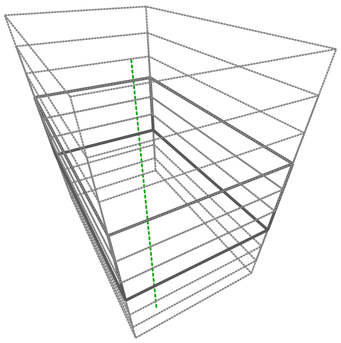
An image I can best be modeled as a multidimensional function. A monochrome image maps locations in the image to a scalar value and is therefore defined as in eq. (2.35). Here, Ω is the two-dimensional image region and is a subset of $\mathbb{R}^2$, which maps to an intensity value. Then the expression $i = I(p)$ denotes $i \in \mathbb{R}$ to be equal to the intensity of the point $p \in \mathbb{R}^2$

$$I : \Omega \to \mathbb{R}, \quad \Omega \subset \mathbb{R}^2 \tag{2.35}$$



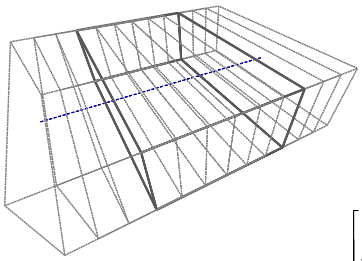
$$\boldsymbol{\xi}_1 = \begin{bmatrix} \Delta p \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

(a) Perturbation of $\boldsymbol{\xi}_1$



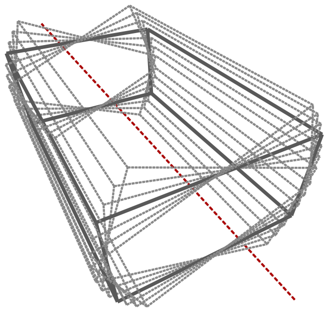
$$\boldsymbol{\xi}_2 = \begin{bmatrix} 0 \\ \Delta p \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

(b) Perturbation of $\boldsymbol{\xi}_2$



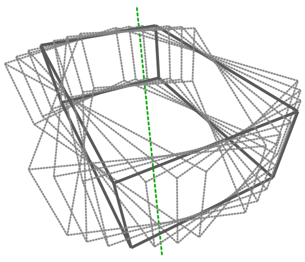
$$\boldsymbol{\xi}_3 = \begin{bmatrix} 0 \\ 0 \\ \Delta p \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

(c) Perturbation of $\boldsymbol{\xi}_3$



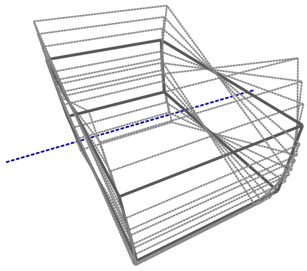
$$\boldsymbol{\xi}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \Delta p \\ 0 \\ 0 \end{bmatrix}$$

(d) Perturbation of $\boldsymbol{\xi}_4$



$$\boldsymbol{\xi}_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \Delta p \\ 0 \end{bmatrix}$$

(e) Perturbation of $\boldsymbol{\xi}_5$



$$\boldsymbol{\xi}_6 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \Delta p \end{bmatrix}$$

(f) Perturbation of $\boldsymbol{\xi}_6$

Figure 2.4: Perturbations of $\boldsymbol{\xi}$

The model for monochrome images in eq. (2.35) can be extended to RGB color images as shown in eq. (2.36). Here each point maps to a three-value vector representing the red, green and blue intensity values.

$$I : \Omega \to \mathbb{R}^3, \quad \Omega \subset \mathbb{R}^2 \tag{2.36}$$

Both eq. (2.35) and eq. (2.36) can be extended with a temporal parameter to represent video. The representation in eq. (2.37) shows the extension of eq. (2.35) to represent video. Now i in $i = I(p,t)$ is assigned the value of point $p \in \mathbb{R}^2$ at time $t \in \mathbb{R}^+$.

$$I : \Omega \times \mathbb{R}^+ \to \mathbb{R}, \quad \Omega \subset \mathbb{R}^2 \tag{2.37}$$

Digital Quantization

A real-world camera is not able to capture the real world with infinite precision, neither spatially or temporally and must, therefore, perform quantization, which is the process of converting real-value functions to discretized values.

A camera's image sensor, as seen in fig. 2.5 is a large number of light-sensitive patches arranged in a grid.

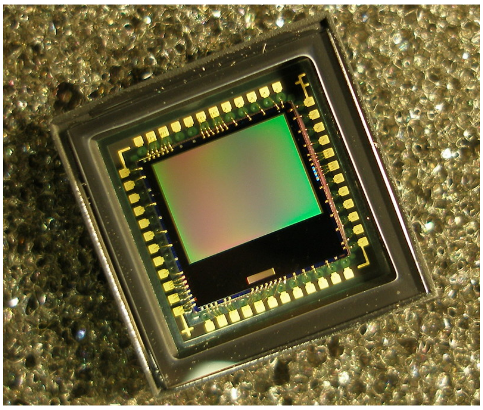


Figure 2.5: CMOS image sensor [4]

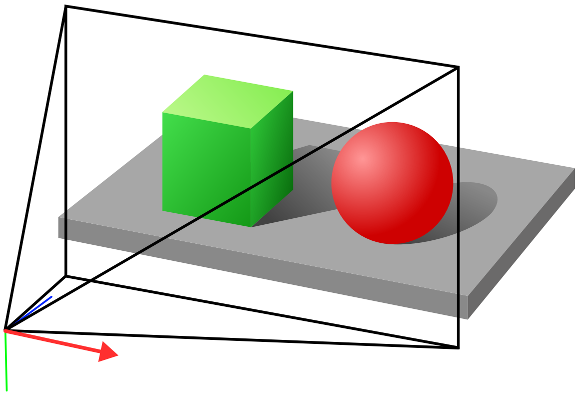
Each patch senses the number of photos within a range of wavelengths hitting it within a certain time period, effectively integrating over the light seen by the camera both spatially and temporally. This can be modeled by the triple integral given in eq. (2.38), where $I_t[p]$ is the discretized image at time t evaluated at pixel p and e is the exposure time of the image.

$$I_t[p] = \int_{t-e}^{t} \int_{p_x-0.5}^{p_x+0.5} \int_{p_y-0.5}^{p_y+0.5} I(q,u)\, dq_y\, dq_x\, du \tag{2.38}$$

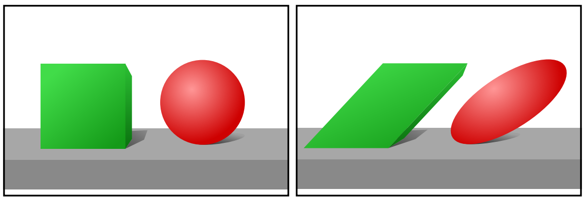
Global & Rolling Shutter

In a global shutter camera, every single pixel in the image sensor is active at the same time. In a rolling shutter camera, the pixels are activated in sequence over a small period of time.

Usually, the image is scanned one line at the time, starting from the top, which can cause distortion artifacts in the image if the camera is moving during this period, as illustrated in figs. 2.6a to 2.6c.



(a) Rightward camera motion



(b) Global shutter image **(c)** Rolling shutter image

Figure 2.6: Effects of global and rolling shutter with camera motion

This distortion can cause significant problems for VO, and SLAM algorithms and rolling-shutter cameras are therefore best avoided for these applications. If the use of rolling shutter cameras is unavoidable, then these distortions must be explicitly accounted for in the algorithm, such as in [28].

2.2.2 Projection

The projection model of a camera describes how any three-dimensional point is projected into the two-dimensional image. Pinhole projection is the most common form of camera projection model and works well for cameras with a field of view that is below 180.

The pinhole camera projection is modeled using an intrinsic camera matrix $\mathbf{K}$. This matrix is shown in eq. (2.39), which is a three-by-three matrix parameterized by four values.

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.39}$$

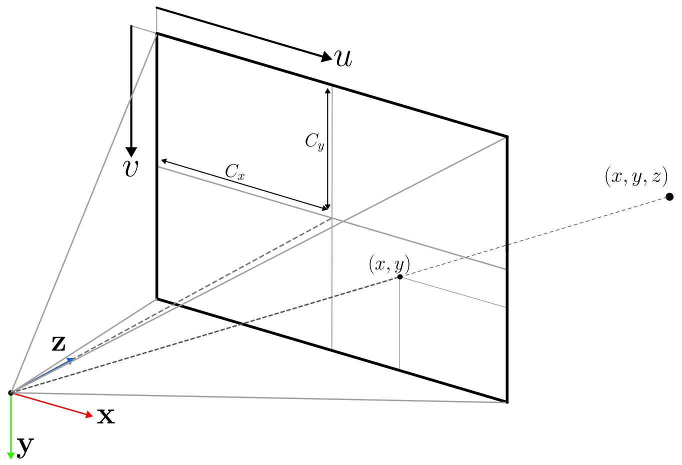


Figure 2.7: Pinhole projection

The f_x and f_y give, respectively, the horizontal and vertical field of view of the projection. The c_x and c_y values provide the coordinates for the principal point in the projection, which is the point at which the projection vector is orthogonal with the image plane. These values are obtained through a calibration process with a calibration object of known dimensions.

$$Z \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \underbrace{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}_{\mathbf{x}} \tag{2.40}$$

The camera matrix is used, as shown in eq. (2.40). A three-dimensional point $\mathbf{x}$ is multiplied by the camera matrix to create a two-dimensional homogeneous point. This point can be converted to an inhomogeneous point as described in section 2.1.3 to retrieve the final image coordinates.

We can extend the equation with an additional matrix $\mathbf{\Pi}_0$. This matrix allows us to use the same equation with a homogeneous point $\tilde{\mathbf{x}}$. The matrix $\mathbf{\Pi}_0$ is a modified identity matrix which removes the last element from the matrix. Therefore the homogeneous point $\tilde{\mathbf{x}}$ must be scaled to have the fourth element be 1 to function correctly.

$$Z \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{\Pi}_0} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\tilde{\mathbf{x}}} \tag{2.41}$$

So far, the point x was required to be expressed in the camera's coordinate system. We can extend eq. (2.41) further by adding a $\mathbf{SE}(3)$ transformation matrix $\mathbf{T}$, giving us eq. (2.42). Using the appropriate transformation $\mathbf{T}$, we can now transform a point into the camera's coordinate-system and project it into the image in a single operation by multiplying $\mathbf{K\Pi}_0\mathbf{T}$ into a single matrix.

$$Z\begin{bmatrix}x\\y\\1\end{bmatrix} = \underbrace{\begin{bmatrix}f_x & 0 & c_x\\0 & f_y & c_y\\0 & 0 & 1\end{bmatrix}}_{\mathbf{K}}\underbrace{\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\end{bmatrix}}_{\mathbf{\Pi}_0}\underbrace{\begin{bmatrix}\mathbf{R} & \mathbf{t}\\\mathbf{0} & 1\end{bmatrix}}_{\mathbf{T}}\underbrace{\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}}_{\tilde{\mathbf{x}}} \tag{2.42}$$

When projecting a large number of points, we can stack the homogeneous column vectors in a matrix and perform the projection of multiple points as a single matrix-matrix multiplication as shown in eq. (2.43).

$$\left(\begin{bmatrix}f_x & 0 & c_x\\0 & f_y & c_y\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\end{bmatrix}\begin{bmatrix}\mathbf{R} & \mathbf{t}\\\mathbf{0} & 1\end{bmatrix}\right)\begin{bmatrix}X_1 & X_2 & \cdots & X_n\\Y_1 & Y_2 & \cdots & Y_n\\Z_1 & Z_2 & \cdots & Z_n\\1 & 1 & \cdots & 1\end{bmatrix} \tag{2.43}$$

2.2.3 Epipolar Geometry

When the same point is seen through two different cameras, there exists a geometric relation between the projected two-dimensional position in the two images and the transformation between the two cameras. This relation is known as epipolar geometry and is illustrated in fig. 2.8.

A three-dimensional point $\mathbf{p}$ is seen by two cameras, giving the two, two-dimensional projected image points $\mathbf{x}_1$ and $\mathbf{x}_2$. Here we denote $\mathbf{x}_1$ and $\mathbf{x}_2$ to be the two-dimensional point extended to a three-dimensional vector by setting the third vector value to 1. This formulation lets us think of all the two-dimensional image point as points on a plane one unit in front of the camera origin and lets us simplify our calculations.

If we only know the two-dimensional projected point position $\mathbf{x}_1$, we can deduce that the point $\mathbf{p}$ must lie somewhere on the line going from the camera origin $\mathbf{o}_1$, through the point on the image-plane $\mathbf{x}_1$. This calculation may be written as multiplying the image-plane position $\mathbf{x}_1$ by a scalar, as in eq. (2.44). We can do the same with $\mathbf{x}_2$ after transforming it to its coordinate system, as in eq. (2.45).

$$\lambda_1\mathbf{x}_1 = \mathbf{p} \tag{2.44}$$

$$\lambda_2\mathbf{x}_2 = \mathbf{Rp} + \mathbf{t} \tag{2.45}$$

By inserting eq. (2.44) into eq. (2.45), we get eq. (2.46).

$$\lambda_2\mathbf{x}_2 = \mathbf{R}(\lambda_1\mathbf{x}_1) + \mathbf{t} \tag{2.46}$$

We can then left-multiply eq. (2.46) with the skew-symmetric matrix $\hat{\mathbf{t}}$. This eliminates $\mathbf{t}$, as $\hat{\mathbf{t}} \cdot \mathbf{t} = \mathbf{t} \times \mathbf{t} = \mathbf{0}$ and gives us eq. (2.47).

$$\lambda_2\hat{\mathbf{t}}\mathbf{x}_2 = \lambda_1\hat{\mathbf{t}}\mathbf{R}\mathbf{x}_1 \tag{2.47}$$

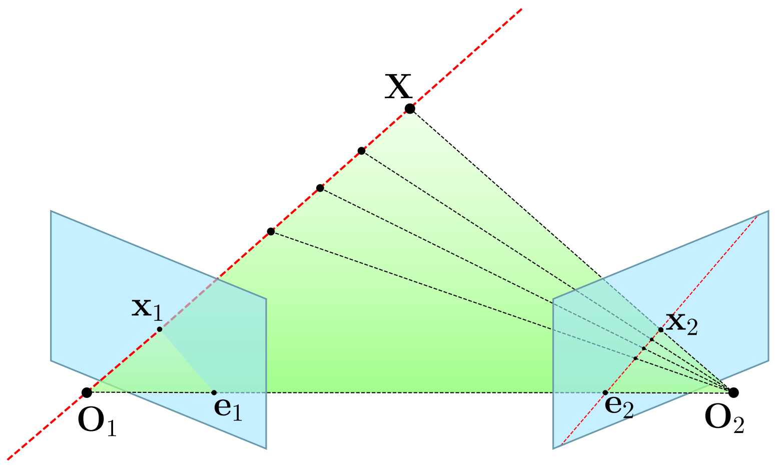


Figure 2.8: Epipolar geometry

Finally we left-multiply eq. (2.47) with $\mathbf{x}_2^T$, which gives us eq. (2.48). We have that $\mathbf{x}_2^T\hat{\mathbf{t}} \cdot \mathbf{x}_2 = 0$, since $\hat{\mathbf{t}} \cdot \mathbf{x}_2$ will always be perpendicular to $\mathbf{x}_2$, causing the dot product of the two to always be zero.

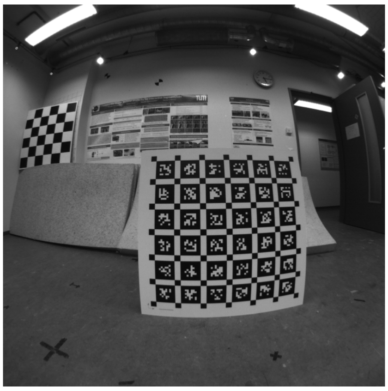
$$\mathbf{x}_2^T\hat{\mathbf{t}}\mathbf{R}\mathbf{x}_1 = 0 \tag{2.48}$$

The constraint given in eq. (2.48) is known as the epipolar constraint. The equation can be geometrically interpreted as stating that the three vectors $\overrightarrow{\mathbf{o}_1\mathbf{p}}$, $\overrightarrow{\mathbf{o}_1\mathbf{o}_2}$ and $\overrightarrow{\mathbf{o}_2\mathbf{p}}$ all lie on a plane. We have that $\overrightarrow{\mathbf{o}_1\mathbf{p}} \propto \mathbf{x}_1$, $\overrightarrow{\mathbf{o}_1\mathbf{o}_2} \propto \mathbf{t}$ and $\overrightarrow{\mathbf{o}_2\mathbf{p}} \propto \mathbf{R}\mathbf{x}_2$. Additionally, given three vectors in a plane, taking the cross product of two of them results in a vector that is perpendicular to both and therefore the dot product of this vector with the remaining vector must be zero. Therefore we have that $\mathbf{x}_2^T(\mathbf{t} \times \mathbf{R}\mathbf{x}_1) = \mathbf{x}_2^T\hat{\mathbf{t}}\mathbf{R}\mathbf{x}_1 = 0$.

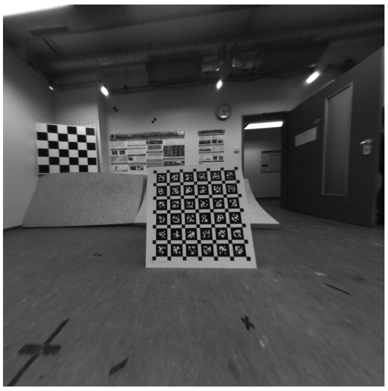
2.2.4 Lense Distortion

Cameras use lenses to capture and focus light better. A wide angle lens enables the camera to sense a large portion of the environment in a single image but also introduces lens distortion artifacts. The most common form of lens distortion can be seen in fig. 2.9a, with images taken from [29], and is known as barrel distortion. We can see that straight lines in the environment are no longer straight in the image. The image is shown in fig. 2.9b is the same image which has been undistorted using the methods described in this section.

