Master's thesis

2019

Hans Kristian Holen

NTNU
Norwegian University of
Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Hans Kristian Holen

Camera Pose Estimation using Line Segments

A study of DLT-Plücker-Lines and its use case in a real time system

July 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Camera Pose Estimation using Line Segments

A study of DLT-Plücker-Lines and its use case in a real time system

Hans Kristian Holen

Manufacturing Technology
Submission date: July 2019
Supervisor: Olav Egeland

Norwegian University of Science and Technology
Department of Mechanical and Industrial Engineering

Camera Pose Estimation using Line Segments

A study of DLT-Plücker-Lines and its use case in a real time system

Hans Kristian Holen

2019-07-15

Contents

List of Figures

Abstract

This Master's thesis is aimed to study the camera pose estimation problem through using line segments, Perspective-n-Lines (PnL). Pose estimation is essential for mobile robots to navigate or operate a robot only using vision. Another aspect of the thesis has been to look into use cases and comprehending methods that is necessary for real world cases. Perspective-n-Lines uses line correspondences between the image 2D plane and the available 3D model of the surroundings.

DLT-Plücker-Lines is the method that has been implemented and tested on real world datasets. Experiments were conducted on two different surroundings and both orientation and translation errors were measured along with the speed. The results were then compared to three other state of the art algorithms and the results were very promising. One of the methods had a slightly better accuracy, however it is worth mentioning this method is found to be slower by other sources.

A point worth mentioning from the test is the speed of the algorithm. The results were somewhat worse than expected, but there has not been that much focus on making the algorithm as fast as possible either. However, if being used in a real time system, this has to be looked further into.

Through the thesis there are described methods that are necessary for estimating camera pose without using predescribed datasets, wheres one of these methods are the Line Segment Detector. This method has been implemented in code using the open source library OpenCV and then been compared to the data in the datasets. Using the standard parameterization gave a result of five times as many line segments than were in the datasets, however this can be lowered by changing the parameters from the standards to customized. Finally there is a proposed suggestion for solving the Simultaneous Localization and Mapping (SLAM) problem using the described methods, where camera pose estimation is an essential part.

Sammendrag

Denne masteroppgaven har som mål å se nærmere på estimering av kameraposisjon ved å benytte seg av linjer samt bruksområder rundt dette feltet. Posisjonsestimering med linjer bruker korrespondansen mellom linjer gjenkjent i 2D kamerabilde og linjer fra tilgjengelig 3D-modell av omgivelsene. Estimering av posisjon er essensielt ved mobile roboter eller dersom en robot estimerer bevegelser kun ved hjelp av syn.

DLT-Plücker-Lines er metoden som har blitt implementert og testet på datasett fra virkelige scenarioer. Det ble gjort målinger på avviket mellom estimert og virkelige målinger på både translasjon samt orientering. Resultatene ble sammenlignet med tre andre anerkjente metoder og viste seg å være svært gode. En av disse algoritmene ga bedre nøyaktighet, dog bare med små marginer. Det må også påpekes at denne algoritmen er funnet å være en tregere algoritme av andre kilder.

Et annet punkt verdt å bemerke seg fra resultatene er hastigheten til algoritmen. Det virker noe tregt, uten at det har vært spesielt i fokus i denne omgang å gjøre algoritmen så effektiv som mulig. Skulle det derimot være behov for å bruke den i sanntidssystemer må dette fokuseres mer på.

Gjennom oppgaven er det beskrevet metoder som må til for at estimering av kameraposisjon skal være mulig uten bruk av ferdige datasett, hvor en av disse metodene er Line Segment Detector. Denne metoden er også implementert i kode ved hjelp av open-source biblioteket OpenCV for å sammenligne resultater med data fra datasett. Med standardparametrene blir det funnet om lag fem ganger flere linjer i bildene enn det er oppgitt i datasettet, men dette kan enkelt løses ved å manuelt stille parametrene. Til slutt er det foreslått hvordan problemet rundt Simultaneous Localization and Mapping (SLAM) kan bli løst ved hjelp av disse metodene, hvor estimering av kameraposisjon er helt essensielt.

Acknowledgements

First and foremost, I would like to thank my supervisor Olav Egeland for lots of valuable help and input during my work. During our weekly meeting Olav always had some new theory for me to look into and was always up to date on relevant papers. As a consequence, a lot of the theory that lies behind this thesis and acquired knowledge comes from these papers.

I would also like to thank all people at the masters office for being excellent discussion partners. Also, sitting at the office after midnight in the weekends would be a lot more depressing if it were not for you. We have made each other better and helped each other whenever we got stucked.

Chapter 1.

Introduction

As automation gets an increasing importance in industry and society in general the focus on finding new robots and computer vision solutions is essential. In some way a lot of these solutions is about simulating human tasks, which could include simulating our movements or vision.
The goal of computer vision is to allow computers to see, like a human or even better. A big part for this to happening is achieved through the exploitation of features. These features are parts of an image that contains distinctive information in some way, which can be lines, points, regions, shape etc. If having a scene of images geometric relations can be obtained using these features, which makes it possible to construct a 3D model of the scene, localize as well as navigate a mobile robot.
A well known challenge within robotics is Simultaneous Localization and Mapping (SLAM), which is the problem of determine both the localization as well as the surroundings of a mobile robot if placed in an unknown environment. This has to be solved by incrementally make and update a map of observed features as well as keeping track of the robots localization within it. A crucial part for this application to work is pose estimation, which is the task of determining the relative position and orientation of a camera and an object to each other in 3D space.
Pose estimation methods using point features have been in the main focus for some time as the mathematical complexity is lower. While points hold information about an exact location, lines describes directions and are also more robust as they can still be viewed be used even if partially occluded, hence could be very useful in some cases. Line pose estimation uses line-line correspondences between lines segments found in the image plane and lines from a 3D model. The advantage of an accurate camera pose estimation in a SLAM system is both getting the location and orientation as accurate as possible which in turn results in better accuracy of new features discovered.

1.1. Objective

The aim of this project is to examine the DLT-Plücker-Lines method, and look into the theory that lies behind. Through the period there is planned implementing the algorithm and test it on two different datasets, with different properties. The goal is to get results that can be compared to various state of the art methods as well as looking into methods that are necessary for implementation into a real world applications. Looking into use cases where the method is applicable is also desired, such as how it can be used improving mobile robots. Hence, a major part of the study will be used looking into methods that are necessary to combine with DLT-Plücker-Lines, such as feature detection, to make a system like this work. Ultimately there will be a proposal for how this can be implemented with other methods to become a SLAM system.

1.2. Scope

This thesis will focus on pose estimation using line segments and the methods, such as detecting features and matching of features from different images, that are essential for it to work in real world applications. The DLT-Plücker-Lines method will be coded and results will be compared to other state of the art methods. Finally there is proposed a potential SLAM application, based on the methods described, where pose estimation is essential. Because a total system for these kind of applications consist of several complex methodologies, only a few will be described in the thesis. The ones described are selected on the basis of the background research of which methods seems most fitted.

Chapter 2.

Background theory

2.1. Bayes filtering

One of the most general algorithms for calculating beliefs is given by the Bayes Filter algorithm. The algorithm calculates the posterior distribution bel_t over state, x_t based on measurement, z_t, and control inputs, u_t, at a given time, t. The belief $\text{bel}(x_t)$ is calculated from the belief $\text{bel}(x_{t-1})$ at time $t-1$.
The Bayes Filter makes a Markov assumption which means the state is describing all past states and implies that the belief is adequate to represent the robot's moves.

Algorithm 1 Bayes Filter

```
procedure BAYES(bel(x_{t-1}), u_t, z_t)
    for all x_t do
        \overline{bel}(x_t) = ∫ p(x_t|u_t, z_t bel(x_{t-1}) dx_{t-1})
        bel(x_t) = η p(z_t|x_t) \overline{bel}(x_t)
    end for
    return bel(x_t)
```

The algorithm above consists of mainly two steps, the prediction step and the update step. In the prediction step, the previous belief, or posterior distribution, is extrapolated to the time of the measurement according to the control input. While in the update step, the new measurement is taken into account for improving the prediction.

2.1.1. Kalman Filter

The Kalman Filter is one of the most used techniques for implementing Bayes filters [21] and is a method for filtering and prediction in linear Gaussian systems. A linear Gaussian is a state transition probability on the form $x_t = A_t x_{t-1} +$

$B_t u_t + \epsilon$, which as seen is linear in its arguments with additive Gaussian noise. The Kalman Filter represents posterior distribution by the moment parametrization: for each time t, the posterior distribution μ_t and covariance Σ_t. For the posterior to be Gaussian the following three properties are required:

1. The state transition probability $p(x_t|u_t, x_{t-1})$ must be a linear function with added Gaussian noise, expressed as:

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon \tag{2.1}$$

   Both x_t and x_{t-1} are state vectors, and u_t is the control vector at the time t. A_t and B_t are matrices representing the linear state transition $(n \times n)$ and control input models $(n \times m$ respectively. n being the dimension of x_t and m of u_t. ϵ is a random noise variable and represents the uncertainty introduced by the state transition. The random noise is a Gaussian distributed vector with zero mean and covariance Q_t. The mean of the posterior state is given by 2.1 and the covariance Q_t as shown in 2.2.

$$\begin{aligned} p(x_t|u_t, z_t) = \det(2\pi Q_t)^{-1/2} \\ \exp\{-\frac{1}{2}(x_t - A_t x_{t-1} + B_t u_t)^T Q_t^{-1}(x_t - A_t x_{t-1} + B_t u_t)\} \end{aligned} \tag{2.2}$$

2. The measurement probability $p(z_t|x_t)$ also have to be linear with added Gaussian noise:

$$z_t = C_t x_t + \delta_t \tag{2.3}$$

   z_t is the measurement vector with dimension k, while C_t is the linear measurement model, with the size $k \times n$. The vector δ_t represent measurement noise and its distribution is a multivariate Gaussian with zero mean and covariance R_t. The measurement probability will then become:

$$p(z_t|x_t) = \det(2\pi R_t)^{-1/2}\exp\{-\frac{1}{2}(z_t - R_t x_t)^T R_t^{-1}(z_t - R_t x_t)\} \tag{2.4}$$

3. The last and third property required is that the initial belief bel(x_0) need to be normally distributed with mean μ_0 and covariance Σ_0:

$$bel(x_0) = p(x_0) = \det(2\pi R_t)^{-1/2}\exp\{-\frac{1}{2}(x_0 - \mu_0)^T \Sigma_0^{-1}(z_0 - \mu_0)\} \tag{2.5}$$

With these three assumptions the posterior bel(x_t) is assured to alway be Gaussian, for any time and state.

The algorithm of the Kalman filter [21], as illustrated in 2, is similar to the algorithm for the Bayes filter with a prediction and a update step.

Algorithm 2 Kalman Filter

procedure KALMANFILTER($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)
Prediction step:
$\overline{\mu}_t = A_t \mu_{t-1} + B_t u_t$
$\overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$

Correction step:
$K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + Q_t)^- 1$
$\mu_t = \overline{\mu}_t + K_t (z_t - C_t \overline{\mu}_t)$
$\Sigma_t = (I - K_t C_t) \overline{\Sigma}_t$
return μ_t, Σ_t

In the correction step, the measurement is used to estimate the posterior distribution bel(x_t). The variable K is the Kalman gain, which decides whether or not the new measurement should be included in the new state estimate. In μ_t in the correction step the mean is adjusted in proportion to the Kalman gain and the innovation. The innovation is the difference between the measurement z_t and the expected measurement C_t and $\overline{\mu}_t$ in the last line in the algorithm.

2.2. Transformations

A homography is a linear transformation on homogeneous 3-vectors represented by a homogeneous, non-singular 3×3 matrix. A characteristic of homographies is that they preserve lines and the homographies can be divided into seven subgroups.

2.2.1. Translation

A translation transformation is the simplest homography with only two degrees of freedom which is described as

$$\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{t} \tag{2.6}$$

where the only transformation is a movement of the object.

2.2.2. Euclidean

An eucledian is a transformation with three degrees of freedom described as

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.7}$$

where the object is both translated as well orientated.

2.2.3. Similarity

A similarity has four degrees of freedom and described as

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} sr_{11} & sr_{12} & t_x \\ sr_{21} & sr_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.8}$$

where the object is translated, oriented and scaled between each image frame. However, corresponding lines are still parallel.

2.2.4. Affine

An affine translation has six degrees of freedom described as

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.9}$$

where the object is transformed just as in the similarity, but the corresponding lines are not longer parallel. However the length ratios between corresponding lines in each image frame are the same for all lines.

2.2.5. Projectivity

A full homography, or projectivity, is an invertible mapping from P^2 to P^2. The mapping can be represented by a non-singular 3×3 matrix H, which is referred to as the homography. The mapping is written $x' = Hx$, which can be written out as

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.10}$$

The homography H has 9 entries. As the scale is arbitrary and H is homogeneous, there are 8 independet entries.

In a projective transformation

$$H = \begin{bmatrix} A & t \\ v^T & v_3 \end{bmatrix} \tag{2.11}$$

where A is an nonsingular matrix, the H matrix can be be decomposed in the three matrices H_s, H_A and H_P

$$H = H_s H_A H_P = \begin{bmatrix} sR & t \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} I & 0 \\ v^T & v_3 \end{bmatrix} = \begin{bmatrix} sRK + t\nu^T & 0 \\ \nu^T & \nu \end{bmatrix} \tag{2.12}$$

here the H_s matrix is the similarity transformation, describing rotation, scaling and translation. The H_A matrix is the affine transformation and, K being the upper triangular with det $K = 1$. The las matrix H_P is the projective transformation,

2.2.6. Transformation of lines

Just as points between two image planes can be described by a homography, so can lines. Lines are transformed according to

$$\boldsymbol{\ell}' = \boldsymbol{H}^{-T}\boldsymbol{\ell} \tag{2.13}$$

like points, four lines are needed to calculate the homography.