To undistort the image, we must model the distortion mathematically, and then apply the inverse transformation to the image to create the undistorted image. To achieve the proper result, this inverse distortion transformation must be applied before the projection described in section 2.2.2. The parameters of the projection are usually obtained in the same calibration operation used to find the projection parameters. We will now introduce the two most common distortion models used for this purpose.



(a) Distorted image



(b) Undistorted image

Figure 2.9: Lense distortion effect on images from [29]

Radial-tangental Distortion Model

A good description of the radial-tangental model is given in [33]. The model is a combination of radial distortion, which is distortion towards or away from the center depending on the distance from the center point. The function from the distorted points x and y to the undistorted points x' and y' is given in eq. (2.49). Here x_c and y_c is the centered point such that $(x_c, y_c) = (0, 0)$ is the principal point of the image, and r_c is the distance from this point, that is $r_c = \sqrt{x_c^2 + y_c^2}$. The parameters k_1, k_2, p_1, and p_2 must be obtained through calibration.

$$\begin{aligned} x' &= x + \underbrace{x_c(k_1 r_c + k_2 r_c^4)}_{\text{Radial}} + \underbrace{2p_1 xy + p_2(r_c^2 + 2x^2)}_{\text{Tangential}} \\ y' &= y + \overbrace{y_c(k_1 r_c + k_2 r_c^4)} + \overbrace{2p_1 xy + p_2(r_c^2 + 2y^2)} \end{aligned} \tag{2.49}$$

Equidistant Distortion Model

The equidistant model is described in [13]. This is a more recently introduced model that can produce a better result for lenses with a large fish-eye distortion [26]. For a distorted point $\mathbf{x} = (x, y)$, eq. (2.50) transforms the point to the undistorted point $\mathbf{x}'$. k_1, k_2, k_3, and k_4 are the parameters of the distortion model that must be calibrated beforehand.

$$\mathbf{x}' = \frac{(\theta + k_1\theta^3 + k_2\theta^5 + k_3\theta^7 + k_4\theta^9)}{\sqrt{x^2 + y^2}}\mathbf{x}, \quad \theta = \arctan(\sqrt{x^2 + y^2}) \tag{2.50}$$

2.2.5 Photometric Model

In section 2.2.2 and section 2.2.4 we have given an account of what is collectively referred to as the geometric model. This model describes the relation of a point's location in three-dimensional space, with its projected position within a two-dimensional image. A photometric model is a mathematical model describing how the intensity of a point may be altered by camera effects such as vignetting, non-linear response function, and exposure time. The geometric model is sufficient for indirect methods that use robust feature detectors and earlier direct methods such as [6] were able to function without a photometric model, but the use of it in methods like [5] significantly improved performance.

The photometric model attempts to model the three most significant effects that affect the perceived intensity of a point:

- **Exposure Time:** The amount of time the camera is collecting light for a single image. The length of time affects the overall brightness of the entire image.
- **Vignetting:** The lens is not perfectly transparent, but absorb a small amount of light. Therefore pixels towards the edge of the image will appear slightly darker than pixels in the center.
- **Non-linear response function:** The relation between the intensity of a pixel in an image, and the amount of light hitting that pixel in the image sensor will be non-linear for most cameras. As the other photometric effects might sift the amount of light, the non-linearity of the image sensor will alter it further and must, therefore, be taken into account.

We model these three effects as show in eq. (2.51). Here $G : \mathbb{R}^+ \to \mathbb{R}^+$ is a monotonically increasing function representing the non-linear response function, mapping irradiance to the pixel value. An example of such a non-linear response function is shown in fig. 2.11. The function $V : \Omega \to [0, 1]$ is the vignetting effect, which darkens pixels depending on their position in the image. An example of this is shown in fig. 2.10. The exposure time $t \in \mathbb{R}^+$ affects all pixels in the image equally. $B : \omega \to \mathbb{R}^+$ is the irradiance, which we can think of as the true intensity of a pixel, and is the unknown value we wish to estimate.

$$I(\mathbf{x}) = G(t \cdot V(\mathbf{x})B(\mathbf{x})) \tag{2.51}$$

The vignetting V and non-linear response function G must be obtained through photometric calibration, as described in [8]. The exposure time t must be read from the camera together with each image. We can then use the inverse of eq. (2.51) together with these known values to obtain an image of the real irradiance value without the intensity altering camera artifacts described above.

2.2.6 Stereo Cameras

A stereo camera is a camera system containing multiple image sensors. When the transformation between the image sensors, and the necessary calibrations are performed, it is possible to use the stereo image pair obtained from the stereo camera to estimate depth in the image. While it is possible to use a single monocular camera for VO and SLAM, the performance is often significantly improved when the system is extended to stereo vision, as in [21], [7] and [36].

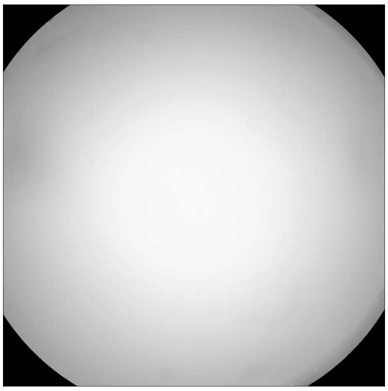


Figure 2.10: Vignetting from [29]

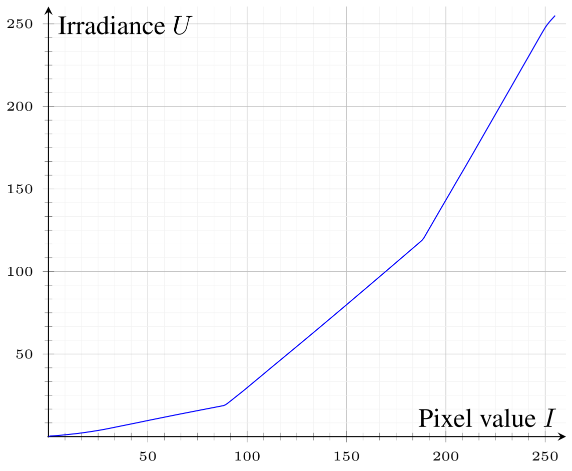


Figure 2.11: Non-linear response function generated from [8]

Section 2.2.6 describes the necessary image transformation step required to make points in one image lie on the same horizontal line as in the other. In fig. 2.13b, we will explain how we can locate the same position in both images, and use the distance between the image coordinates to estimate the depth of the point.

Stereo Rectification

The goal of stereo rectification is two warp the stereo images, such that they are equivalent with images captured with two cameras with co-linear z-axis, as illustrated in fig. 2.12, which means that the two axes through their respective image centers are parallel.

This goal can be achieved by first rotating the two cameras such that the vector of their viewing direction is perpendicular to the line joining the camera origins. Next, the cameras are both rotated around the viewing direction such that their respective y-axis also is perpendicular to the line joining them.

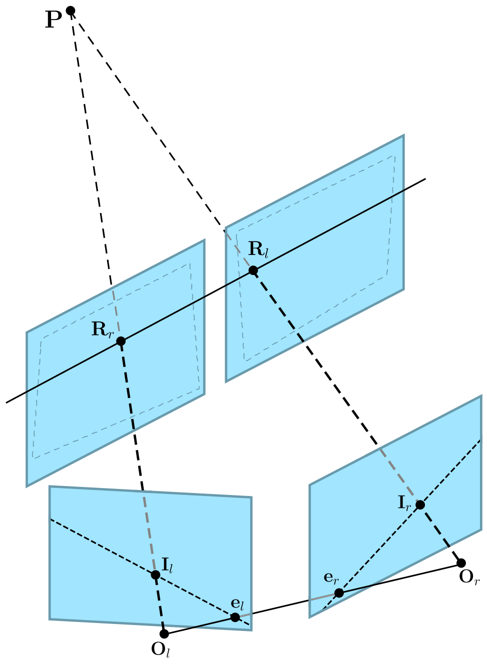


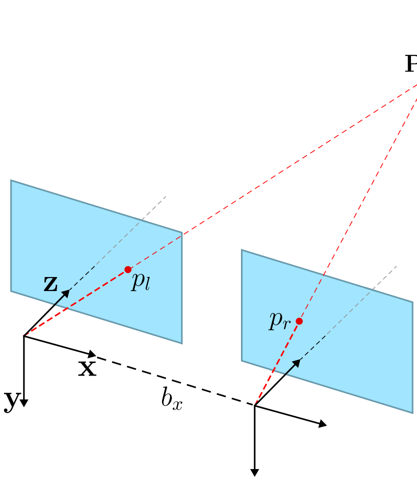
Figure 2.12: Stereo rectification

The original image is then re-projected into these new camera orientations, which ensures that corresponding epipolar lines are horizontal, and points at infinity have the same image coordinate in both images.

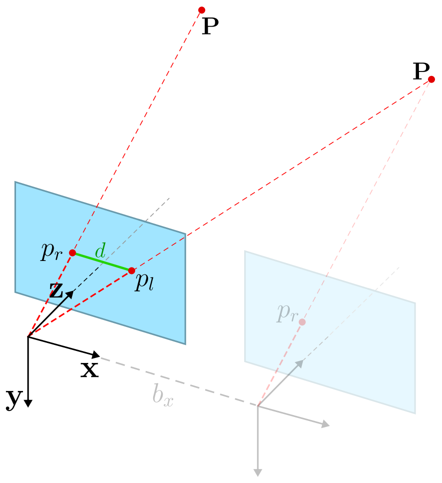
Disparity

The disparity is a measure of distance, measured in pixels, between the two image coordinates. We calculate the disparity of a point by searching for the same point along the horizontal line of the point in the other image. To search, we take a small patch around the potential point pair in both images end to evaluate how well they match. A common matching function is NCC(Normalized Cross Correlation), which is shown in eq. (2.52). We can calculate the NCC value of all potential matches with our given point if none are above a certain threshold we set the disparity to be unknown. Otherwise, we calculate the disparity between the points as shown in fig. 2.13b, which gives us a disparity image like fig. 2.14b.

$$\mathrm{NCC}(\mathbf{x}_1, \mathbf{x}_2) = \frac{\sum_{\mathbf{v}\in\mathcal{N}} I_1(\mathbf{x}_1 + \mathbf{v}) \cdot I_2(\mathbf{x}_2 + \mathbf{v})}{\sqrt{\sum_{\mathbf{v}\in\mathcal{N}} [I_1(\mathbf{x}_1 + \mathbf{v})^2] \cdot \sum_{\mathbf{v}\in\mathcal{N}} [I_2(\mathbf{x}_2 + \mathbf{v})^2]}} \tag{2.52}$$



(a) Stereo projection



(b) Stereo disparity

Figure 2.13: Stereo vision

Depth from Disparity

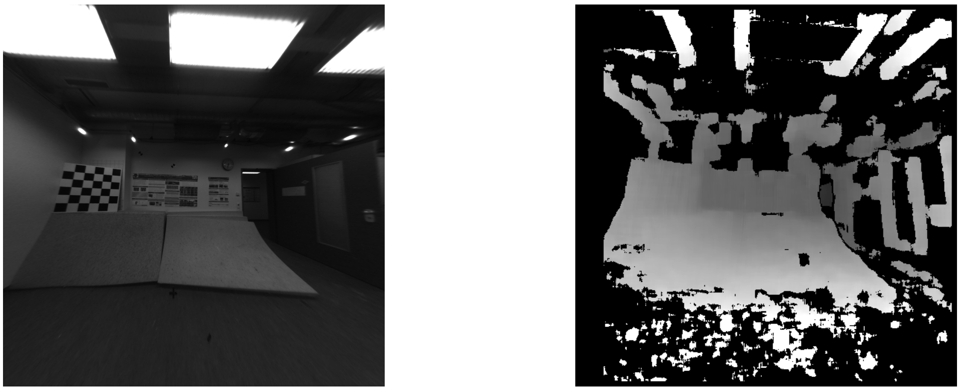
By knowing the disparity of a point, the intrinsic parameters of the camera, and the distance between the cameras, we can calculate the depth of the point. The formula for this calculation is given in eq. (2.53), where f_x is the horizontal focal length of the camera, b_x is the distance between the camera origins and d is the disparity of the point.

$$z = f_x \frac{b_x}{d} \tag{2.53}$$

The accuracy of this depth measurement will depend on the accuracy of the point matching algorithm and will be more accurate for points closer to the camera as they give a greater disparity.

Structured-light Stereo

Another way to produce stereo images is by using a structured-light stereo method. This method used a camera and a projector and a camera instead of multiple cameras. The projector will project a known light pattern onto the scene, which the cameras detects, as illustrated in fig. 2.15. Despite the different setup the calculation needed to find depths is almost the same as with a regular stereo setup. Here we use the known light pattern to estimate the disparity, and then estimate depth in the same way as before. Often an infrared projector is used with an infrared camera, in addition to a common visible-light camera as this allows the camera to produce both depth images in addition to color or gray-scale image where the light pattern is not visible. An example of images created with a structured light sensor is shown in fig. 2.16a and fig. 2.16b.



(a) Left stereo image **(b)** Stereo disparity image

Figure 2.14: Disparity from stereo calculated with images from [29]

2.3 Non-Linear Optimization

In visual odometry, we are trying to find the parameters that best describe some motion of the camera. These kinds of problems are, in most cases, solved as an optimization problem. An optimization problem are usually formalized as in eq. (2.54), where we are trying to find the input vector $\mathbf{x}^*$ that minimizes the value of the cost function $E(\mathbf{x})$. Here E is a function that assigns a number to how well our input $\mathbf{x}$ works as a solution to our problem, where a lower value is better. The key to solving any problem using optimization is therefore to design a suitable cost function, which will have a minima as close to our desirable solution as possible given our data, while still being tractable to optimize numerically within our given limitations.

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}}\, E(\mathbf{x}), \quad E : \mathbb{R}^n \to \mathbb{R} \tag{2.54}$$

In this section, we will describe the methods that can be used to solve such a problem, primarily focusing on those that are characterized by non-linear functions. Most of the interesting problems modeling real-world phenomenons will turn out to be non-linear functions. It is, therefore, vital that we can solve them robustly and efficiently.

In section 2.3.1 we first start by introducing a linear least squares problem, and the method for solving them. Section 2.3.2 presents non-linear least squares, which is the types of an optimization problem that will be most relevant to us. Section 2.3.3 covers gradient descent, which is the simplest and most naive method of solving non-linear optimization problems. In section 2.3.4 we improve on gradient descent by taking into account the Hessian of the function. Section 2.3.5 expands on this further by introducing a less costly method of approximating the Hessian. In section 2.3.6 we describe a method of combining both gradient descent and Newton's method to improve stability on specific problems. Section 2.3.7 shown how we can create robust optimizations that can handle outliers in the data using a method called iteratively reweighted least squares. Section 2.3.8 shows how we can apply optimization methods to solve problems over Lie group

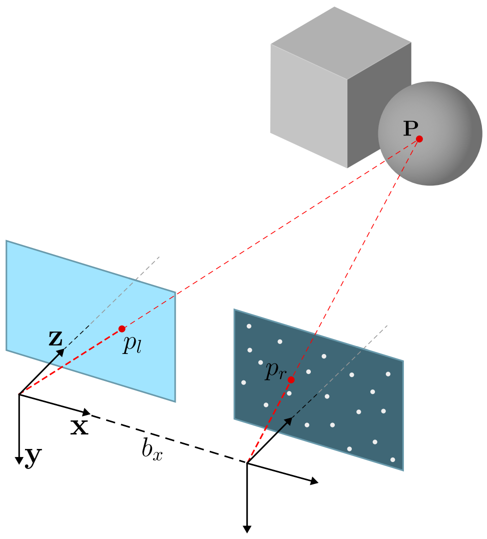


Figure 2.15: Structured-light stereo

manifolds.

A more detailed introduction to the theory given in sections 2.3.1 to 2.3.7 can be seen in [24], while the theory in section 2.3.8 is covered extensively in [1].

2.3.1 Linear Least Squares

In a linear least squares problem we are trying to find a vector $\mathbf{x}$ that best solves the equation given in eq. (2.55). Here, a_i is a function of $\mathbf{x}$ with some gaussian noise η_i with zero mean and a diagonal covariance matrix on the form $\mathbf{\Sigma} = \sigma^2\mathbf{I}$.

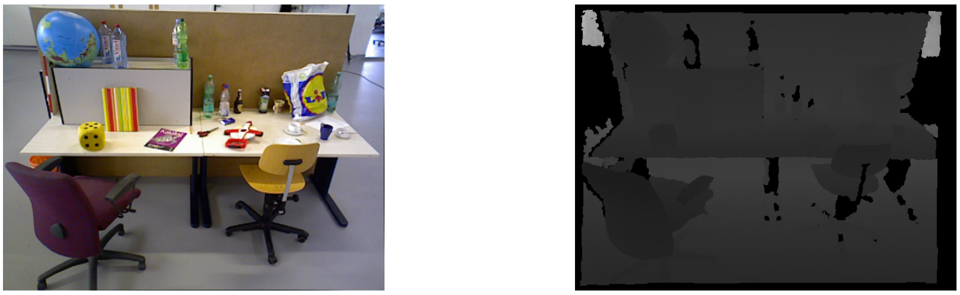
$$a_i = \mathbf{b}_i^T\mathbf{x} + \eta_i, \quad a_i \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^m, \mathbf{b}_i \in \mathbb{R}^m, i \in \{1, \dots, n\}, \eta \sim \mathcal{N}(0, \mathbf{\Sigma}) \tag{2.55}$$

This equation leads to the minimization problem given in eq. (2.56).

$$\underset{\mathbf{x}}{\operatorname{argmin}}(\mathbf{Ax} - \mathbf{a})^T\mathbf{\Sigma}^{-1}(\mathbf{Ax} - \mathbf{a}) \tag{2.56}$$

As this is a linear problem, the optimal input $\mathbf{x}^*$ has a closed form solution as shown in eq. (2.57). Which allows us to solve the problem directly.

$$\begin{aligned}\mathbf{x}^* &= \underset{\mathbf{x}}{\operatorname{argmin}}(\mathbf{Ax} - \mathbf{b})^T\mathbf{\Sigma}^{-1}(\mathbf{Ax} - \mathbf{b}) \\ &= (\mathbf{A}^T\mathbf{\Sigma}^{-1}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{\Sigma}^{-1}\mathbf{b}\end{aligned} \tag{2.57}$$



(a) Structured light stereo color **(b)** Structured light stereo depth

Figure 2.16: Structured light stereo images from [32]

2.3.2 Non-linear Least Squares

Most optimization problem we are interested in solving within the context of visual odometry is non-linear, and often take the form of non-linear least squares problems. The general form of non-linear least squares problems are given in eq. (2.58), for some non-linear function f. As the problem is non-linear, we will not be able to solve it directly as in section 2.3.1. Instead, we must use iterative methods to find a solution. For non-convex problems, we are also not guaranteed that our solution is an optimal one, as the iterative method might get stuck in local a minima.

$$\underset{\mathbf{x}}{\operatorname{argmin}} \sum_i r_i(\mathbf{x})^2, \quad r_i(\mathbf{x}) = a_i - f(b_i, \mathbf{x}) \tag{2.58}$$

2.3.3 Gradient Descent

Gradient descent is the most basic iterative method for solving non-linear optimization problems like the one introduced in section 2.3.2. Given an initial value $\mathbf{x}_0$, we iteratively improve the input according to eq. (2.59). We evaluate the gradient at our current value of $\mathbf{x}$ and alter $\mathbf{x}$ in the direction most negative gradient, with a step length λ.

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \lambda \left. \frac{dE}{d\mathbf{x}} \right|_{\mathbf{x}=\mathbf{x}_{k-1}}, \quad k = 1, 2, 3, \dots \quad \lambda > 0 \tag{2.59}$$

We continuously iterate over $\mathbf{x}$ according to eq. (2.59) until we reach a local minimum where we are unable to improve the input further. Gradient descent is robust and will find the global minimum for convex functions. It can, however, be quite slow to converge, and we will, therefore, introduce other methods which are faster.

2.3.4 Newton's Method

Newton's method is a second order, meaning it employs both the gradient and the Hessian to estimate the iteration step. A geometrical intuition of Newton's method is that we are for each step fitting a quadratic function to the gradient and Hessian of our current value of $\mathbf{x}$, and solving this new quadratic problem to get a new value of $\mathbf{x}$ for the next iteration.

Given a cost function E, we can approximate it as a quadratic function by it's second order Taylor expansion at point $\mathbf{x_k}$, as shown in eq. (2.60). Here $\mathbf{g}$ and $\mathbf{H}$ is the gradient and Hessian respectively, at point $\mathbf{x}_k$

$$\begin{gathered} E(\mathbf{x}) \approx E(\mathbf{x}_k) + \mathbf{g}^T(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T \mathbf{H}(\mathbf{x} - \mathbf{x}_k) \\ \mathbf{g} = \left.\frac{dE}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_k}, \quad \mathbf{H} = \left.\frac{d^2E}{d\mathbf{x}^2}\right|_{\mathbf{x}=\mathbf{x}_k} \end{gathered} \tag{2.60}$$

For this second order approximation, the optimality condition is given by eq. (2.61).

$$\mathbf{g} + \mathbf{H}(\mathbf{x} - \mathbf{x}_k) = 0 \tag{2.61}$$

Solving for $\mathbf{x}$ in eq. (2.61) lead us to the iterative formula given in eq. (2.62).

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \mathbf{H}^{-1}\mathbf{g} \tag{2.62}$$

In practice, we may wish to limit the step-size with a variable λ, as shown in eq. (2.63).

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \lambda \mathbf{H}^{-1}\mathbf{g}, \quad 0 < \lambda \leq 1 \tag{2.63}$$

2.3.5 The Gauss-Newton Algorithm

The Hessian may be costly to calculate for more complex functions. The Gauss-Newton algorithm introduces a method for approximating the Hessian and thereby simplifying the calculations necessary for non-linear optimization.

The elements of the Hessian are defined as shown in eq. (2.64).

$$\mathbf{H}_{jk} = 2\sum_i \left(\frac{\partial r_i}{\partial \mathbf{x}_j}\frac{\partial r_i}{\partial \mathbf{x}_j} + r_i \frac{\partial^2 r_i}{\partial \mathbf{x}_j \partial \mathbf{x}_k} \right) \tag{2.64}$$

Given that the function is not to non-linear, we can assume that eq. (2.65) holds, and drop the second term of the sum without to much error.

$$\left| \frac{\partial r_i}{\partial \mathbf{x}_j}\frac{\partial r_i}{\partial \mathbf{x}_k} \right| \gg \left| r_i \frac{\partial^2 r_i}{\partial \mathbf{x}_j \partial \mathbf{x}_i} \right| \tag{2.65}$$

We can, therefore, approximate the elements of the Hessian using the Jacobian, as shown in eq. (2.66).

$$\mathbf{H}_{jk} \approx 2\sum_i \mathbf{J}_{ij}\mathbf{J}_{ik}, \quad \mathbf{J}_{ij} = \frac{\partial r_i}{\partial \mathbf{x}_j} \tag{2.66}$$

Using this together with $\mathbf{g} = 2\mathbf{J}^T\mathbf{r}$ gives us the iterative formula given in eq. (2.67).

$$\mathbf{x}_k = \mathbf{x}_{k-1} - (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{r} \tag{2.67}$$

2.3.6 The Levenberg-Marquardt Algorithm

For specific optimization problems, a large valued Hessian may have unwanted effects on the algorithm when using Newton's method. To mediate this, we can use the Levenberg-Marquardt algorithm, which is given in eq. (2.68).

$$\mathbf{x}_k = \mathbf{x}_{k-1} - (\mathbf{H} + \lambda \mathbf{I})^{-1}\,\mathbf{g} \tag{2.68}$$

This algorithm now is a mix of gradient descent and Newton's algorithm, with λ as a parameter for controlling the interpolation between the two. The algorithm will be equal to Newton's method when $\lambda = 0$, and equal to gradient descent when $\lambda \to \infty$.

We can combine Levenberg-Marquardt with Gauss-Newton, and get the formula as given in eq. (2.69).

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left(\mathbf{J}^T\mathbf{J} + \lambda \mathbf{I}\right)^{-1} \mathbf{J}^T\mathbf{r} \tag{2.69}$$

We may also alter the formula slightly by using the diagonal of the Hessian, or approximated Hessian, as shown in eq. (2.70), which will help the algorithm to avoid slow convergence when the gradient is small.

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left(\mathbf{J}^T\mathbf{J} + \lambda \mathrm{diag}(\mathbf{J}^T\mathbf{J})\right)^{-1} \mathbf{J}^T\mathbf{r} \tag{2.70}$$

2.3.7 Iteratively Reweighted Least Squares

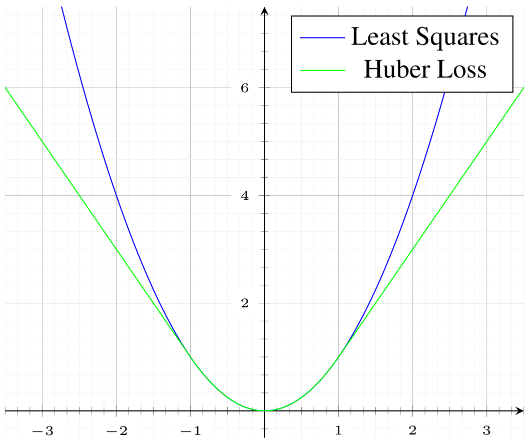


Figure 2.17: Weighted error functions.

The formulation described in section 2.3.2 works well in many problems, but performance may suffer significantly in the presence of outliers. An outlier is defined as a data-point that does not fit the model we are optimizing and is usually included in the data by some error in a previous step. Some amount of outliers is to be expected in most real-world optimization applications and must, therefore, be accounted for.

Iteratively reweighted least squares are one such method of improving robustness of non-linear least squares methods to be able to handle outliers in the data. The formulation is shown in eq. (2.71) and is similar to eq. (2.58) but with a additional weighting term function w_i, that is recalculated for each iteration.

$$\underset{\mathbf{x}}{\operatorname{argmin}} \sum_i w_i(\mathbf{x}) r_i(\mathbf{x})^2, \quad r_i(\mathbf{x}) = a_i - f(b_i, \mathbf{x}) \tag{2.71}$$

The iterative solution to this is as given in eq. (2.72), where $\mathbf{W}$ is a diagonal matrix of all the weights.

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left(\mathbf{J}^T\mathbf{W}\mathbf{J}\right)^{-1}\mathbf{J}^T\mathbf{W}\mathbf{r} \tag{2.72}$$

There exist a large number of different weighting functions, each with their strengths and weaknesses. One of the most commonly used weighting functions is Huber Loss, which was first published in [12]. By applying the Huber weights shown in eq. (2.73) we turn the error function into a combination of a linear and a squared function, as is illustrated in fig. 2.17. This figure shows the error function for both regular least squares and Huber loss with $k = 1$. We see here that both functions are equal in the range -1 to 1, but the Huber loss is linear beyond this. This change will significantly reduce the error of outliers as the error now grows linearly instead of quadratic outside this range. The tuning constant k is usually adapted to the distribution of residual in each iteration. A common method is to set $k = 1.345\sigma$, where σ is the standard deviation of the residuals, which gives a 95-percent efficiency for Gaussian distributed weights and still offers protection against outliers.

$$w_H(e) = \begin{cases} 1 & \text{for } |e| \le k \\ k/|e| & \text{for } |e| > k \end{cases} \tag{2.73}$$

2.3.8 Optimization over Lie Groups

The theory we have established so far in this chapter is enough to implement optimization algorithms for most non-linear problems if they are trivially mapped to Euclidean space. Many problems in VO and SLAM are however not trivially mapped to Euclidean space. One such example is rotations, which we in section 2.1.5 showed was best represented as a Lie Group manifold. The way we optimize over a Lie Group is to use its respective Lie Algebra as the parameterization and using the linearized gradient of the exponential function, which maps an element of the Lie Algebra to its corresponding element in the Lie Group.

We will illustrate this method with a simple example. Given a set $\mathbf{P}$ of points $p_i \in \mathbb{R}^3$ we can can create a set $\mathbf{Q}$, which is a set of points that have been transformed by an unknown transformation function $g(\exp(\boldsymbol{\xi}), \mathbf{p})$ witch transforms a point $\mathbf{p}$, using $\exp(\boldsymbol{\xi})$, and add a independent and identically distributed, zero-mean gaussian noise $\boldsymbol{\eta} \in \mathbb{R}^3$, as given in eq. (2.74).

$$\begin{gathered} \mathbf{q}_i = g(\exp(\boldsymbol{\xi}), \mathbf{p}_i) + \boldsymbol{\eta} \\ \mathbf{p}_i \in \mathbf{P}, \mathbf{q}_i \in \mathbf{Q}, \boldsymbol{\eta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}), \boldsymbol{\Sigma} = \operatorname{diag}\left((\sigma^2, \sigma^2, \sigma^2)^T\right) \end{gathered} \tag{2.74}$$

We now wish to recover the unknown transformation $\mathbf{T}$ using non-linear optimization. To achieve this, we must first design a cost function that will recover the transformation. For this we use the function given in eq. (2.75), which takes a $\mathfrak{se}(3)$ element $\boldsymbol{\xi}$ as a parameter and calculates the sum of the squared Euclidean distance between the corresponding point after the points in $\mathbf{P}$ have been transformed with the given transformation.

$$E(\boldsymbol{\xi}) = \sum_i \|g(\exp(\boldsymbol{\xi}), \mathbf{p}_i) - \mathbf{q}_i\|_2^2 \tag{2.75}$$

We rewrite the function slightly to better work with the iterative method we will use to optimize the cost function. Instead of changing the existing transformation, we concatenate it with a small transformation away from identity $\exp(\Delta\boldsymbol{\xi})$. This change will make the linearization we soon will perform work much better, as it is only accurate in a small neighborhood around the identity transformation. The altered cost function is shown in eq. (2.76). We can no linearize the function g with respect to $\Delta\boldsymbol{\xi}$ as shown in eq. (2.77) as to turn each iteration of the optimization to a linear optimization problem. This linearization gives us the Jacobian $\mathbf{J}$ of g which is defined as in eq. (2.81). By rearranging the terms and defining two new variables $\mathbf{A}$ and $\mathbf{b}$ as shown in eq. (2.78) we get the much more familiar form of eq. (2.79). We can get rid of the summation by stacking the terms as shown in eq. (2.82) to get eq. (2.80).

$$\begin{aligned}
\Delta\boldsymbol{\xi}^* &= \underset{\Delta\boldsymbol{\xi}}{\operatorname{argmin}} \sum_i \|g(\exp(\boldsymbol{\xi}) \cdot \exp(\Delta\boldsymbol{\xi}), \mathbf{p}_i) - \mathbf{q}_i\|_2^2 && (2.76)\\
&\approx \underset{\Delta\boldsymbol{\xi}}{\operatorname{argmin}} \sum_i \|g(\exp(\boldsymbol{\xi}), \mathbf{p}_i) + \mathbf{J}_i \cdot \Delta\boldsymbol{\xi} - \mathbf{q}_i\|_2^2 && (2.77)\\
&= \underset{\Delta\boldsymbol{\xi}}{\operatorname{argmin}} \sum_i \left\| \underbrace{\mathbf{J}_i}_{\mathbf{A}} \cdot \Delta\boldsymbol{\xi} - \underbrace{(\mathbf{q}_i - g(\exp(\boldsymbol{\xi}), \mathbf{p}_i))}_{\mathbf{b}} \right\|_2^2 && (2.78)\\
&= \underset{\Delta\boldsymbol{\xi}}{\operatorname{argmin}} \sum_i \|\mathbf{A}_i \cdot \Delta\boldsymbol{\xi} - \mathbf{b}_i\|_2^2 && (2.79)\\
&= \underset{\Delta\boldsymbol{\xi}}{\operatorname{argmin}} \|\mathbf{A} \cdot \Delta\boldsymbol{\xi} - \mathbf{b}\|_2^2 && (2.80)
\end{aligned}$$

$$\mathbf{J}_i = (\mathbf{I} | -\hat{\mathbf{p}}_i) \tag{2.81}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_n \end{bmatrix} \tag{2.82}$$

Just as with eq. (2.56) we can rewrite eq. (2.80) into the matrix form given in eq. (2.83).

$$\Delta\boldsymbol{\xi}^* = \underset{\Delta\boldsymbol{\xi}}{\operatorname{argmin}} \left(\mathbf{A} \cdot \Delta\boldsymbol{\xi} - \mathbf{b}\right)^T \left(\mathbf{A} \cdot \Delta\boldsymbol{\xi} - \mathbf{b}\right) \tag{2.83}$$

This allows us to solve the minimization just as we did in section 2.3.1 which gives us eq. (2.84).

$$\Delta\boldsymbol{\xi}^* = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{2.84}$$

For each iteration, we use the value of $\Delta\boldsymbol{\xi}^*$ to update our current value of $\boldsymbol{\xi}$, with each iteration bringing the value closer to a local minimum. Normally this is done using simple addition. But as we are optimizing over a manifold, and not simply in Euclidean space, this will not work. Instead we must use eq. (2.85), where convert both values to their $\mathbf{SE}(3)$ Lie group representation, multiply them using matrix-matrix multiplication, and convert the resulting $\mathbf{SE}(3)$ element back to a $\mathfrak{se}(3)$ element using the exponential and logarithm functions described in section 2.1.5.

$$\boldsymbol{\xi} := \log(\exp(\boldsymbol{\xi}) \cdot \exp(\Delta\boldsymbol{\xi}^*)) \tag{2.85}$$

Putting this all together gives us the alg. 1, which solves the optimization problem iteratively. The algorithm converges to the optimal transformation $\boldsymbol{\xi}^*$, after a sufficient number of iterations.

Algorithm 1: Optimization of transformation between P and Q

```
Data: Set of points $P$ and $Q$, each with $m$ points
Result: Computes the optimal transformation $\boldsymbol{\xi}^*$
$\mathbf{P}^{(0)} \longleftarrow \mathbf{P}$
$\boldsymbol{\xi}^{(0)} \longleftarrow \mathbf{0}$
for $i \leftarrow 1$ to $n$ do /* Optimize for n iterations.   */
    $\mathbf{T}^{(i)} \longleftarrow \exp\left(\boldsymbol{\xi}^{(i-1)}\right)$
    for $j \leftarrow 1$ to $m$ do
        $\mathbf{p}_j^{(i)} \longleftarrow g(\mathbf{T}^{(i)}, \mathbf{p}_j^{(0)})$
        $\mathbf{b}_j^{(i)} \longleftarrow \mathbf{p}_j^{(i)} - \mathbf{q}_j$
        $\mathbf{A}_j \longleftarrow \left(\mathbf{I}| - \hat{\mathbf{p}}_j^{(i)}\right)$
    end
    $\Delta\boldsymbol{\xi} \longleftarrow (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$
    $\boldsymbol{\xi}^{(i)} \longleftarrow \log(\exp(\boldsymbol{\xi}^{(i-1)}) \cdot \exp(\Delta\boldsymbol{\xi}))$
end
$\boldsymbol{\xi}^* \longleftarrow \boldsymbol{\xi}^{(n)}$
```

Chapter 3

CUDA & GPGPU Programming

3.1 Introduction

In this chapter, we will introduce the programming of General-purpose Graphical Processing Units (GPGPUs) and illustrate how it differs from the programming of Central Processing Units (CPUs). GPGPUs allows us to implement an algorithm in a massively parallel manner, where hundreds of threads will be run simultaneously.