2.2.7. Camera model

The perspective camera model is a mathematical model describing the correspondence between world coordinates and pixels in the image. The transformation between 3D world coordinates and the 2D image plane is found by using a coordinate frame to represent the camera. Normally the perspective camera model is divided into two parts, extrinsic and intrinsic, both commonly represented by a homogeneous matrix.
The extrinsic parameters describes the position and orientation of the camera in space. This camera pose is found through a transition from the world to the camera coordinate system, first by a translation followed by a rotation.
The intrinsic parameters describes how coordinates in the 2D image plane mat to its image coordinates in pixels. This mapping can be expressed by an upper triangular 3×3 camera calibration matrix as

$$\mathrm{K} = \begin{bmatrix} s_x & k & x_0 \\ 0 & s_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

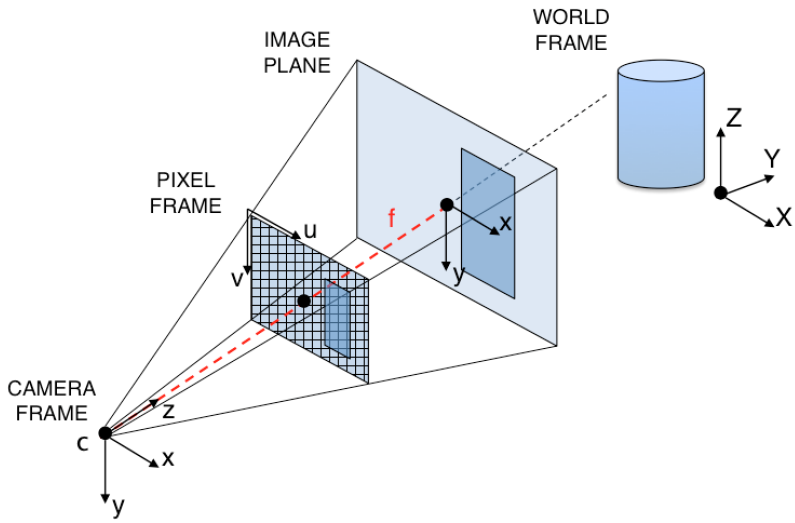


Figure 2.1.: The world coordinate frame (right), the camera coordinate system(left), where the transition between them are through a translation bmT and rotation $\boldsymbol{R}$

where s_x is the scale factor in the x direction while s_y is the scale factor in the y direction of an image. k is the skew factor, found by $k = s_y \tan\theta$, θ being the angle between the x and y image axis. (x_0, y_0) is the coordinate of the point in the image plane where the plane meets the camera Z-axis.
Putting the camera calibration matrix (the intrinsic parameters) together with the translation and rotation (the extrinsic parameters) gives the full perspective camera model. This model is typically presented as

$$\tilde{\boldsymbol{u}} = K[R \quad \boldsymbol{t}]^W \tilde{\boldsymbol{X}} \tag{2.15}$$

which describes the relation between the 3D point $\boldsymbol{X}$ and its projection $\boldsymbol{u}$ in the image. Expanded this will look like

$$\tilde{\boldsymbol{u}} = \begin{bmatrix} s_x & k & x_0 \\ 0 & s_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R_{3\times3} & \boldsymbol{t}_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix}^W \tilde{\boldsymbol{X}} \tag{2.16}$$

The Euclidean transformation of points from W to C is described by the extrinsic part. Further, the pose of the camera relative to the world frame can be

represented by a homogeneous transformation

$$
{}^{W}\zeta_C = \begin{bmatrix} {}^{W}R_C & {}^{W}\boldsymbol{t}_C \\ 0_{1\times 3} & 1 \end{bmatrix} \tag{2.17}
$$

$$
{}^{W}\tilde{X} = {}^{W}\zeta_C {}^{C}\tilde{X}
$$

The other way around, the Euclidean transformation from W to C is given as

$$
\begin{bmatrix} R_{3\times 3} & \boldsymbol{t}_{3\times 1} \\ 0_{0\times 3} & 1 \end{bmatrix} = {}^{W}\zeta_C^{-1} = \begin{bmatrix} {}^{W}R_C & {}^{W}\boldsymbol{t}_C \\ 0_{0\times 3} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} {}^{W}R_C^T & -{}^{W}R_C^T {}^{W}\boldsymbol{t}_C \\ 0_{0\times 3} & 1 \end{bmatrix} \tag{2.18}
$$

$$
{}^{C}\tilde{\boldsymbol{X}} = {}^{W}\zeta_C^{-1} {}^{W}\tilde{\boldsymbol{X}}
$$

2.2.8. Plücker coordinates

Plücker coordinates is a way to assign six homogeneous coordinates to each line in the projective 3D space. While a 3D line has 4 DOF, Plücker coordinates is a vector with 6 DOF which has the benefit being a convenient linear projection of 3D lines onto the image plane. Given a line joining two distinct 3D points in homogeneous coordinates, $\boldsymbol{X} = (X_1 X_2 X_3 X_4)^T$ and $\boldsymbol{Y} = (Y_1 Y_2 Y_3 Y_4)^T$, the projected line in 2D image plane is

$$
\lambda_1 \boldsymbol{x} = \boldsymbol{X}, \quad \text{and} \quad \lambda_2 \boldsymbol{y} = \boldsymbol{Y} \tag{2.19}
$$

where λ is a shared constant while x and y are the two corresponding endpoints in the 2D image plane, defined as $\boldsymbol{x} = (x_1 x_2 x_3)^T$ and $\boldsymbol{y} = (y_1 y_2 y_3)^T$. Given that

$$
\boldsymbol{x}^T \ell = 0 \quad \text{and} \quad \boldsymbol{y}^T \ell = 0 \tag{2.20}
$$

and the fact that λ is just a constant gives

$$
\boldsymbol{X}^T \ell = 0 \quad \text{and} \quad \boldsymbol{Y}^T \ell = 0 \tag{2.21}
$$

which also means that the line ℓ can be defined with the two 3D coordinates as

$$
\boldsymbol{\ell} = \boldsymbol{X} \times \boldsymbol{Y} \tag{2.22}
$$

The normal of the line can be defined as

$$
U = (\boldsymbol{X}, \boldsymbol{Y}) \times V \tag{2.23}
$$

where V is the directional vector of the line $\boldsymbol{L}$. Hence, from this, the 3D line can be represented in Plücker coordinates as $\boldsymbol{L} \approx (\boldsymbol{U}^T \boldsymbol{V}^T)^T = (L_1 L_2 L_3 L_4 L_5 L_6)^T$,

where

$$\boldsymbol{U}^T = (L_1 L_2 L_3) = (X_1 X_2 X_3) \times (Y_1 Y_2 Y_3) \tag{2.24}$$
$$\boldsymbol{V}^T = (L_4 L_5 L_6) = X_4(Y_1 Y_2 Y_3) - Y_4(X_1 X_2 X_3)$$

here $\boldsymbol{U}$ describes the position of the line in space, while $\boldsymbol{V}$ describes the direction of the line. Because $\boldsymbol{U}$ is a normal of the plane made up by the line $\boldsymbol{L}$ and the origin of the camera coordinate system, $\boldsymbol{L}$ must satisfy a bilinear constraint $\boldsymbol{U}^T\boldsymbol{V} = 0$.

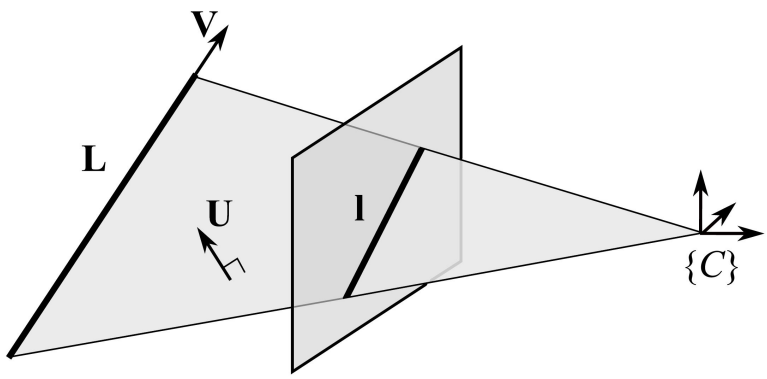


Figure 2.2.: 3D line projection. $\boldsymbol{L}$ is parameterized by its direction vector $\boldsymbol{V}$ and a normal $\boldsymbol{U}$ of its interpretation plane, and the projected 2D line l lies at the intersection of the interpretation plane and the image plane.

2.3. Line feature detection

Line detection is the process of finding line features in an image by taking a collection of n edge points and find all the lines which these points lie. The most common line detectors are the Hough transform and convolution based techniques.

2.3.1. Edge detection

One of the first steps in detecting features and identities in images are to find edges that are distinctive and can easily be restored and found in other images. Edges are defined as places in the image where there are strong signs of change in form of boundaries. These boundaries may come in the form of object shapes, shadows etc. Further edges may be grouped into curves or contours, straight line segments and piecewise linear contours. A challenge however is to determine how much change will be considered as an edge.

Sudden changes in pixel intensities in images is used to find these edges. By calculating the first derivative of the image intensity function, the extremas will illustrate the edges. The image gradient can be described as

$$\nabla \boldsymbol{f} = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] \tag{2.25}$$

which is a vector made up by the derivative in both x and y direction. This vector will hence have the magnitude

$$\| \nabla \boldsymbol{f} \| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 \left(\frac{\partial f}{\partial y}\right)^2} \tag{2.26}$$

and the gradient orientation can be estimated as

$$\theta = \arctan\left(\frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}}\right) \tag{2.27}$$

however, while this method reacts badly on image noise the derivative of Gaussian would give a smoother result

$$\frac{\partial}{\partial u} h_\sigma(u, v) \tag{2.28}$$

where

$$h_\sigma(u, v) = \frac{1}{2\pi\sigma^2} e^{-(\frac{u^2+v^2}{2\sigma^2})} \tag{2.29}$$

σ representing the width of the kernel, where a small σ meaning a finer feature is detected while a large σ on the other hand meaning only large scale edges have been detected.

2.3.2. Edge operators

A concern already mentioned is to determine how much change would be categorized as an edge. A way to remove spikes and noise as well as create an image to emphasise edges is to apply an edge operator. Depending on what kind of noise is represented in the image there are different edge operators suited for each case. These edge operators are also known as correlation filters using a weighted moving average, and can be sorted in either uniform or nonuniform filters. The general form of theses filters can be

$$G = H \otimes F \tag{2.30}$$

H being the kernel or mask that is used to change the intensity in the images and F represents each pixel in the image.

A uniform filter can be expressed as

$$G[i,j] = \frac{1}{(2k+1)^2} \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[i+u, j+v] \tag{2.31}$$

where the first fraction of the equation is a uniform weight for each pixel, while the last part is a loop for all pixels in the neighbourhood around image pixel $F[i,j]$. The window size would be $2k+1 \times 2k+1$. An example of a kernel in this filter could look like

$$H(u,v) = \frac{1}{9}\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline\end{array}$$

which would be an averaging filter that could be used to blur images, and $k = 1$. The non-uniform filter can be expressed as

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i+u, j+v] \tag{2.32}$$

An example of a non-uniform filter could be a Gaussian kernel as described in 2.29 could look like

$$H(u,v) = \frac{1}{16}\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline\end{array}$$

In an 1D signal from an image, applying a Gaussian kernel would have the effect that is illustrated in 2.3. The signal is first smoothed by adding it to the kernel and after finding the first derivative the peak represents the location of the edge.

Some of the most common operators for computing gradient are the Prewitt operator and the Sobel operator, where there are two kernels applied, both for x and y axis. The Prewitt operator is described as

$$G_x = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline\end{array}, \quad G_y = \begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline\end{array}$$

While the Sobel operator is described as

$$G_x = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline\end{array}, \quad G_y = \begin{array}{|c|c|c|}\hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline\end{array}$$

Using both Prewitt and Sobel can transform the images to binary images with

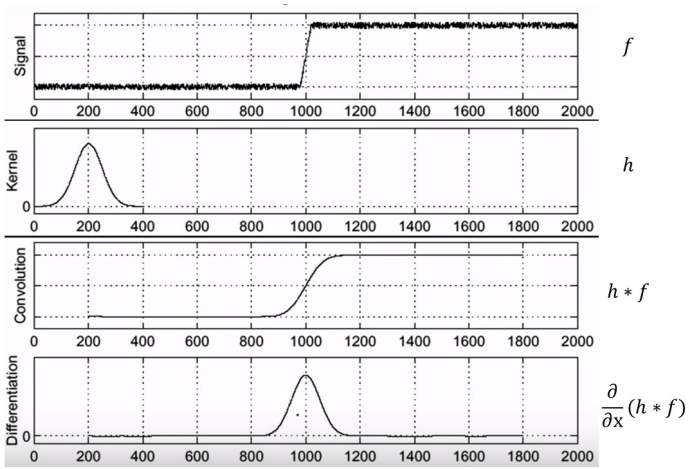


Figure 2.3.: A noisy signal f is smoothed by added the kernel h, and then by finding the first derivative the edge is found.

isolated edges. The gradient being $\nabla I = [G_x, G_y]^T$.

2.3.3. Canny Edge detector

Even though the Canny Edge detector came out in the late 80s, it is still the most widely used edge operator [referanse]. The Canny operator gives single pixel wide images with good continuation between adjacent pixels. The Canny edge algorithm can be composed in five steps:

1. Noise reduction
2. Gradient calculation
3. Non-maximum suppression
4. Double treshold
5. Edge tracking by hysteresis

The first step in the Canny Edge Detector is to smooth the image with Gaussian filter with spread θ. The next step is to find regions of significant gradient, using edge detection operators, to detect the edge intensity and direction. As an optimal solution the final image should have as thin edges as possible, which is conducted using Non-Maximum Suppression. In this step the algorithm goes through all points in the gradient intensity matrix and finds the pixel with the maximum value in edge directions. After thinning the double treshold step is aimed to sort each pixel into: strong, weak or non-relevant. If the pixel intensity value is so

that it is certain it contributes to the final edge, the pixel is sorted into strong. Weak pixels have an intensity value that is not enough to be considered as strong, but still not small enough to be discarded. The rest of the pixels are considered non-relevant for the edge. The final step is Edge Tracking by Hysteresis, which builds on the results from the double treshold step and sorts the weak pixels further. Looping over all weak pixels, they are transformed either into strong pixels or non-relevant. If one (or more) out of the nine pixels surrounding a weak pixel turns out to be strong, the weak pixel will be transformed to a strong pixel, otherwise set to non-relevant.

2.3.4. Line detection - Hough transform

The purpose of the Hough Transform for finding line segments is detecting groups of edge points by a voting procedure over a set of parameterized objects. This way imperfections in the image data, like missing points or spatial deviations between the ideal line and noisy edge points, have less impact compared to using an edge detector previously described.
Hough transform can be used to detect several shapes, however straight lines is the simplest case. A straight line is typically represented as $y = ax + b$, however because vertical lines will cause a problem lines are instead represented as

$$\rho = x \cos\theta + y \sin\theta \tag{2.33}$$

where ρ is the shortest distance from the origin to the straight line, while θ is the angle between the x-axis and the line connecting origin to the closest point. Therefore each line can be associated with (ρ, θ), often referred to as Hough Space. Given a single point in the plane, the set of all straight lines going through that point corresponds to a sinusoidal curve in the (ρ, θ) plane. Therefore a set of two or more points that form a straight line will produce sinusoids which will cross at (ρ, θ) for that line.
For each pixel at (x, y) and its neighborhood, the algorithm determines weather or not there is enough evidence of a straight line at that pixel. Further the line hypothesis' get their parameters (ρ, θ) calculated and sorts it into accumulator bins. Adding a new item to the bin then increments the value of that bin, and the bins with the highest values are most likely to be a straight line. These are observed through local maxima, combined with some sort of threshold, in the accumulator space.
As lines discovered do not come with information about the length, different techniques are required to find which parts of the image that match up with each lines.

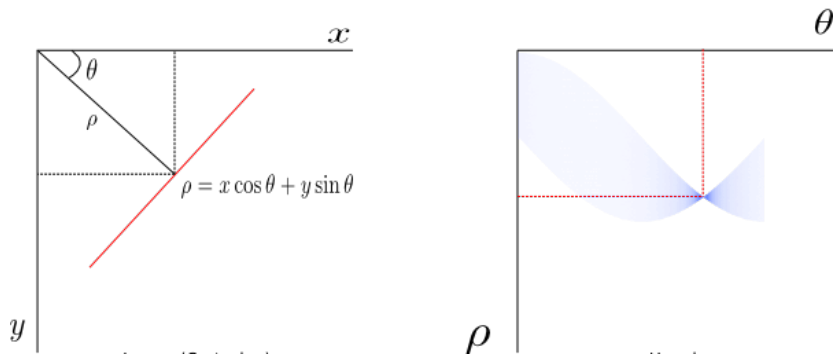


Figure 2.4.: The left illustrates the pixel parameters while the right side illustrates the accumulated space and detected line segment.

Chapter 3.

Theory

3.1. Line detection - Line Segment Detector (LSD)

The classic line segment detection methods usually starts with applying Canny [6] edge detector followed by a Hough transform [3] for extracting all lines contained by a number of edge points exceeding a treshold [9]. However, the Line Segment Detector method defines line segments as regions using only gradient information based on Burns' algorithm [5], combined with the validation criterion from Desolneux [1]. The algorithm is consisting of the following 4 steps: finding the line support regions, finding the rectangular approximation of every line-support regions, validation of each potential line segment and finally an improved approximation of line-support region and validation.