We will in this thesis focus on the the CUDA library[25] created by the Nvidia Corporation. This is a proprietary, closed-source library that allows GPGPU programming of Nvidia GPUs. Other GPGPU programming libraries exist, such as OpenACC which provides a single library targeting Nvidia, AMD and Intel GPUs, and OpenCL which in addition to GPGPUs is able to target Digital Signal Processors (DSPs) and Field-programmable Gate Arrays (FPGAs).

While these alternative libraries can target a larger and more diverse group of hardware, CUDA can still be the better choice in many cases. The dominance of Nvidia hardware in the GPGPU space has caused the CUDA-ecosystem to be far more active, with a larger selection of supported libraries and development tools. CUDA has also been shown to beat both OpenCL and OpenAAC in performance tests when run on Nvidia hardware [14][11].

3.1.1 From GPU to GPGPU

Graphical Processing Units (GPUs) were introduced as a dedicated processor for rendering 3D graphics. As the GPU was specially created for a singular purpose, they could be optimized for the characteristics of 3D graphics computation. This optimization mainly involved the parallel nature of 3D rendering, where one pixel can be computed independently from any other pixel, and implementing many commonly used operations in hardware rather than software.

It later became apparent that massively parallel nature of GPUs could be exploited to compute certain problems far faster than what was possible with CPUs. This discovery led to the introduction of GPGPU programming languages and libraries, which allowed such algorithms to be implemented far easier than using the graphics-specific languages and libraries for purposes they were not designed. Nvidia introduced their own GPGPU library, CUDA, in 2007 [23].

3.1.2 GPGPU vs CPU Programming

To illustrate some of the main differences between CPU and GPGPU programming, we will use the implementation of a Single-precision A, X plus Y (SAXPY) in both paradigms. SAXPY is a common linear algebra operation where a vector X is multiplied by a scalar A and added to vector Y. This operation is an easily parallelizable problem as each value of the solution vector is independent of all the others.

Listing 3.1 shows the serial CPU implementation, written in C. We see that this implementation is based on serially looping through the vectors. While it would be possible to optimize this code using Single Instruction, Multiple Data (SIMD) instructions or multithreading, we will still be forced to use a loop because of the limited parallelism of a CPU.

```c
void saxpy(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
    {
        y[i] = alpha*x[i] + y[i];
    }
}

saxpy(n, 2.0, x, y); // Invoke SAXPY C Function
```

Listing 3.1: C SAXPY

Listing 3.2 shows the same function implemented in CUDA. We can see that this code does not use any loops. Instead, the code implements the functionality of a single CUDA thread, and the CUDA library is responsible for executing the code across multiple threads. As the code is identical for all threads, we must instead let each thread calculate its index in the array. This is in listing 3.2 done in line 4, using values stored in blockIdx, blockDim and threadIdx which are internal CUDA objects that will be explained in more detail in section 3.2.3.

```
1  __global__
2  void saxpy(int n, float alpha, float *x, float *y)
3  {
4      int i = blockIdx.x*blockDim.x + threadIdx.x;
5      if(i >= n) { return; }
6
7      y[i] = alpha*x[i] + y[i];
8  }
9
10 int nblocks = (n + 255) / 256;
11 saxpy<<<nblocks, 256>>>(n, 2.0, x, y); // Invoke SAXPY CUDA kernel.
```

Listing 3.2: CUDA SAXPY

3.2 CUDA Execution Model

In this section, we will describe how a GPU executes CUDA code and how it differs from how a CPU executes code.

3.2.1 Kernels

To execute code on a GPGPU, we submit jobs to be executed from the CPU. In CUDA, these jobs are referred to as kernels and are executed as function calls with a special syntax. We will from here on be referring to the CPU as the host, and the GPU as the device as this in common terms in the context of GPGPU programming.

```
kernel<<<grid_size, block_size, shared_memory_size, stream>>>(a, b, c);
```

Listing 3.3: CUDA kernel launch example

Listing 3.3 shows an example of the syntax for launching a CUDA kernel from the host code. We can see that the syntax is similar to a regular C function call, but with the addition of the triple angle brackets (<<<...>>>) to set the parameters that are needed to launch the CUDA kernel.

The first to parameters is the grid size and the block size for the execution of the kernel, and are described in section 3.2.3. These parameters are mandatory and must be included in all kernel launches. The next are shared memory size, which is described in section 3.3.2, and stream, which is described in section 3.2.5. Both of these values are optional. The function parameters work similar to regular C function parameter and are accessible from each thread in the kernel. A CUDA kernel is however not able to return a value in the same manner as a standard C function and must therefore always be defined with void as the return type.

3.2.2 Thread Divergence

An essential difference between the execution of host code and device code is the concept of thread divergence. Listing 3.4 shows a CUDA kernel where different threads will take different paths in the branch.

```
__global__
void foo()
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    A;
    if (i % 2 == 0) {
        B;
    } else {
        C;
    }
    D;
}
```

Listing 3.4: Branching Kernel

This branching introduces a phenomenon known as thread divergence, illustrated in fig. 3.1. The issue is that CUDA is not able to execute different instructions of a single kernel at the same time. All threads execute A, but when the threads reach a branching part of the code, the threads not executing the branch are temporarily disabled and are not doing useful work.

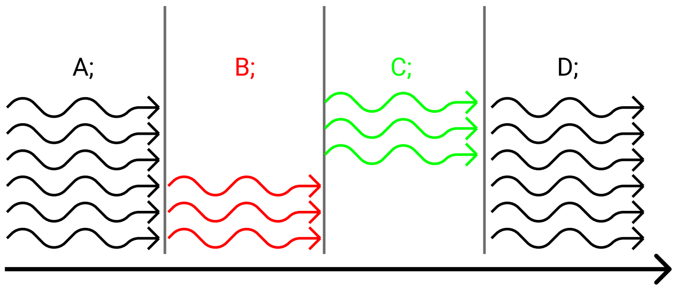


Figure 3.1: Thread Divergence when executing kernel in listing 3.4.

Branching can, therefore, be detrimental to performance as a single thread can hold up all other threads from doing useful work by taking a different branch. The problem gets exponentially worse with multiple levels of branching as each level can potentially double the time taken to execute the branching part of the kernel. Because of this, the CUDA kernel code should limit the amount of branching, where different threads may take different paths.

3.2.3 Execution Hierarchy

The execution of a CUDA kernel is organized into a hierarchy of 4 levels and is illustrated in fig. 3.2. These levels are:

- **Block:** Threads are grouped into thread blocks. The dimensions of a thread-block are adjustable by the user and can be set to a one, two, or three-dimensional configuration. The size of each dimension can be set by the user, with some limitations, and the total number of threads in a thread-block may not exceed 1024 threads on all current hardware[25].

- **Warp:** The threads in a thread-block is organized into groups of threads called a *Warp*. All currently existing Nvidia hardware uses a warp size of 32 threads and is not controllable by the user[25]. There exist a collection of methods for sharing data between threads in a warp without using shared memory.

- **Grid:** The collection of thread-blocks are again organized into a grid. The grid can like the thread-block be arranged in a user-specified configuration of up to three dimensions. Threads in different thread block are not able to communicate directly, and any data transfer must be done through the global memory.

- **Device:** A single CUDA program can cooperatively use multiple Nvidia GPGPUs and directly transfer data between them. Multi-GPU programming is however outside the scope of this thesis, and will therefore not be covered further.

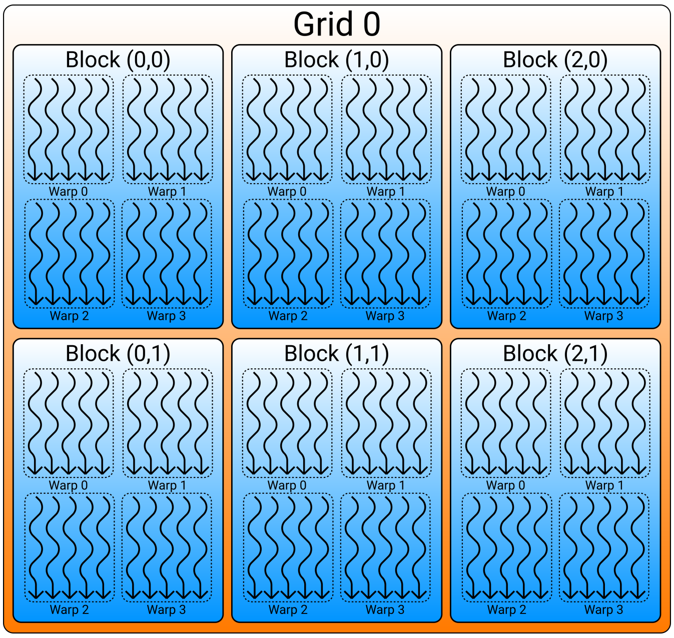


Figure 3.2: CUDA Execution Hierarchy

3.2.4 Kernel Synchronization

While a large number of threads in the execution of a CUDA kernel may be run simultaneously, not all of them will be, and groups of threads will be run in succession. The order of execution is set by the CUDA scheduler and is not controllable by the user. Synchronization is therefore required if any thread, at any point needs to utilize data from a different thread.

CUDA provides two main methods for performing synchronization within a kernel:

- **Block-level synchronization:** Performed by calling `__syncthreads ()`. This call will ensure that all threads in a block have reached this line in its execution before any thread in the block is allowed to continue past it.
- **Warp-level synchronization:** Performed by calling `__syncwarp()`. This call will ensure that all threads in a warp have reached this line before continuing past it.

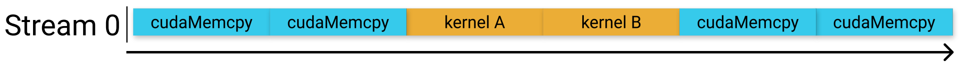
Until recently there was no way to synchronize across a grid in CUDA, but CUDA 9.0 added a feature known as *Cooperative Groups* which added the functionality of syncing across arbitrary groups of threads, and even across multiple GPUs[25]. This feature was, however, not used for this thesis, so we will not go into more details here.

In addition to the synchronization methods, CUDA provides a collection of atomic functions. These methods are guaranteed to perform their action without allowing any other thread to access the data in an intermediate state. One such atomic function is atomicInc which increments an integer stored in shared or global memory and returns its value before the incrimination. The atomic nature of the function assures the correctness of the affected value despite multiple threads incrementing the same variable without any synchronization.

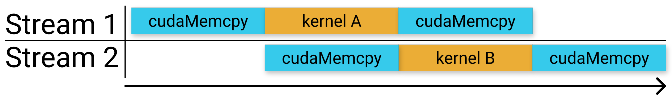
3.2.5 CUDA Streams

CUDA streams is a method of optimizing and simplifying the orchestration of launching multiple CUDA kernels. A CUDA program will often have some kernels with dependency on the result on other kernels, and must, therefore, be run in order, while others have no such dependencies. The use of CUDA streams allows us to launch multiple series of kernels, where the kernels of each series are computed in order, but the CUDA scheduler is allowed to switch between streams or run multiple operations at the same time if possible.

An example of this is given in fig. 3.3, where we illustrate the benefits using an example where we have two independent kernels that perform some work on the GPU. We need to copy some memory to the GPU before the kernel and copy some data back after it has run. In fig. 3.3a, we are only using a single stream, and all the operations will, therefore, be completed serially. In fig. 3.3b we are however assigning the work to two separate streams. As the execution of a kernel and the copying of data utilizes separate pieces of the GPU hardware, we can perform each in parallel and therefore increase throughput.



(a) Single CUDA stream



(b) Multiple CUDA streams

Figure 3.3: CUDA streams

Recent versions of CUDA have also introduced functionality for defining a computational graph of kernel launches [25], this is however outside the scope of this thesis and will not be covered here.

3.3 CUDA Memory

One of the most significant differences between CUDA programming and regular CPU programming is a large number of different memory types a CUDA programmer must be aware. While CPU code should take into account the behavior of the CPU cache to ac hive maximum performance, and therefore isn't entirely trivial, a CUDA program has access to a wide array of different memory types. Each memory type has its advantages and caveats that must be taken into account for maximum utilization and performance when writing GPU code.

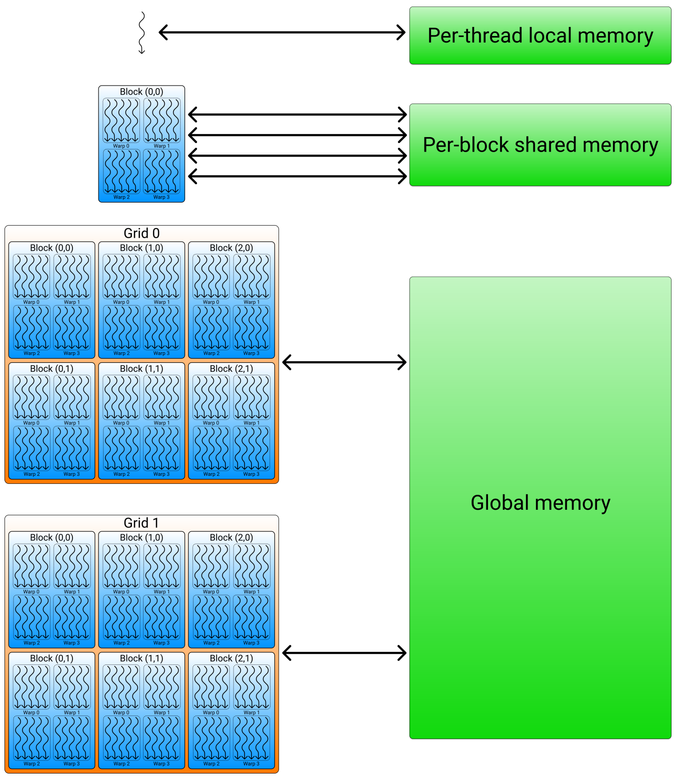


Figure 3.4: Memory Hierarchy

We will, in this section, give an overview of the different memory types available when writing CUDA programs and try to explain how they best can be used.

3.3.1 Local Memory

The local memory is the collective term for the set of GPU registers that are available to a CUDA program. While a CPU often only has around 10-20 general-purpose register per CPU core, modern GPUs has several hundred registers per CUDA thread [25]. The local memory is the fastest memory available in CUDA code, and it is what is used to store the input parameters, variables defined within a CUDA kernel or any intermediate computation result.

3.3.2 Shared Memory

Shared memory is the second level of the CUDA memory hierarchy. The shared memory is shared among all threads in a thread-block, and not accessible by any thread in any other-thread block. This memory is used as the primary method of communication between threads in a thread-block. Most systems have between 48KB and 96KB of shared memory per thread block.

Shared memory can be defined at run-time by using the shared memory size parameter when launching the kernel, as shown in listing 3.5. Here the shared memory is defined in the CUDA kernel using the extern keyword together with the __shared__ keyword.

```
__global__ void foo()
{
    extern __shared__ float s[];

    /* Kernel code using the shared memory */
}

// Launching kernel with run-time shared memory size
foo<<<grid_size, block_size, shared_memory_size>>>();
```

Listing 3.5: Run-time shared memory definition

Shared memory can also be defined at compile time, as shown in listing 3.6, by specifying the size of the shared memory array. When this is done, there is no need to determine the shared-memory size when launching the CUDA kernel.

```
__global__ void foo()
{
    __shared__ float s[128];

    /* Kernel code using the shared memory */
}
```

Listing 3.6: Compile-time shared memory definition

3.3.3 Global Memory

Global memory is the main memory of the GPU. The size depends on the specific device and is usually multiple gigabytes on a modern GPU. Global memory is allocated by calling cudamalloc, or one of its variants. These functions return a pointer to the allocated GPU memory which will continue to exist until freed by a call to cudaFree or the process terminates.

The global memory is slow, and without caching, it is however required to store data that will last across multiple kernels and can be read from and written to from different thread blocks.

3.3.4 Constant Memory

Constant memory is a read-only version of the global memory that can be advantageous if a kernel only needs to read from global memory, and not write. Constant memory is physically part of the global memory, but it is accessed as constant memory the GPU can use caching to speed significantly up parallel or repeated memory access.

Constant memory is defined using the __constant__ prefix, as shown in listing 3.7. It is not possible to write to constant memory from kernel code, it is, however, possible to write from host code using the cudaMemcpytoSymbol function.

```
__constant__ float c[128];
```

Listing 3.7: Constant memory definition

3.3.5 Texture Memory

Like constant memory, texture memory is a special way for kernel code to access global memory in a read-only way. Texture memory does, however, utilize specific hardware capabilities to significantly increase performance when the data is arranged in a one, two, or three-dimensional form.

The caching used for texture memory is a spatial locality type cache, which greatly increases the performance when multiple threads in a thread-block, access points in the texture memory that is spatially close to each other.

Texture memory is also able to perform extremely fast, hardware-implemented, linear interpolation, which makes it possible to sample the texture using a real-valued coordinate and get the interpolated value of the neighboring points in the texture.

3.4 CUDA Libraries

CUDA has many public libraries that implement commonly used functionality. Some of these libraries are created by Nvidia themselves, while others are a community open-source effort.

We will in this section give a short introduction to the two CUDA libraries that are relevant for this thesis. Section 3.4.1 provides an introduction to cuBLAS, which is an official library from Nvidia for performing many common linear algebra operations on a GPGPU. We also give an overview of Nvidia VisionWorks in section 3.4.2, which is an Nvidia library covering commonly used computer vision algorithms.

3.4.1 cuBLAS

The cuBLAS library is a GPGPU implementation commonly used linear algebra operations, such as matrix-matrix and matrix-vector multiplication. The library is based on the BLAS[3] (The Basic Linear Algebra Subprograms) library, which is a CPU linear algebra library for C and Fortran that has been in use for several decades. The cuBLAS library uses the same API (Application Programming Interface) as BLAS to simplify the conversion from BLAS to cuBLAS.

The cuBLAS library API is divided into three levels.

- **Level-1 Functions:** Scalar and vector based operations.
- **Level-2 Functions:** Matrix-vector operations.
- **Level-3 Functions:** Matrix-matrix operations.

All the function names in the cuBLAS library follow a common naming pattern to simplify the process of selecting the correction function for the task. This naming pattern is cublas<t><f>, where <t> is the datatype and <f> is the operation performed on the data. The different data types are shown in table 3.1.

Datatype	Meaning
S	Real single-precision
D	Real double-precision
C	Complex single-precision
Z	Complex double-precision

Table 3.1: Table to test captions and labels

As an example, we will take the level-2 function for calculating a double-precision matrix-vector multiplication, which is cublasDgemv.

This function is defined to calculate eq. (3.1), where α and β are scalars and op either has no effect or is a transposition.

$$y = \alpha \cdot \text{op}(\mathbf{A})\mathbf{x} + \beta \cdot \mathbf{y} \tag{3.1}$$

The complete list of cuBLAS functionality is given in [25].

3.4.2 Nvidia VisionWorks

Nvidia VisionWorks is a CUDA library released by Nvidia for computing common computer vision operations on a GPU. The API is split into three versions with different advantages and disadvantages.

- **OpenVX Immediate Mode:** This API follows the OpenCV standard set by the Khronos Group. Each operation is called immediately and blocks execution until it finishes. The data is stored in opaque structures where the internal data is not immediately accessible by the user.
- **OpenVX Graph Mode:** Like the immediate mode, this API follows the OpenVX standard, but here the computation is defined as a computational graph that may be executed multiple times on different data. As the computation is defined beforehand, VisionWorks can perform several optimizations on the graph to improve performance.
- **CUDA API:** This is a low-level CUDA API for VisionWorks. Unlike the other APIs, the data is completely transparent and accessible, and memory management is left to the user.

3.5 OpenGL Interoperability & Visualization

The ability to visualize is an essential tool when working with large amounts of data or complex programs. The visualization of CUDA programs have both advantages and disadvantages of the visualization of CPU programs, as GPU memory is slightly more opaque and harder to inspect but we may also take advantage of some of the graphical abilities to visualize the data.

CUDA provides the functionality to inter-operate with both OpenGL and Vulcan, which are both graphics libraries supported by most modern GPUs. By using this functionality, we can map memory used in the CUDA program to be accessible by OpenGL or Vulcan code.

Chapter 4

Direct Visual Odometry Algorithm & Implementation

In this chapter, we will give a detailed explanation of how the Direct Visual Odometry (DVO) algorithm functions, using the theory we established in chapter 2. We will also show how DVO was implemented on an Nvidia GPU using the CUDA library, and give extra attention to how this implementation differs from the CPU implementation detailed in [15].

A video of the implementation running can be seen at `https://youtu.be/oioTzMCSmH4`.

The full code of the implementation can be viewed and downloaded at `https://github.com/matiasvc/gpu-dvo`.

4.1 Direct Image Alignment

In this section, we will detail the theory behind direct image alignment as given in [30] and [15]. This method is the core algorithm of DVO and allows us to align two views of the same scene using direct non-linear optimization. Section 4.1.1 introduces the photo-consistency assumption, which is the base assumption on which direct image alignment is based. Section 4.1.2 details the warping function that is used to mathematically model the projection of a pixel in one image into another. In section 4.1.3, we show the probabilistic reasoning behind the cost function used for the image alignment. Section 4.1.4 shows how we can optimize the cost function to find the most probable transformation of the camera. Section 4.1.5 shows how the warp function and its optimization is implemented on a GPGPU.

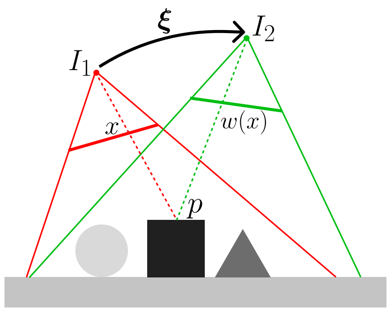


Figure 4.1: Photo-consistency assumption

4.1.1 Photo-consistency Assumption

The algorithm is based on the photo-consistency assumption, which is illustrated in fig. 4.1. This assumption states that color or gray-scale intensity of a point viewed through a camera at one position will be equal to the same point view from a different position. This assumption is not entirely descriptive of the real world, as it assumes all surfaces to be perfectly Lambertian as any specular surface will break this assumption. Despite this, the assumption is accurate enough for DVO to work in most real-world environments. The robustification methods detailed in section 4.2 will also help to minimize the effect from surfaces breaking the assumption and other outliers.

4.1.2 Warping function

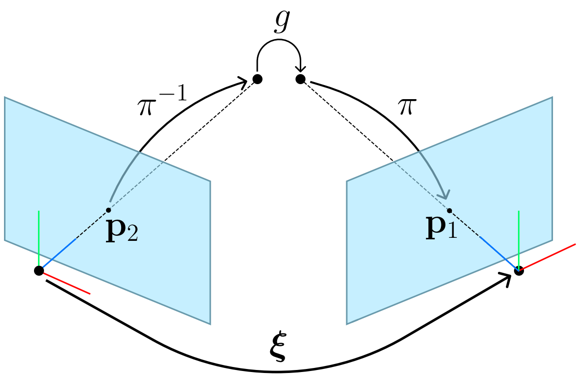


Figure 4.2: Warp function

The warp function describes how a pixel in the image of one camera is projected into three-dimensional space with a known depth, and then re-projected back into the image of a camera with a known transformation to the first image, as illustrated in fig. 4.2.

The projection of a pixel with know depth into three-dimensions is a static transformation and is not dependent on the transformation we will optimize over. We can, therefore, do this once as a pre-processing step for each image. We use the inverse of the projection described in section 2.2.2 and shown in eq. (4.1), where we project each pixel to a point on an image plane, one unit in front of the camera origin. By multiplying this vector with the known depth of the pixel $D(\mathbf{x})$, we get the three-dimensional point in the coordinate system of the camera.

$$\begin{aligned}\mathbf{p} &= \pi^{-1}(\mathbf{x}, D(\mathbf{x})) \\ &= D(\mathbf{x}) \left(\frac{u + c_x}{f_x}, \frac{v + c_y}{f_y}, 1 \right)^T\end{aligned} \tag{4.1}$$

To transform a three-dimensional point $\mathbf{p}$ we use the function T given in eq. (4.2) which transform the point according to a given transformation $g \in \mathbf{SE}(3)$.

$$T(g, \mathbf{p}) = \mathbf{R}\mathbf{p} + \mathbf{t} \tag{4.2}$$

We will need to use a minimal parameterization for the transformation to be able to optimize over it, as described in eq. (2.74). We will, therefore, parameterize the $\mathbf{SE}(3)$ element using a $\mathfrak{se}(3)$ element as shown in eq. (4.3).

$$g(\boldsymbol{\xi}) = \exp(\hat{\boldsymbol{\xi}}) \tag{4.3}$$

After the three-dimensional points have been transformed into the coordinate system of the other camera, we need to project them back into the two-dimensional image. To do this, we use a simple pinhole projection, as outlined in section 2.2.2. This process is summarized in the function given in eq. (4.4).

$$\pi(\mathbf{p}) = \left(\frac{f_x x}{z} - c_x, \frac{f_y y}{z} - c_y \right) \tag{4.4}$$

All of these operations are composed into the full warp function τ as shown in eq. (4.5).

$$\begin{aligned}\tau(\boldsymbol{\xi}, \mathbf{x}) &= \pi(T(g(\boldsymbol{\xi}, \mathbf{p}) \\ &= \pi(T(g(\boldsymbol{\xi}, \pi^{-1}(\mathbf{x}, D(\mathbf{x}))\end{aligned} \tag{4.5}$$

4.1.3 Probabilistic Model

Based on the photo-consistency assumption from fig. 4.1 we can define a residual for a given pixel i as shown in eq. (4.6). Here the residual is the difference in brightness between the original color of the pixel in the first frame, and the color of the point hit when projecting the pixel into the second frame, given the transformation $\boldsymbol{\xi}$.

$$r_i(\boldsymbol{\xi}) = I_2(\tau(\boldsymbol{\xi}, \mathbf{x}_i)) - I_1(\mathbf{x}_1) \tag{4.6}$$

Because of errors in measurement and the the photo-consistency assumption the residuals in eq. (4.6) will be distributed according to a probabilistic sensor model $p(r_i|\boldsymbol{\xi})$. We assume that the distribution of all residuals are independent and equally distributed, and we may, therefore, model the distribution over all the residuals as the product of each of them as shown in eq. (4.7).

$$p(\mathbf{r}|\boldsymbol{\xi}) = \prod_i p(r_i|\boldsymbol{\xi}) \tag{4.7}$$

Using Bayes' rule we can rewrite it to the probability of $\boldsymbol{\xi}$ given $\mathbf{r}$ as in eq. (4.8).

$$p(\boldsymbol{\xi}|\mathbf{r}) = \frac{p(\mathbf{r}|\boldsymbol{\xi})p(\boldsymbol{\xi})}{p(\mathbf{r})} \tag{4.8}$$

We will model $p(\mathbf{r})$ as a uniform distribution, and we can therefore rewrite the equation as given in eq. (4.9).

$$p(\boldsymbol{\xi}|\mathbf{r}) \propto p(\mathbf{r}|\boldsymbol{\xi})p(\boldsymbol{\xi}) \tag{4.9}$$

Here $p(\boldsymbol{\xi})$ is interesting as it models the probability distribution of all possible transforms. By choosing a fitting distribution, we can include a priori knowledge of the dynamics of the system or use a probability based on the measurements from a Inertial measurement unit (IMU). We will show how we do the former in section 4.2.2.

Our goal is to find the most probable transformation $\boldsymbol{\xi}$ given the residuals $\mathbf{r}$, which gives us the formula in eq. (4.10). We use eq. (4.7) to rewrite the equation to eq. (4.11). We can then take the log of the formula to get eq. (4.12). This formulation is far easier to optimize as we have replaced the product with a sum and does not change the solution as the minimum is not altered by the log operator as it is a strict monotonically increasing function.

$$\begin{aligned}
\boldsymbol{\xi}_{\text{MAP}} &= \underset{\boldsymbol{\xi}}{\operatorname{argmax}}\, p(\boldsymbol{\xi}|\mathbf{r}) && (4.10)\\
&= \underset{\boldsymbol{\xi}}{\operatorname{argmax}} \prod_i p(r_i|\boldsymbol{\xi})p(\boldsymbol{\xi}) && (4.11)\\
&= \underset{\boldsymbol{\xi}}{\operatorname{argmin}} - \sum_i \log\left(p(r_i|\boldsymbol{\xi})\right) - \log\left(p(\boldsymbol{\xi})\right) && (4.12)
\end{aligned}$$

We will for now ignore the motion model term $log(p(\boldsymbol{\xi}))$ for the sale of simplicity, and re-introduce it later in section 4.2.2. To minimize eq. (4.12) will we take the gradient and set it equal to zero as shown in eq. (4.13).

$$\sum_i \frac{\partial \log p(r_i|\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} = \sum \frac{\partial \log p(r_i)}{\partial r_i}\frac{\partial r_i}{\partial \boldsymbol{\xi}} = 0 \tag{4.13}$$

By factoring out part of the equation into a separate function $w(r_i)$, we get the equation given in eq. (4.14).

$$\frac{\partial r_i}{\partial \boldsymbol{\xi}} w(r_i) r_i = 0, \quad w(r_i) = \frac{\partial \log p(r_i)}{\partial r_i}\frac{1}{r_i} \tag{4.14}$$

This minimization becomes equivalent to the least squares minimization over a Lie group as we introduce in section 2.3.8. This result gives us the final minimization problem, as shown in eq. (4.15).

$$\boldsymbol{\xi}_{\text{MAP}} = \underset{\boldsymbol{\xi}}{\text{argmin}} \sum_i w(r_i)(r_i(\boldsymbol{\xi}))^2 \tag{4.15}$$

4.1.4 Optimization

We now wish to optimize the non-linear optimization problem eq. (4.15). The solution to this uses the theory we established in section 2.3. We will for now let the weighting function $w(r_i) = 1$ for the sake of simplicity, and reintroduce it later on section 4.2.1. The equation we must solve is therefore as given in eq. (4.16). We linearize the equation and write it as a small transformation $\Delta\boldsymbol{\xi}$ away from the current transformation $\boldsymbol{\xi}$ as we did in section 2.3.8 which gives us eq. (4.17), which we can rewrite as eq. (4.18). This equation diverges slightly from formula used in [15], where the linearization is done at the identity transformation $\tilde{\boldsymbol{\xi}} = \mathbf{0}$ instead of at the current transformation $\tilde{\boldsymbol{\xi}} = \boldsymbol{\xi}$, we will come back to the reason for this change in section 4.1.5.

$$\boldsymbol{\xi}^* = \underset{\boldsymbol{\xi}}{\text{argmin}} \sum_i (r_i(\boldsymbol{\xi}))^2 \tag{4.16}$$

$$\approx \underset{\Delta\boldsymbol{\xi}}{\text{argmin}} \sum_i \left(r_i(\boldsymbol{\xi}) + \left.\frac{\partial r(\tau(\tilde{\boldsymbol{\xi}}, \mathbf{x}_i))}{\partial \tilde{\boldsymbol{\xi}}}\right|_{\tilde{\boldsymbol{\xi}}=\boldsymbol{\xi}} \Delta\boldsymbol{\xi} \right)^2 \tag{4.17}$$

$$= \underset{\Delta\boldsymbol{\xi}}{\text{argmin}} \sum_i (r_i(\boldsymbol{\xi}) + \mathbf{J}_i \Delta\boldsymbol{\xi})^2 \tag{4.18}$$

Using the same method as in section 2.3.8 we can rewrite the problem as the linear optimization problem eq. (4.19).

$$\mathbf{J}^T\mathbf{J}\Delta\boldsymbol{\xi} = -\mathbf{J}^T\mathbf{r} \tag{4.19}$$

Which we can see in eq. (4.20) is just a simple linear equation we easily can solve.

$$\underbrace{\mathbf{J}^T\mathbf{J}}_{\mathbf{A}} \underbrace{\Delta\boldsymbol{\xi}}_{\mathbf{x}} = \underbrace{-\mathbf{J}^T\mathbf{r}}_{\mathbf{b}} \tag{4.20}$$

The residual jacobian J_i is the Jacobian of the warp function of point i. The warp-function is the concatenation of the point transformation, point projection, and image intensity function. We can, therefore, use the chain-rule and calculate the warp-function Jacobian as the product of the jacobians of each of the three composed functions as given in eq. (4.21).

$$\mathbf{J}_i = \nabla I(\mathbf{r})\big|_{\mathbf{r}=\pi(T(\exp(\boldsymbol{\xi}),\mathbf{p}_i))} \cdot \left.\frac{\partial \pi(\mathbf{q})}{\partial \mathbf{q}}\right|_{\mathbf{q}=T(\exp(\boldsymbol{\xi}),\mathbf{p}_i)} \cdot \left.\frac{\partial T(\exp(\boldsymbol{\xi}), \hat{\mathbf{p}})}{\partial \boldsymbol{\xi}}\right|_{\hat{\mathbf{p}}=\mathbf{p}_i} \tag{4.21}$$

The Jacobian of the image function is simply the image gradient as shown in eq. (4.22) while the jacobians of the point projection and point transformation are given in eq. (4.23) and eq. (4.24) respectively.

$$\nabla I(\mathbf{p}) = \begin{bmatrix} I_x(\mathbf{p}) & I_y(\mathbf{p}) \end{bmatrix} \tag{4.22}$$

$$\frac{\partial \pi(\mathbf{p})}{\partial \mathbf{p}} = \begin{bmatrix} \frac{f_x}{p_z} & 0 & -\frac{f_x x}{p_z^2} \\ 0 & \frac{f_y}{p_z} & -\frac{f_y y}{p_z^2} \end{bmatrix} \tag{4.23}$$

$$\frac{\partial T(\exp(\boldsymbol{\xi}), \mathbf{p})}{\partial \boldsymbol{\xi}} = \begin{bmatrix} 1 & 0 & 0 & 0 & p_z & -p_y \\ 0 & 1 & 0 & -p_z & 0 & x \\ 0 & 0 & 1 & p_y & -p_x & 0 \end{bmatrix} \tag{4.24}$$

The product of these three jacobians gives us the full warp jacobian, as shown in eq. (4.25).

$$\mathbf{J}_i = \begin{pmatrix} \frac{I_x f_x}{p_z} \\ \frac{I_y f_y}{p_z} \\ -\frac{I_x f_x p_x}{p_z^2} - \frac{I_y f_y p_y}{p_z^2} \\ -I_y f_y - p_y \frac{I_x f_x p_x + I_y f_y p_y}{p_z^2} \\ I_x f_x - p_x \frac{I_x f_x p_x + I_y f_y p_y}{p_z^2} \\ \frac{I_y f_y p_x}{p_z} - \frac{I_x f_x p_y}{p_z} \end{pmatrix}^T \tag{4.25}$$

4.1.5 Implementation

We will here give an overview how the most basic version of the direct image alignment was implemented using C++ and CUDA, later in section 4.2.3 we will expand on this implementation with the implementation of the robustification features described in section 4.2.