Step 1: in the first step, a line segment is defined as a region of a cluster of points in a connected region. To find the line-support region those pixels that share an approximately same gradient orientation angle as well as having a gradient magnitude greater than a treshold, ρ, is clustered. The pixels are grouped and then sorted into a finite number of groups based on their gradient. Each region starts with a single pixel and initializes the line-support region angle with the gradient angle of this pixel. If an adjacent pixel share the same characteristics and the following is satisfied

$$\text{abs}(\text{angle}(\overline{p} - \theta_{\text{region}})) < \tau \tag{3.1}$$

the pixel is added to the line-support region. Here $\overline{p}$ is the adjacent pixel, while τ is the treshold of the difference between adjacent pixel's angle and line-support region angle, where $\tau = 22.5°$. Line-support region angle is updated as it grows and expressed as

$$\theta_{\text{region}} = \arctan(\frac{\Sigma_i \sin(\text{angle}_i)}{\Sigma_i \cos(\text{angle}_i)}) \tag{3.2}$$

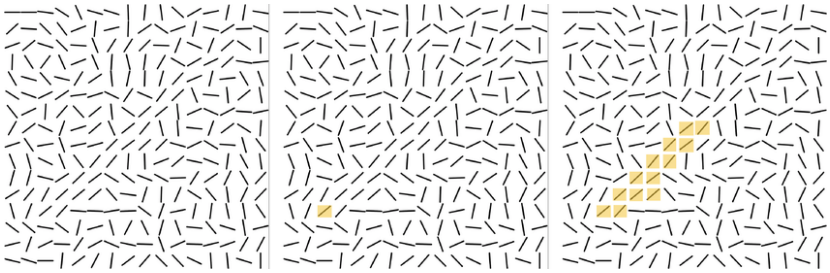


Figure 3.1.: The growing process of a region of aligned points.

Step 2: the second step is to define a line segment that is made up by the line-support region. This line-support region is characterized as an approximation of a rectangle, defined with parameters such as center, orientation angle, length and width.

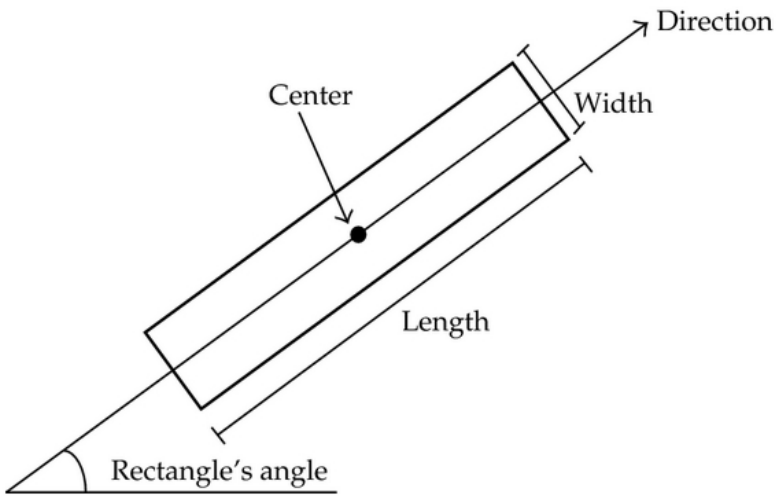


Figure 3.2.: Line segment characterized by a rectangle.

As shown in 3.2 the centroid of mass of the rectangular approximation is set as the center, when the gradient magnitude is used as pixel's mass.

Step 3: after the rectangular approximations of the line-support regions is found, each line segment approximation is validated using the number of aligned points and total number of pixels. Aligned points are defined as pixels with a gradient angle is the same as the line segment, within a tolerance of τ.

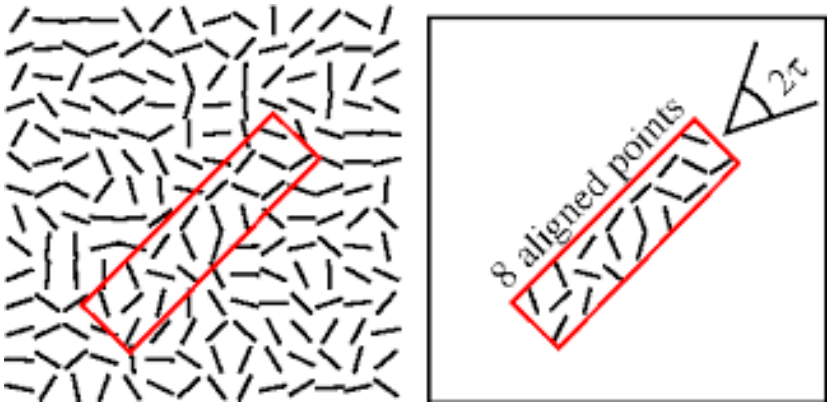


Figure 3.3.: The number of aligned points up to the angular tolerance is counted for each line segment. There are nine points among twenty in this case.

All potential line segments in the image are to be tested, and those that satisfy a treshold criterion based on length and number of aligned points k are kept as valid line segments. If there are r potential rectangles in the image x, a same amount of tests needs to be conducted to detect the final amount of lines. Each test relies on the statistics $k(r, x)$, where the H_0 is to be rejected if $k(r, x) \geq k_r$. Here H_0 is a Gaussian white noise model of the image background, that represents well isotropic zones, while straight edges are highly anisotropic zones. However, in practice a set of pixels will not be accepted as a line segment if it could have been formed by an isotropic process. Hence, k_r must be fixed to have a control of the expected number of false alarms under H_0, where the number of false alarms of a rectangle $r \epsilon R$ in an image x is

$$\text{NFA}(r, x) = \sharp R \cdot IP_{H_0}[k(r, X) \geq k(r, x)] \tag{3.3}$$

where X is a random image under model H_0, $\sharp R$ is the number of potential rectangles in image X and $IP_{H_0}\lceil k(r, X) \geq k(r, x)\rceil$ is the probability of $k(r, X)$ being greater or equal to $k(r, x)$. The smaller the NFA value is, the more significant the rectangle r is. Because each pixel's gradient is independent in rectangle, the number of aligned point $k(r)$ have a binomial distribution so that

$$IP_{h_0}[k(r, X) \geq k(r, x)] = b(n(r), k(r), p) \tag{3.4}$$

where $b(n, k, p) = \Sigma_{i=k}^{n}(n/i)p^i(1-p)^{n-i}$, which makes it possible to calculate NFA as

$$\text{NFA}(r) = N^5 \cdot b(n(r), k(r), p) \tag{3.5}$$

as there are N^4 potential line segments in a $N \times N$ image for the start and end point both have N^2 possible (with one pixel accuracy), hence $\sharp R = N^5$.

If NFA is less than a treshold ϵ, the rectangle is accepted as a plausible line segment.

Step 4: To get an even better NFA five dyadic precision steps are applied to adjust the width of the approximation and the probability p followed by another five dyadic steps.

3.2. Matching of lines

The methods for finding the pose estimation based on lines is based on finding the the corresponding 3D lines and 2D image lines. These methods are either based on geometric constraints, on their appearance, or a combination. Matching based on appearance, a descriptor is used to find a feature vector out of the line's appearance, for instance using its neighbour.

3.2.1. Fast Directional Chamfer Matching

Micusik and Wildenauer [16] has developed a descriptor free visual indoor localization using line segments. The approach is searching through the parameter space of camera poses to generate tens of thousands of virtual camera views, and then compared with the real line segments by Fast Directional Chamfer Matching [14].
Using Chamfer Distance [4] will get rid of the problem where the complexity of aligning one set of line segments with a second set (given a pair of images containing line segments) is quadratic, as each line in the first set needs to be compared to all of lines in the second. This method measures the discrepancy of two contours, defined as the cost of aligning two edge maps $\epsilon = \{x_{ei}\}_{i=1}^{N_\epsilon}$ and $\tau = \{x_{ti}\}_{i=1}^{N_\tau}$ as

$$d(\epsilon, \tau) = \frac{1}{N_\tau} \Sigma_{x_t \in \tau} \min_{x_t \in \epsilon} \parallel x_e - x_t \parallel \tag{3.6}$$

However, the min function can be replaced by a look-up in the distance transform (DT) image $I_{DT\epsilon}$ of the edge map ϵ so that

$$d(\epsilon, \tau) = \frac{1}{N_\tau} \Sigma_{x_t \in \tau} I_{DT\epsilon}(x_t, \gamma) \tag{3.7}$$

Where γ is a threshold for truncating the DT values, to achieve robustness against noise in edgels.
The edge map and the template is divided into discrete orientation channels θ_i, and sum the individual Chamfer scores. Then line detection algorithms, such as Line Segment Detector, are used to split Edgels in ϵ into piece-wise linear segments. A well-used segmentation is using 16 discrete orientation channels,

$0, 14, 26.6 \ldots 166°$, and assign each edgel the discrete orientation closest to the orientation of the fitted line segment. This means sixteen DT images $I^{\theta_i}_{DT\epsilon}$ are computed on filtered binary images, which is an image composed of edgels from edge map ϵ assigned with discrete orientation θ.
Given the 3D model consisting of line segments, the space of possible locations and orientations of the query image is sampled. At each sampled location line segments are projected into virtual views. In the template map τ, which corresponds to a virtual view, the projected lines of the 3D model are splitted in smaller ones and snapped into the discrete orientations. Then the template map becomes a map of line segments expressed as

$$\tau^l = \{\boldsymbol{l}\}_{i=1}^{N_{\tau^l}} \tag{3.8}$$

where $\boldsymbol{l}_i = [x_i^s, x_i^e]$ is a four element vector. The distance between the two edge maps is then

$$d(\epsilon, \tau) = \frac{1}{N_\tau^l} \Sigma_{x_t \in \tau^l} d(\epsilon, l) \tag{3.9}$$

Because the sum operation in 3.7 is expensive, [14] suggests to use integral contours. Integral distance transform (IDT) images $I^{\theta_i}_{DTE\epsilon}$ are made by summing pixels along a respective scan line. To compute it off-line, it needs one pass though the query image. As a result, instead of summing along hundreds of pixels of the edgels, only the endpoints of the line segments are used in $I^{\theta_i}_{DTE\epsilon}$ and then divided by its length so that

$$d(\epsilon, l) = \frac{I^{\theta(l)}_{IDT\epsilon}(x^e, \gamma) - I^{\theta(l)}_{IDT\epsilon}(x^s, \gamma))}{l_{leng}} \tag{3.10}$$

where the function $\theta(\boldsymbol{l})$ assigns a discrete orient to a line segment $\boldsymbol{l}$ as its closest snapping orientation. The length of the line is found by $\| x^e - x^s \|$.
Finally the template τ is found searching through the set of templates τ_{set} which minimizes the distance d

$$\tau* = \arg \min_{\tau \in \tau_{set}} d(\epsilon, \tau) \tag{3.11}$$

however [16] suggests to rather find the best template by replacing shortest distance with a cost function as

$$\tau* = \arg \max_{\tau \in \tau_{set}} c(\epsilon, \tau) \tag{3.12}$$

where

$$c(\epsilon, \tau) = \frac{1}{N_{\tau^l}} \sum_{l \in \tau^l} \delta(d(\epsilon, l) < \gamma) \tag{3.13}$$

with N_{τ^l} being the number of line segments in the template map τ^l, γ is the treshold and the summed expression being a binary function returning 0 if expression

is equal, and 1 otherwise. In this way, the algorithm selects the template map with the highest number of matched lines.
Applying a search optimization will allow the algorithm to reach the maximum without going through all elements in the sum operation. All template maps are processed at the same time switching between, and one line is evaluated at the time, the upper bound of the cost will then become

$$\overline{c}(\epsilon, \tau_{1:i}) = \frac{N_{\tau^l} - i + i * c(\epsilon, \tau_1 : i)}{N_{\tau^l}} \tag{3.14}$$

where $\tau_{1:i}$ represents the template map with the ith first line segments.
The algorithm in its full is then

Algorithm 3 Search for the K best alignments to a query

```
procedure LineSegmentDetection(ϵ)
    Detect line segments in query, from ϵ
    Compute integral contour images I^θ_{IDTϵ} at 16 orientations of θ
    Set upper bounds c̄(ϵ, τ) ← 1 for all virtual views τ ∈ τ_set
    set counters of processed lines i_τ ← 1 for all τ
    set counters of best matches k ← 0
    push c̄(ϵ, τ) of all τ into the priority queue
    repeat until k = K
        take τ from the priority queue with the highest c̄(ϵ, τ)
        evaluate i_τth line by scoring c̄(ϵ, τ_{1_{iτ}})
        i_τ ← i_τ + 1
        ifi_τ = N_{τ^l}
            k ← k + 1
        else
            push c̄(ϵ, τ_{1_{iτ}}) into the priority queue
    return(K)
```

3.3. Structure from motion

Structure from motion is the problem of simultaneously estimate the structure of the environment as well as motion from visual input of a moving camera. Typically points are used as they are easily detected and matched in images of natural scenes, however especially when considering man-made objects, line segments contain far more information about a scene.
There are several ways to solve this problem, however Micusik and Wildenauer [15] suggest this method that consists of three stages:

1. **Initial 3D model** is this stage the main goal is to form an initial 3D structure building a base to which follow-up frames have to be incrementally added. When building this model, both estimating relative camera motion and triangulation corresponding features from the images needs to be estimated simultaneously.

   When working with lines, a minimum of three views are necessary for making an initial 3D model. The relative pose is estimated in two stages by decoupling rotation and translation. This way, only five correspondences are necessary to find the translation with a linear solver, instead of 13 otherwise. At the same time the line segment matching algorithm has to establish a sufficient number of initial 2D-2D correspondences. From this available information the translation and 3D model can be made by i.e. a RANSAC procedure[7].

2. **Pose in 3D model** The absolute pose of each new input image are calculated using the 3D model built so far. A matching algorithm is used to establish enough correspondences between line segments in the query and the 3D model.

3. **Update 3D model** As the initial 3D model is somewhat incomplete and inaccurate, the 3D model is continuously updated. Using the pose estimation from the previous step and then triangulate the detected line segments.

3.4. PnP - Perspective-n-Point

The Perspective-n-point problem was first introduced in 1981 by Fischler and Bolles [8] for estimating the camera pose. This estimation is made by finding point correspondences between known real features and their image plane counterparts. Going from n 3D reference points to their 2D projections at least 3 points are needed, hence there are 6 degrees of freedom in the form of rotation and translation, which follows the perspective project model for cameras:

$$sp_c = K[R\,|\,T]p_w \tag{3.15}$$

where $p_c = [u,\, v,\, 1]^T$ is the homogeneous image point, while $p_w = [x,\, y,\, z,\, 1]^T$ is the corresponding homogeneous world point. K is the matrix of intrinsic camera parameters, s is a scale factor for the image point and R and T are 3D rotation and translation matrices of the camera.

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \gamma & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.16}$$

In the K matrix, f_x and f_y are scaled focal lengths, γ is the skew parameter, while (u_0, v_0) is the principal point.
A solution for the PnP problem can be classified in two ways, either as an iterative or a noniterative method[20]. The noniterative methods are more efficient, but have less accuracy when affected by noise (especially when number of features are low) compared to the iterative method.
EPnP, DLT and P3P are some of the most common solutions for this problem which will be further described.