For testing the algorithm, the TUM RGB-D dataset[32] was primarily used. The implementation loads corresponding pairs of RGB and depth images into CPU memory, which is then copied into GPU memory. All further image operation is then in the GPU, and no images are ever copied out to CPU memory again.

Image Pyramid Creation

While most of the robustification features will be skipped here to be covered in section 4.2 instead, there is one feature that is so vital to the direct image alignment that it would not function properly without it. This feature is the use of an image pyramid, as illustrated in fig. 4.3. An image pyramid is created by taking an image and then blurring it before lowering the resolution, usually by half, to create a new layer. This procedure is done repeatedly until the desired number of layers are created. Our implementation uses a Gaussian image pyramid, which blurs the image with the 5×5 gaussian kernel given in eq. (4.26) and halves the resolution for each level in the pyramid. All image pyramids in the implementation using 5 levels, which means that as the base resolution is 640×480 pixels, the final level has a resolution of 40×30 pixels.

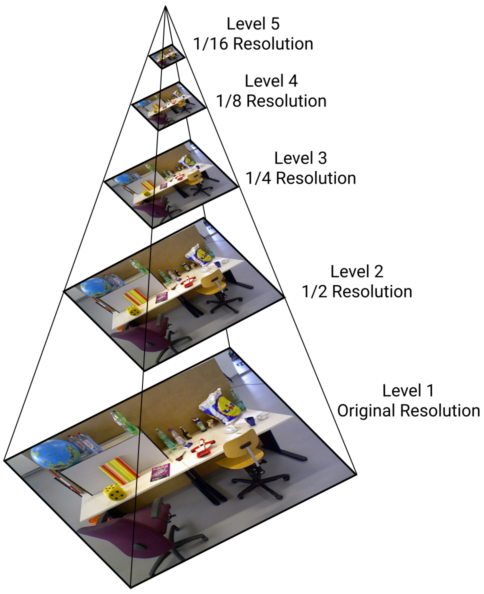


Figure 4.3: Image pyramid

$$\mathbf{G} = \frac{1}{256} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{4.26}$$

The creation of the depth pyramid is done slightly different as the depth images have missing data as can be seen in fig. 4.4f. We use the same blurring kernel eq. (4.26), but when a pixel is missing depth data it is ignored, and the division used to set the average kernel value to one is adjusted to only include the kernel values that are used.

In addition to this, two more pyramids are created for the horizontal and vertical gradient of the brightness image. This operation is done by applying a 3×3 Scharr operator, as shown in eq. (4.27), on each level of the brightness pyramid.

$$\mathbf{G}_x = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}, \quad \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix} \tag{4.27}$$

An example of the resulting brightness and depth pyramids are shown in figs. 4.4a to 4.4j.

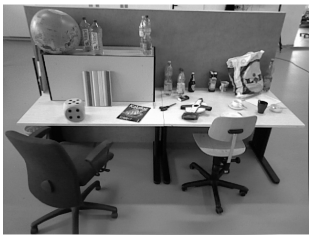
Warp Jacobian & Residual Calculation

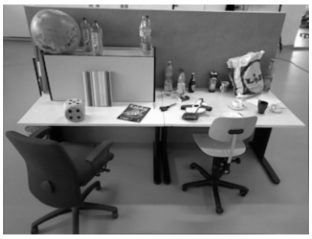
As soon as the second image frame is received, the algorithm will start performing direct image alignment between pairs of frames, as shown in fig. 4.5. Each alignment uses the depth and brightness from the previous frame to create a three-dimensional point-cloud of points with an associated brightness. Each image alignment produces a transformation between two image pairs. We assume the first frame to be at an identity transformation in relation to the origin and compose new transformations with the existing estimate to continuously estimate the camera transform as the algorithm runs. The algorithm starts aligning the level in the image pyramid with the lowest resolution. Once this level has converged, the algorithm moves the next level, until the final layer has converged. As a new frame enters, it replaces the memory used by the oldest frame. This procedure ensures that only frames are kept in memory at any time, keeping the memory usage low. This process is illustrated in fig. 4.5. A visualization of points warped into another frame is shown in fig. 4.6a with the resulting error shown in fig. 4.6b.

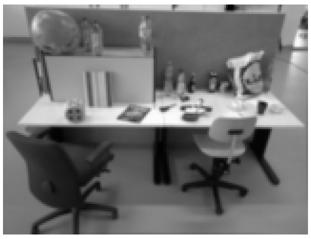
As mention in section 4.1.4, we chose to implement the calculation of the Jacobian slightly differently than in [15]. The difference is that the CPU implementation described in the original paper calculates the warp jacobian once for each point in the first iteration, and then uses the same Jacobian for each iteration. This method makes sense for a serial CPU implementation as the small performance gain by not recomputing the Jacobian adds up as the code loops over each pixel, and the small inaccuracies caused by not recomputing the warp jacobians does not affect the ability to converge to the true transformation.

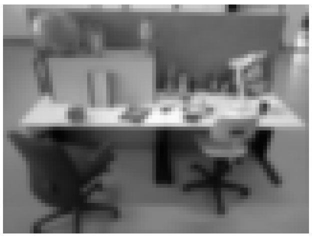
The situation is, however, different when implementing this on a GPU. First of all, the small increase in performance by not recomputing the Jacobian becomes insignificant when running in parallel as it is not added together many times as with the loop in a serial implementation. We would also need to store the Jacobian in the first iteration in memory and load it back for each iteration, and such memory operations would take far more time than the few instructions we save by not recomputing the jacobians. Therefore our implementation is more suited for a GPU.

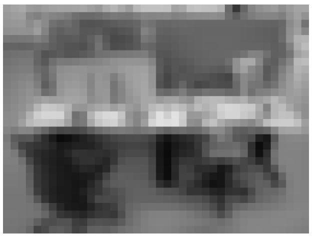
The core part of the CUDA kernel computing the residuals and Jacobians is given in listing 4.1. Here line 10 to 15 corresponds to computing eq. (4.25). We also make use of the hardware implemented bi-linear texture sampling described in section 3.3.5 to sample the gradients, on line 1 and 2, and brightness values on line 17. The operation at line 22 allows us to count the number of active points, which is used by later operations, as the given code is only run on points that fall within the image and points outside the image have their Jacobian and residual values set to zero. A visualization of the resulting Jacobian values for each of the six columns in the final Jacobian is shown in figs. 4.7a to 4.7f.

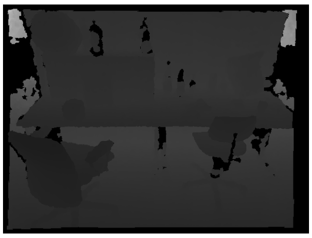

(a) Level 1 brightness

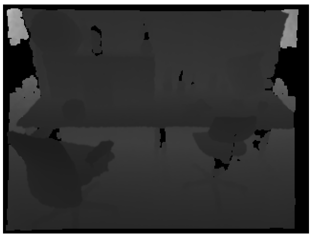

(b) Level 2 brightness

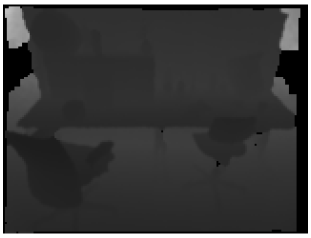

(c) Level 3 brightness

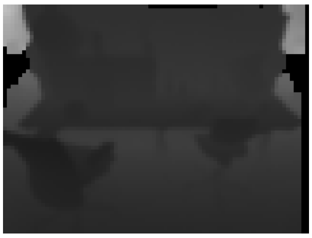

(d) Level 4 brightness


(e) Level 5 brightness


(f) Level 1 depth


(g) Level 2 depth


(h) Level 3 depth


(i) Level 4 depth

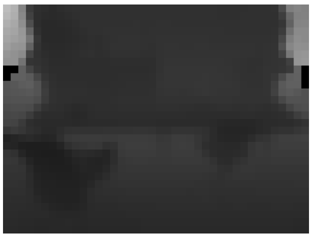
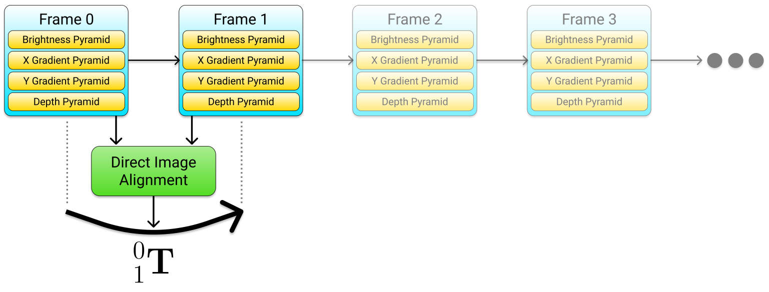
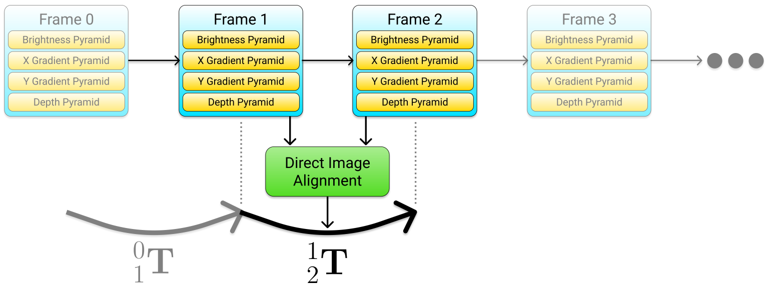

(j) Level 5 depth

Figure 4.4: Brightness and depth pyramids created from images from [32]



(a) Alignment of frame 0 & frame 1



(b) Alignment of frame 1 & frame 2

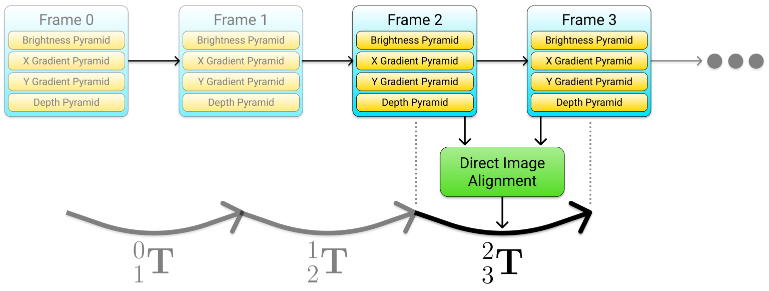
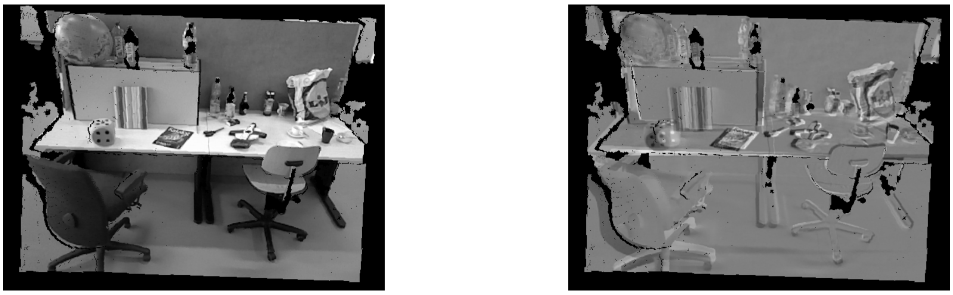


Figure 4.5: Direct image alignment of image stream



(a) Warp Image **(b)** Warp Error

Figure 4.6: Warp Function: Image & Error

```
const float gradXValue = tex2D<float>(gradXTexture, static_cast<float
    >(u + 0.5), static_cast<float>(v + 0.5));
const float gradYValue = tex2D<float>(gradYTexture, static_cast<float
    >(u + 0.5), static_cast<float>(v + 0.5));

const double Ix = static_cast<double>(gradXValue);
const double Iy = static_cast<double>(gradYValue);

const double z_div = 1.0/z;
const double z2_div = 1.0/(z*z);

Jptr[pointId + 0*nPoints] =  (Ix*fx)*z_div;
Jptr[pointId + 1*nPoints] =  (Iy*fy)*z_div;
Jptr[pointId + 2*nPoints] = -(Ix*fx*x)*z2_div - (Iy*fy*y)*z2_div;
Jptr[pointId + 3*nPoints] = -Iy*fy - y*((Ix*fx*x) + (Iy*fy*y))*z2_div;
Jptr[pointId + 4*nPoints] =  Ix*fx + x*((Ix*fx*x) + (Iy*fy*y))*z2_div;
Jptr[pointId + 5*nPoints] = (Iy*fy*x)*z_div - (Ix*fx*y)*z_div;

const double currentImageValue = static_cast<double>(tex2D<float>(
    currentImageTexture, static_cast<float>(u + 0.5), static_cast<
    float>(v + 0.5)));
const double previousImageValue = static_cast<double>(point.intensity)
    /std::numeric_limits<uint8_t>::max();
const double residual = currentImageValue - previousImageValue;
residualPtr[pointId] = residual;

atomicInc(nActivePointsPtr, 0xFFFFFFFF); // Count up the number of
    points that are active in the optimization
```

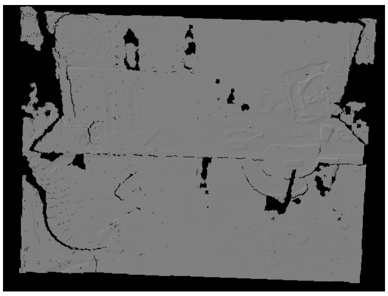
Listing 4.1: DVO Jacobian CUDA code

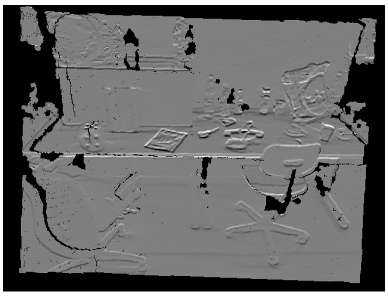
Linear Equation Solver

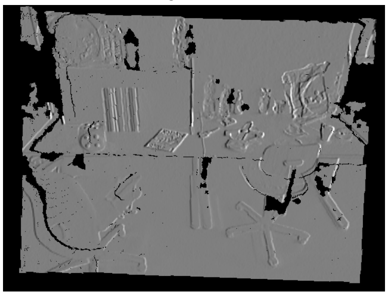
Once the Jacobian matrix and the residual vector is computed, we use the cuBLAS library introduced in section 3.4.1 to compute the matrix $\mathbf{A} = \mathbf{J}^t\mathbf{J}$ and the vector $\mathbf{b} = -\mathbf{J}^T\mathbf{r}$. This computation is done, as shown in line 4 and 15, respectively, in listing 4.2. The resulting matrix and vector are quite small, at a size of only 6×6 elements for the matrix $\mathbf{A}$

(a) ξ_1 Gradient

(b) ξ_2 Gradient


(c) ξ_3 Gradient


(d) ξ_4 Gradient


(e) ξ_5 Gradient

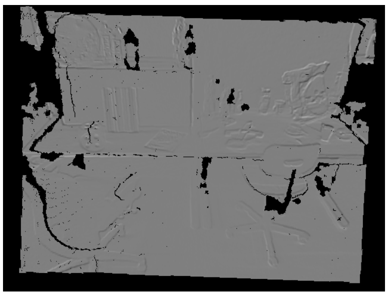

(f) ξ_6 Gradient

Figure 4.7: Warp Function gradients

and 6 elements for the vector **b**. We can therefore quickly copy them to the CPU memory and do the final equation solving using the robust Cholesky decomposition implemented in the Eigen C++ library, as shown on line 32.

```
1  const double identityScalar = 1.0;
2  const double zeroScalar = 0.0;
3
4  cublasDgemm_v2(cublasHandle,
5                 CUBLAS_OP_T, CUBLAS_OP_N,
6                 6, 6, nPoints,
7                 &identityScalar,
8                 alignImageBuffers.d_J, nPoints,
9                 alignImageBuffers.d_J, nPoints,
10                &zeroScalar,
11                alignImageBuffers.d_A, 6);
12
13 const double negativeIdentityScalar = -1.0;
14
15 cublasDgemm_v2(cublasHandle,
16                CUBLAS_OP_T, CUBLAS_OP_N,
17                6, 1, nPoints,
18                &negativeIdentityScalar,
19                alignImageBuffers.d_J, nPoints,
20                alignImageBuffers.d_residual, nPoints,
21                &zeroScalar,
22                alignImageBuffers.d_b, 6);
23
24 Matrix6d A;
25 cudaMemcpyAsync(static_cast<void*>(A.data()), static_cast<void*>(
       alignImageBuffers.d_A), 6*6*sizeof(double), cudaMemcpyDeviceToHost
       , stream);
26
27 Vector6d b;
28 cudaMemcpyAsync(static_cast<void*>(b.data()), static_cast<void*>(
       alignImageBuffers.d_b), 6*sizeof(double), cudaMemcpyDeviceToHost,
       stream);
29
30 CUDA_SAFE_CALL( cudaStreamSynchronize(stream) );
31
32 const Vector6d xi_delta = A.ldlt().solve(b);
33 xi = Sophus::SE3d::log(currentPose * Sophus::SE3d::exp(xi_delta));
```

Listing 4.2: DVO Jacobian CUDA code

This whole process is done run multiple times until the Euclidean length of the resulting $\Delta\xi$ is sufficiently small. The algorithm then moves onto the next level in the image pyramid, using the current transformation found in the last layer as the starting point for the next. This method of optimizing over increasingly higher resolution images in the image pyramid has two primary purposes.

The first is robustness, as lowering the resolution as the effect of smoothing out the residual landscape. This effect can be seen in section 4.1.5 where we have plotted the value of the residual resulting from different changes in components of the transformation ξ at different levels. We can see that the residual curves of the higher levels are smoother than the other ones. While it may appear that residuals are fully convex even at the lower levels, it is essential to remember that this figure is only a small sampling along the principal axis of the $\mathfrak{se}(3)$ vector space. The real landscape is a highly complex multi-dimensional landscape that could in certain situations contain local minimas which would be smoothed away at lower resolution levels.

Another important gain is performance. As the lower levels contain far fewer pixels, they are much faster to optimize over. Starting the optimization on the lower resolution images means we can achieve a rough initial estimation of the transformation quickly, and then only spend a few iterations on the more computationally expensive higher layers to fine-tune the transformation.

4.2 Robust Estimation

In this section, we will expand on the method and implementation we described in section 4.1. This implementation does function on its own, but [15] outlines two methods of significantly increasing the robustness of the implementation. These are outlier rejection, which makes the direct image alignment more robust against errors in the data. We will cover this feature in section 4.2.1. We will also cover the implementation of a motion model in section 4.2.2. The use of a motion model dramatically reduces the amount of error in cases of input with a lot of error, like motion blur, and makes the optimization more likely to converge to the global minimum in the instances of a non-convex residual landscape.

4.2.1 Outlier Suppression

The outlier suppression is based on the theory we established in section 2.3.7, where we use a weighting function to reduce the effect of outliers.

The addition of weights to the optimization changes eq. (4.19) to eq. (4.28). Where the matrix $\mathbf{W}$ is a diagonal matrix of all the weights.

$$\mathbf{J}^T\mathbf{W}\mathbf{J}\Delta\boldsymbol{\xi} = -\mathbf{J}^T\mathbf{W}\mathbf{r} \tag{4.28}$$

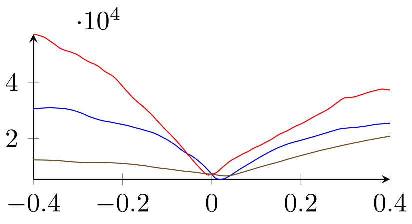
We can in eq. (4.29) see that this does not change the basic structure of the final equation, and can, therefore, be solved in the same manner. We calculate the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ in a slightly different way.

$$\underbrace{\mathbf{J}^T\mathbf{W}\mathbf{J}}_{\mathbf{A}}\underbrace{\Delta\boldsymbol{\xi}}_{\mathbf{x}} = \underbrace{-\mathbf{J}^T\mathbf{W}\mathbf{r}}_{\mathbf{b}} \tag{4.29}$$

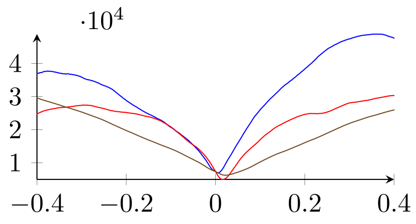
The original DVO paper[15] advocates for using a student-t weighting. This method does, however, require iterating over all the residuals multiple times to solve for the standard deviation of the student-t distribution. We chose instead to use Hubert weighting, as described in section 2.3.7, as the standard deviation to be used in the weights, can be calculated in a single pass, and still provide great robustness against outliers.

4.2.2 Motion Model

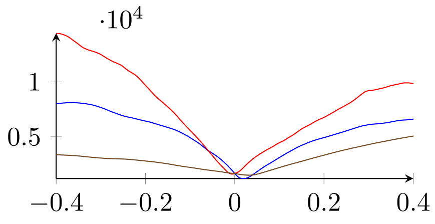
As we are estimating the motion of a physical system in the real world, not all possible motions are equally likely. The system has mass and is therefore limited in how fast it may



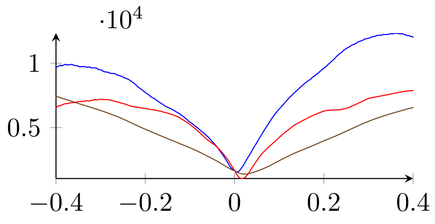
(a) Level 1 translational residuals



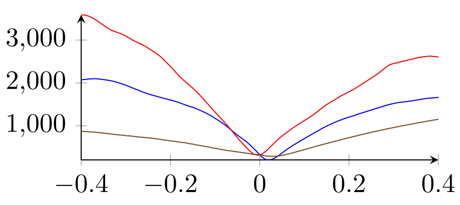
(b) Level 1 rotational residuals



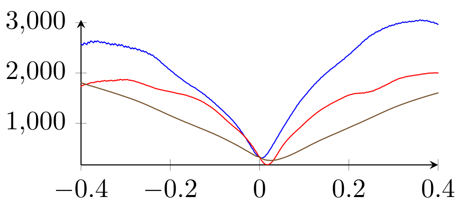
(c) Level 2 translational residuals



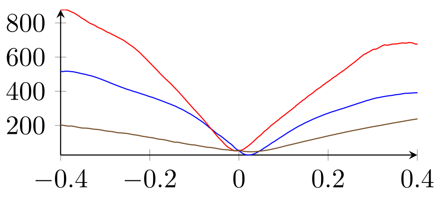
(d) Level 2 rotational residuals



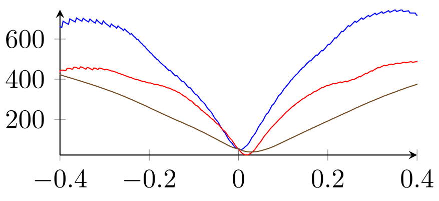
(e) Level 3 translational residuals



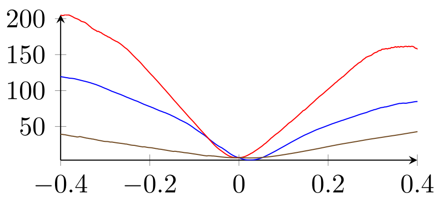
(f) Level 3 rotational residuals



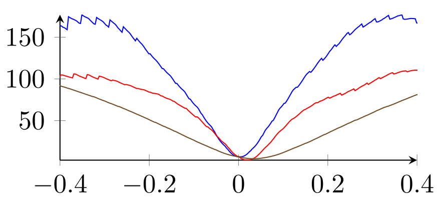
(g) Level 4 translational residuals



(h) Level 4 rotational residuals



(i) Level 5 translational residuals



(j) Level 5 rotational residuals

Figure 4.8: Translational & rotational residuals at different image pyramid levels

accelerate. This knowledge is not embedded anywhere in our algorithm, but it is possible for us to do so by making a few alterations.

First of all, we can change what transformation our optimization starts with. Our current implementation begins with an identity transformation and starts optimizing from there. We change this by letting the optimization algorithm start optimizing from the transformation of the previous frame. This change improves performance, as we will quite often already be close to the true transformation. This change also improves robustness as we are more likely to already be in a convex area around the global minima, where we are guaranteed to converge to the optimal solution.

We may also penalize the optimization from moving too far away from the transformation of the last frame. This is achieved by further modifying eq. (4.28) into eq. (4.30). Here $\boldsymbol{\xi}_{t-1}$ is the transformation from the previous frame and $\boldsymbol{\xi}_t$ is the transformation from the previous iteration. The matrix $\boldsymbol{\Sigma}$, is a diagonal 6×6 matrix as shown in eq. (4.31). Each element in the diagonal matrix defines the strength of the motion model for its respective term in the transformation vector $\boldsymbol{\xi}$, where a smaller value increases the penalization for value away from the motion model.

$$\left(\mathbf{J}^T\mathbf{W}\mathbf{J} + \boldsymbol{\Sigma}^{-1}\right)\Delta\boldsymbol{\xi} = -\mathbf{J}^T\mathbf{W}\mathbf{r} + \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{\xi}_{t-1} - \boldsymbol{\xi}_t\right) \tag{4.30}$$

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \sigma_6 \end{bmatrix} \tag{4.31}$$

As before, we do not alter the structure of the linear equation we must solve, and we are instead redefining the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ as shown in eq. (4.32).

$$\underbrace{\left(\mathbf{J}^T\mathbf{W}\mathbf{J} + \boldsymbol{\Sigma}^{-1}\right)}_{\mathbf{A}}\Delta\boldsymbol{\xi} = \underbrace{-\mathbf{J}^T\mathbf{W}\mathbf{r} + \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{\xi}_{t-1} - \boldsymbol{\xi}_t\right)}_{\mathbf{b}} \tag{4.32}$$

4.2.3 Implementation

We first give an overview of how the implementation was changed to include robust weighting. The cuBLAS library does not support diagonal matrices, and we wanted to avoid having to create the actually $\mathbf{W}$ matrix in memory, as it is a very large matrix with almost only zero values. We chose a more efficient implementation where we calculate the weights as a vector, as shown in listing 4.3. We then multiply each element of the vector with the corresponding row of the matrix $\mathbf{J}$ to get the weighted matrix $\mathbf{J}'$. We then calculate the matrix $\mathbf{A}$ with cuBLAS as before, now using $\mathbf{A} = \mathbf{J}^T\mathbf{J}'$. We use the same matrix $\mathbf{J}'$ to calculate the vector $\mathbf{b} = -(\mathbf{J}')^T\mathbf{r}$.

```
__global__ void calculateHuberWeights(double* residualArray, double*
    weightArray, double* variancePtr, const uint32_t nArrayElements,
    uint32_t* nActivePointsPtr)
{
  const uint32_t arrayIndex = blockIdx.x * blockDim.x + threadIdx.x;
  if (arrayIndex >= nArrayElements) { return; }

  // Value of 1.345 gives 95% efficiency in cases of gaussian
    distribution and is commonly used with huber weights
  const double k = 1.345*sqrt((1.0/(*nActivePointsPtr))*(*variancePtr)
    );
  const double absError = abs(residualArray[arrayIndex]);

  if (absError > k)
  {
    weightArray[arrayIndex] = k/absError;
  }
  else
  {
    weightArray[arrayIndex] = 1.0;
  }
}
```

Listing 4.3: Huber weight calculation

To implement the motion model, we can alter the final linear solving that is done on the CPU. This is done as shown in listing 4.4. We also multiply $\Delta\xi$ with a step length variable which starts at a value of 1 at the beginning of each level and is decreased a bit for each level. This step length variable was found to increase the convergence rate of the optimization slightly.

```
Matrix6d A;
cudaMemcpyAsync(static_cast<void*>(A.data()), static_cast<void*>(
    alignImageBuffers.d_A), 6*6*sizeof(double), cudaMemcpyDeviceToHost
    , stream);

Vector6d b;
cudaMemcpyAsync(static_cast<void*>(b.data()), static_cast<void*>(
    alignImageBuffers.d_b), 6*sizeof(double), cudaMemcpyDeviceToHost,
    stream);

CUDA_SAFE_CALL( cudaStreamSynchronize(stream) );

const Vector6d xi_delta = stepLength * (A + sigma_inv).ldlt().solve(b
    + sigma_inv*(initialXi - xi));
```

Listing 4.4: Linear solver with motion model

4.3 Visualization

A system for visualizing the algorithm while it is running was also implemented, which is seen in fig. 4.9. While there exist many tools for easy visualization of CPU programs, there is far fewer to choose from when it comes to CUDA programs. We, therefore, to implement our system using the OpenGL interoperability functionality that exists in CUDA, as described in section 3.5. The system is capable of visualizing the estimated trajectory alongside the ground truth trajectory and the pixel projected into the scene using the depth image. It also displays the image and depth pyramid.

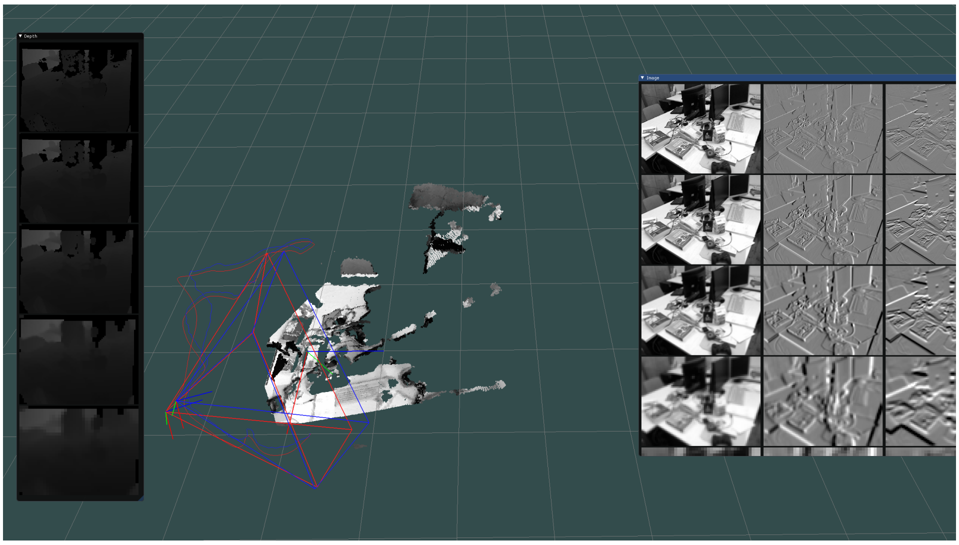


Figure 4.9: DVO Visualization

Chapter 5

Evaluation

We will, in this chapter, evaluate the performance of our DVO implementation, in terms of both speed and accuracy. For this we will use the RGB-D dataset from [32]. From this dataset, we will focus on the three sequences given in table 5.1, which together showcase several aspects of the implementation. The tests where run on a machine using an AMD Ryzen 5 2600X CPU and an Nvidia Geforce 1070 Ti GPU.

5.1 Functionality

The *freiburg1_desk* sequence is a shorter sequence and will be used to demonstrate the basic functionality of the implementation, while the *freiburg3_long_office_household* is a longer, more complex sequence that we will use to show the effect of changing the parameters of the algorithm. Finally, we will look at the *freiburg2_pioneer_slam* sequence, which illustrate some of the limitations of the implementation and shows how the performance is affected by motion blur.

Sequence	Length	Description
freiburg1_desk	23.40s	Shows several sweeps over four desks in a office environment.
freiburg3_long_office_household	87.09s	The camera moves in a large circle around a office scene in the center.
freiburg2_pioneer_slam	155.72s	The camera is mounted on top of a Pioneer robot as it is driving around a hall with several large objects.

Table 5.1: Dataset sequences

Figure 5.1 shows a plot of the XZ-position of both the ground-truth and the estimated trajectory of the *freiburg1_desk* sequence. We can from this plot see that the implementation can track the path of the camera reasonably well. There is some drifting over time, but this is to be expected as each transformation between two frames will have some small error that adds up over time.

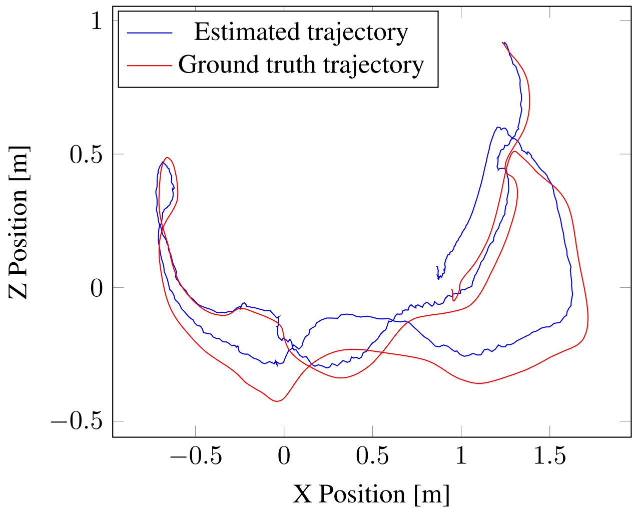


Figure 5.1: Trajectory of *freiburg1_desk* sequence

Figure 5.2 shows the number of iterations to converge for each of the levels. There is a spike in the number of iterations in the beginning as the motion model is initialized, but the optimization then stabilizes at around 18-20 iterations per frame. We can see that most of the iterations are done in the higher levels, which are much cheaper to compute, and just 1-2 iterations are done on the final layer.

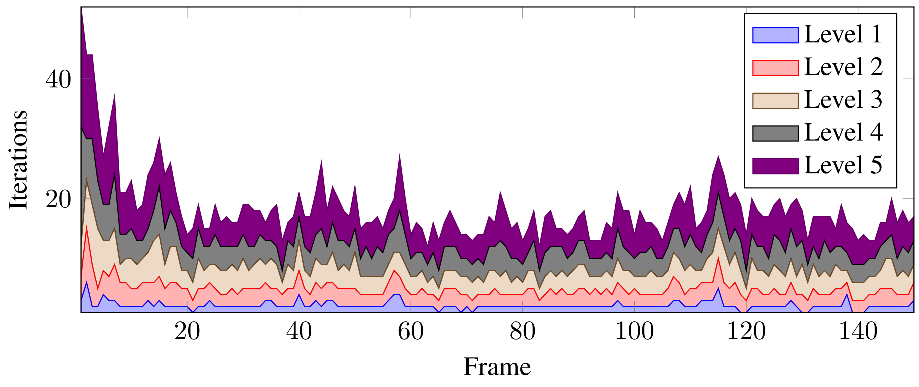


Figure 5.2: Iterations per level for first 150 frames

5.2 Stopping Criteria

We will now look at how changing some of the parameters of the implementation changes the accuracy and the performance of the implementation. We evaluate the accuracy of the path using the relative pose error with a distance of 1 second, as detailed in [32].

In table 5.2 we see the result from running the implementation on the *freiburg3_long_office_household* sequence. The optimization is set to run on each level until the euclidean length of $\Delta\xi$ becomes less than or equal to ξ_{min}. We can see that a lower value of ξ_{min} does decrease the accuracy of the trajectory, not by much however.