3.4.1. P3P

The minimal form of the PnP problem is when $n = 3$, solved with three correspondences[11] and is usually used in combination with RANSAC to remove outliers [24].
To find the camera projection center point C, given three observed feature points P_1, P_2 and P_3, assuming the vectors pointing toward these points $\vec{f_1}, \vec{f_2}$ and $\vec{f_3}$ also are known. The new camera frame is defined as $\tau = (C, \vec{t_x}, \vec{t_y}, \vec{t_z})$ where

$$\vec{t_x} = \vec{f_1}$$

$$\vec{t_z} = \frac{\vec{f_1} \times \vec{f_2}}{\| \vec{f_1} \times \vec{f_2} \|}$$

$$\vec{t_y} = \vec{t_z} \times \vec{t_x}$$

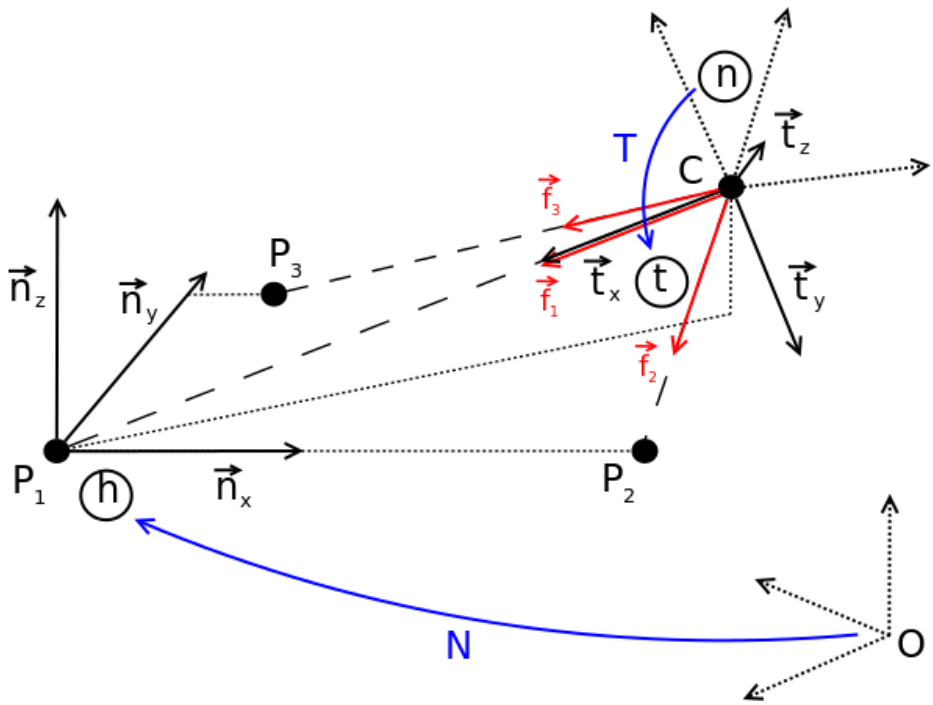


Figure 3.4.: Illustraion of the camera frame τ and the world frame η.

Through the transformation matrix $T = [\vec{t_x}, \vec{t_y}\vec{t_z}]^T$, feature vectors can be transformed to τ by

$$\vec{f_i}^T = T \cdot \vec{f_i} \tag{3.17}$$

The new world frame η from the world points P_1, P_2 and P_3. The new spatial frame is defined as $\eta = (P_1, \vec{n}_x, \vec{n}_y, \vec{n}_z)$ where

$$\vec{n_x} = \frac{\vec{P_1P_2}}{\parallel \vec{P_1P_2} \parallel}$$

$$\vec{n_z} = \frac{\vec{n_x} \times \vec{P_1P_3}}{\parallel \vec{n_x} \times \vec{P_1P_3} \parallel}$$

$$\vec{n_y} = \vec{n_z} \times \vec{n_x}$$

Using the transformation matrix $N = [\vec{n_x}, \vec{n_y}, \vec{n_z}]^T$ the world points can be transformed to μ using

$$P_i^\mu = N(P_i - P_1) \tag{3.18}$$

The points P_1 and P_2 combined with the camera center C form a triangle where two parameters are known, the distance d_{12} between P_1 and P_2. The second parameter is the angle β between $\vec{f_1}$ and $\vec{f_2}$ which can be obtained by the dot-

product $cos\beta = \vec{f_1} \dots \vec{f_2}$. Further the sine-law can give a way to obtain d_{12}

$$\frac{\parallel \vec{CP_1} \parallel}{d_{12}} = \frac{\sin(\pi - \alpha - \beta)}{\sin \beta} \tag{3.19}$$

A plane π containing P_1, P_2 and C, hence also $\vec{n_x}, \vec{t_x}, \vec{t_y}, \vec{f_1}$ and $\vec{f_2}$, can be defined where the position of the camera center C inside the plane is

$$C^{\pi}(\alpha)(\alpha) = \begin{pmatrix} \cos\alpha \cdot \parallel \vec{CP_1} \parallel \\ \cos\alpha \cdot \parallel \vec{CP_1} \parallel \\ 0 \end{pmatrix} = \begin{pmatrix} d_{12}\cos\alpha\sin(\alpha+\beta)\sin^{-1}\beta \\ d_{12}\sin\alpha\sin(\alpha+\beta)\sin^{-1}\beta \\ 0 \end{pmatrix} \tag{3.20}$$

where α is the angle between $\vec{n_x}$ and $\vec{t_x}$. For $C, \vec{t_x}, \vec{t_y}$ and t_z to be inside μ, a new parameter is introduced, the rotation θ of π around $\vec{n_x}$. This rotation matrix is expressed by

$$R_\theta = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \tag{3.21}$$

the camera center C is then defined as

$$C^{\eta}(\alpha,\theta) = R_\theta C^{\pi} \begin{pmatrix} D_{12}\cos\alpha(\sin\alpha \cdot \cot\beta + \cos\alpha) \\ D_{12}\sin\alpha\cos\theta(\sin\alpha \cdot \cot\beta + \cos\alpha) \\ D_{12}\sin\alpha\sin\theta(\sin\alpha \cdot \cot\beta + \cos\alpha) \end{pmatrix} \tag{3.22}$$

the tranformation matrix from μ to τ is given by

$$Q(\alpha,\theta) = [R_\theta(\vec{t_x^{\pi}}, \vec{t_y^{\pi}}, \vec{t_z^{\pi}})]^T \begin{pmatrix} -\cos\alpha & -\sin\alpha\cos\theta & -\sin\alpha\sin\theta \\ \sin\alpha & -\cos\alpha\cos\theta & -\cos\alpha\sin\theta \\ 0 & -\sin\alpha & \cos\theta \end{pmatrix} \tag{3.23}$$

The values α and θ can be found by transforming the third point P_3^{η} into τ, setting the direction of the point equal to one of $\vec{f_3^T}$.

$$\begin{aligned} P_3^{\tau} &= Q(\alpha,\theta)(P_3^{\eta} - C^{\eta}(\alpha,\theta)) \\ &= \begin{pmatrix} -\cos\alpha \cdot p_1 - \sin\alpha\cos\theta \cdot p_2 + d_{12}(\sin\alpha \cdot \cot\beta \ \cos\alpha) \\ \sin\alpha \cdot p_1 - \cos\alpha\cos\theta \cdot p_2 \\ -\sin\theta \cdot p_2 \end{pmatrix} \end{aligned} \tag{3.24}$$

$$\phi_1 = \frac{f_{3,x}^{\tau}}{f_{3,z}^{\tau}}, \phi_2 = \frac{f_{3,y}^{\tau}}{f_{3,z}^{\tau}} \tag{3.25}$$

these two conditions combined with 3.24 will finally result in

$$\cot\alpha = \frac{\frac{\phi_1}{\phi_2}p_1 + \cos\theta \cdot p_2 - d_{12} \cdot \cot\beta}{\frac{\phi_1}{\phi_2}\cos\theta \cdot p_2 - p_1 + d_{12}} \tag{3.26}$$

expanding ϕ_2 gives

$$\sin^2\theta \cdot f_2^2 p_2^2 = \sin^2\alpha(p_1 - \cot\alpha\cos\theta \cdot p_2)^2$$

$$(1-\cos^2\theta)(1+\cot^2\alpha)f_2^2p_2^2 = p_1^2 2\cot\alpha\cos\theta \cdot p_1p_2 + \cot^2\alpha\cos^2\theta \cdot p_2^2 \tag{3.27}$$

Combining 3.26 and 3.27, expanding eventually leads to a fourth order polynomial in the form of

$$a_2\cos^4\theta + a_3\cos^3 + a_2\cos^2 + a_1\cos^1 + a_0 = 0 \tag{3.28}$$

where

$$\begin{aligned}
a_4 &= -\phi_2^2p_2^4 - \phi_1^2p_2^4 - p_2^4 \\
a_3 &= 2p_2^3d_{12}\cot\beta + 2\phi_2^2p_2^3d_{12}\cot\beta - 2\phi_1\phi_2p_2^3d_{12} \\
a_2 &= -\phi_2^2p_1^2p_2^2 - \phi_2^2p_2^2d_{12}^2\cot^2\beta - \phi_2^2p_2^2d_{12}^2 + \phi_2^2p_2^4 + \phi_1^2p_2^4 + 2p_1p_2^2d_{12} \\
&\quad + 2\phi_1\phi_2p_1p_2^2d_{12}\cot\beta - \phi_1^2p_1^2 + 2\phi_2^2p_1p_2^2d_{12} - p_2^2d_{12}^2\cot^2\beta - 2p_1^2p_2^2 \\
a_1 &= 2p_1^2p_2d_{12}\cot\beta + 2\phi_1\phi_2p_2^3d_{12} - 2\phi_2^2p_2^3d_{12}\cot\beta - 2p_1p_2d_{12}^2\cot\beta \\
a_0 &= -2\phi_1\phi_2p_1p_2^2d_{12}\cot\beta + \phi_2^2p_2^2d_{12}^2 + 2p_1^3d_{12} - p_1^2d_{12}^2 + \phi_2^2p_1^2p_2^2 - p_1^4 \\
&\quad - 2\phi_2^2p_1p_2^2d_{12} + \phi_1^2p_1^2p_2^2 + d_{12}^2\cot^2\beta
\end{aligned}$$

Solving this fourth order polynomial give four real solutions for $\cos\theta$. Each value for θ will also lead to one value for α, which makes it possible to solve the camera center position and rotation

$$C = P_1 + N^TC^\eta, \quad R = N^TQ^TT \tag{3.29}$$

As the P3P provides many solutions with just three points, a fourth point is often used in practice which gives a Å4P problem [12].

3.4.2. Direct linear transformation

A direct linear transformation is an algorithm to solve for a homography from a set of point mappings. There are several algorithms for DLT, all have in common that four point mappings are needed to find the eight independent entries. The transformation in DLT is given by the equation

$$x_i' = Hx_i \tag{3.30}$$

where the transformation matrix H is to be found. If the jth row of H is denoted by h_j^T 3.30 can be rewritten to

$$\lambda x_i' = Hx_i = \begin{bmatrix} h_1^T x_i \\ h_2^T x_i \\ h_3^T x_i \end{bmatrix} \tag{3.31}$$

writing $x_i' = (x_i', y_i', w_i')^T$ the cross product will be given as

$$x_i' \times Hx_i = \begin{pmatrix} y_i' h_3^T x_i - w_i' h_2^T x_i \\ w_i' h_1^T x_i - x_i' h_3^T x_i \\ w_i' h_2^T x_i - y_i' h_1^T x_i \end{pmatrix} \tag{3.32}$$

As $h_j^T x_i = x_i^T h_j$, this gives a set of three equations for solving H, which can be expressed as

$$\begin{bmatrix} 0^T & -w_i' x_i^T & y_i' x_i^T \\ w_i' x_i^T & 0^T & -x_i' x_i^T \\ -y_i' x_i^T & x_i' x_i^T & 0^T \end{bmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = 0 \tag{3.33}$$

finally giving the equation to be solved as

$$x_i' \times Hx_i = 0 \tag{3.34}$$

to give a simple linear solution for H to be derived. Further to find all elements in matrix H is rewritten as

$$A_i h = 0 \tag{3.35}$$

where h is a 9×1 vector consisting all elements in H, all of whom unknown, and A_i being a 3×9 matrix consisting a function of measurements. The equation gives two independent equations to be solved as [10]

$$\begin{bmatrix} A_1 h \\ A_2 h \\ \vdots \\ A_N h \end{bmatrix} = Ah = 0 \tag{3.36}$$

which gives a homogeneous system. This system can be solved by applying Singular Value Decomposition which is in the form of

$$P_{m \times n} = U_{2N \times 9} \Sigma_{9 \times 9} V_{9 \times 9}^T \tag{3.37}$$

both $U_{m \times n}$ and $V_{n \times n}^T$ are orthogonal matrices, and the last column of V will give h.

3.4.3. EPnP

Efficient PnP is a non-iterative solution to the PnP problem developed by Lepetit [13]. The solution is expressed as a vector that lies in the kernel of a matrix, denoted as M, either of size $2n \times 12$ or $2n \times 9$. In this case n reference points in the camera coordinate system is expressed as a weighted sum of virtual control points. The four control points used to express world coordinates will be $c_j,\ j = 1, ..., 4$ while the known 3D coordinates known in the world coordinate system will be $p_i,\ i = 1, ..., n$. For describing which coordinate system the point coordinates are expressed, p_i^w are used for the world coordinate system, while p_i^c is used for the camera coordinate system. Each reference point is then expressed as a weighted sum of the control points

$$p_i = \sum_{j=1}^{4} a_{ij} c_j, \quad \sum_{j=1}^{4} a_{ij} = 1 \tag{3.38}$$

To derive the matrix M first step is to let K be the camera internal calibration matrix and expand

$$w_i \begin{bmatrix} u_i \\ 1 \end{bmatrix} = K p_i^c = K \sum_{j=1}^{4} a_{ij} c_j^c \tag{3.39}$$

where $u_i = 1, ..., n$ are the 2D projections of the $p_i = 1, ..., n$ reference points, while w_i are scalar projective parameters. Expanding the expression for specific 3D coordinates $[x_j^c, y_j^c, z_j^c]$ of each c_j^c control point, 2D coordinates $[u_i, v_i]^T$ of the u_i projections, as well as focal lengths coefficients f_u, f_v and the (u_c, v_c) principal point gives

$$w_i \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} f_u & 0 & u_c \\ 0 & f_v & v_c \\ 0 & 0 & 1 \end{bmatrix} \sum_{j=1}^{4} a_{ij} \begin{bmatrix} x_j^c \\ y_j^c \\ z_j^c \end{bmatrix} \tag{3.40}$$

There are $12 + n$ unknown parameters of this linear system, which are the 12 control point coordinates $(x_j^c, y_j^c, z_j^c)_{i=j,...,4}$ and the n projective parameters $w_{i i=1,...,n}$. In the last row it is implied that $w_i = \Sigma_{j=1}^{4} a_{ij} z_j^c$, which gives two linear equations from the first two rows for each reference point

$$\sum_{j=1}^{4} a_{ij} f_u x_j^c + a_{ij}(u_c - u_i) z_j^c = 0 \tag{3.41}$$

and

$$\sum_{j=1}^{4} a_{ij} f_v y_j^c + a_{ij}(v_c - v_i) z_j^c = 0 \tag{3.42}$$

linking these equations for all n reference points can give a linear system in the form of

$$Mx = 0 \tag{3.43}$$

where M is a $2n \times 12$ matrix made up by 3.41 and 3.42 for each reference point. On the other side, x is a vector with the size of 12, expressed as $x = [c_1^{cT}, c_2^{cT}, c_3^{cT}, c_4^{cT}]^T$ and are made up by the unknowns. Further, x can be expressed as

$$x = \sum_{i=1}^{N} \beta_i v_i \tag{3.44}$$

here v_i represent the columns of the right-singular vectors of M and N being the number of null singular values in M.

3.5. SLAM

Simultaneous Localization and Mapping (SLAM) is the process of updating a map of an unknown environment while keeping track of the position of the robot.
When applying SLAM, there are two givens, the robot's controls or path and relative observations and landmarks. These can represented as:
The robot controls:

$$U_{1:t} = \{u_1, u_2, ..., u_t\}$$

Already observed landmarks:

$$Z_{1:t} = \{z_1, z_2, ..., z_t\}$$

On the other hand, there are two unknowns that are wanted, the actual path of the robot, and the map of all landmarks. These can be represented as:
Path of the robot:

$$X_{1:t} = \{x_1, x_2, ..., x_t\}$$

Map of features:

$$m = \{m_1, m_2, ..., z_k\}$$

As shown in figure 3.5 this problem can be illustrated graphical where the top layer illustrates the robot's path, next layer control inputs, third layer observed landmarks and final layer a map of the observed landscape. However, putting these four factors together gives the following equation for the SLAM problem:

$$p(x_{0:t}, m | z_{1:t}, u_{1:t}) \tag{3.45}$$

There are three main paradigms within solving the SLAM problem. The first is using an Extended Kalman filter, the second is using a particle filter and the

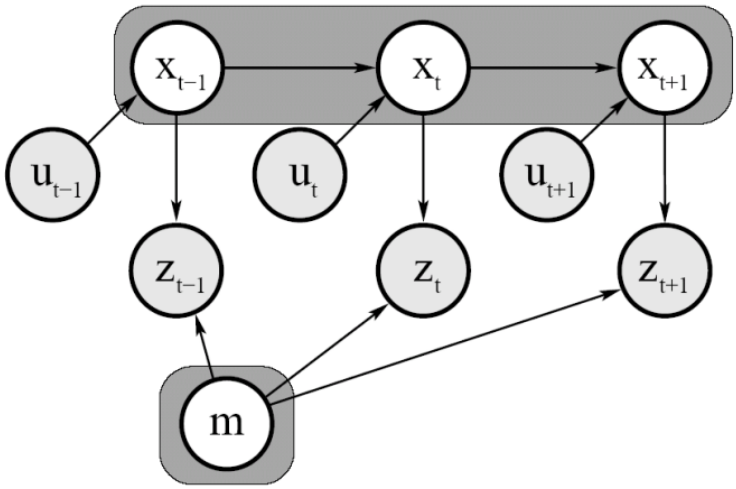


Figure 3.5.: Illustration of the SLAM problem

third is Graph-based [2]. All three all have their different use cases and different setbacks. However, in this thesis, only EKF SLAM is described further.

3.5.1. EKF SLAM

SLAM based on the Extended Kalman Filter is one of the first SLAM methods introduced [21] and also one the more simple ones. The method uses a linearized Gaussian probability distribution model and is based on online SLAM using maximum likelihood data association. A limitations using this method is because of computational difficulties the number of landmarks are quite low (<1000), hence the most ideal map is feature based. This is a result of the complexity of the algorithm as it exponentially grows with the number of landmarks. As the EKF algorithm also makes a Gaussian noise assumption for robot motion and perception, the amount of uncertainty in the posterior must be relative small, otherwise the linearization results in intolerable errors and the algorithm fails. When using EKF SLAM with maximum likelihood it uses EKF framework to find a solution to:

$$p(x_t, m|z_{1:t}, u_{1:t}) = \eta \underbrace{p(z_t|p_t, m)}_{\text{Observation model}} \int \underbrace{p(p_t|p_{t-1}, u_t)}_{\text{Motion model}} \underbrace{p(p_{t-1}|z_{1:t-1}, u_{1:t-1})}_{\text{Prior distribution}} dp_{t-1} \tag{3.46}$$

To represent the localization of the robot there is a 3x1 pose vector and a 3x3

covariance matrix.

$$x_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix}, C_k = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{y\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_\theta^2 \end{bmatrix}$$

This can be expanded to include every observed landmarks, for every timestep k, growing both the state vector and the covariance matrix.

$$x_k = \begin{bmatrix} x_R \\ m_1 \\ m_2 \\ \vdots \\ m_n \end{bmatrix}_k, C_k = \begin{bmatrix} C_R & C_{RM_1} & C_{RM_2} & \dots & C_{RM_n} \\ C_{M_1R} & C_{M_1} & C_{M_1M_2} & \dots & C_{M_1M_n} \\ C_{M_2R} & C_{M_2M_1} & C_{M_2} & \dots & C_{M_2M_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{M_nR} & C_{M_nM_1} & C_{M_nM_2} & \dots & C_{M_n} \end{bmatrix}_k$$

Using these representations can be put together making a state representation of both the robots pose and every observed landmark. Using Gaussian distribution, this will become a (3+2N)-dimensional matrix that can handle hundreds of dimensions.

$$\mathrm{Bel}(x_t, m_t) = \left(\underbrace{\begin{bmatrix} x \\ y \\ \theta \\ l_1 \\ l_2 \\ \vdots \\ l_n \end{bmatrix}}_{\mu}, \underbrace{\begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xl_1} & \sigma_{xl_2} & \dots & \sigma_{xl_n} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\theta} & \sigma_{yl_1} & \sigma_{yl_2} & \dots & \sigma_{yl_n} \\ \sigma_{x\theta} & \sigma_{y\theta} & \sigma_{theta}^2 & \sigma_{\theta l_1} & \sigma_{\theta l_2} & \dots & \sigma_{\theta l_n} \\ \sigma_{xl_1} & \sigma_{yl_1} & \sigma_{\theta l_1} & \sigma_{l_1}^2 & \sigma_{l_1l_2} & \dots & \sigma_{l_1l_n} \\ \sigma_{xl_2} & \sigma_{yl_2} & \sigma_{\theta l_2} & \sigma_{l_1l_2}^2 & \sigma_{l_2}^2 & \dots & \sigma_{l_2l_n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \sigma_{xl_n} & \sigma_{yl_n} & \sigma_{\theta l_n} & \sigma_{l_1l_n}^2 & \sigma_{l_2l_n} & \dots & \sigma_{l_n}^2 \end{bmatrix}}_{\Sigma} \right) \tag{3.47}$$

Next, when having the state representations and a matrix to store the observed landmarks, it is time to set up the EKF Algorithm. As described in chapter 2.1.1 on a higher level, the Kalman Filter consists of two steps, the prediction step, and then the correction step. However, to fill these matrices demands a lot of calculations as will be described.
The first step in the Prediction step, $\overline{\mu}_t = g(u_t, u_{t-1})$ will be calculated as in 3.48.

Algorithm 4 Extended Kalman Filter Algorithm

procedure EXTENDEDKALMANFILTER($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)
Prediction step:
$\overline{\mu}_t = g(u_t, u_{t-1})$
$\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$

Correction step:
$K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + Q_t)^{-1}$
$\mu_t = \overline{\mu}_t + K_t(z_t - h(\overline{\mu}_t))$
$\Sigma_t = (I - K_t H_t)\overline{\Sigma}_t$
return μ_t, Σ_t

$$\begin{pmatrix} x' \\ y' \\ \theta' \\ \vdots \end{pmatrix} = \underbrace{\begin{pmatrix} x \\ y \\ \theta \\ \vdots \end{pmatrix} + \underbrace{\begin{pmatrix} 1 & 0 & 0 & \underbrace{0 \cdots 0}_{2N \text{ cols}} \\ 0 & 1 & 0 & 0 \cdots 0 \\ 0 & 0 & 1 & 0 \cdots 0 \end{pmatrix}^T}_{F_x^T} \begin{pmatrix} -\frac{v_t}{\omega_t}\sin\theta - \frac{v_t}{\omega_t}\sin(\theta + \omega_t \Delta t) \\ \frac{v_t}{\omega_t}\cos\theta - \frac{v_t}{\omega_t}\cos(\theta + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix}}_{g(u_t, x_t)} \tag{3.48}$$

The second and last step in the Prediction step is a little more complex. However, $\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$ will be calculated as in 3.50. G_t^x is found by calculating the Jacobian of the motion model, but for simplicity, only the result is shown in refunder.

$$G_t^x = \begin{pmatrix} 1 & 0 & -\frac{v_t}{\omega_t}\cos\theta + \frac{v_t}{\omega_t}\cos(\theta + \omega_t \Delta t) \\ 0 & 1 & -\frac{v_t}{\omega_t}\sin\theta + \frac{v_t}{\omega_t}\sin(\theta + \omega_t \Delta t) \\ 0 & 0 & 1 \end{pmatrix} \tag{3.49}$$

$$\overline{\Sigma}_t = \underbrace{\begin{pmatrix} G_t^x & 0 \\ 0 & I \end{pmatrix}}_{G_t} \underbrace{\begin{pmatrix} \Sigma_{xx} & \Sigma_{xl} \\ \Sigma_{lx} & \Sigma_{ll} \end{pmatrix}}_{\Sigma_{t-1}} \underbrace{\begin{pmatrix} (G_t^x)^T & 0 \\ 0 & I \end{pmatrix}}_{G_t^T} + \underbrace{F_x^T R_t^x F_x}_{R_t} \tag{3.50}$$

For the second step of the algorithm, the correction step, what is necessary to finish is the K_t. To solve this the Jacobian is needed. First the low dimensional Jacobian is calculated, further it is mapped to the high dimensional space.

$$^{\text{low}}H_t^i = \frac{1}{q}\begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \sqrt{q}\delta_x & \sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x \end{pmatrix} \tag{3.51}$$

$$H_t^i = {}^{\text{low}}H_t^i F_{x,j}, \quad \text{where } F_x, j = \begin{pmatrix} 1 & 0 & 0 & 0\cdots0 & 0 & 0 & 0\cdots0 \\ 0 & 1 & 0 & 0\cdots0 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 1 & 0\cdots0 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 0 & 0\cdots1 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 0 & \underbrace{0\cdots0}_{2j-2} & 1 & 0 & \underbrace{0\cdots0}_{2N-2j} \end{pmatrix} \tag{3.52}$$

This step happens when a new landmark with index j is observed at measurement i at time t. The expected observation is computed using the Jacobian of h according to the current estimates:

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \overline{\mu}_{j,x} - \overline{\mu}_{t,x} \\ \overline{\mu}_{j,y} - \overline{\mu}_{t,y} \end{pmatrix}, q = \delta^T\delta, \hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ \text{atan2}(\delta_y, \delta_x) - \overline{\mu}_{t,\theta} \end{pmatrix} = h(\overline{\mu}_t)$$

Chapter 4.

Line pose estimation

This chapter contains the work that has been conducted through the master thesis. First, the latest development within pose estimation and computer vision has been discussed and analyzed. The main focus with this thesis has been to look at methods for finding the camera pose estimation based on detected line segments, hence this will be the majority of this chapter. Another part of this chapter will consist of methods and algorithms that would be necessary to make an independent top of the state SLAM system. This includes among other things feature detection, matching of features detected and structure from motion.

4.1. Perspective-n-Lines

Camera pose estimation based on 3D/2D correspondences is a fundamental task required for various applications within robotics and computer vision, such as localization and navigation of a robot, augmented reality or operation of robots based on visual information. While PnP is a well studied topic, PnL is lagging a little behind because of its mathematical complexity of handling lines. However, lines are more robust as they can still be viewed be used even if partially occluded [18].
DLT-Plücker-Lines are the PnL method that has been in focus in this thesis and described in depth in 4.2. Some other state of the art PnL methods are ASPnL, Mirzaei and RPnL.
RPnL [23] works with four lines or more for retrieved the optimal solution from a sixteenth order cost. The method uses an intermediate model coordinate system which is aligned with a 3D line of the longest projection. These lines are divided into triples, and for each a P3L (the same method as explained in 3.4.1, but using lines instead) polynominal is formed. The optimal solution of the polynomial system is found from the roots of its derivative in terms of a least squares residual.
ASPnL (Accurate Subset based PnL)[22] is a modified version of RPnL, to be used on small line sets for better accuracy. A drawback of this method however,

it is stated in the article that the speed is not among the best.
Mirzaei and Roumeliotis [17] proposed the PnL problem as nonlinear least squares, by solving it as an eigenvalue problem. It is solved through the eigendecomposition of a 27×27 multiplication matrix obtained through constructing a polynomial system with 27 candidate solutions.

4.2. DLT-Plücker-Lines

The process of DLT-Plücker-Lines can be devided into three parts[19]. First the input data is prenormalized to achieve good conditioning. Then there is estimated a projection matrix using homogeneous linear least squares, hence the effect of prenormalization will be reversed. Finally the pose parameters are extracted from the estimated projection matrix.
DLT Plücker-Lines takes advantage of the linear projection of 3D lines parameterized using Plücker coordinates onto the image plane, which have the benefit of higher accuracy, compared to for instance regular DLT-lines. The transformation between the 3D line $\boldsymbol{L}$ and its projection $\boldsymbol{l}$ is described by the projection matrix $\overline{P}$ as

$$\boldsymbol{l} \approx \overline{P}\boldsymbol{L} \tag{4.1}$$

where $\overline{P}$ is a 3×6 matrix. Hence, as $\overline{P}$ has 18 entries, a minimum of 9 lines are needed. $\boldsymbol{l}$ is a homogeneous 2D line in the normalized image plane, described as (l_x, l_y, l_w).

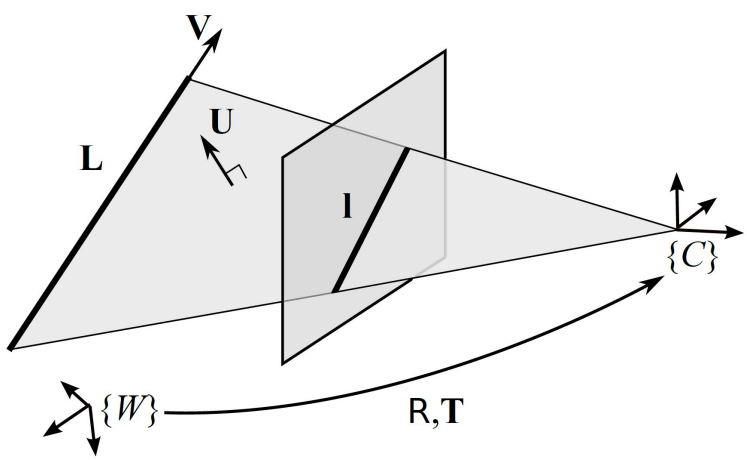


Figure 4.1.: A 3D line $\boldsymbol{L}$ parameterized using Plücker coordinates is defined by a normal $\boldsymbol{U}$ and direction vector $\boldsymbol{V}$

The transformation from the world coordinate system to camera coordinate system can be transcribed as

$$\boldsymbol{L}_C = \boldsymbol{T}_W^C \boldsymbol{L}_W \tag{4.2}$$

where T is a 6×6 line motion matrix defined as

$$\boldsymbol{T} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{R}(-\boldsymbol{t})_\times \\ 0_{3\times3} & \boldsymbol{R} \end{pmatrix} \tag{4.3}$$

$\boldsymbol{R}$ being a 3×3 rotation matrix and t being the translation vector, used for finding the skew-symmetric matrix $(\boldsymbol{t})_\times$. As shown in 2.2.8, the line projection matrix $\boldsymbol{P}$ is equal to the upper half of $\boldsymbol{T}$ and can be found by

$$\boldsymbol{P} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{R}(-\boldsymbol{t})_\times \end{pmatrix} \tag{4.4}$$

Having defined the core of camera pose estimation, using the line projection matrix $\boldsymbol{P}$, which contains all parameters to be found, $t_x, t_y, t_z, \alpha, \beta, \gamma$. This system of linear equations can in 4.1 be solved as in 3.4.2 by transforming each equation of 4.1 into a homogeneous system

$$\boldsymbol{M}\boldsymbol{p} = 0 \tag{4.5}$$

where $\boldsymbol{M}$ being a $2n \times 18$ measurement matrix containing coefficients of equations from 3D lines and projection correspondences. The matrix $\boldsymbol{M}$ is given by

$$\begin{aligned} \boldsymbol{m}^{(2i-1)} &= [l_{iw}L_{i1}\ 0\ -l_{ix}L_{i1}\ -l_{iw}L_{i2}\ 0\ -l_{ix}L_{i2}\ \cdots\ -l_{iw}L_{i6}\ 0\ -l_{ix}L_{i6}] \\ \boldsymbol{m}^{(2i)} &= [0\ -l_{iw}L_{i1}\ -l_{iy}L_{i1}\ 0\ -l_{iw}L_{i2}\ -l_{iy}L_{i2}\ \cdots\ 0\ -l_{iw}L_{i6}\ -l_{iy}L_{i6}] \end{aligned}$$

where $\boldsymbol{m}^{(j)}$ is row j of $\boldsymbol{M}$, and where the rows $\boldsymbol{m}^{(2i-1)}$ and $\boldsymbol{m}^{(2i)}$ are due to line correspondence i between $\boldsymbol{L}_i$ and $\boldsymbol{l}_i$ where $i = 1 \ldots n$, $n \geq 9$. Here $\boldsymbol{l}_i = [l_{ix}, l_{iy}, l_{iz}]^T$ are the homogeneous coordinates of a 2D line $\boldsymbol{l}_i$ in the normalized image plane, while $\boldsymbol{L}_i = [L_{i1}, L_{i2}, L_{i3}, L_{i4}, L_{i5}, L_{i6}]^T$ are the Plücker coordinates of a corresponding 3D line $\boldsymbol{L}_i$

Further the known elements in 4.1 need to be prenormalized as shown in 4.2.1.