ξ_{min}	Translation RMSE	Translation RMSE	Iterations average (Level 1-5)
1e−5	0.015693 m	0.682646 deg	8.03 6.65 10.35 34.72 44.19
1e−4	0.015762 m	0.685008 deg	3.26 3.34 3.53 3.54 3.83
1e−3	0.016141 m	0.701809 deg	1.08 1.60 2.03 2.09 2.50
1e−2	0.016190 m	0.710133 deg	1.00 1.04 1.12 1.12 1.06
1e−1	0.017754 m	0.729568 deg	1.00 1.00 1.00 1.00 1.00

Table 5.2: Stopping criteria

We can in fig. 5.3 see that the change in accuracy, has minimal effect on the accuracy of the path. Astonishingly, even when we only run the implementation for a single iteration on each level, the path only degrades slightly. We found these results very surprising, as we expected the optimization landscape to be far more challenging to optimize. It should, however, be noted that this sequence is only 87 seconds long, and even a small increase in error will quickly grow over time. These results do, however, show the implementation can be run with very few iterations, and therefore run very quickly, in applications where some error over long periods is not detrimental to the functioning of the system.

5.3 Limitations & Challenges

We will, in this section, utilize the *freiburg2_pioneer_slam* to illustrate some of the limitations of the implementation. This sequence is recorded in a larger room with several large objects. The camera is placed on top of a Pioneer wheeled robot, which is remotely operated. The sequence is especially challenging as it contains several short segments of intensive motion blur as the robot drives over objects on the ground.

The Y-position of a small segment of the sequence is shown in fig. 5.4, at multiple different values for the standard deviation of the motion model. This plot shows how the position jumps at several places. We can see that high uncertainty in the motion model gives a much more significant jump; however, a very low uncertainty causes the estimation to calculate a wrong path outside the jumps. This sequence does unfortunately not have a working ground-truth path, but as the camera always stays the same height of the ground, we know the path should be a straight line.

Figure 5.5 show the same segment as in fig. 5.4, but showing the XZ-coordinates. In this plot, we can see how a very low uncertainty in the motion model causes the path to drift away from the others.

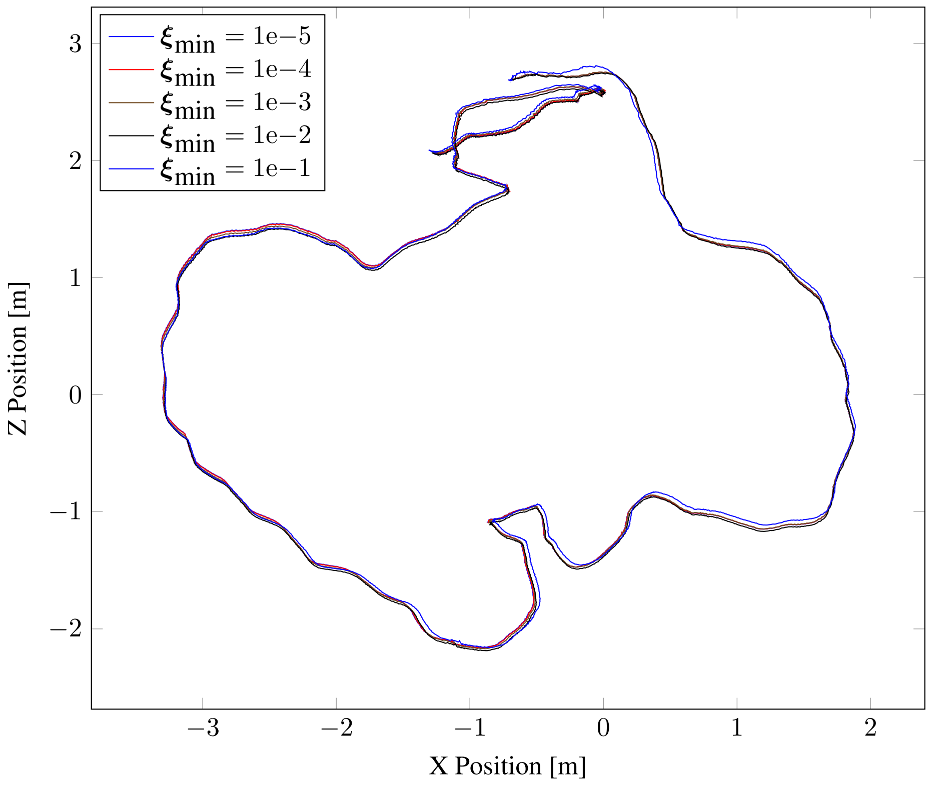


Figure 5.3: Trajectory of *freiburg3_long_office_household* sequence

We can see that we can decrease the effect of motion blur using our motion model, but we must be careful not to use too low uncertainty as it causes the path to drift significantly. No value does, however, remove the effect of the motion blur and we are not able to avoid it having a significant impact on the performance of the implementation.

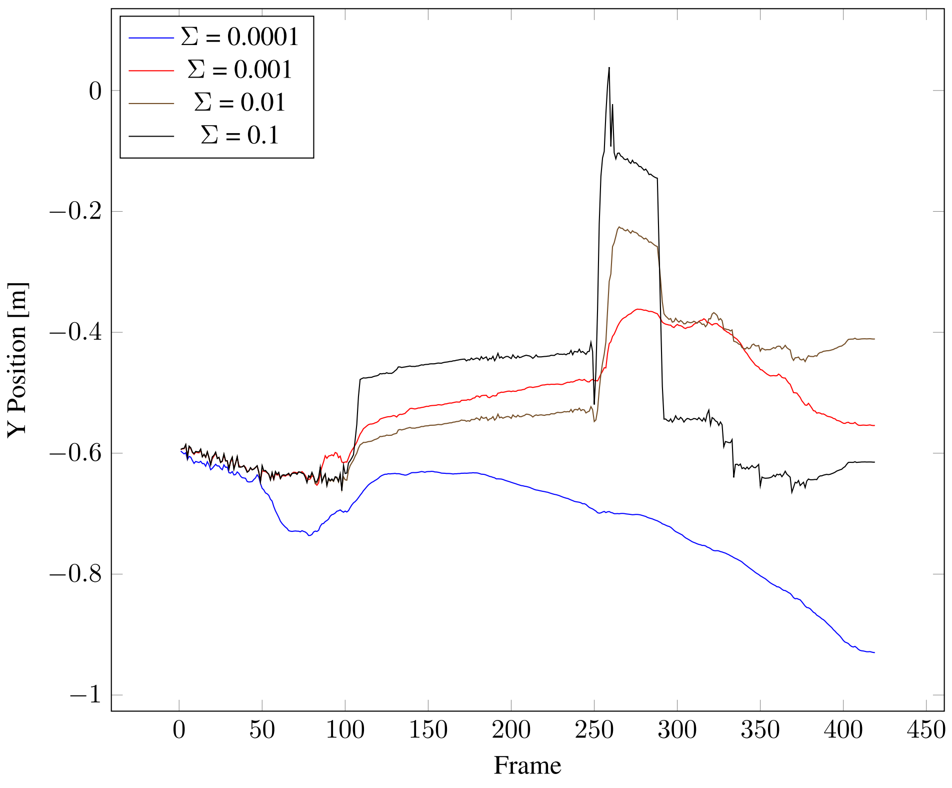


Figure 5.4: Y-position of challenging segment of the *freiburg2_pioneer_slam* sequence

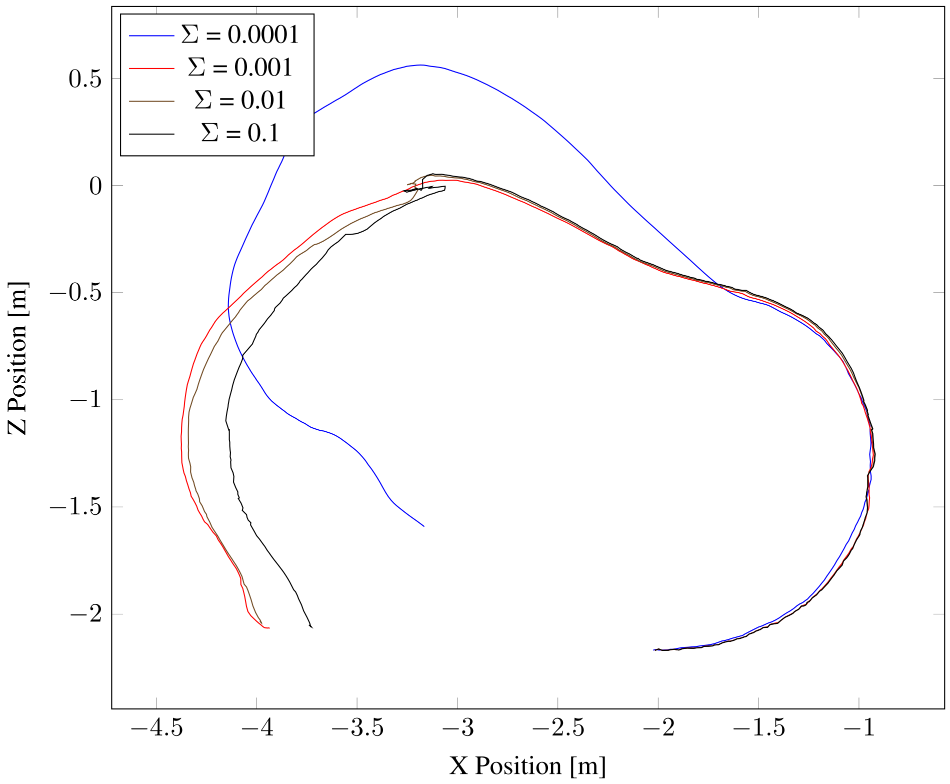


Figure 5.5: XZ-position of challenging segment of the *freiburg2_pioneer_slam* sequence

Chapter 6

Conclusions & Further Work

In this thesis, we have shown how the DVO algorithm can be implemented on a GPU effectively and efficiently. We saw that the CUDA library, together with the cuBLAS and Nvidia VisionWorks libraries provided us with the tools necessary to implement the algorithm in a massively parallel way on a GPGPU.

Through the implementation, we saw that some design decisions that was a good fit in a CPU implementation, where better implemented differently on a GPU. This difference was especially true with regards to memory access. A GPU has multiple different types of memory, and access to the global memory is an expensive operation causing the re-computation of particular values to be more efficient than fetching it from memory. We also saw that we could make use of some of the unique hardware features found in GPUs, like texture memory with hardware bi-linear interpolation.

Testing the algorithm with different parameters, we found that the implementation functioned surprisingly well with just a few iterations. We also saw how the motion prior was able to minimize significantly larger jumps caused by errors in the optimization but needed to be appropriately adjusted not negatively to affect the optimization.

There is, however, still room for improvements. We did not make use of the cuSolve library, created by Nvidia. This library is a GPU linear solver library that could have been used to avoid having to move data to the CPU memory to perform the final solving in the optimization. We could also have improved the motion model by using data from an IMU (Inertial Measuring Unit) so that instead of merely using the previous transformation, we would use the measurement from the IMU. This change would most likely greatly improved the issues that were observed with motion blur in the images.

DVO is a relatively simple direct visual odometry algorithm. Therefore it would be an interesting future work to investigate possible GPU implementation of more advanced algorithms, such as [5], [6] or [9]. Some of these more advanced methods use multiple processes running simultaneously with coordination and data transmitting between them. These are complexities which would make a GPU implementation much more challenging to implement and optimize.

We have also only shown the implementation running on a desktop computer, and it remains to be shown how well this implementation is capable of running on integrated platforms, such as Nvidia Jetsons, or Nvidia Drive. While the implementation should be able to run without any alterations, it should be altered slightly to take advantage of the hardware feature available in such platforms, such as unified GPU-CPU memory, tensor cores, and vision processing units.

Bibliography

[1] Barfoot, T. D., 2017. State Estimation for Robotics. Cambridge.

[2] Bay, H., Tuytelaars, T., Van Gool, L., 2006. Surf: Speeded up robust features. In: European conference on computer vision. Springer, pp. 404–417.

[3] Blackford, L. S., Petitet, A., Pozo, R., Remington, K., Whaley, R. C., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., et al., 2002. An updated set of basic linear algebra subprograms (blas). ACM Transactions on Mathematical Software 28 (2), 135–151.

[4] Commons, W., 2019. Cmos image sensor.
URL `https://commons.wikimedia.org/wiki/File:Matrixw.jpg`

[5] Engel, J., Koltun, V., Cremers, D., July 2016. Direct sparse odometry. In: arXiv:1607.02565.

[6] Engel, J., Schöps, T., Cremers, D., September 2014. LSD-SLAM: Large-scale direct monocular SLAM. In: European Conference on Computer Vision (ECCV).

[7] Engel, J., Stückler, J., Cremers, D., 2015. Large-scale direct slam with stereo cameras. In: Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on. IEEE, pp. 1935–1942.

[8] Engel, J., Usenko, V., Cremers, D., July 2016. A photometrically calibrated benchmark for monocular visual odometry. In: arXiv:1607.02555.

[9] Forster, C., Pizzoli, M., Scaramuzza, D., 2014. Svo: Fast semi-direct monocular visual odometry. In: Robotics and Automation (ICRA), 2014 IEEE International Conference on. IEEE, pp. 15–22.

[10] Harris, C., Stephens, M., 1988. A combined corner and edge detector. In: Alvey vision conference. Vol. 15. Citeseer, pp. 10–5244.

[11] Hoshino, T., Maruyama, N., Matsuoka, S., Takaki, R., 2013. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application.

In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. IEEE, pp. 136–143.

[12] Huber, P. J., 1992. Robust estimation of a location parameter. In: Breakthroughs in statistics. Springer, pp. 492–518.

[13] Kannala, J., Brandt, S. S., 2006. A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses. IEEE transactions on pattern analysis and machine intelligence 28 (8), 1335–1340.

[14] Karimi, K., Dickson, N. G., Hamze, F., 2010. A performance comparison of cuda and opencl. arXiv preprint arXiv:1005.2581.

[15] Kerl, C., Sturm, J., Cremers, D., 2013. Robust odometry estimation for rgb-d cameras. In: 2013 IEEE International Conference on Robotics and Automation. IEEE, pp. 3748–3754.

[16] Klein, G., Murray, D., 2007. Parallel tracking and mapping for small ar workspaces. In: Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on. IEEE, pp. 225–234.

[17] Krames, J., 1941. Zur ermittlung eines objektes aus zwei perspektiven.(ein beitrag zur theorie der "gefährlichen örter".). Monatshefte für Mathematik und Physik 49 (1), 327–354.

[18] Lowe, D. G., 1999. Object recognition from local scale-invariant features. In: Computer vision, 1999. The proceedings of the seventh IEEE international conference on. Vol. 2. Ieee, pp. 1150–1157.

[19] Matthies, L., Szeliski, R., Kanade, T., 1988. Incremental estimation of dense depth maps from image sequences. In: Computer Vision and Pattern Recognition, 1988. Proceedings CVPR'88., Computer Society Conference on. IEEE, pp. 366–374.

[20] Moutarlier, P., Chatila, R., 1990. An experimental system for incremental environment modelling by an autonomous mobile robot. In: Experimental Robotics I. Springer, pp. 327–346.

[21] Mur-Artal, R., Tardós, J. D., 2017. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. IEEE Transactions on Robotics 33 (5), 1255–1262.

[22] Newcombe, R. A., Lovegrove, S. J., Davison, A. J., 2011. Dtam: Dense tracking and mapping in real-time. In: Computer Vision (ICCV), 2011 IEEE International Conference on. IEEE, pp. 2320–2327.

[23] Nickolls, J., Buck, I., Garland, M., 2008. Scalable parallel programming. In: 2008 IEEE Hot Chips 20 Symposium (HCS). IEEE, pp. 40–53.

[24] Nocedal, J., Wright, S. J., 2006. Numerical Optimization, 2nd Edition. Springer.

[25] Nvidia, 2019. Cuda C Programming Guide.
URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[26] Richardson, A., Strom, J., Olson, E., 2013. Aprilcal: Assisted and repeatable camera calibration. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, pp. 1814–1821.

[27] Rublee, E., Rabaud, V., Konolige, K., Bradski, G., 2011. Orb: An efficient alternative to sift or surf. In: Computer Vision (ICCV), 2011 IEEE international conference on. IEEE, pp. 2564–2571.

[28] Schubert, D., Demmel, N., Usenko, V., Stückler, J., Cremers, D., 08 2018. Direct sparse odometry with rolling shutter.

[29] Schubert, D., Goll, T., Demmel, N., Usenko, V., Stückler, J., Cremers, D., 2018. The tum vi benchmark for evaluating visual-inertial odometry. arXiv preprint arXiv:1804.06120.

[30] Steinbrücker, F., Sturm, J., Cremers, D., 2011. Real-time visual odometry from dense rgb-d images. In: Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on. IEEE, pp. 719–722.

[31] Strasdat, H., Montiel, J., Davison, A. J., 2010. Real-time monocular slam: Why filter? In: Robotics and Automation (ICRA), 2010 IEEE International Conference on. IEEE, pp. 2657–2664.

[32] Sturm, J., Engelhard, N., Endres, F., Burgard, W., Cremers, D., Oct. 2012. A benchmark for the evaluation of rgb-d slam systems. In: Proc. of the International Conference on Intelligent Robot Systems (IROS).

[33] Szeliski, R., 2011. Computer Vision: Algorithms and Applications. Springer.

[34] Tomasi, C., Kanade, T., 1991. Detection and tracking of point features.

[35] Trajković, M., Hedley, M., 1998. Fast corner detection. Image and vision computing 16 (2), 75–87.

[36] Wang, R., Schwörer, M., Cremers, D., 2017. Stereo dso: Large-scale direct sparse visual odometry with stereo cameras. In: International Conference on Computer Vision (ICCV). Vol. 42.