After the data is prenormalized the given equations in 4.5 can be solved by finding the Singular Value Decomposition of $\boldsymbol{M}$. Further the projection matrix $\boldsymbol{P}$ can be made from the 18-vector $\boldsymbol{p}$.

Even though $\boldsymbol{P}$ has 18 entries it only has 6 degrees of freedom, $t_x, t_y, t_z, \alpha, \beta, \gamma$. Of 12 independent linear constraints left, the first 6 are imposing the rotation matrix $\boldsymbol{R}$ and the last 6 imposed by the skew-symmetric matrix $(\boldsymbol{t})_\times$.

4.2.1. Prenormalization

The choice of coordinate system is important for the DLT method to work properly, hence it is necessary to prenormalize the data to get the measurement matrix $\boldsymbol{M}$ properly conditioned. However, the optimal transformations are unknown so in practice the forcus is to reduce the large values of point coordinates. A way to do this is to center the data around the origin as well as scaling the coordinates so the average absolute value is 1. Hence, the average distance to the origin is $\sqrt{2}$ in 2D, and $\sqrt{3}$ in 3D.

Algorithm 5 Prenormalization of 3D lines

1: **procedure** PRENORMALIZATION($L_j, j = 1 \dots m$)
2: For all lines: $\boldsymbol{L}_j = \frac{\sqrt{3}}{\|\boldsymbol{V}_j\|}$
3: For all lines: $\boldsymbol{L}_j = \begin{bmatrix} I & [-\boldsymbol{T}]_\times \\ 0 & I \end{bmatrix} \boldsymbol{L}_j$
4: $S_x = \frac{\text{mean}(\boldsymbol{V}_j)}{\text{mean}(L_{j,1})}$
5: $S_y = \frac{\text{mean}(\boldsymbol{V}_j)}{\text{mean}(L_{j,2})}$
6: $S_z = \frac{\text{mean}(\boldsymbol{V}_j)}{\text{mean}(L_{j,3})}$
7: For all lines: $L_j = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
8: **return** Set of m prenormalized 3D lines $\boldsymbol{L}_j, j = 1 \dots m$

In this case, because the $\boldsymbol{V}$-part of the 3D line $\boldsymbol{L}$ is the direction of the line, only the $\boldsymbol{U}$-part is affected by both translation and scaling. Hence $\boldsymbol{L}$ is first multiplied so that $\| \boldsymbol{V} \| = \sqrt{3}$, followed by an applied translation so that the average magnitude of $\boldsymbol{U}$ is minimized.

4.2.2. Estimation of projection matrix

The corresponding measurement matrix $\overline{\boldsymbol{M}}$ has 18 columns and is made up by the line projection matrix $\overline{P}$ and its estimate $\overline{P}'$, both of whom with size 3×6. The number of rows are determined from the number of 3D/2D line correspondences. Each of these correspondences generates three rows structured as

$$\overline{\boldsymbol{M}}_{(3j-2:3j,:)} = \boldsymbol{L}_j^T \otimes [\boldsymbol{l}_j]_\times \tag{4.6}$$

which will be a $3m \times 18$ matrix, with the line correspondences $\boldsymbol{L}_j \leftrightarrow \boldsymbol{l}_j, (j = 1 \dots m)$.

4.2.3. Pose parameters

Before extracting the pose parameters from the estimated projection matrix $\overline{P}$ it is first necessary to scale the matrix to $s\overline{P}'$ so that all three singular values are equal to 1.

Algorithm 6 Extraction of pose parameters from $\overline{P}'$

1: **procedure** PoseExtract($\overline{P}', s$)
2: $\overline{P}'_2 \leftarrow$ right 3×3 submatrix of $\overline{P}'$
3: $U\Sigma V^T \leftarrow SVD(s\overline{P}'_2)$
4: $Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
5: $W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
6: $q = \frac{\Sigma_{1,1}+\Sigma_{2,2}}{2}$
7: Compute 2 candidate solutions (A, B):
$R_A = UW\mathrm{diag}(11 \pm 1)V^T, \quad [\boldsymbol{T}]_{\times A} = qVZV^T$
$R_B = UW\mathrm{diag}(11 \pm 1)V^T, \quad [\boldsymbol{T}]_{\times B} = qVZV^T$
8: Use the most physically plausible solution so that the scene lies in front of the camera
$R = R_A, \quad \boldsymbol{T} = \boldsymbol{T}_A$ or
$R = R_B, \quad \boldsymbol{T} = \boldsymbol{T}_B$
9: **return**$(R, \boldsymbol{T})$

From this, the camera pose parameter is found from the right part of $s\overline{P}'$, getting an estimated 3×3 skew-symmetric matrix multiplied with a rotation matrix. From this matrix, the translation can be found by

$$\boldsymbol{T} = qVZV^T \tag{4.7}$$

where V is from the SVD process, z is a where q is a variable found by

$$q = \frac{\Sigma_{1,1}\Sigma_{2,2}}{2} \tag{4.8}$$

which is the average of the first two singular values of $s\overline{P}'_2$. The rotation matrix R is found Also, so that $\det(R_A) = \det(R_B) = 1$, either $+1$ or -1 needs to be put in the last diagonal.

4.3. Premises for PnL

As the datasets that were used already consisted of predefined line segments, and matched lines, there were conducted experiments to look into the deviations. This to check whether or not the algorithm is suitable for real world applications, or the potential in different situations.

4.3.1. OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. This library, with its 2500 optimized algorithms, has been proven useful for the thesis in the way of testing several vision methods in a time efficient way. It is written natively in C/C++ and are very well documented for both C++ and Python. As most of the coding has been conducted using Python, this is a major advantage. OpenCV is built in a modular structure, some of whom that has been used in the thesis are described below:

1. **Core**: this module defines basic data structure and basic functions used by all other modules.

2. **Imgproc**: is the module for image processing, which includes functions to manipulate images, like image filtering, finding image gradients, edge detection, Hough transform etc.

3. **line_descriptor**: is a module for binary descriptors for lines extracted from an image, like Line Segment Detector.

4. **calib3d**: is a module for multiple-view algorithms, like camera calibration, 3D reconstruction and object pose estimation.

4.3.2. Line Segment Detector

The OpenCV library has a built in Line Segment Detector which can be found in the "Image Processing" module. When programming in Python the first step is to import the library by writing

```
import cv2
```

The input needed is an image that first need to be converted to grey scale for the algorithm to work. Next step is to create a smart pointer to a LineSegmentDetector Object and initialize it. There are seven different parameters that can be tweaked for specific use cases, or the standard values can be applied

```
img = cv2.imread(image,0)
LSD = cv2.createLineSegmentDetector([refine[,scale
```

```
[,sigma_scale[,quant[,ang_th[,log_eps
[,density_th[,n_bins]]]]]]]]])
```

where the parameters are

- **refine**: this parameter set if the standard parameters is to be used or to customized.
- **scale**: the scale of the image that will be used to find the lines, in the range of 0 to 1.
- **sigma_scale**: sigma for the Gaussian filter.
- **quant**: bound to the quantization error on the gradient norm.
- **ang_th**: gradient angle tolerance, given in degrees.
- **log_eps**: detection treshold.
- **density_th**: the minimum density of aligned region points in the enclosing rectangle.
- **n_bins**: the number of bins in pseudo-ordering of gradient modulus.

Next information of the line segments can be found from

```
LSD.detect(image[,lines[,width[,prec[,nfa]]]])
```

where the output parameters are

- **lines**: gives a vector of four elements as an output, representing the end-points of a line as $(x1, y1, x2, y2)$.
- **width**: gives a vector describing the widths of the regions.
- **prec**: vector of precision with which the lines are found.
- **nfa**: vector containing number of false alarms in the region.

Chapter 5.

Results

The main goals of this thesis was to implement the DLT-Plücker-Lines algorithm and then determine the possibility of using the method in SLAM. To verify, the accuracy and speed were measured on two different use cases and compared to different state of the art methods. These measurements should cover majority of applications for which pose estimation are used.
To determine the possibility of using the method in SLAM also included looking into the premises of the algorithm to work, such as detecting the line segments. Hence, the Line Segment Detector method were also evaluated, as this is considered as a crucial step.

5.1. Data

The experiments where conducted on datasets from University of Oxford and even though quite some time were used on understanding, converting and implementing the data into the code, the results were satisfactory. An advantage of using these datasets is that a lot of methods and algorithms have used the same data, hence it is easy to compare the results. Also, there are information about what is to be found which means results can easily be checked.
The datasets used in this case consisted of three images of a building as well as eleven images of a hallway, all from different views. For each image, there were three corresponding files attached, line segments given as $(x0, y0, x1, y1)$ for each line, then affine multiscale Harris corners given as (x, y), and finally a 3×4 camera projection matrix per image. In addition, the dataset consists of four more files. One file consisting 3D points in the image given as $(x\ y\ z)$ for each detected point, one with 3D line segments given as $(x0, y0, z0, x1, y1, z1)$ for each line. In addition, the last two files are information about the matching of features, one for corner matches and the other for line matches. These files describes which features that corresponds to each other in all images.
The files essential for the PnL algorithm are the line segments for each image,

the camera projection matrix', the 3D line segment file and the matching file for lines. The images has also been implemented for illustrating the results.
The dataset can be retrieved at: http://www.robots.ox.ac.uk/ vgg/data/data-mview.html

5.2. Experiments

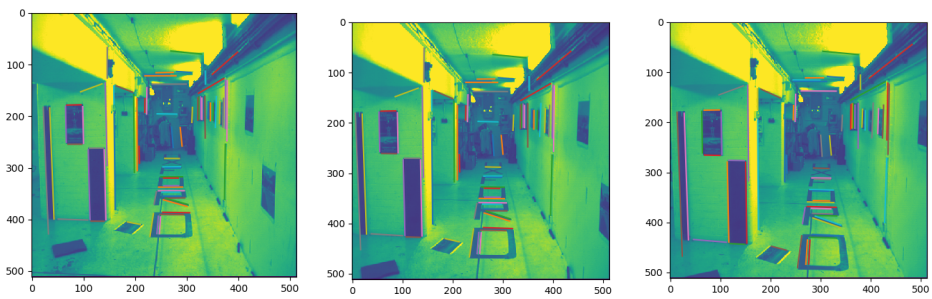
The experiments were conducted on data described in 5.1 on two quite different use cases. The first one is a corridor or hallway where the camera can clearly be observed moving further inwards. The second dataset is an outdoor environment, with a building from Oxford in focus. Another differences between the datasets except the environments are the number of lines, camera movement and number of images. The error measurements have been found by:

- **Rotation error**: is calculated as the absolute value of $(R^T R')$ and then converted from radians to degrees.
- **Translation error**: is the absolute value of the difference between estimated and true position, $\| \boldsymbol{t}_{est} - \boldsymbol{t} \|$.

The third measurement is the speed. The efficiency of all results was found by measuring runtime on a desktop PC with Intel core i7-6500U 2.50GHz CPU and 8GB of RAM.

5.2.1. Results hallway

The hallway dataset consists of eleven camera views, five of whose results are displayed, and a total of 69 line segments. In 5.2 the different images, or camera views, are overlaid with reprojections of 3D line segments found from the DLT-Plücker-Lines algorithm.



The results from each view is shown in 5.2.1, which are rotation and translation error from image to image as well as time spent for each.

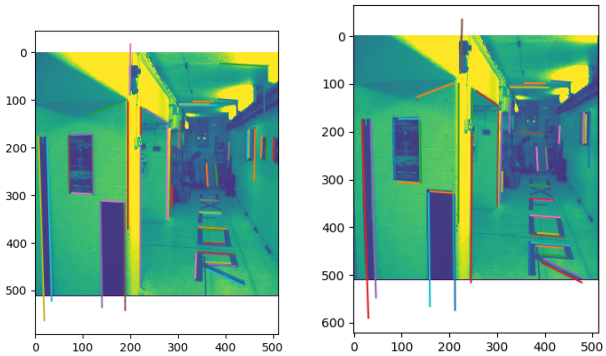


Figure 5.2.: Results of the hallway from five angles.

Image nr.	1	2	3	4	5
Rotation error (°)	0.158	0.062	0.086	0.161	0.347
Translation error (m)	0.322	0.0154	0.0175	0.0303	0.0614
Time (s)	0.327	0.206	0.225	0.209	0.240

5.2.2. Results Oxford building

The Oxford building dataset consists of three camera views, all of whom are displayed, and a total of 302 line segments. In 5.3 the different images, or camera views, are overlaid with reprojections of 3D line segments found from the DLT-Plücker-Lines algorithm.

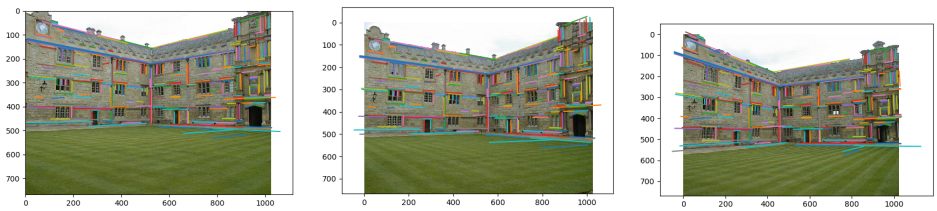


Figure 5.3.: Results of the Oxford building from three angles.

The results from each view is shown in 5.2.2, which are rotation and translation error from image to image as well as time spent for each.

Image nr.	1	2	3
Rotation error (°)	0.224	0.095	0.694
Translation error (m)	0.062	0.101	0.291
Time (s)	0.527	0.309	0.297

5.2.3. Results compared

The table 5.2.3 consists of all measurements from the different datasets, where the errors are the average of all views.

Dataset	Rotation error	Translation error	Time per image	Nr. of line segments
Hallway	0.163	0.031	0.241	69
Oxford building	0.338	0.151	0.378	302

As can be seen in the data the calculation time of the building is a little higher than the hallway, which is not unexpected as the number of lines are higher. However, the time does not seem to increase drastically with the number of lines. What might be more surprisingly is that both the translation as well as the rotation error is smaller for the hallway than for the building. Even though there are more lines in the building dataset there can be seen from the images that the distance between each camera view is far further. While the camera moves a long distance between each image in the building data, the hallway camera barely moves from image to image that can explain these results. The speed of the algorithm seems to be some high, especially if using the algorithm in a real time SLAM. That being said, the code can be written more efficient and in C/C++.
Further the results have been compared to results from other state of the art PnL methods.

	Hallway		Oxford building	
Dataset	Rotation error	Translation error	Rotation error	Translation error
Dlt-Plücker-Lines	0.163	0.031	0.338	0.151
ASPnL	0.10	0.03	0.20	0.08
Mirzaei	0.22	0.10	15.47	7.37
RPnL	0.40	0.13	0.43	0.22

As can been seen in the table 5.2.3 there is not much difference from the best displayed. The ASPnL algorithm doing only slightly better. The time is also a part of the factor, but as the results from this theses can not be directly compared, because of different hardware, programming language etc., these numbers has been left out. However, the ASPnL method has been referred to as a considerable slower compared to the rest.

5.2.4. Line Segment Detector

The Line Segment Detector algorithm were applied on the images from the datasets to compare the results to the line segments in the datasets. The results are shown in 5.4 for the line detected in the hallway images and 5.5 for the Oxford building.

Figure 5.4.: Line segments found in different images from hallway.

Figure 5.5.: Line segments found in the different camera views of the Oxford building.

In 5.2.4 both the number of line segments found and in the data is listed from each images. No refinements were applied in this case and standard parameters were used. As can be seen from the results, the number of lines found is approximately five times as high. This will result in a way higher time spent on running the matching and PnL algorithms. From the images there can be seen that the lines lies very close together in the Oxford building images, which can make them hard to match as they are so close together as well as it would be unnecessary with so many for a good results in the matching process. To get fewer and keep the better line segments, the detection treshold or the sigma for the Gaussian filter can be changed and tweaked.

		1	2	3
Hallway	From LSD	378	395	358
	From data	69	69	64
Building exterior	From LSD	1827	1699	1758
	From data	302	302	302

Chapter 6.

Discussion

As shown in the results, the accuracy of the DLT-Plücker-Lines method is very good as well as stable through all images. An interesting point worth mentioning is that the accuracy error for the Oxford building dataset is actually worse than the hallway, even the number of line segment was close to five times more. A reasonable reason for this could be that the movement between each image Hallway dataset is much smaller, hence the error with respect to total distance could be a better measurement, but unfortunately unavailable. However, the results from each images are very volatile and because there were only three images in the Oxford building dataset, the peak drastically increases the average.
The speed of the algorithm has been mentioned as an issue in the previous chapter, and if implemented into a real time system this should be focused more on. Coding in C/C++ instead of python could be a solution for making the program more efficient, but is hard to quantify. Decreasing the number of lines could also be a solution, as can be read from the results, the time spent on estimating each pose was 50% higher for the Oxford building. On the other hand, reducing the number of line segment could decrease the accuracy, and would need to be considered and evaluated from cases to case.
The Line Segment Detector results seems to have an accuracy within reason, though the output gave about five times as many line segments were detected compared to the datasets. Having in mind that there is advantageous to improve the speed for pose estimation. Because of this, the standard parameters from OpenCV should be changed for filtering the line segments, and reduce the total amount. Both the treshold as well as the sigma parameter in the Gaussian filter can be changed for this to happen.

6.1. Implementation in SLAM

As seen from the results, DLT-Plücker-Lines is a method that has great accuracy and would therefor be suitable for estimating the camera pose in a SLAM system.

However, a concern would be the speed of the algorithm and if not improved significantly, there would only be about two calculations per second. This might work if the camera is not moving at a high speed, but improving the speed would be beneficial anyway. As the methods mentioned in this thesis is based upon using line segments, man made features such as indoor environments would be very well fitted use case.
A version of SLAM is proposed in 6.1, which combines different methods described and discussed so far. The suggestion is based on EKF SLAM, using only a single monocular camera as input source.

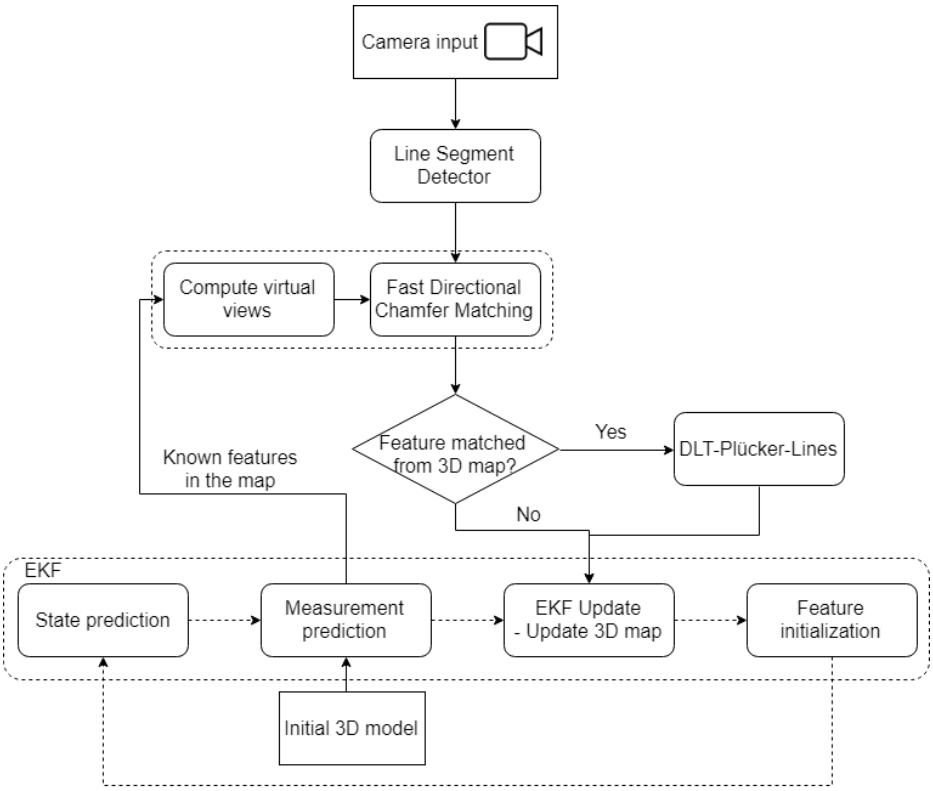


Figure 6.1.: Block diagram of how a SLAM application could be made using the methods explained in the thesis.

Fast Directional Chamfer Matching [16] has been referred to by [18] as one of the most suitable methods for such cases. However, the implementation of this method would take some time as it is somewhat advanced. The Line Segment Detector is as previous described one of the best available line segment detectors at the moment, both with respect to speed as well as accuracy. The implementation of this method is a lot harder than the well known and often used Hough Line

Transform, but using OpenCV the implementation time would be slightly longer. Using line segment instead of points, which has been the most common problem to research, will give results of higher quality when it comes to texture-less scenes, such as indoor environments. This is a result of line features being more stable than point features and more robust to occlusions. In addition, point based pose estimation is only limited to cases with enough distinctive points, which can be challenging considering indoor environments. On the other side, detecting lines is a lot more challenging than detecting points with respect to the mathematical difficulty.

Chapter 7.

Conclusion

This thesis has been focusing on pose estimation through lines along with preconditions for it to work in an independent system, such as SLAM. Some of the preconditions described are state of the art methods, both for finding line segment as well as matching line segments from a 3D model to line segments found. The DLT-Plücker-Lines method were programmed and tested on real world datasets and compared to other state of the art methods.
The results from the datasets were very promising, with an accuracy that was the second best on the datasets used and only slightly behind the best. The method were tested on two surroundings, one outdoor environment with relatively long distance between the camera views and more than 300 line segments and the other of an indoor environment with 69 line segments. The algorithm took a little more time to run on the dataset with the most lines, however, the time does not increase dramatically. On both datasets the results were among the best compared to the other methods. On the other side, a concern is the speed measured on the implemented algorithm. If being used in real-time systems 0.3 seconds per image seems a little too high. This can be improved by focusing on making the code more efficient or programming in C/C++ language.
Testing the line segment detection gave satisfying results in the form of accuracy but implemented in a system it would be beneficial to decrease the number of line segments. Being more selective when choosing line segments to use makes the system more efficient without compromising the accuracy.
Finally there were proposed a SLAM composed of the Line Segment Detector, Fast Direct Chamfer Matching, Extended Kalman Filter and DLT-Plücker-Lines. This shows a new way of using the pose estimation algortihm, DLT-Plücker-Lines, in a real world and real time applicant/system for determine the location of a moving robot simultaneously as mapping the surroundings.

References

[1] L. Moisan A. Desolneux and J.M. Morel. "Meaningful Alignments". In: *Int'l J. Computer Vision* 40.1 (2000), pp. 7–23.

[2] O Khatib B Siciliano. *Handbook of Robotics*. Springer International Publishing, 2016. ISBN: 978-3-3193-2552-1.

[3] D.H. Ballard. "Generalizing the Hough Transform to Detect Arbitrary shapes". In: *Pattern recognition* 13.2 (1981), pp. 111–122.

[4] H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. "Parametric Correspondence and Chamfer Matching: Two New Techniques for Image Matching". In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'77. Cambridge, USA: Morgan Kaufmann Publishers Inc., 1977, pp. 659–663. URL: http://dl.acm.org/citation.cfm?id=1622943.1622971.

[5] J. B. Burns, A. R. Hanson, and E. M. Riseman. "Extracting Straight Lines". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.4 (July 1986), pp. 425–455. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767808.

[6] J. Canny. "A Computational Approach to Edge Detection". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 8.6 (Nov. 1986), pp. 1129–1140.

[7] A. Elqursh and A. Elgammal. "Line-based relative pose estimation". In: *CVPR 2011*. June 2011, pp. 3049–3056. DOI: 10.1109/CVPR.2011.5995512.

[8] M. Fischler and R. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Comm. ACM* 24.6 (1981), pp. 381–395.

[9] Rafael Grompone von Gioi, Jérémie Jakubowicz, Jean-Michel Morel, and Gregory Randall. "LSD: a Line Segment Detector". In: *Image Processing On Line* 2 (2012), pp. 35–55. DOI: 10.5201/ipol.2012.gjmr-lsd.

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

[11] Laurent Kneip, Davide Scaramuzza, and Roland Siegwart. "A novel parametrization of the perspective-three-point problem for a direct computation of absolute camera position and orientation". In: June 2011, pp. 2969–2976. DOI: 10.1109/CVPR.2011.5995464.

[12] Laurent Kneip, Davide Scaramuzza, and Roland Siegwart. "A novel parametrization of the perspective-three-point problem for a direct computation of absolute camera position and orientation". In: June 2011, pp. 2969–2976. DOI: 10.1109/CVPR.2011.5995464.

[13] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. "EPnP: An accurate O(n) solution to the PnP problem". In: *International Journal of Computer Vision* 81 (Feb. 2009). DOI: 10.1007/s11263-008-0152-6.

[14] M. Liu, O. Tuzel, A. Veeraraghavan, and R. Chellappa. "Fast directional chamfer matching". In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. June 2010, pp. 1696–1703. DOI: 10.1109/CVPR.2010.5539837.

[15] Branislav Micusik and Horst Wildenauer. "Structure from Motion with Line Segments Under Relaxed Endpoint Constraints". In: *International Journal of Computer Vision* 124.1 (Aug. 2017), pp. 65–79. ISSN: 1573-1405. DOI: 10.1007/s11263-016-0971-9. URL: https://doi.org/10.1007/s11263-016-0971-9.

[16] Branislav Micusík and Horst Wildenauer. "Descriptor free visual indoor localization with line segments". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 3165–3173.

[17] F. M. Mirzaei and S. I. Roumeliotis. "Globally optimal pose estimation from line correspondences". In: *2011 IEEE International Conference on Robotics and Automation*. May 2011, pp. 5581–5588. DOI: 10.1109/ICRA.2011.5980272.

[18] B. Přibyl, P. Zemčík, and M. Čadík. "Absolute Pose Estimation from Line Correspondences using Direct Linear Transformation". In: *Computer Vision and Image Understanding* (2017). ISSN: 1077-3142.

[19] Bronislav Přibyl, Pavel Zemcík, and Martin Cadik. "Camera Pose Estimation from Lines using Plücker Coordinates". In: Sept. 2015. DOI: 10.5244/C.29.45.

[20] C. Lu S. Li and M. Xie. "A Robust $O(n)$ Solution to the Perspective-n-Point Problem". In: *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE* 34.7 (July 2012), pp. 1444–1450.

[21] W Burgard S Thrun and D Fox. *Probabilistc Robotics, Intelligent Robotics and Autonomous series*. MIT Press, 2006. ISBN: 978-0-262-20162-9.

[22] C. Xu, L. Zhang, L. Cheng, and R. Koch. "Pose Estimation from Line Correspondences: A Complete Analysis and a Series of Solutions". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6 (June 2017), pp. 1209–1222. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2016.2582162.

[23] Lilian Zhang, Chi Xu, Kok-Meng Lee, and Reinhard Koch. "Robust and Efficient Pose Estimation from Line Correspondences". In: *Computer Vision – ACCV 2012*. Ed. by Kyoung Mu Lee, Yasuyuki Matsushita, James M. Rehg, and Zhanyi Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 217–230. ISBN: 978-3-642-37431-9.

[24] Yinqiang Zheng, Yubin Kuang, Shigeki Sugimoto, Kalle Åström, and Masatoshi Okutomi. "Revisiting the PnP Problem: A Fast, General and Optimal Solution". In: Dec. 2013, pp. 2344–2351. DOI: 10.1109/ICCV.2013.291.

Appendix A.

DLT-Plücker-Lines

```
import csv
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import statistics
import time

l3DFileName = 'bt.l3d'
lineMatchFileName = 'bt.nview-lines'
img1Nr = 0
imgNrs = 11
imgPartName = str('00')
imgExt = '.jpg'

for imgNr in range(imgNrs):
    t0=time.time()
    #Read 3D line endpoints (x0,y0,z0,x,y,z):
    endPts3D_raw = read_from_file('bt.l3d')
    endPts3D_raw = np.array([float(x) for x in
        endPts3D_raw.split()])
    lengde=int(len(endPts3D_raw)/6)
    endPts3D = np.reshape(endPts3D_raw,(lengde,6))

    #Read 2D lines
    endPtsImg_temp = read_from_file('bt.00'+str(imgNr)+'.
        lines')
    endPtsImg_temp = [float(x) for x in endPtsImg_temp.
        split()]
```

```
lengde=int(len(endPtsImg_temp)/4)
endPtsImg_temp = np.reshape(endPtsImg_temp, (lengde,4)
    )
endPtsImg = np.array([0,0,0,0])

#Read 3D-2D line endpoints:
matches_raw = read_from_file(lineMatchFileName) #nview
    lines, which line matches
matches_raw = [x for x in matches_raw.split()]
matches_temp = []
for i in range(imgNr,len(matches_raw),imgNrs):
    matches_temp.append(matches_raw[i])
NLINES=0
matches = []
line3DEndPts = np.array([1, 1, 1, 1])
for i in range(len(matches_temp)):
    if matches_temp[i] !='*':
        matches.append(int(matches_temp[i]))
        line3DEndPts = np.vstack((line3DEndPts, np.
            ones((2,4))))
        n = NLINES*2
        line3DEndPts[n][0]=endPts3D[i][0]
        line3DEndPts[n][1]=endPts3D[i][1]
        line3DEndPts[n][2]=endPts3D[i][2]
        line3DEndPts[n+1][0]=endPts3D[i][3]
        line3DEndPts[n+1][1]=endPts3D[i][4]
        line3DEndPts[n+1][2]=endPts3D[i][5]

        endPtsImg = np.vstack((endPtsImg,
            endPtsImg_temp[matches[NLINES]]))
        NLINES += 1
line3DEndPts = line3DEndPts[:-1,:]
endPtsImg = endPtsImg[1:,:]

#Read camera projection matrix
matches_raw = read_from_file('bt.00'+str(imgNr)+'.P')
matches_raw = [float(x) for x in matches_raw.split()]
P_pm = [[],[],[]]
length = len(matches_raw)/3
for j in range(3):
    last = int(length*(j+1))
    first = int(j*length)
```

```
    P_pm[j]=matches_raw[first:last]
K_R_T = KR_from_P(P_pm)
K = K_R_T[0]
R = K_R_T[1]
t = K_R_T[2]

lineEndPtsImg_estim = np.array([[float(1) for col in
    range(3)] for row in range(2*NLINES)])
for i in range(len(line3DEndPts)):
    if i % 2 == 0:
        n=int(i/2)
        lineEndPtsImg_estim[i][0]=endPtsImg[n][0]
        lineEndPtsImg_estim[i][1]=endPtsImg[n][1]
        lineEndPtsImg_estim[i+1][0]=endPtsImg[n][2]
        lineEndPtsImg_estim[i+1][1]=endPtsImg[n][3]
line2DEndPts_estim = np.dot(np.linalg.inv(K),np.
    transpose(lineEndPtsImg_estim))

#Camera pose estimation
R_t_estim = linePoseEstimation(line3DEndPts,
    line2DEndPts_estim, NLINES)
R_estim = R_t_estim[0]
t_estim = R_t_estim[1]

rotError(R,R_estim)
transError(t,t_estim)

R_cam = rotMatrix2EulerAngles(R_estim)
rotX_cam = R_cam[0]
rotY_cam = R_cam[1]
rotZ_cam = R_cam[2]
TM_estim = getTransformationMatrix(rotX_cam, rotY_cam,
    rotZ_cam, t_estim)

line2DEndPts_estim = np.dot(TM_estim,np.transpose(
    line3DEndPts))
lineEndPtsImg_estim = np.dot(K,line2DEndPts_estim)

for i in range(3):
    lineEndPtsImg_estim[i] = lineEndPtsImg_estim[i,:]/
        lineEndPtsImg_estim[2,:]
```

```
    im = plt.imread('bt.00'+str(imgNr)+'.pgm')
    implot = plt.imshow(im)

    t1=time.time()
    total = t1-t0
    print('Tid: ',total)

    plt.plot([lineEndPtsImg_estim[0,::2],
        lineEndPtsImg_estim[0,1::2]], [lineEndPtsImg_estim
        [1,::2], lineEndPtsImg_estim[1,1::2]])
    plt.show()

def rotError(r_real,r_est):
    r_real_t = np.transpose(r_real)
    r_ab = np.dot(r_real_t,r_est)
    error = np.rad2deg(np.arccos((np.trace(r_ab) - 1) / 2)
        )
    print('rError',error)
    return error

def transError(t_real,t_est):
    error = (np.sqrt((t_real[0]+t_est[0])**2)+np.sqrt((
        t_real[1]+t_est[1])**2)+np.sqrt((t_real[2]+t_est
        [2])**2))/3
    print('tError',error)
    return error

def closestPoint2SetOfLines(line3DEndPts,NLINES):
    #minimizes sum of squared distances

    dim = len(line3DEndPts[0])-1
    #normalize the homogeneous coordinates of endpoints
    line3DEndPts = np.array([np.divide(line3DEndPts[:,0],
        line3DEndPts[:,3]),np.divide(line3DEndPts[:,1],
        line3DEndPts[:,3]),np.divide(line3DEndPts[:,2],
        line3DEndPts[:,3]),np.divide(line3DEndPts[:,3],
        line3DEndPts[:,3])])

    e = np.subtract(line3DEndPts[:dim,1::2],line3DEndPts[:
        dim,0::2])
    l2norm = np.sqrt((e * e).sum(axis=0))
    e = np.divide(e,(l2norm.reshape(1,len(e[0]))))
```

```
    e[np.isnan(e)] = float(0)
    oned = e.flatten('F')
    A = np.array([oned,oned,oned]) * np.kron(e,np.ones((1,
        dim)))
    M = np.eye(dim)
    for i in range(NLINES-1):
        M = np.concatenate((M,np.eye(dim)))
    M = M-np.transpose(A)

    C = line3DEndPts[0:dim,::2]
    C = np.kron(np.transpose(C),np.ones((dim,1)))
    c=np.array([])
    for i in range(len(M)):
        c = np.append(c,np.dot(M[i],C[i]))

    closest_point = np.linalg.solve(np.dot(np.transpose(M)
        ,M),(np.dot(np.transpose(M),c)))
    return closest_point

def pluckerLines(line3DEndPts):
    np.array(line3DEndPts)
    pts_start = line3DEndPts[:,::2]
    pts_end = line3DEndPts[:,1::2]
    pluck_mat_part1 = np.array([pts_start.flatten('F'),
        pts_start.flatten('F'),pts_start.flatten('F'),
        pts_start.flatten('F')]) * np.kron(pts_end,np.ones
        ((1,4)))
    pluck_mat_part2 = np.array([pts_end.flatten('F'),
        pts_end.flatten('F'),pts_end.flatten('F'),pts_end.
        flatten('F')]) * np.kron(pts_start,np.ones((1,4)))

    plucker_matrices = pluck_mat_part1 - pluck_mat_part2

    u = np.array([plucker_matrices[2,1::4],
        plucker_matrices[0,2::4],plucker_matrices[1,0::4]])
    v = np.array([plucker_matrices[0,3::4],
        plucker_matrices[1,3::4],plucker_matrices[2,3::4]])
    pluckerLines = np.concatenate((u,v))
    return pluckerLines

def getMeasurementMatrix(lines_3D, lines_2D, NLINES):
```

```
    M1 = np.array([lines_2D[2,:]*lines_3D[0,:], np.zeros(
        NLINES), -lines_2D[0,:]*lines_3D[0,:], lines_2D
        [2,:]*lines_3D[1,:], np.zeros(NLINES), -lines_2D
        [0,:]*lines_3D[1,:], lines_2D[2,:]*lines_3D[2,:],
        np.zeros(NLINES), -lines_2D[0,:]*lines_3D[2,:],
        lines_2D[2,:]*lines_3D[3,:], np.zeros(NLINES), -
        lines_2D[0,:]*lines_3D[3,:], lines_2D[2,:]*lines_3D
        [4,:], np.zeros(NLINES), -lines_2D[0,:]*lines_3D
        [4,:], lines_2D[2,:]*lines_3D[5,:], np.zeros(NLINES
        ), -lines_2D[0,:]*lines_3D[5,:]])
    M2 = np.array([np.zeros(NLINES), lines_2D[2,:]*
        lines_3D[0,:], -lines_2D[1,:]*lines_3D[0,:], np.
        zeros(NLINES), lines_2D[2,:]*lines_3D[1,:], -
        lines_2D[1,:]*lines_3D[1,:], np.zeros(NLINES),
        lines_2D[2,:]*lines_3D[2,:], -lines_2D[1,:]*
        lines_3D[2,:], np.zeros(NLINES), lines_2D[2,:]*
        lines_3D[3,:], -lines_2D[1,:]*lines_3D[3,:], np.
        zeros(NLINES), lines_2D[2,:]*lines_3D[4,:], -
        lines_2D[1,:]*lines_3D[4,:], np.zeros(NLINES),
        lines_2D[2,:]*lines_3D[5,:], -lines_2D[1,:]*
        lines_3D[5,:]])

    M = np.concatenate((M1,M2),axis=1)
    return M

def fitLineProjectionMatrix(lines_3D,lines_2D,NLINES):
    M = getMeasurementMatrix(lines_3D,lines_2D,NLINES)
    W = np.eye(NLINES*2)

    u, s, v = np.linalg.svd(np.dot(W,np.transpose(M)),
        full_matrices=True)
    proj_mat_lines = v[-1,:]
    P = np.transpose(np.reshape(proj_mat_lines, (6,3)))

    P_M = [P,M]
    return P_M

def rotMatrix2EulerAngles(R):
    Y1 = -math.asin(R[0][2])
    Y2 = math.pi - Y1
    Z1 = math.atan2(R[0][1]/math.cos(Y1),R[0][0]/math.cos(
        Y1))
```

```
    Z2 = math.atan2(R[0][1]/math.cos(Y2),R[0][0]/math.cos(
        Y2))
    X1 = math.atan2(R[1][2]/math.cos(Y1),R[2][2]/math.cos(
        Y1))
    X2 = math.atan2(R[1][2]/math.cos(Y2),R[2][2]/math.cos(
        Y2))

    angleX = (X1+math.pi)%(2*math.pi)-math.pi
    angleY = (Y1+math.pi)%(2*math.pi)-math.pi
    angleZ = (Z1+math.pi)%(2*math.pi)-math.pi

    angles = [angleX,angleY,angleZ]
    return angles

def rotationMatrix(rot_x,rot_y,rot_z):
    Rx = np.array([[1, 0, 0],[0, math.cos(rot_x), math.sin
        (rot_x)],[0, -math.sin(rot_x), math.cos(rot_x)]])
    Ry = np.array([[math.cos(rot_y), 0, -math.sin(rot_y)
        ],[0, 1, 0],[math.sin(rot_y), 0, math.cos(rot_y)]])
    Rz = np.array([[math.cos(rot_z), math.sin(rot_z),
        0],[-math.sin(rot_z), math.cos(rot_z), 0],[0, 0,
        1]])
    R = np.eye(3)
    R = np.dot(Rx,np.dot(Ry,np.dot(Rz,R)))
    return R

def getTransformationMatrix(rotX_cam, rotY_cam, rotZ_cam,
   t_estim):
    tv = np.array([[t_estim[0]],[t_estim[1]],[t_estim
        [2]]])
    Rm = rotationMatrix(rotX_cam, rotY_cam, rotZ_cam)

    T = np.dot(Rm,np.concatenate((np.eye(3),tv),axis=1))
    return T

def getProjParam(P_line_est):
    P1 = P_line_est[:,:3]
    s = 1/np.linalg.det(P1)
    if s<0:
        s = (-s)**(1/3)
        s = -s
    else:
```

```
        s = s**(1/3)

    Pscale = s*P_line_est
    P2scale = Pscale[:,3:]
    U,sigma,V = np.linalg.svd(P2scale)
    Z = [[0,1,0],[-1,0,0],[0,0,0]]
    W = [[0,-1,0],[1,0,0],[0,0,1]]
    ave_sigma_val = (sigma[0]+sigma[1])/2
    da = np.linalg.det(np.dot(U,np.dot(np.transpose(W),np.
        transpose(V))))
    db = np.linalg.det(np.dot(U,np.dot(np.transpose(W),np.
        transpose(V))))

    Ra = np.dot(np.dot(np.dot(U,W),np.diag(np.array([1,1,
        da]))),V)
    Rb = np.dot(np.dot(np.dot(U,np.transpose(W)),np.diag(
        np.array([1,1,db]))),V)

    angles_a = rotMatrix2EulerAngles(Ra)
    angles_b = rotMatrix2EulerAngles(Rb)

    tXa = np.dot(ave_sigma_val,np.dot(np.transpose(V),np.
        dot(Z,V)))
    tXb = np.dot(ave_sigma_val,np.dot(np.transpose(V),np.
        dot(np.transpose(Z),V)))
    ta = np.array([tXa[2][1],tXa[0][2],tXa[1][0]]) #skew
        symmetric
    tb = np.array([tXb[2][1],tXb[0][2],tXb[1][0]])

    x = [-ta[0],-tb[0]]
    y = [-ta[1],-tb[1]]
    z = [-ta[2],-tb[2]]

    rot_x = [angles_a[0],angles_b[0]]
    rot_y = [angles_a[1],angles_b[1]]
    rot_z = [angles_a[2],angles_b[2]]

    projParam = [x,y,z,rot_x,rot_y,rot_z]
    return projParam

def linePoseEstimation(line3DEndPts,line2DEndPts, NLINES):
    #Centroid of 3D lines endpoints
```

```
line3DEndPts = np.array(line3DEndPts)
center_3D = [np.mean(np.divide(line3DEndPts[:,0],
    line3DEndPts[:,3])),np.mean(np.divide(line3DEndPts
    [:,1],line3DEndPts[:,3])),np.mean(np.divide(
    line3DEndPts[:,2],line3DEndPts[:,3]))]

#Pre-normalization of 3D plucker lines
shift_3D = closestPoint2SetOfLines(line3DEndPts,
    NLINES)
pre_tform_3D_pts = [[float(1) for i in range(4)]for j
    in range(4)]
pre_tform_3D_pts = np.diag(np.diag(pre_tform_3D_pts))
for i in range(len(pre_tform_3D_pts)-1):
    pre_tform_3D_pts[i][3] = -shift_3D[i]
line_3D_end_pts = np.dot(pre_tform_3D_pts,np.transpose
    (line3DEndPts))

#Create Plucker representation of 3D lines
lines_3D = pluckerLines(line_3D_end_pts)

#Construct 2D line equations from projected endpoints
lines_2D = []
lines_2D = np.cross(np.transpose(line2DEndPts[:,0::2])
    ,np.transpose(line2DEndPts[:,1::2]))
#Pre-normalization of 2D lines - treat as 2D points
lines_2D_nlines = np.dot(np.transpose(lines_2D),np.
    diagflat(np.ones(NLINES))) #set centroid of lines
    as origin
shift_2D = [statistics.mean(np.divide(lines_2D_nlines
    [0],lines_2D_nlines[2])), statistics.mean(np.divide
    (lines_2D_nlines[1],lines_2D_nlines[2]))] #skjer
    noe grums her!!
pre_shift_2D_lines = np.array([[1, 0, -shift_2D
    [0]],[0, 1, -shift_2D[1]],[0, 0, 1]])
lines_2D_shift = np.array(np.dot(pre_shift_2D_lines,np
    .transpose(lines_2D)))
    #Scale lines
scale_2D = math.sqrt(2) / statistics.mean(np.sqrt(np.
    square(np.divide(lines_2D_shift[0,:],lines_2D_shift
    [2,:]))+np.square(np.divide(lines_2D_shift[1,:],
    lines_2D_shift[2,:]))))
pre_scale_2D_lines = np.zeros(9).reshape(3,3)
```

```
pre_scale_2D_lines = np.diag([scale_2D,scale_2D,1])
pre_tform_2D_lines = np.dot(pre_scale_2D_lines,
    pre_shift_2D_lines)    #combine scale and
    translation
lines_2D = np.dot(pre_tform_2D_lines,np.transpose(
    lines_2D))

#Estimate camera pose
p_line_est = fitLineProjectionMatrix(lines_3D,lines_2D
    ,NLINES)
P = p_line_est[0]
M = p_line_est[1]

P_line_est = np.linalg.solve(pre_tform_2D_lines,P)
cam_pose_param = getProjParam(P_line_est)
cam_pose_x0 = cam_pose_param[0]
cam_pose_y0 = cam_pose_param[1]
cam_pose_z0 = cam_pose_param[2]
cam_rot_x0 = cam_pose_param[3]
cam_rot_y0 = cam_pose_param[4]
cam_rot_z0 = cam_pose_param[5]

R1 = rotationMatrix(cam_rot_x0[0],cam_rot_y0[0],
    cam_rot_z0[0])
t1 = np.array([-cam_pose_x0[0],-cam_pose_y0[0],-
    cam_pose_z0[0]])
R2 = rotationMatrix(cam_rot_x0[1],cam_rot_y0[1],
    cam_rot_z0[1])
t2 = np.array([-cam_pose_x0[1],-cam_pose_y0[1],-
    cam_pose_z0[1]])

test1 = -1 * np.dot(R1[2],(center_3D-shift_3D-t1)/np.
    linalg.norm(center_3D-shift_3D-t1))
test2 = -1 * np.dot(R2[2],(center_3D-shift_3D-t2)/np.
    linalg.norm(center_3D-shift_3D-t2))

if test1>0 and test2<0:
    R = R1
    t = t1-shift_3D
else:
    R = R2
    t = t2-shift_3D
```

```
    pose = [R,t]
    return pose

def read_from_file(filename):
    f = open(filename, 'r')
    innhold=f.read()
    f.close()
    return innhold

#Extract K and R from camera matrix
def KR_from_P(P):
    P = np.array(P)
    L = int(len(P))
    H = np.zeros([L,L])
    for i in range(L):
        for j in range(L):
            H[i][j]=P[i][j]
    #H = np.array(P[:,:L])

    A = np.array(np.transpose(H))
    R, K = np.linalg.qr(A[::-1,::-1])
    K = np.transpose(K)
    R = np.transpose(R)
    K = K[::-1,::-1]
    R = R[::-1,::-1]

    K = K/(K[L-1][L-1])
    t = np.linalg.solve(-H,P[:,-1])
    K_R_t = [K,R,t]
    return K_R_t
```

Appendix B.

Line Segment Detector

```
import cv2
import math
import numpy as np

imgNrs = 9

for imgNr in range(imgNrs):
    #Read gray image
    img = cv2.imread('bt.00'+str(imgNr)+'.pgm',0)

    #Parametrization LSD
    lsd = cv2.createLineSegmentDetector(2, 0.8, 0.6, 2.0,
        22.5, 0, 0.7, 1024)

    #Detect lines in the image
    lines_2D = lsd.detect(img)[0] #first row is the
        detected lines. Each row is a detected line,
        expressed as (x0,y0,x1,y1)
    nfa = lsd.detect(img)
    print(len(lines_2D))
    drawn_img = lsd.drawSegments(img,np.array(lines_2D))
    #Show image
    cv2.imshow("LSD",drawn_img )
    cv2.waitKey(0)
```